



The DevOps Standard

*The Vendor-Neutral Model
for the AI-Driven World!*

Official Book



The DevOps Standard

***The Vendor-Neutral Model
for the AI-Driven World!***

Official Book



Unlocking your potential to achieve more

PeopleCert is the global leader in **professional, language, and vocational qualifications**. Our vision is “to empower people and organizations to achieve what they are capable of” and our mission is “to own and develop Global Best Practice frameworks, qualifications, and certifications that improve organizational efficiency and enhance the lives and careers of people”. Our core values of Quality, Innovation, Passion, Integrity, Velocity, and Clarity have remained intact since our foundation.

Positioned as a **Global House of Brands**, our qualification portfolio includes **City & Guilds®, ITIL®, PRINCE2®, DEVOPS INSTITUTE®, LANGUAGECERT®, ILM®, IASSC®** and our training portfolio **The Oxford Group®, Intertrain®, TS4U®** and **Gen2®**. With presence in 200+ countries, we deliver our portfolio to **millions of learners annually**,

55,000 leading companies (including 82% of the Fortune 500) and 800 government organizations in c.50 countries, 7,500 partners.

The driving force behind PeopleCert’s success is our team of **2,500 employees** and **3,000 associates** from **40 nationalities**, strategically located across the UK, US, EMEA, India, China, and Australia. We foster a culture of diversity, inclusion, and meritocracy. PeopleCert has received over **60 awards** across various domains, including entrepreneurship, business, technology, diversity, and sustainability.

PeopleCert has a **long-standing commitment to ESG**, actively supporting initiatives that deliver **positive social impact** while creating **long-term value** for global stakeholders and communities.

Our motto is “**Dream it, do it.**”

Published by PeopleCert International Limited
ISBN: 978-9925-39-700-6 (Digital)
ISBN: 978-9925-39-699-3 (Print)
ISBN: 978-9925-39-698-6 (ePub)
Published in Cyprus
Publication printed in Greece or reproduced electronically in Greece

Version 1.0 (October 2026)

Copyright © 2026 PeopleCert International Limited and its affiliates (“PeopleCert”)

All rights reserved. No part of this document or the information in it may be copied, distributed, disclosed or used other than as authorized by PeopleCert. Information identified as being under a Creative Commons license may be used in accordance with that license. ITIL®, PRINCE2®, DEVOPS INSTITUTE®, IASSC®, LANGUAGECERT®, and the Swirl logo are registered trademarks of PeopleCert.

AI and Machine Learning Restriction

No part of PeopleCert’s intellectual property, including but not limited to Official Books, Official Training Materials, syllabi, live exams, sample papers, mock exams, assessment materials, digital content, and any other intellectual property, may be used, reproduced, extracted, processed, mined, analysed, or otherwise exploited for the purpose of text and data mining, or for training, fine-tuning, validating, testing, or developing any Artificial Intelligence (AI) or Machine Learning (ML) system, including but not limited to Large Language Models (LLMs), foundation models, and generative AI systems.

Disclaimer

This publication is designed to provide helpful information to the recipient. Although care has been taken by PeopleCert in preparation of this publication, no representation or warranty (either express or implied) is given by PeopleCert with respect to the completeness, accuracy, or suitability of the information or advice contained within it, and PeopleCert shall not be held responsible for any loss or damage whatsoever relating to such information or advice.

DEVOPS INSTITUTE® is dedicated to the global DevOps community whose passion for collaboration, continuous improvement, and value delivery continues to advance the practice of DevOps. Their experience, innovation, and commitment to better ways of delivering value have shaped the ideas, practices, and insights shared throughout this book.



Join our
Global DEVOPS INSTITUTE
Community!



Contents

01

Chapter

PART ONE The case for the global DevOps standard

28

The state of DevOps today

30

- 1.1 Introduction 30
- 1.2 Why DevOps needs a shared standard 31
- 1.3 What organizations need 36
- 1.4 Chapter summary 39

02

Chapter

What DevOps is

40

- 2.1 Definition of DevOps 40
- 2.2 DevOps as an organization-wide system for value delivery 42
- 2.3 DevOps across ITIL, PRINCE2 Project Management, PRINCE2 Agile, and SRE 43
- 2.4 Common DevOps misconceptions 47
- 2.5 DevOps maturity: outcomes, not activities 48
- 2.6 Chapter summary 50

03

Chapter

AI-native DevOps

52

- 3.1 How AI changes the DevOps system while the fundamentals still apply 52
- 3.2 Governing AI-native delivery 56
- 3.3 AI implications for the DevOps Nine Pillars 61
- 3.4 Chapter summary 62

04

Chapter

PART TWO The DEVOPS INSTITUTE® Operating Model, architecture and practices

64

The DEVOPS INSTITUTE® Operating Model

66

- 4.1 The DOI® Operating Model as an integrated capability system 66
- 4.2 The DevOps Nine Pillars as interdependent practice domains 69
- 4.3 The DevOps Blueprint as a four-layer delivery architecture 70
- 4.4 Connecting the DevOps Nine Pillars and the DevOps Blueprint 74
- 4.5 Chapter summary 78

05

Chapter

Leadership and Collaborative Culture

80

- 5.1 Leadership as a system enabler 80
- 5.2 Collaborative Culture as a system enabler 86
- 5.3 Leadership, security, and shared ownership 92
- 5.4 Chapter summary 94

06

Chapter

Design for DevOps

96

- 6.1 Architecture as a DevOps input: decisions that enable or constrain delivery 96
- 6.2 Design enablers and governed constraints 104
- 6.3 Chapter summary 108

07 Chapter

Continuous Integration and Continuous Testing	110
7.1 Continuous Integration as a practice, not a pipeline	110
7.2 Continuous Testing: quality as a system property, not a gate	114
7.3 Chapter summary	123

08 Chapter

Continuous Delivery and Continuous Deployment	124
8.1 Continuous Delivery vs Continuous Deployment	124
8.2 Release readiness and decision governance	126
8.3 Controlled deployment, exposure, and recovery	130
8.4 Deployment frequency, release flow, and capability maturity	134
8.5 Chapter summary	136

09 Chapter

Elastic Infrastructure and Platform Engineering	138
9.1 Infrastructure as a DevOps capability	139
9.2 Platform Engineering as a discipline	142
9.3 Governed infrastructure and platform capability	148
9.4 Chapter summary	149

10 Chapter

Continuous Security	150
10.1 DevSecOps: from security gate to system property	150
10.2 Modern supply chain and AI security	155
10.3 Continuous security operations and improvement	158
10.4 Chapter summary	163

11 Chapter

Continuous Monitoring and Observability	164
11.1 Observability foundations	164
11.2 From telemetry to operational decisions	169
11.3 Extending the scope of observability	172
11.4 Alerting, on-call, and blameless learning	174
11.5 Chapter summary	178

12 Chapter

PART THREE Putting it into practice	180
Assessing where you are	182
12.1 Assessment foundations	182
12.2 Conducting and interpreting the assessment	188
12.3 Improvement investment, benchmarking, and governance	191
12.4 Honest assessment as a leadership practice	193
12.5 Chapter summary	194

13 Chapter

The DEVOPS INSTITUTE® Transformation Playbook	196
13.1 Transformation sequence: the first 90 days	196
13.2 Scaling and governing the transformation	204
13.3 Adapting and sustaining the transformation	206
13.4 Chapter summary	209

14 Chapter

Measuring what matters	210
14.1 Measurement system foundations	210
14.2 Core delivery and outcome measures	214
14.3 AI-specific metrics	221
14.4 Building a measurement programme	223
14.5 Chapter summary	226

15 Chapter

DevOps at scale	228
15.1 Why DevOps at scale is different	228
15.2 Design for scale	229
15.3 The organization-wide DOI® Operating Model	233
15.4 Chapter summary	235

16 Chapter

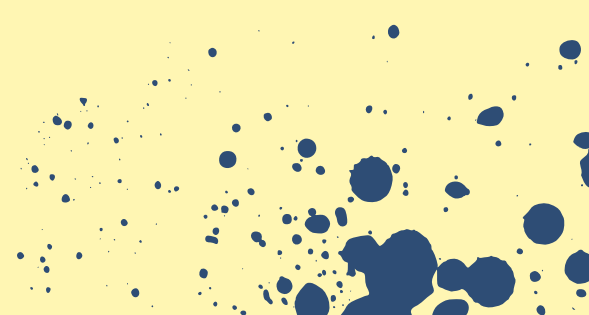
PART FOUR Future Horizons and next steps	236
Future horizons: applying DevOps principles to emerging change	238
16.1 Durable DevOps principles under emerging pressure	238
16.2 Applying DevOps principles to emerging capability	241
16.3 Chapter summary	245

17 Chapter

Your next steps	246
17.1 For leaders and decision-makers	246
17.2 For senior practitioners	249
17.3 For multipliers: consultants, trainers, and coaches	251
17.4 Chapter summary	255

Extras

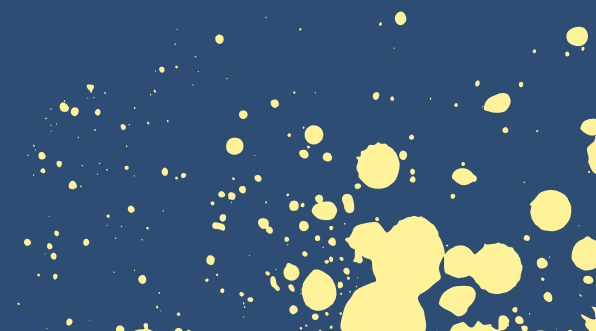
Appendix A The DEVOPS INSTITUTE® Maturity self-assessment tool	256
Appendix B Recommended metrics reference	260
Glossary	268
Bibliography	274
Index	278
Endnotes	282



List of figures

Figure 1.1	The cost of fragmentation	33
Figure 2.1	Modern DevOps definition	41
Figure 2.2	Adjacent disciplines across the value stream	44
Figure 3.1	The three roles of AI in the delivery system	54
Figure 3.2	AI output validation funnel	57
Figure 3.3	AI governance controls across the DevOps value stream	59
Figure 4.1	The DEVOPS INSTITUTE® Operating Model	68
Figure 4.2	The DevOps Nine Pillars	69
Figure 4.3	The DevOps Blueprint: four-layer delivery architecture	71
Figure 5.1	Leadership as the DevOps system enabler	81
Figure 5.2	DevOps team topologies and interaction modes	89
Figure 6.1	Architecture as a DevOps input	97
Figure 6.2	Designing for deployability	101
Figure 7.1	Continuous Integration as a team practice	111
Figure 7.2	Unified pipeline for human and AI-generated artifacts	113
Figure 7.3	Quality as a system property	115
Figure 8.1	Continuous Delivery versus Continuous Deployment	125
Figure 8.2	Application Release Orchestration and embedded governance	128
Figure 8.3	AI-assisted risk-based deployment decision model	129
Figure 8.4	Deployment is not the same as release	132
Figure 9.1	Infrastructure as a DevOps capability	138
Figure 9.2	Infrastructure as Code lifecycle	140
Figure 9.3	Internal developer platform capabilities	144
Figure 9.4	Platform team as a product team	145
Figure 10.1	Contrasting DevSecOps with traditional late-stage security	151
Figure 10.2	Shift-left security checks across the pipeline	153
Figure 10.3	Security as code and continuous compliance evidence	154
Figure 10.4	AI-specific security controls in DevSecOps	157

Figure 11.1 The DevOps Observability feedback loop	165
Figure 11.2 Monitoring versus observability decision tree	166
Figure 11.3 Telemetry pipeline architecture	167
Figure 11.4 SLI, SLO, and error budget decision model	171
Figure 11.5 AI observability control loop	173
Figure 12.1 DevOps maturity levels	184
Figure 12.2 Internal assessment process	188
Figure 12.3 From assessment to investment roadmap	191
Figure 13.1 Transformation sequence	197
Figure 13.2 The first 90 days	198
Figure 13.3 Value Stream Mapping and pilot design	202
Figure 14.1 Balanced measurement model	213
Figure 14.2 DORA in context	215
Figure 14.3 Flow metrics across the value stream	217
Figure 14.4 AI measurement stack	221
Figure 15.1 From team DevOps to organization-wide DevOps	228
Figure 15.2 The organization-wide DOI® Operating Model: core elements and relationships	233
Figure 16.1 From assisted delivery to governed autonomy	239
Figure 16.2 Operating model adaptability	240



List of tables

Chapters

Table 2.1	Complementary contributions of management and delivery disciplines	46
Table 2.2	DevOps maturity activity indicators and outcome evidence	49
Table 3.1	Agent authorization practices	55
Table 4.1	Relationship between the DevOps Nine Pillars and the DevOps Blueprint	74
Table 4.2	Automation and orchestration in the delivery system	75
Table 4.3	Applications and evidence for each of the DevOps Nine Pillars	77
Table 5.1	Leadership responsibilities	82
Table 5.2	Leadership shift from control to enablement	83
Table 5.3	Leading indicators of DevOps culture	91
Table 6.1	Different validation levels and their purposes	100
Table 6.2	Design review checklist	109
Table 7.1	Layered test strategy across the pipeline	118
Table 8.1	Release readiness evidence checklist	127
Table 8.2	Deployment patterns for controlled exposure	131
Table 8.3	From release event to release flow	134
Table 9.1	Cost governance as an engineering feedback loop	146
Table 10.1	Minimum software-supply-chain evidence before promotion	156
Table 11.1	AIOps use, practical purpose, and required control	169
Table 11.2	Observability maturity path	177
Table 12.1	Assessment criteria across the DevOps Nine Pillars	185
Table 12.2	Interpreting common DevOps maturity patterns	187
Table 13.1	Transformation cases	200
Table 13.2	Transformation 90-day startup activities	203
Table 13.3	Transformation governance practices and pitfalls	205
Table 13.4	Transformation anti-patterns and corrections	207
Table 13.5	Context-specific starting points	208
Table 14.1	Types and use cases of measurement	210
Table 14.2	DORA metrics use	216

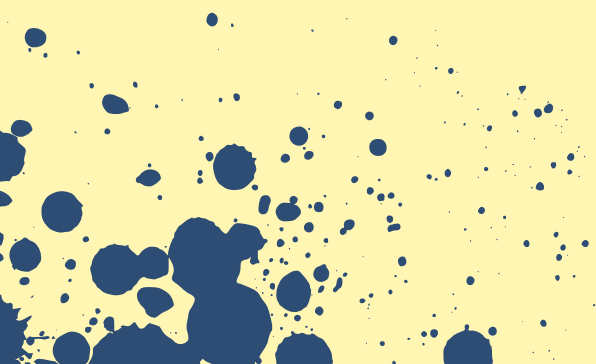
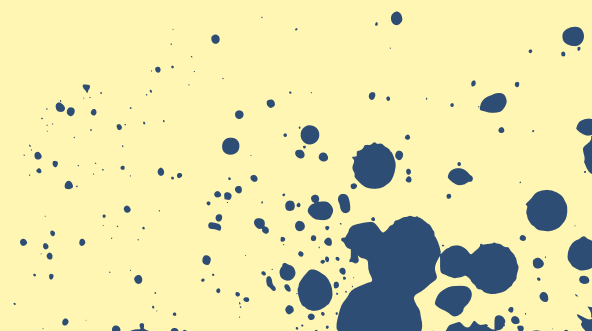
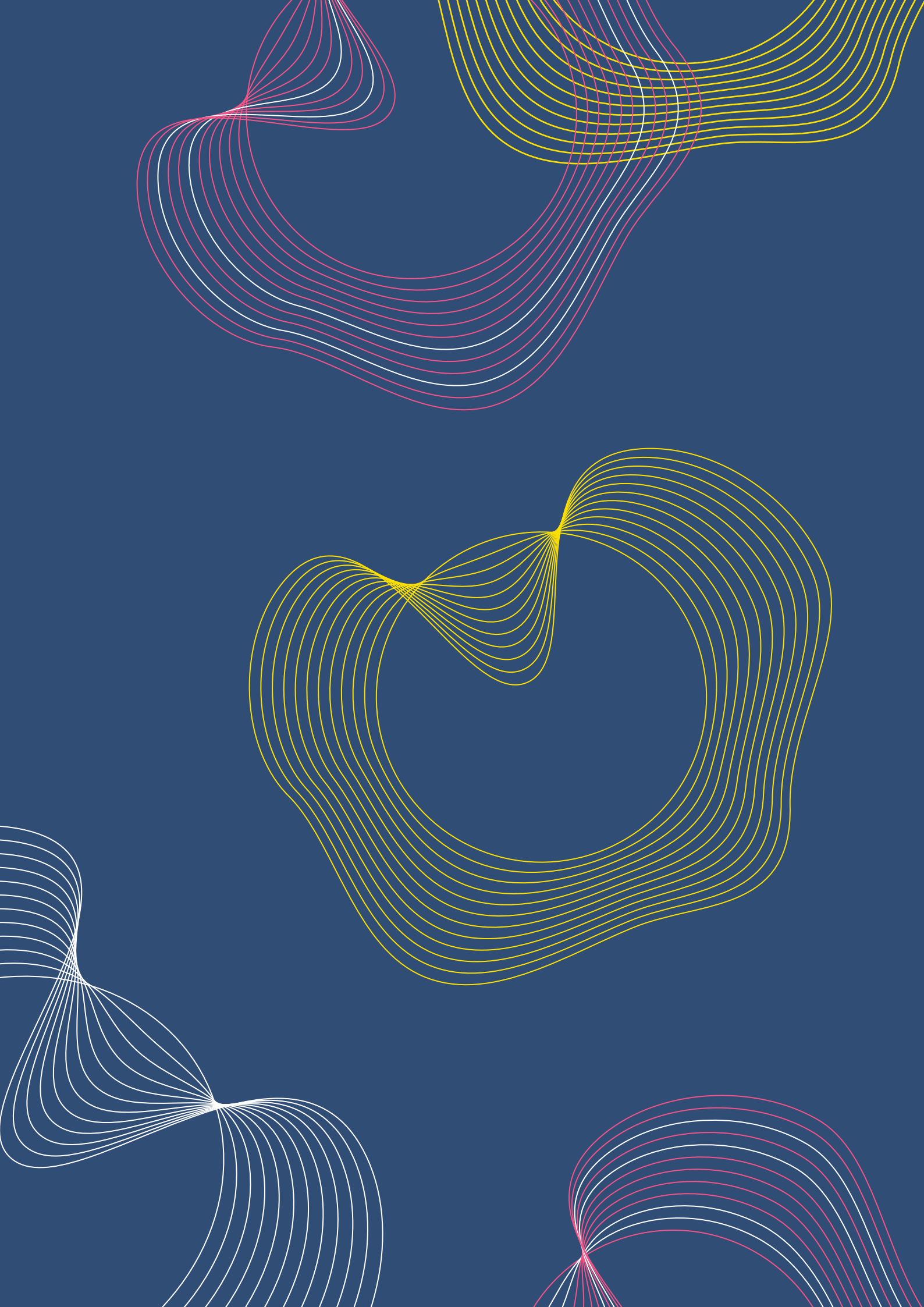


Table 15.1	Team-level and organization-wide design concerns	229
Table 15.2	Platform-portfolio decisions at scale	231
Table 15.3	Scaling DevOps from teams to organizational capability	234
Table 16.1	Evaluating an emerging capability	242
Table 16.2	Evaluation decisions and their meaning	242
Table 16.3	DevOps response to change in various domains	243
Table 17.1	Translating delivery issues into business impact	248
Table 17.2	Senior practitioners' guidance	250
Table 17.3	Typical DevOps transformation workshops, practical outputs, and evidence of success	253
Table 17.4	Guidance for leaders, practitioners, and multipliers	254
Table 17.5	Your DevOps next step	254

Extras

Table A.1	DevOps assessment fields and their purpose	256
Table A.2	Maturity levels and scoring guidance for the DevOps assessment	257
Table B.1	Stepwise metric selection	260
Table B.2	Metric definition template	261
Table B.3	Flow metrics definitions and interpretations	262
Table B.4	Quality metrics definitions and interpretations	263
Table B.5	Stability metrics definitions and interpretations	263
Table B.6	Value metrics definitions and interpretations	264
Table B.7	DORA metrics: definitions, collections, and interpretations	265
Table B.8	Pillar-aligned metrics	266
Table B.9	AI-specific metrics	267





Foreword by the CEO

At PeopleCert, we believe that excellence is a journey, not a destination.

From the very beginning, our vision has been clear: to empower organizations and individuals to achieve what they are capable of. Our core values of Quality, Innovation, Passion, Integrity, Velocity and Clarity have guided every decision we make since our inception in 2000, enabling us to become the global leader in professional, language and vocational qualifications. Our product portfolio, delivered in more than 200 countries every year, now includes multiple iconic qualification brands (City & Guilds®, ITIL®, PRINCE2®, DEVOPS INSTITUTE®, LANGUAGECERT®, ILM®, IASSC®) and training brands (The Oxford Group®, Intertrain®, TS4U® and Gen2®).

While PeopleCert is a house of brands, we consider it our responsibility to honour each brand as our most cherished one, nurtured with care and pride. As a family-owned business with a long-term horizon and perspective, we are committed to preserving the legacy of DEVOPS INSTITUTE and acting as its guardian and steward. DEVOPS INSTITUTE is the leading player in the vendor-neutral DevOps certification sector, adopted worldwide by organizations and professionals across every industry, delivering software-driven value reliably and at scale, in the AI-driven world.

A heartfelt thank you goes to our community, whose trust, dedication, collaboration, and commitment have been central to our shared success. Through our extensive network of 7,500 partners, 3,500 recognized academic institutions, and our award-winning technology ecosystem, we are proud to serve millions of learners every year, including 55,000 leading companies (82% of the Fortune 500) and 800 government organizations. To date, we have received more than 60 prestigious awards, and in 2023, PeopleCert was included as an INSEAD Business Case Study, following its emergence as the first unicorn of Greek origin and serving as a model of global success. This would not have been possible without you.

As we look ahead, we remain dedicated to our purpose-driven success. As an AI frontrunner, we are committed to transforming, leading, and empowering our ecosystem. Our journey is powered by our community and by our unwavering belief that, together, we will continue to shape the future of skills.

Dream it, do it!

Byron Nicolaides

Chairman and CEO, PeopleCert



Preface

DevOps has transformed how organizations deliver technology-enabled products and services. What began as a movement to improve collaboration between development and operations has evolved into a foundational capability for modern organizations. Today, DevOps is no longer emerging. It is embedded in how organizations innovate, compete, and create value. Yet despite its widespread adoption, DevOps remains one of the least consistently defined disciplines in technology.

The world in which DevOps operates has changed dramatically. Cloud computing, platform engineering, site reliability engineering, automation, and increasingly artificial intelligence have reshaped how software is built, deployed, and operated. AI is no longer an emerging technology at the edge of delivery. It is becoming an active participant in the delivery system itself, generating code, producing tests, assisting decisions, and accelerating change at a pace never before possible.

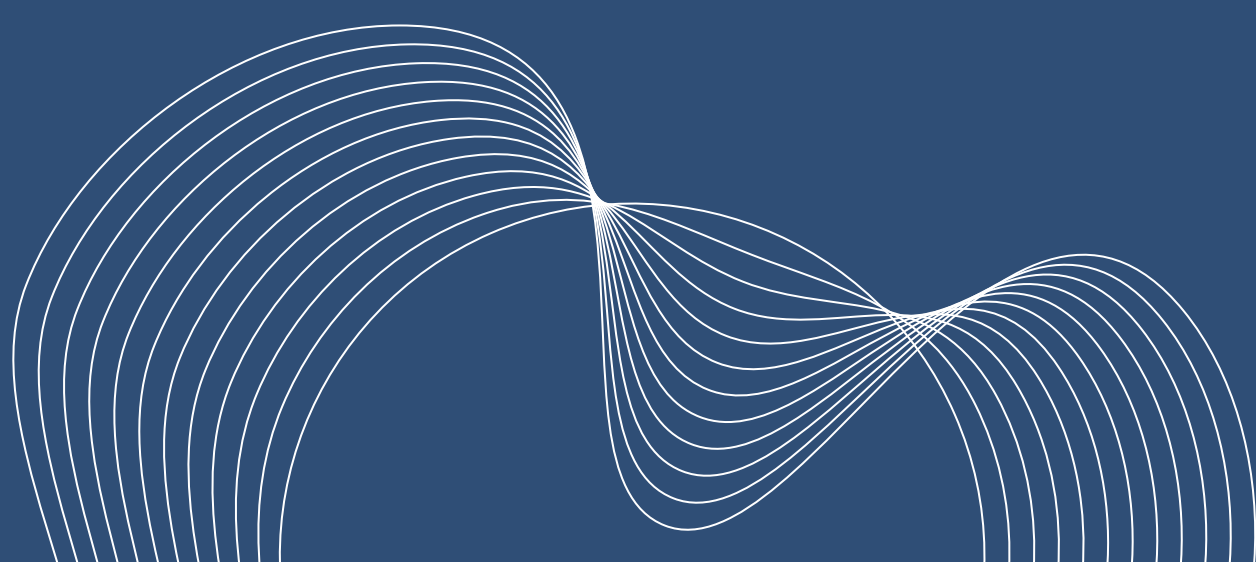
Yet speed alone does not create value.

As organizations adopt new tools, platforms, and AI capabilities, many find themselves facing a different challenge: fragmentation. Different teams define DevOps differently. Vendors promote their own versions of DevOps through proprietary platforms and ecosystems. Organizations invest heavily in technology while struggling to establish shared practices, governance, accountability, and meaningful ways to assess progress. The result is often inconsistency, complexity, and difficulty translating DevOps activity into business outcomes.

This reality creates an urgent need for a common reference.

Not another tool guide. Not another collection of isolated practices. Not another technology-specific implementation playbook.

What organizations need now is a vendor-neutral standard that defines DevOps as a complete operating model: one that integrates people, processes, technology, governance, measurement, and continuous learning into a coherent system for delivering value in the AI-driven world.



This book was created to meet that need.

The DEVOPS INSTITUTE® DevOps Standard establishes a common language, structured operating model, and practical framework that organizations can use regardless of industry, technology choices, methodology, or level of maturity. It positions DevOps not as a collection of tools or a team structure, but as an organization-wide system for value delivery that connects strategy, engineering, operations, security, governance, and business outcomes.

Importantly, this edition recognizes a reality that previous generations of DevOps guidance could not fully address the rise of AI-native delivery. As AI accelerates the creation of software and decisions, organizations must simultaneously strengthen validation, governance, observability, security, and accountability. The future belongs not to organizations that simply adopt AI faster, but to those that can govern and operationalize it effectively within a trusted delivery system.

This book introduces the DEVOPS INSTITUTE® Operating Model, the DevOps Nine Pillars, and the DevOps Blueprint, providing organizations with a practical and assessable framework for building mature, scalable, and AI-enabled DevOps capabilities. Whether you are a leader shaping organizational strategy, a practitioner improving delivery performance, or an educator helping others build capability, this book is intended to serve as both a guide and a reference throughout your journey.

DevOps has reached an important point in its evolution.

The industry no longer needs another interpretation of DevOps. It needs a shared standard.

This is that standard.

The DEVOPS INSTITUTE® DevOps Standard.

For practitioners and leaders.

For organizations of every size.

And for everyone committed to delivering value in an AI-driven world.

The DEVOPS INSTITUTE® Team



Contributors



With sincere appreciation, we acknowledge and thank our global community for shaping DEVOPS INSTITUTE into what is today. We are deeply grateful for your continued support, and we recognize your outstanding contributions.

Share the excitement, be part of the DEVOPS INSTITUTE movement, and let's celebrate together!

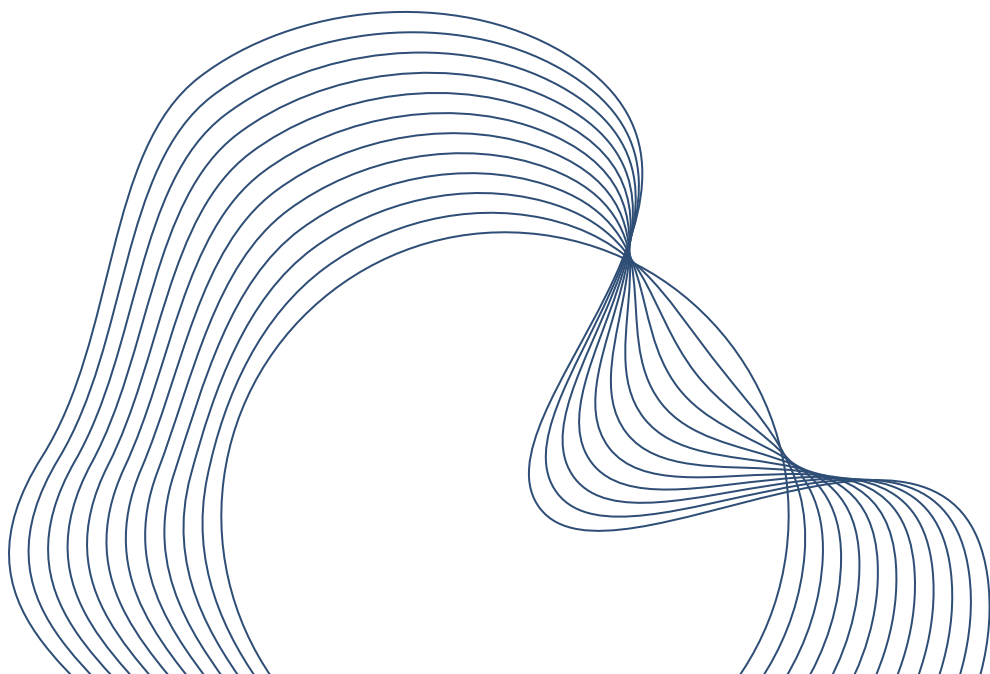
DEVOPS INSTITUTE portfolio has been developed under the strategic direction and active stewardship of **Eleni Nicolaides** and **Markus Bause**.

Special thanks also go to our internal team who worked tirelessly to bring this book to life including Dimitris Georgiadis, Adam Griffith, Dimitris Hatzidakis, Faidra Inta, Anastasia Kalogirou, Spyros Kaoukis, Panagiota Kokkosi, Konstantina Lili, Vicky Mavragani, Marina Petritsi, Rinku Sachdeva, Richa Thapliyal, Neill Treacy, Susan Tuttle, Roman Zhuravlev.

Marc Hornbeek

Lead Contributor & Senior DevOps Advisor, DEVOPS INSTITUTE®

Marc Hornbeek, known as DevOps-the-Gray, is an award-winning engineer, executive advisor, author, educator, and transformation leader with five decades of experience in software and systems engineering. He specializes in DevOps, DevSecOps, SRE, Platform Engineering, Continuous Testing, AI-native delivery, and governance. Marc has led more than 95 transformations across enterprises, manufacturers, service providers, and government organizations. He is an IEEE Outstanding Engineer and Life Member and the author of books on secure, reliable, and human-centred engineering.



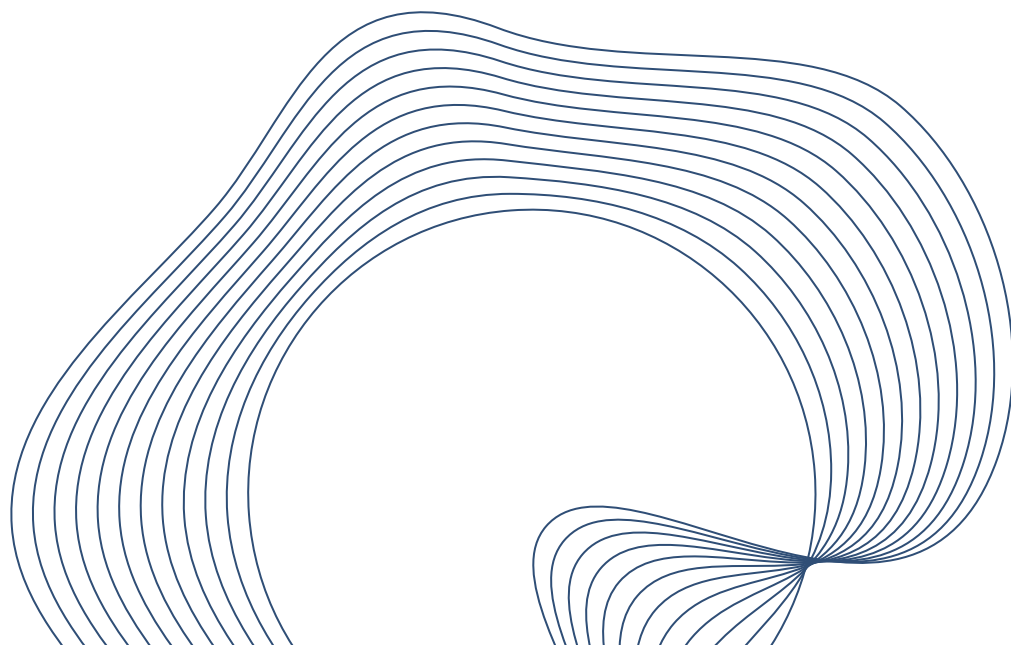
We are deeply grateful to the following professionals for their dedication, expertise and active engagement in shaping this book.

Ken Adachi (Dynatrace), Gemini Agaloos (Marriott International), Sunil Agarwal (Cisco System Inc), Viralkumar Ahire (Adobe), Priyank Ahuja (Accenture), Mojtaba Alfardan (Artificial Labs), Luis Allegretti (Choice Properties), Abdelalah Alzaghloul (Qomrah 360), Christopher Amick (Peraton), Wissal Amine (Leyton), Chin Tiong Ang (Kuok Group Singapore), Andrés Felipe Mahecha Arevalo (TaskUs), Matías Armándola (Mati Armándola), Luz Alicia Acevedo Avila (Tata Consultancy Services - Santander Peru), Wiem Batita Ayari (Ahead Technologies), Ayokunle Ayoola (BOOFEM INC), Mina Aziz (Hitachirail), Thierry Bachmann (IB SA), Lucas Renan Vicente Bandeira (Capgemini), Indrani Basu (VISA), Helen Beal (PeopleCert), Jean-Claude Beaudry (Qualiti7 Canada), Vipin Bhardwaj, Debashis Bhattacharyya (Opus Technologies), Signe-Marie H. Bjerke (Glasspaper), Jorge Blanco (Glumin Networks SC), Krzysztof Bona (Bayer), Catherine Bordinat (Qualiti7), Roman Bartolomej Borovsky (T-Systems), Christophe Bourcier (CB CONSEILS), Alexander Bremer (SERVIEW GmbH), Kunliang Cao (Shenzhen Aito Xianfeng Enterprise Management Consulting Co., Ltd.), Jose Luis Perez Carmona (COSMO CONSULT), Ahmet Turan Cetin, Martin Chan, Eliezer Chavez (NTT Data), Alberto Ciardelli (HRV.SWISS), Lukasz Ciecwierz (MB Group), Marc Cluet (UBS), Ethan Cohen (Blue Soft Group), Matthias Cohn (EWERK), Massimo Colombo (Bip), Julio Corteza (Orchard Software), Joaquin Crespo (NATO), Alejandro Debenedet (GUETCHU), Eckhard Dechau (ICH), Mieke Deenen (Portaal), Stephen Dick (BetterCloud), Ja'Mesa Dixon (Zayo Group), Paulina Dubas (Dubas Consulting), Bernd Ebert (iTSM Consulting GmbH), Antonio Elizondo (Global Advisory Solutions), Gabriel Espinosa (Everforth Apex Systems), Paul Farrow (HM Treasury), Xiao Feng (Norn Technology),

Leon Fester (IBM), Michael Fosong (CompuFIX (UK) Ltd.), Patricia Fridman (Comuni), Anton Genchev (DraftKings Inc.), Radu Mihnea Gheorghiu (Serview GmbH), Stephen Gilmer (Ericsson), Mareike Glanz (DATAGROUP Operations GmbH), Aj Godahewa (Caterpillar), Sesha Kiran Gonaboyina (IEEE), Eva López Gorria (Advanced Quality Solutions), Adam Griffith (PeopleCert), Dr. Atul Grover (Advance Solutions Corp.), Suresh Gupta (TaUB Solutions), Alex Harding (Runshaw College), Gábor Harsányi (Providing Technical Leadership Kft.), Yazmani Reyes Hernandez (Epicurus Fund Management Asesor Independiente), Pham Minh Hoang (VNPT IT), David Hogan (Service Reliability Engineering), Ali Honina (YKB), Tibor Horvath (British Airways), Stephen William Howard, Lauri Humina (Wakaru Oy), Marcelo Inhesta, Feisal Ismail (Sapience Consulting Pte Ltd), Ramkumari Iyer (Reliscale consulting Pvt Ltd), Maciej Jarosz (IF ELSE), Jakub Jasinski (P&G), Ajinkya Kadam (Microsoft), Anesu Kafesu (ZimWorX), Aray Kaken (AGILEPM HUB), Sunil Kamarajugadda (Robert Half), Sekhar Khanal (CloudCast Nepal), Ihab Khiri (Workplace Consultants BV), Stephane Kittler (DS2I), Wania Konageski (Logicalis), Marcin Krakowiak (ATOS), Radhakrishnan Krishna Kripa (ANSYS INC), Helena Krol-Horvath (CGI), Madha Lakshminarayanan (Microsoft), Yuk Wa Law (HKBN Enterprise Solutions), Wai-man Lee (XBO Technologies), Benjamin Lemoine (Bouygues Construction IT), Zhichao Li (Huaxi Securities), Eleftherios Liakakos (NSPA), Toto Liong (NSW), Jens Lorscheider (Landesamt für Geoinformation und Landentwicklung (LGL) - Baden-Württemberg), Alcides Magule, Jack Maher (ValueStreamPro.com), Catherine Malcolm, Sercan Marangoz (IBM), Roberto Marini (DSV), Ray Marshall (Sopra Steria), Belisario Martinic (Pontificia Universidad Católica de Chile), Sam McDonald (SitusAMC), Ross Andrew Mckellar (Paypal), Vishal Mehta (Rabobank), Ernesto Olivares Mena (Project Management Institute),

Evgenii Metelkin (Ministry of Public and Business Service Delivery and Procurement | Ontario Public Service), Kedi Mhelembe, Danylo Mikula (Medidata Solutions), Caspar Miller (Teknologisk), Rahul Miragi (Syngenta), Abhishek Mishra (High hope services), Kaihei Miyamoto (Cantabia Consulting, Ltd.), Adit Modi (Lauren), Simone J. Moore (SJM), Victorio Mosso (ANALYTICA MTY), Philip Müller (Agile Forge), Santhosh Nc (Thoughtworks), Pavlo Nesterov (EPAM), Alysson Ribeiro das Neves (BNDES), Clint Nieuwendijk (In Groupe), Giuseppe Nigro (Almaviva Spa), William David O'Donohoe (HealthcareE), Eveline Oehrlich (Serviceware), Sjoerd Olink (ALDI Sued), Sven Ossenber (Glenfis AG), Mourad Ouannes (MED TRAINING AND CONSULTING), Michael Oudermans (LLMO), Patrizia Panichi (Municipia SpA), Venkata Reddy Pasam (Boardwalk Pipelines), Arpita Patel (Alabama Water Institute), Shilpan Patel (PrimeOps Limited), Agnieszka Pawłowska (UBS), Michael Pearson (proAV), Arun Pandiyan Perumal (Adobe), Mark Thomas Peters II (BrainGu), Priyanka Janet Austin Phoras (MSD), Davi Piovezan (MRS), Krzysztof Politowicz, Chandra M R (CMR Excel), Ali Ramadan (Global Knowledge Middle East & Africa), Jayakumar Ramalingam (SiriusXM), Nathalie Rijkssen (Kembit), Junior Omar Castro Riofrio (BANCO DE CRÉDITO DEL PERÚ), Sam Rosbergen (SN),

Michael James Rossiter (Enablis), Alberto Gavancho Ruiz, Vignesh Kumar P S (Kyndryl), Roberto Salas (Segex), Claude Sam-foh (York University), Neelima Savardekar (The Ohio State University), Marc Schachteli (Ninja Training), Mahendran Selvakumar (Tata Consultancy Services), Salman Shaikh (Royal Cyber Inc.), Faisal Adnan Siddiqui (Mindfluent Solutions Ltd), Cheryl Simpson (Learning Tree International), Durgesh Singh (Tata Consultancy Services), Satbir Singh (Cisco System Inc), Subhaan Singh (Eduqe), Hassan Smahi (Experis France), Andy Stevens (Chenega Analytic Business Solutions), Rossana Suarez (Naranja X), Marcin Szybkowski (ImmerSim), Nikolaos Tantouris (ENISA), Ajanta Basu Tarasiuk (Worldline), Doug Tedder (Tedder Consulting LLC), Jenny Thomas (TK Elevator Australia), Ernest Wee Keong Toh (United Overseas Bank Limited), David "tommo" Tomlinson (QA Ltd), Mahamadou Traore (MTEC), Susan Tuttle (PeopleCert), Anuj Tyagi (RingCentral Inc), Sergiu Ungureanu (YPTO Belgium), Wieger Vellema (BT), Jonathan Wafford (Eli Lilly and Company), Richard Wijers (ITIL4MASTER), Linda Sandberg Wrånge (Länsförsäkringar), Xinkuan Yeh (Delta), Sunoj Kumar Yeluru (Wipro Ltd), Abba Soungui Younouss, Eric Yuen (Digitaleads Education), Fahmi Yusuf (Worley), Roman Zhuravlev (PeopleCert).





How to use this book

This book is designed as a structured reference, not a linear narrative. While it can be read from beginning to end, its primary value lies in how it supports different roles, use cases, and stages of DevOps maturity. Readers are encouraged to engage with the content in a way that aligns with their responsibilities and objectives.

Persona-based navigation

The content is organized to support three primary audience tiers. Each group will find specific value in different parts of the book:

- **Leaders and executives** should focus on the core doctrine, the DevOps Blueprint, and the sections on governance, measurement, and outcomes. These areas provide the strategic context needed to guide transformation, establish accountability, and align DevOps with business objectives.
- **Practitioners and engineers** should focus on the practice domains and architectural layers. These sections define the capabilities required to build, integrate, test, secure, and operate systems effectively. They provide clarity on how individual practices fit into the larger system.
- **Assessors, auditors, and educators** should focus on the structured definitions, capability models, and appendices. These sections support evaluation, benchmarking, certification, and training. They provide the consistency and precision required for objective assessment.

This persona-based structure allows each reader to navigate the book with purpose, rather than treating it as a single, uniform experience.

Reading paths

There are three primary ways to use this book, depending on the reader's intent:

- **Full read (system understanding).** Reading the book sequentially provides a complete understanding of DevOps as a system. This approach is recommended for those seeking to build a comprehensive mental model, including how principles, capabilities, and architecture connect.
- **Reference use (targeted application).** The book is designed to function as a reference. Readers can navigate directly to specific domains, capabilities, or architectural components as needed. This is particularly useful for practitioners solving specific problems or designing parts of the system.
- **Assessment use (evaluation and benchmarking).** The structured nature of the framework enables its use as an assessment tool. Readers can evaluate their current state against the defined capabilities and outcomes, identify gaps, and prioritize improvements. This approach is relevant for transformation programmes, audits, and maturity assessments.

Each reading path reflects a different stage of engagement, from understanding to implementation to evaluation.

Navigating the DevOps Standard

The DevOps Standard brings together several connected concepts and areas of focus for different purposes. The following map shows how each is used and where readers can find the primary guidance for assessment, design, delivery, operation, measurement, transformation, and scale.

Focus	How it is used	Primary reference
DevOps definition	Establishes the common definition of DevOps as an organization-wide system for value delivery.	Chapter 2
AI-native DevOps	Defines how AI-assisted and AI-enabled work is validated, governed, monitored, and kept accountable.	Chapter 3
The DEVOPS INSTITUTE® (DOI®) Operating Model	Provides the integrated structure that connects the book's practice and implementation views.	Chapter 4
DevOps Nine Pillars (part of DOI® Operating Model)	Define the nine interdependent practice domains. Each is inclusive of the three practice dimensions: people, process, and technology.	Chapter 4 Chapters 5–11
DevOps Blueprint (part of DOI® Operating Model)	Provides the four-layer implementation architecture: Elastic Infrastructure, CI/CD Pipeline, Application Release Orchestration, and Value Stream Management.	Chapter 4
Assess	Establishes current maturity, evidence quality, constraints, and improvement priorities using the DEVOPS INSTITUTE® Maturity self-assessment tool.	Chapter 12 Appendix A
Design	Applies architecture and design principles for observability, testability, deployability, security, operability, and governed change.	Chapter 6
Deliver	Applies Continuous Integration, Continuous Testing, Continuous Delivery and Deployment, infrastructure, platforms, and security.	Chapters 7–10
Operate	Uses observability, operational feedback, SLOs, incident learning, and production evidence to operate and improve the system.	Chapter 11
Measure	Defines balanced measurement across delivery, flow, quality, stability, value, and AI-specific outcomes.	Chapter 14 Appendix B
Transform	Provides the practical sequence from assessment and the first 90 days through scaling and sustained capability building.	Chapter 13
Scale	Defines the enduring organization-wide operating design across teams, value streams, platforms, governance, architecture, and funding	Chapter 15

Note: For ease of reading, DOI® is used as an abbreviation for DEVOPS INSTITUTE® where appropriate throughout this book.

Understanding the format

To support clarity and practical application, the book uses several recurring format elements:

- **Key messages and tips** present real-world observations and insights drawn from organizational experience. They highlight how concepts are applied in practice and where challenges commonly arise.
- **Anti-pattern boxes** identify common failure modes and misconceptions. They are intended to help readers recognize and avoid patterns that lead to poor outcomes.
- Recurring composite cases and worked examples apply the operating model to three continuing organization contexts. Four anchor cases appear as numbered subsections; shorter continuations sit beside the practice they illustrate.
- **Decision and action guides** identify the executive decision, the chapter-relevant roles expected to act, and the evidence that should be reviewed.
- **AI-related content boxes** highlight chapter-specific implications of AI-native delivery. Core AI concepts and controls are defined in Chapter 3 and remain part of the main narrative where AI materially changes the practice being discussed.

These elements are not supplementary. They are integral to the structure of the book, reinforcing both understanding and application.

Recurring case threads and worked examples

This edition uses three recurring composite organizations instead of introducing a different case in every chapter. The cases¹ synthesize recurring patterns from published research, standards, and practitioner material. Names, organizational details, sequences, and any uncited quantitative values are illustrative; no case represents one named organization.

- **Case A - Northstar Financial Services:** a regulated multinational financial-services organization with a mixed legacy and cloud estate, more than one hundred product and platform teams, demanding evidence obligations, and expanding use of AI-assisted engineering.
- **Case B - Harbor Digital Commerce:** a high-volume customer-facing digital product organization using cloud platforms, an internal developer platform, progressive delivery, and Site Reliability Engineering (SRE) practices while managing reliability, security, and cost pressure.
- **Case C - CivicLink Public Services:** a citizen-facing critical service delivered across multiple suppliers, with accessibility, privacy, continuity, public accountability, and high-stakes AI considerations.

Readers may follow one case thread across chapters or use each example independently. The four anchor cases in Chapters 2, 4, 13, and 15 are included in the table of contents. Other chapters use shorter case continuations or worked-example boxes placed next to the practice they demonstrate.

Integration with DEVOPS INSTITUTE® learning pathways

This book serves as a foundational body of knowledge for the DEVOPS INSTITUTE® qualification portfolio within PeopleCert. Readers are encouraged to use this book in conjunction with:

- DEVOPS INSTITUTE® certification pathways, which support structured learning and validate understanding and application of the framework.
- Implementation playbooks, which provide practical guidance for applying the framework in real-world environments

Together, these elements form a complete system for learning, applying, and advancing DevOps capabilities.

What comes next

This book is not an endpoint. It is a starting point for structured transformation. Readers are expected to move beyond understanding to application. This includes assessing current capabilities, defining target states, and implementing changes aligned with the framework. The goal is not simply to adopt DevOps practices, but to operate DevOps as a coherent, measurable system.

Each persona has a role in this process:

- **Leaders** establish direction and governance.
- **Practitioners** build and operate the system.
- **Assessors** and **educators** ensure consistency and continuous improvement.

By using this book as both a reference and a guide, organizations can move from fragmented implementations to mature, high-performing DevOps systems aligned with modern, AI-enabled delivery.

This section provides the foundation for that journey.



Please take a few minutes to give us your feedback on the DevOps Standard Official Book by completing the online evaluation survey [here](#).

Icon map

**KEY
MESSAGE**



DEFINITION



TIP



SCENARIO



**AI-RELATED
CONTENT**





The case for the global DevOps standard

PART 1

Chapter 1

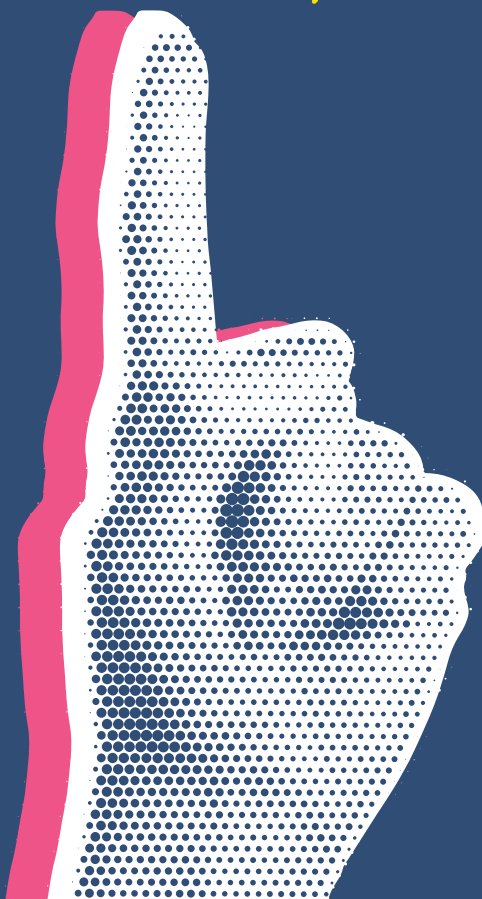
The state of DevOps today

Chapter 2

What DevOps is

Chapter 3

AI-native DevOps



DevOps is no longer emerging. It is established, widely adopted, and foundational to how modern organizations deliver software. Yet despite this maturity, it remains one of the least consistently defined disciplines in organizational technology. Organizations invest in DevOps initiatives, implement tools, restructure teams, and adopt new practices, but often without a shared understanding of what DevOps is as a system. The result is uneven performance, difficulty in scaling, and an inability to measure outcomes with confidence. This part of the book establishes the case for why a global DevOps standard is necessary now.

The industry has reached an inflection point. The speed and complexity of software delivery have increased significantly, driven by cloud platforms, distributed architectures, and more recently, AI-assisted development. These advances have amplified both capability and risk. Code can be generated faster than ever before, but without corresponding advances in validation, governance, and system-level design that speed does not translate into reliable outcomes. Organizations are no longer struggling to adopt DevOps. They are struggling to make it work consistently at the level required by modern systems.

At the same time, the DevOps landscape has become fragmented. Vendors provide powerful platforms and tools, but each defines DevOps through the lens of its own ecosystem. Practices vary across teams and organizations. Guidance is often either too abstract to implement or too tied to specific technologies to generalize. Without a unifying structure, organizations are left to assemble their own interpretation of DevOps, often resulting in gaps between intention and execution.

PART ONE addresses this problem directly.

Chapter 1

Examines the current state of DevOps and the consequences of operating without a standard. It makes the case for a vendor-neutral, structured approach by highlighting the real costs of fragmentation, including tool sprawl, inconsistent practices, and failed transformations.

It also clarifies the role of vendors and platforms, positioning them as essential enablers but not substitutes for a coherent operating model.

Chapter 2

Establishes what DevOps is. It introduces the book's precise definition of DevOps and explains its organization-wide scope. It clarifies the relationship between DevOps and adjacent frameworks such as ITIL, PRINCE2, PRINCE2 Agile, and Site Reliability Engineering, resolving common misconceptions and positioning these disciplines as complementary components within a unified model.

Chapter 3

Introduces AI-native DevOps as a defining characteristic of modern systems. Artificial intelligence is not treated as an external enhancement, but as an integrated participant in the delivery system. This shift requires new approaches to validation, governance, and accountability. The chapter establishes the foundational principles for incorporating AI into DevOps while maintaining control, reliability, and trust.

Together, these chapters establish the intellectual foundation for the rest of the book. They define the problem space, clarify the core concepts, and introduce the perspective that underpins the DEVOPS INSTITUTE® Operating Model. This section does not attempt to provide implementation guidance. Its purpose is to create alignment. It ensures that the reader understands not only what DevOps is, but also why a structured, shared reference is necessary before moving into how it is realized.

1.1 Introduction

DevOps is now part of normal enterprise technology work. Most organizations have adopted some combination of pipelines, cloud platforms, automated testing, infrastructure automation, observability tools, and cross-functional practices. Many teams release more frequently than they did a decade ago, and many leaders can explain the basic idea of DevOps without needing a long definition (Forsgren et al., 2018; DORA, 2024).

That adoption has created a false sense of maturity. In many organizations, DevOps remains a collection of effective but disconnected practices rather than a coherent system for delivering value. One team may excel at continuous integration, another at deployment controls, and a third at incident response. Yet the organization may still face inconsistent ownership, uneven measurement and governance, excessive manual coordination, and cultural barriers to shared accountability.

This chapter is most useful for leaders, transformation sponsors, senior practitioners, and assessors who need a clear explanation of why DevOps now requires a structured, shared reference. It sets the problem before the book introduces the DEVOPS INSTITUTE® Operating Model.

The chapter also establishes an AI-native premise. AI increases the speed and volume of change entering the delivery system. That speed raises the required standard for validation, governance, measurement, security, and operational feedback.

AI-assisted engineering further highlights gaps in the delivery system. AI can rapidly generate code, tests, configuration as code, documentation, operational summaries, and deployment recommendations, increasing the volume and pace of change entering the value stream (see Glossary). This creates sustainable value only when the delivery system can absorb it through appropriate validation, governance, security, telemetry, and accountability. As change volume grows, weaknesses in the delivery system become constraints sooner (DORA, 2025). This is why the conversation about DevOps has changed. Organizations are no longer asking whether DevOps is useful; they are asking how to make it consistent, assessable, governable, scalable, and fit for AI-native delivery.

A useful standard provides a shared language, a stable reference model, and a structured way to assess progress while allowing organizations to adapt practices to their context. It helps distinguish isolated activity from system capability and connects culture and collaboration with flow, quality, reliability, security, governance, and value. The present-day gap is therefore not a lack of DevOps practices, tools, or guidance, but the absence of a common, vendor-neutral structure that integrates them into an organization-wide system.

This book addresses that gap through the DOI® Operating Model, supported by the DevOps Nine Pillars and DevOps Blueprint. The model is designed to be adapted across organization sizes, sectors, regions, regulatory environments, operating models, and technology landscapes, including AI-native delivery.

The state of DevOps today

1.2 Why DevOps needs a shared standard

DevOps emerged from practice². It grew out of operational pain, failed handoffs, slow releases, fragile production systems, and the frustration of teams that understood the cost of separating development from operations. Early practitioners experimented with automation, collaboration, continuous integration, Infrastructure as Code (IaC), and faster feedback because they needed working answers, not formal doctrine (Humble & Farley, 2010; Kim et al., 2021).

That origin mattered. A rigid standard in the early years would have slowed discovery. Teams needed room to test patterns, adapt to local conditions, and learn from production experience. The field advanced because practitioners shared what worked through conferences, internal engineering communities, blogs, books, and open discussion. In this way, its strength came from practical learning.

DevOps spread before it was formally defined and that history still creates ambiguity today. Different organizations interpreted it through their own pressures. Some treated it as a cultural initiative, while others treated it as automation. Some created a team called DevOps, whereas others bought a toolchain and assumed the operating model would follow. These interpretations share vocabulary, but they often produce different behaviours. What once supported experimentation now creates friction on an organizational scale.

Leaders struggle to compare maturity across value streams. Practitioners struggle to know which capabilities matter most. Risk and audit teams struggle to assess whether controls are improving or being bypassed. Lastly, training teams struggle to create a consistent development path for engineers, testers, operations staff, security professionals, and platform teams.

A DevOps standard should therefore describe a working system. It should define the practice domains, show how those domains connect, describe maturity, identify governance expectations, and give organizations a way to measure improvement, including the additional validation, provenance, and accountability requirements introduced by AI-native delivery.

1.2.1 The cost of fragmentation

Many DevOps environments that look mature are still fragmented in practice. Fragmentation occurs when teams adopt useful practices in isolation. While one team automates builds, others introduce deployment scripts, add security scanning tools, or build dashboards. Meanwhile, product teams track delivery in a separate planning system. Each improvement has value. The combined system still contains gaps, handoffs, duplicated effort, and weak feedback (Forsgren et al., 2018; Kim et al., 2021).

Tool sprawl is a visible part of the problem. Organizations accumulate tools for source control, build automation, and test automation, along with deployment and security.

They also add tools for vulnerability scanning, observability, incident response, service management, and portfolio reporting. These tools rarely share the same definitions of work, risk, ownership, and evidence. Consequently, people create manual workarounds to connect the systems and keep work moving.

Inconsistent practice is harder to see, which weakens feedback and learning. One team may require peer review for every change while another relies on informal approval. One team may maintain reliable test automation while another depends on manual testing. Similarly, one team may use feature flags and progressive release while another still coordinates large deployment windows. Organizational leaders see uneven outcomes and often misread the cause as team competence rather than system design.

Fragmentation also weakens governance. Controls applied differently across teams produce inconsistent evidence. Security checks may be automated in one pipeline and handled through email in another. Approval records may exist, but the decision's basis might be unclear. AI-generated code, scripts, or recommendations add further complexity because the organization must capture provenance and validation evidence consistently across the delivery system.

The implication is clear: fragmented DevOps increases the cost of change. As work waits in queues, teams duplicate effort, production issues recur, and leaders struggle to identify the real constraint. Consequently, the organization may invest heavily yet still experience slow flow, poor reliability, or weak audit confidence.

Figure 1.1 shows how these forms of fragmentation combine to increase delay, rework, instability, and audit pain as change volume rises.

Anti-pattern: “We bought the tools, we did DevOps”

A common failure mode appears when leaders equate tool adoption with DevOps maturity. The organization buys pipeline tools, cloud services, dashboards, test automation, security scanners, and AI assistants. The investment is visible, procurement can report completion, and teams may become busier. The underlying operating model may remain unchanged.

The symptoms are familiar. Teams wait for approvals that are disconnected from the evidence in the pipeline. Security findings still arrive late and create release pressure. Environments drift.

Incident reviews produce weak follow-through. Product leaders make roadmap decisions without production telemetry. AI-generated code may enter the repository faster, but review, testing, and governance do not keep pace.

Tools can make a good system faster, but they can also make a weak system fail more quickly. Automation increases the rate at which work moves through existing structures. If those structures contain poor ownership, weak validation, fragile environments, or unclear decision rights, automation exposes the weakness. AI increases the same effect because it accelerates artifact creation and operational analysis (Kim et al., 2021; DORA, 2025).

This anti-pattern persists because tools are easier to buy than operating models are to change. Tool implementation has a budget line, a schedule, and a completion date. Changing ownership, incentives, governance, and measurement requires leadership discipline. It also needs patience because the meaningful outcomes appear through sustained improvement rather than a single deployment of technology.



Figure 1.1 The cost of fragmentation

"We bought the tools, we did DevOps"

The belief: Tool adoption proves DevOps maturity. The organization expects improved outcomes because modern platforms and automation are in place.

The reality: Tools provide capability. The operating model determines whether that capability produces value. Without shared

ownership, validated flow, embedded governance, trustworthy telemetry, and measurement, the organization has acquired components rather than built a system.

The better pattern: Start with the value stream, identify the constraint, define the control model, select tools that support the model, and measure outcomes. AI-generated and human-generated work should enter the same governed delivery system.

KEY MESSAGE



A mature organization asks a different set of questions that reveal the health of the DevOps system more accurately than a tool inventory:

- Can teams integrate change safely?
- Can the organization prove the evidence behind a release decision?
- Can production telemetry influence the next backlog decision?
- Can security controls operate continuously?
- Can AI-assisted work be traced, validated, and explained?

1.2.2 What cloud giants and tool vendors provide and what they do not

Cloud providers and major tool vendors have advanced DevOps practices in important ways. They provide scalable infrastructure, managed services, and pipeline capabilities, along with deployment automation. They also offer observability services, identity controls, and security tooling, alongside increasingly sophisticated AI-assisted engineering capabilities. These capabilities have reduced barriers that once made modern delivery difficult.

The value of these platforms is real. A team can provision infrastructure in minutes and create repeatable deployments with AI-assisted development. Major cloud providers also publish adoption frameworks and operating guidance that address organizational capabilities, governance, people, processes, and transformation within their ecosystems.

The distinction is also important. Cloud providers and vendor platforms define capabilities and operating guidance primarily in relation to their own ecosystems. The DOI® Operating Model complements this guidance by providing a vendor-neutral structure for applying DevOps across mixed technology environments. Organizations still need to establish clear governance across several key areas. They must decide how work is owned and how maturity is assessed, while defining

how risk is accepted and evidence is captured. They also need to determine how outcomes are measured and how AI-generated outputs are reviewed.

This distinction matters because most organizations operate in mixed environments. They may use more than one cloud platform and maintain legacy systems while integrating third-party, open-source and customized platforms. Furthermore, they may use internal tooling and rely on external partners. Their DevOps system must work across that reality.

Cloud providers and platform vendors provide valuable adoption and operating guidance within their respective ecosystems. The gap addressed by this book is different: organizations working across multiple clouds, platforms, legacy systems, delivery domains, partners, and governance obligations need a vendor-neutral structure that integrates these capabilities across the organization.

AI reinforces this need for cross-ecosystem integration. Platforms may provide code generation, model services, deployment insights, automated recommendations, and operating guidance, but the organization remains accountable for how those capabilities are governed, validated, evidenced, measured, and used in production. The DOI® Operating Model provides the common structure within which these different technologies and forms of guidance can operate together.

1.2.3 Addressing gaps in existing DevOps guidance

DevOps literature has helped the industry. Important books have explained DevOps principles, values, culture, flow, continuous delivery, reliability engineering, measurement, and systems thinking. Many leaders and practitioners learned the language of DevOps through these works. Their contribution is respected because the field would be weaker without them (Humble & Farley, 2010; Beyer et al., 2016; Forsgren et al., 2018; Kersten, 2018; Hornbeek, 2019; Kim et al., 2021).

Case A - Northstar Financial Services (composite): fragmentation after local success

Northstar had several product teams with strong local delivery practices. One used trunk-based integration and automated security checks; another had dependable deployment automation; a third had mature incident response. Each team had selected tools and approval habits that suited its immediate context. AI coding assistants and incident-summary pilots were also introduced under different local data rules.

The problem: The organization problem appeared when leaders asked cross-value-stream questions. Release-readiness evidence used different names and formats. Deployment and release events were not consistently separated. Because risk classifications varied, a low-risk change in one division could require a different path in another. Audit teams assembled evidence manually, and as a result, portfolio leaders could not see where work was waiting.

The solution: Northstar did not mandate one toolchain. It defined a minimum operating contract: common terminology, a shared release-readiness evidence schema, organization risk tiers, standard deployment and release events, and provenance requirements for AI-assisted artifacts. Teams could keep local tools when those tools produced the required evidence and respected the common guardrails.

The first effect was uncomfortable. Teams had to expose hidden assumptions and retire some local shortcuts. The longer-term effect was less manual reconciliation, clearer conversations regarding dependencies, and more consistent evidence for risk and audit decisions.

Transferable lesson: Scale reusable patterns and evidence contracts, not identical toolchains. Local optimization becomes organization capability only when the organization can understand, govern, and learn from the whole value stream.

SCENARIO



The remaining gap is structural and prescriptive. Much of the literature emphasizes one part of the system more than the whole. Culture-focused guidance often explains why collaboration and learning matter, but gives less detail on organization governance, assessment, maturity, and operating architecture. Technical guidance often explains pipelines, deployment, infrastructure, or observability with depth, but can leave leaders without a complete set of practices and an integrated model for scaling across organizations' portfolios and regulated environments.

Other sources are tied closely to a specific technology choice, toolchain, vendor, or platform. They are valuable when the organization has already selected a technology path. They become less useful when the organization needs a neutral reference across multiple technologies, operating models, and risk contexts. A standard must work across environments because most organizations cannot simplify their reality to one platform or one method.

A modern reference must provide practical engineering practices for a comprehensive DOI® Operating Model. The remaining need is therefore for a vendor-neutral, structured, and prescriptive reference that connects culture, engineering practices, governance, measurement, maturity, transformation, and AI-native controls within one operating model. It should give leaders enough structure to direct investment, practitioners enough specificity to build and improve the system, and assessors enough clarity to evaluate maturity consistently.

This book does not replace the best existing literature. It organizes the work into a structured reference that can be taught, used, assessed, governed, and improved. Readers should study other works on continuous delivery, Site Reliability Engineering (SRE), Platform Engineering, security, testing, and value stream management. The purpose here is to provide connective tissue.

AI-native delivery adds a newer limitation to established DevOps guidance: earlier literature remains useful, but it was not written for systems in which generated artifacts, AI-supported decisions, and agentic workflows are routine. This book addresses that gap through the operating model, while Chapter 3 provides detailed treatment of validation, provenance, governance, monitoring, and accountability.

AI-RELATED CONTENT



1.3 What organizations need

The gaps described above point to a practical requirement: organizations need a common operating model that helps them understand current capability, identify constraints, sequence improvement, govern risk, and measure whether the system is getting better.

The model must describe the practices required for flow, quality, reliability, security, governance, and value. It must address infrastructure and platform concerns, provide meaningful measurement, support organizational scaling, and incorporate AI-native controls for generated artifacts and AI-supported decisions.

Practical transformation also requires sequencing. A team with weak testing should not rush towards high deployment frequency.

A product with poor observability should not increase release exposure aggressively. A regulated value stream cannot treat governance as an afterthought. The next investment should follow the organization's goals, current capability, risk, and principal system constraint.

Organizations also need common language. A clear, well-documented operating model creates enough clarity for meaningful conversation, helps leaders ask better questions, and enables practitioners to identify and explain missing capability without reducing DevOps to tools, culture, automation, cloud migration, or any single practice.

The necessary guidance has the following five characteristics. It must be:

- structured, so the whole DevOps operating system is visible
- actionable, so teams can use it without translation into corporate slogans
- measurable, so progress can be assessed
- scalable, so local success can become an organizational capability
- AI-aware, so speed, validation, governance, and accountability evolve together.

1.3.1 The DEVOPS INSTITUTE® position: above vendors, not against them

The DEVOPS INSTITUTE® position is vendor-neutral by design. It sits above toolchains and platform ecosystems to define a vendor-neutral operating model that integrates capabilities across technology and organizational boundaries. This positioning is important because organizations need both strong technical capabilities and a stable, prescriptive model with clearly defined practices for integrating them across teams, value streams, and governance requirements.

Being above vendors does not mean being indifferent to tools. DevOps depends on

working technology. Pipelines, infrastructure automation, observability, security scanning, deployment orchestration, and AI-assisted engineering environments all matter. Vendor frameworks also provide valuable adoption and operating guidance within their respective ecosystems. The point is that these capabilities and guidance must work together across the organization's different technology ecosystems. Without a vendor-neutral structure for integrating them, tools and practices can remain fragmented across platforms and local environments rather than forming an organization-wide DevOps capability.

A vendor-neutral framework helps organizations retain choice. For instance, a company may use one platform for cloud infrastructure, another for source control, a third for observability, and a fourth for service management. It may also use AI capabilities from multiple sources. To manage this complexity, the DOI® Operating Model provides a way to assess, integrate, and govern DevOps capabilities across these ecosystems without tying the definition of DevOps to any one provider. Ultimately, vendor neutrality means the operating model remains stable while tools, cloud services, AI services, and platform choices change underneath it.

This matters for audit, compliance, and risk management. Control owners need to understand how delivery evidence is created and how it can be trusted. Security leaders need to know where controls live in the workflow. Executives need to see whether investment in platforms and automation is improving flow, stability, and value. Practitioners need a model that allows them to improve engineering practices without being trapped in a vendor argument.



1.3.2 How this book is different

The DEVOPS INSTITUTE® DevOps Standard responds to the gaps through an integrated, organization-wide delivery model. It does not reduce DevOps to culture, automation, cloud adoption, deployment frequency, or a team name. Those elements are all important, but none of them are sufficient. The system matters because outcomes emerge from the interaction among practices of people, processes, technology, governance, measurement, and learning across software, data, AI, security, reliability, and transformation.

The DOI® Operating Model provides the overall structure. The DevOps Nine Pillars define the practice domains that must mature, while the DevOps Blueprint shows how DevOps systems are implemented through infrastructure, pipelines, release orchestration, and value stream management. Later chapters provide practical guidance for assessment, transformation, measurement, scaling, and future adaptation.

The DevOps Standard is useful for organizations at any level of maturity, whether they are starting their DevOps journey or already have some capabilities. It recognizes that readers are dealing with real constraints: legacy systems, regulatory obligations, distributed teams, scarce talent, production incidents, security risk, fragmented tooling, and AI adoption that is moving faster than governance. The guidance is therefore practical, prescriptive, and operational. It is intended to help people make better decisions in all conditions.

For this reason, the requirements of AI-enabled delivery are integrated throughout the model, with each chapter addressing the relevant implications for validation, provenance, oversight, observability, risk-based controls, and accountability.

DECISION AND ACTION GUIDE

Executive decision: The executive sponsor decides whether DevOps will be governed as an organization-wide value-delivery capability rather than a tooling programme, names the accountable sponsor, and authorizes a baseline for the priority value streams.

Role-specific actions:

Transformation and engineering leaders map fragmentation; product, platform, operations, security, risk, and finance roles identify conflicting definitions, queues, controls, and evidence practices.

Evidence to review: A current-state baseline, fragmentation map, tool and control inventory, named sponsor, agreed outcome hypothesis, and date for the first evidence review.

The book separates assessment from judgement. Maturity assessment should reveal the state of the system. It should not be reduced to a simple ranking exercise, or a way to assign criticism. Organizations improve when they can see constraints clearly, do the work required to remove them, learn from evidence, and continuously improve. That is the practical intent behind the model.

The DevOps Standard is a structured reference that is prescriptive without being rigid. It defines what must be true for DevOps to operate well, while allowing organizations to implement practices in ways that fit their specific goals, technologies, risk, scale, and context. That balance is necessary because DevOps has to adapt to widely different applications and ever-changing trends. It must support both fast-moving start-ups and global platforms, while simultaneously meeting the strict demands of regulated organizations, public sector bodies, and AI-enabled delivery environments.



1.4 Chapter summary

DevOps has reached a critical stage of maturity. The early period of discovery established useful principles and practices. Modern DevOps requires consistency, assessment, governance, measurement, and scale. Organizations need a standard because the work has become too important and too complex to depend on local interpretation alone.

The problems described in this chapter are practical. DevOps activity is widespread, but DevOps' capabilities remain uneven. Fragmentation creates delay, risk, and inconsistent outcomes. Vendor platforms provide essential technical capability, but organizations still need a neutral operating model that explains how those capabilities work together.

The DevOps Standard book addresses this need by providing a shared language, a capability model, an architectural view, and a basis for assessment and improvement. It respects the value of tools while making clear that tools require structure, governance, and measurement to produce reliable results.

The next chapter defines what DevOps is. This chapter establishes why the definition matters. An organization cannot govern, assess, improve, or scale a system that it cannot clearly describe.

02.

This chapter establishes the DEVOPS INSTITUTE® definition of DevOps and the central thesis of this book. It clarifies how DevOps connects adjacent disciplines and corrects the misconceptions that keep organizations busy without making them more capable.

DevOps is an organization-wide system for value delivery. It connects people, practices, platforms, governance, measurement, and learning so that software-enabled services can be changed safely, operated reliably, and improved continuously. This chapter gives the reader the shared language used throughout the rest of the book. Chapter 1 explained why the industry needs a shared reference. Chapter 2 defines the system that this reference describes.

The definition matters because loose language creates loose operating models. When one leader uses DevOps to mean tools, another uses it to mean culture, and a third uses it to mean a team name, the organization cannot assess maturity with any confidence. It can buy platforms and still leave the delivery system largely unchanged.

Leaders should use this chapter to establish a common definition before funding transformation. Practitioners should use it to connect daily engineering work to the wider operating model. Assessors, trainers, and coaches should use it as the vocabulary baseline for maturity assessment and improvement planning.

Clarity about the definition of DevOps is especially important in AI-native delivery, where human and AI-assisted work are governed through the same DevOps model; the specific validation and governance implications are developed in Chapter 3.

2.1 Definition of DevOps

DEVOPS INSTITUTE®

A socio-technical system that integrates people, process, and technology practices to support the efficient, safe, and reliable delivery of software-enabled products and services³.

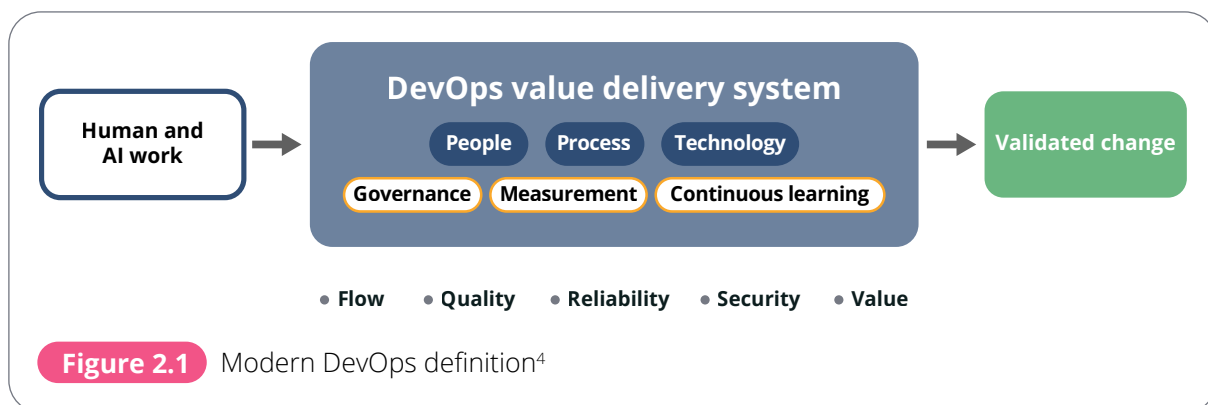
DEFINITION



This definition is deliberately broader than tooling, but narrower than a slogan. Beyond core engineering practices and operational responsibility, this system integrates strategic product direction, platform capability, robust measurement frameworks, and proactive leadership behaviour. It also makes value delivery the purpose of DevOps rather than treating local activity as the goal.

As shown in Figure 2.1, DevOps operates as an organization-wide system for value delivery. Human and AI work enter this value delivery system. Within it, people, practices, platforms, automation, governance, measurement, and continuous learning turn work into validated change and produce flow, quality, reliability, security, and value.

What DevOps is



The key criterion for evaluating a practice is straightforward. If a practice does not improve the organization's ability to deliver useful change safely and repeatedly, it is supporting activity rather than DevOps capability. A pipeline may be fast, but if it moves poorly designed work into production without evidence, it increases risk. A dashboard may look mature, but if it does not change decisions, it is decoration.

The definition also places governance inside the system. This is important in organizations where delivery must satisfy reliability, security, audit, compliance, financial, and customer obligations. Governance should guide the flow of work with clear policies, automated evidence, risk thresholds, and exception paths. Slow approval queues usually indicate that governance has been placed outside the delivery system.

This definition gives organizations a basis for assessment. It allows leaders to ask whether teams have clear ownership, whether work flows effectively, whether feedback is fast

enough, whether security and governance are embedded, and whether production learning improves the next change. These questions turn DevOps from an opinion into an assessable capability model.

A useful definition should influence decisions. It should shape how teams are funded and platforms are built, while guiding how controls are embedded, how incidents are reviewed, and how progress is measured. A definition that cannot be applied in a design review, a maturity assessment, or a transformation plan is too weak to be useful.

The definition also protects the organization from cosmetic adoption. If DevOps is a value delivery system, a tool rollout cannot be presented as successful transformation. If governance is part of the system, approval queues outside the workflow become visible as design problems. If learning is part of the system, production incidents and user feedback become inputs to improvement rather than exceptions to be ignored.

AI-native delivery follows the modern DevOps definition: AI-generated artifacts and AI-assisted decisions enter the delivery system as work that must be validated, traced, secured, and governed with clear accountability. AI reduces some manual effort and raises the importance of disciplined integration and governance control.

2.2 DevOps as an organization-wide system for value delivery

DevOps starts to work when an organization treats delivery as an end-to-end system. That system begins before a developer writes code and continues after the service is running in production. It includes demand, design, build, test, and security, as well as release, operation, measurement, incident learning, and product improvement.

Many organizations improve one part of this system and treat that improvement as evidence of DevOps maturity. A team may automate builds while release approval remains manual. Another may improve deployment speed while incident learning remains weak. A third may adopt observability tooling while product decisions continue to depend on anecdote. These partial improvements are of value, but they do not create system performance on their own.

A value delivery system requires shared ownership across functions, with each group contributing a distinct part of the whole. Product teams decide which problems are worth solving, while engineering teams design and change the system. Operations and SRE teams bring production knowledge, resilience patterns, and incident discipline, ensuring that delivery remains reliable in real conditions. At the same time, security and risk teams define guardrails and acceptable exposure, while platform teams reduce repeated work

and make secure, compliant practices easier to follow. Leadership connects these contributions by aligning funding, priorities, and decision rights.

The important point is that DevOps crosses organizational boundaries. It cannot be delegated to the toolchain group or assigned to a small central team while the rest of the operating model stays unchanged. The organization must manage the whole flow of value, including the friction between teams, the evidence needed for decisions, and the feedback that comes from production use.

Consider a payment change that touches application code, fraud rules, customer messaging, database schema, monitoring thresholds, and support procedures. The work may begin as a backlog item, but it becomes a system change long before it reaches production. The DOI® Operating Model makes those connections visible early enough for the organization to manage risk without burying the team in meetings.

The same point applies to internal platforms. A platform team may provide build templates, deployment workflows, secrets management, environment provisioning, observability, and policy checks. Those capabilities create DevOps value only when product teams can use them to improve flow and control. A platform that exists in isolation becomes another tool estate.

A good DevOps system therefore balances speed with confidence. It gives teams enough autonomy to make decisions close to the work while maintaining alignment through standards, platforms, measurement, and governance. This balance is hard to achieve with structure alone. It requires leaders to pay attention to incentives, constraints, funding, and the lived experience of teams doing the work.

The organization-wide view also distributes explicit, complementary accountabilities across roles. Developers are accountable for the quality and operability of the systems they change. Operations and SRE teams are accountable for reliability practices and production learning. Security and risk teams are accountable for control expectations. Each task or decision must still have one

named accountable owner, although responsibility for performing the work may be distributed or delegated. Therefore, the practical boundary of DevOps is the value stream, not the team chart, meaning that demand, design, build, test, security, release, operations, measurement, and learning must be managed as one connected system.

AI-assisted delivery reinforces the need for an organization-wide system view because increased output only creates value when validation, security, compliance, observability, recovery, and support capacity keep pace. A local team may generate more code, more tests, and more configuration changes in less time. The organization still needs to know whether those changes are valuable, secure, compliant, observable, recoverable, and supportable.

2.3 DevOps across ITIL, PRINCE2 Project Management, PRINCE2 Agile, and SRE

DevOps operates within a broader ecosystem of complementary disciplines. ITIL, PRINCE2 Project Management, PRINCE2 Agile, and Site Reliability Engineering (SRE) each contribute practices that support the creation, delivery, operation, and improvement of digital products and services. Their areas of emphasis overlap, and organizations may apply them in different combinations according to their objectives, risks, regulatory obligations, technology environments, and available resources. These disciplines should not be treated as competing or mutually exclusive approaches (Beck et al., 2001; Beyer et al., 2016; PeopleCert, 2025c; PeopleCert, 2026d).

ITIL provides a best-practice framework for managing digital products and services throughout their lifecycle. PRINCE2 Project Management provides structured project governance, continued business justification, risk management, and control. PRINCE2 Agile helps organizations discover customer needs, experiment, and adapt product direction through iterative delivery and rapid feedback. SRE applies software-engineering principles to the design and operation of reliable services through observability, SLIs, SLOs, error budgets, automation, and continuous operational learning. These disciplines share important principles. They focus on value, progressing iteratively with feedback, promoting collaboration, and continuously improving outcomes. Each addresses a different dimension of organizational capability, and organizations achieve the greatest benefit when they apply these disciplines together rather than treating them as competing approaches.

DevOps complements these disciplines by providing the operating model, engineering and automation practices, and cultural principles required to continuously develop, deliver, operate, and improve digital products and services. It connects product discovery, software engineering, security, operations, governance, and customer feedback through one integrated delivery model. Rather than replacing ITIL, PRINCE2, PRINCE2 Agile, or SRE, DevOps enables their practices to work together as part of an integrated flow of value (Humble & Farley, 2010; Beyer et al., 2016; PeopleCert, 2025c; PeopleCert, 2026d).

For example, a team developing a customer-facing digital service may use PRINCE2 Agile practices to refine customer requirements and validate assumptions through rapid feedback. PRINCE2 Project Management may provide project-level business justification, investment oversight, risk management, and control, while wider programme or portfolio governance coordinates work above the project level.

ITIL provides end-to-end digital product and service management practices that help the organization plan, design, transition, operate, support, measure, and continually improve the products and services.

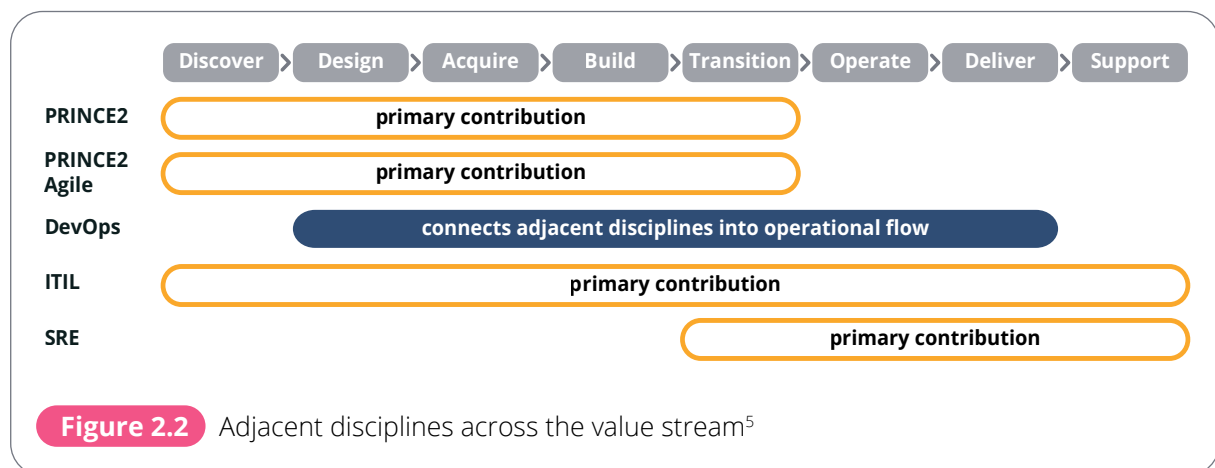
SRE contributes SLIs, SLOs, error budgets, operational automation, and post-incident learning. DevOps integrates these activities into a continuous delivery system supported by automated pipelines, platform capabilities, governance controls, and continuous feedback.

Successful organizations do not attempt to determine which discipline owns the work. Instead, they build an operating model in which each discipline contributes where it provides the greatest value. The result is a coherent system that combines governance with agility, operational stability with delivery speed, and innovation with responsible risk management.

Each discipline also has natural limitations when used in isolation. PRINCE2 Agile planning cannot compensate for an unreliable delivery pipeline. Service management practices cannot create fast flow if governance depends on manual approvals and disconnected processes. Project governance cannot

demonstrate value without operational feedback from production. Reliability engineering cannot succeed if teams lack the authority or capability to remove the causes of recurring failures. DevOps provides the mechanisms that allow these complementary disciplines to reinforce one another throughout the value stream.

Figure 2.2 places PRINCE2 Project Management, PRINCE2 Agile, DevOps, ITIL, and SRE across the value stream activities of Discover, Design, Acquire, Build, Transition, Operate, Deliver, and Support. The arrows indicate the conventional left-to-right direction of value flow. Although all five disciplines contribute to every stage to some extent, the figure highlights where each makes its primary contribution. DevOps connects these adjacent disciplines into a continuous operational flow spanning the entire value stream.



The growing adoption of AI makes DevOps' role as an integrating discipline even more important. As AI-generated outputs, predictions, recommendations, and autonomous actions influence development, testing, operations, security, release decisions, and other workflows across the delivery system, organizations need an

integrated DOI® Operating Model to keep that work validated, traceable, governed, monitored, and continuously improved. AI governance complements the adjacent disciplines by providing explicit controls for data, models, tools, provenance, evidence, and human accountability.

**AI-RELATED
CONTENT**



2.3.1 How adjacent disciplines contribute across the value stream

Figure 2.2 should be read as a decision map rather than an ownership hierarchy. The purpose of integration is to bring the right form of guidance to the decision being made while maintaining one continuous flow from intent to operation and improvement.

- Use PRINCE2 Project Management for project business justification, project risk, project control, and project baseline change (PeopleCert, 2025).
- Use PRINCE2 Agile approaches to discover needs, test assumptions, adapt priorities, and develop incrementally.
- Use DevOps to integrate, validate, secure, deploy, release, operate, measure, and improve digital products and services.
- Use ITIL to manage digital products and services through their lifecycle, including change enablement, incident management, problem management, service configuration, continuity, and continual improvement.
- Use SRE to engineer reliability through SLIs, SLOs, error budgets, observability, automation, toil reduction, and incident learning.

2.3.2 Anchor worked example - Case C: one public-service change across adjacent disciplines

Context. CivicLink Public Services needed to change the eligibility journey for a citizen benefit before a statutory date. On the development side, the changes affected application rules and accessible user journeys, alongside identity checks and document handling. On the operational side, they affected supplier interfaces, support procedures, monitoring, and continuity

arrangements. A proposed AI capability would classify uploaded documents, but a person would remain accountable for the eligibility decision.

Why was one discipline not enough?

The organization initially framed the work as choice between a PRINCE2 Agile product approach and formal project and service governance. That framing created an avoidable debate. The most accurate question was related to which decisions had to be made, what evidence each decision required, and which discipline supplied the strongest guidance.

The investment was governed as a PRINCE2 Project Management project within a wider public service programme. The project business case defined the statutory outcome, cost and time tolerances, named accountable owners, principal risks, and the process for changing the project baseline. The product team used PRINCE2 Agile approaches to test citizen needs, simplify the language, evaluate accessibility, and divide the change into increments that could be validated with users before the statutory deadline.

DevOps connected these contributions into one delivery system. The product increment determined the configuration of software and infrastructure, AI model version, prompts, tests, service documentation, and CI/CD Pipeline. Application Release Orchestration combined technical evidence with risk and service-readiness evidence to determine whether the software was ready to deploy to production as a release to citizens. Feature flags and controlled increments allowed users to be shielded from unvalidated technical installations.

The SRE and service-management perspectives changed the release decision. A rehearsal showed that a document-classification failure would push applicants into a manual queue that had insufficient capacity. The team narrowed the first release and increased the human fallback. Furthermore, they strengthened the SLO for successful application completion, while adding an operational runbook. The statutory date was preserved, but a lower-value automation feature was deferred.

Outcome and trade-off. CivicLink released a smaller, better-evidenced capability rather than the complete original scope. The organization accepted more manual review in the first increment in exchange for lower citizen risk and stronger operational confidence. Subsequently, production telemetry and support feedback guided the next increment. The sponsor communicated the outcome in operational terms: to reduce citizen failure and waiting time while preserving accessibility, privacy, reliability, and explainable decisions.

Transferable lesson. Do not ask which discipline owns the change. Ask which decision must be made, what evidence supports it, and how the disciplines combine across one end-to-end delivery flow. Agile, ITIL, PRINCE2 Project Management, SRE, AI governance, and DevOps become most useful when each contributes where it is strongest. Table 2.1 demonstrates which discipline can be used in each case.

Table 2.1 Complementary contributions of management and delivery disciplines⁶

Decision or need	Primary contribution	Evidence used in the case
Investment, project control, and project baseline change	PRINCE2 Project Management	Approved business case, tolerances, named accountable owners, risk records, and baseline change decisions.
Citizen need, accessibility, and incremental scope	PRINCE2 Agile approaches	Tested hypotheses, accessible prototypes, user feedback, and a prioritized product backlog.
Integrated engineering and value delivery	DevOps	Versioned artifacts, pipeline evidence, controlled deployment and release records, platform guardrails, and production feedback.
Product and service lifecycle management	ITIL	Change enablement, service configuration, continuity, incident and problem records, support readiness, and continual improvement.
Reliability engineering	SRE	Citizen-journey SLI, SLO, error budget, load and failure tests, actionable alerts, and operational-readiness evidence.



2.4 Common DevOps misconceptions

DevOps misconceptions persist because many of them contain a small piece of truth. All factors matter: tools, teams, pipelines, culture, and governance. The problem starts when one part of the system is treated as the whole system. Organizations then invest heavily in visible activity while the underlying constraints remain untouched.

Misconception 1: DevOps means tools.

Tools can automate repeatable work, expose evidence, enforce policy, and reduce toil. They cannot decide ownership, align funding, remove organizational fear, or create a shared definition of value. Tooling is useful when it supports a designed operating model.

Misconception 2: DevOps is a team.

A central DevOps team can help build platforms, coach practices, and provide reusable patterns. It becomes a problem when every delivery team waits for that group to do integration, deployment, environment provisioning, or operational improvement. DevOps capability must live where value is delivered.

Misconception 3: DevOps means CI/CD.

CI/CD is essential, but it is only one part of the delivery system. A pipeline that builds and deploys quickly still depends on product clarity and architectural quality, along with test strategy, security control, observability, release readiness, and recovery design.

Misconception 4: DevOps removes governance.

Mature DevOps improves governance by making it more consistent and evidence based. Policies can be expressed as code; approvals can be risk-aware, and audit trails can be generated automatically. Good governance protects flow by reducing ambiguity and late surprises (NIST, 2022; PeopleCert, 2026a).

Misconception 5: AI makes DevOps optional.

AI may generate work faster, but that speed makes DevOps more necessary, not less, since generated outputs still require validation, traceability, governance, and accountable human review.

These misconceptions survive because they are convenient. Tool purchase is easier to approve than an operating model change. A renamed team is easier to announce than shared ownership. A dashboard is easier to show than a hard conversation about dependencies, reliability, technical debt, or incentives. Seasoned practitioners learn to look past the visible artifact and ask what has changed in the system of work.

The strongest signal of misconception occurs when a gap exists between activity and outcomes. If teams have tools but lead time remains high, the constraint is probably not tooling. Similarly, if pipelines run but releases remain painful, the issue may be testing, architecture, approvals, or environment consistency. Ultimately, when AI-generated code increases output while defects and review burden rise, the delivery system is being overloaded rather than improved.

A practical check is to ask what became easier for the team and what became safer for the user. If a new tool reduces toil, shortens feedback, improves audit evidence, or helps a team recover faster, it is contributing to capability. Also, if it adds another login, another dashboard, or another disconnected workflow, it may increase operational drag.

Misconceptions also persist because they give leaders a simple story. It is easier to say that the organization adopted DevOps than to explain that funding, operating models, platform design, security workflows, and measurement still need work. The discipline of DevOps begins when the simple story is replaced with evidence about the system.

Anti-pattern box: “We renamed our ops team DevOps”

A cosmetic rename creates the appearance of change while preserving the old handoffs. Development still throws work over the wall. Operations still absorb production pain. Security still arrives late. Product leaders still measure feature output while ignoring reliability, rework, and customer impact.

The better pattern is shared capability with clear ownership. Operations knowledge should move into design, deployment, observability, incident learning, and platform practices. The operations function remains important, but its value increases when it helps the whole delivery system become more reliable and easier to operate.

**KEY
MESSAGE**



2.5 DevOps maturity: outcomes, not activities

DevOps maturity should be judged by outcomes. The question is not whether teams have stand-ups, pipelines, cloud environments, dashboards, or automation scripts. The question is whether the organization can deliver valuable change with less delay and rework, fewer uncontrolled risks, faster recovery, stronger security, and better learning (Forsgren et al., 2018; DORA, 2024).

Activity measures are useful during implementation because they show whether practices are being adopted. They should not become the final proof of success. A team may complete training and still struggle to deploy. A platform may exist and still lack adoption. A security scanner may run and still produce findings that nobody can prioritize. Activity is evidence of effort, not evidence of capability.

Capability appears in the behaviour of the system. Work moves in smaller batches. Integration happens frequently. Tests provide meaningful feedback. Security controls run early and consistently (Humble & Farley, 2010; NIST, 2022; DORA, 2024).

Deployments become routine. Incidents create learning rather than blame. Product decisions use evidence from users and production. AI-generated outputs follow the same governed flow as human-created artifacts.

The outcome should include speed, quality, reliability, security, value, and learning. Speed without quality creates rework. Quality without flow can create unnecessary control and delay, and reliability without change can become resistance to improvement. Security without usability becomes a bottleneck, while value without learning becomes accidental. A balanced DevOps system keeps these outcomes visible together.

In AI-native environments, maturity is not shown by the amount of AI-generated work produced, but by the organization’s ability to validate, govern, monitor, and learn from that work. AI-assisted outputs should therefore be assessed through the same outcome lens as any other change: whether they improve flow, quality, reliability, security, value, and learning without creating hidden risk (Autio et al., 2024; NIST, 2024).

Maturity should also be observable. A team should be able to show how a change behaved after release, how quickly problems were detected, what customers experienced, and what the next improvement will be. This is where DevOps becomes a learning system. Production is not the end of the work; it is

where the organization receives the most honest feedback about the work.

For senior practitioners, this perspective changes priorities. It encourages them to look beyond pipeline mechanics and examine the whole path of change. They should ask where work waits, why evidence is weak, where handoffs create defects, and where security slows delivery. Another important question is where AI-assisted work needs stronger review.

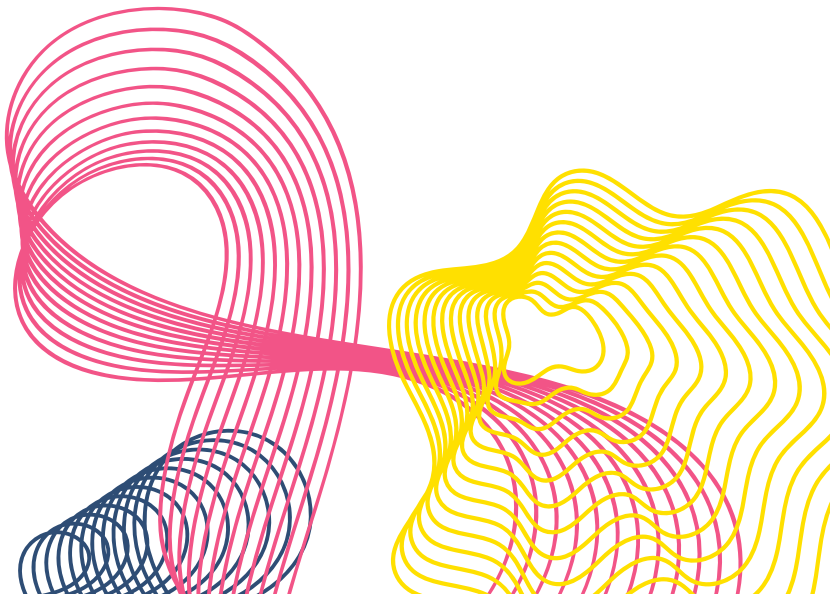
For leaders, the perspective changes what should be funded. It makes platform work, observability, test reliability, security automation, incident learning, and technical debt reduction visible as business investments.

These investments may be less dramatic than launching a new tool, but they are often where DevOps capability improves.

Leaders should be careful not to confuse progress with maturity. A platform launch, a team reorganization, or an improved maturity score may indicate movement, but they do not demonstrate success. True maturity is evident when the organization consistently delivers value, recovers rapidly from failure, learns from evidence, and adapts to change while maintaining control. Table 2.2 compares activity indicators with outcome evidence that suggests DevOps maturity.

Table 2.2 DevOps maturity activity indicators and outcome evidence

Activity indicator	Outcome evidence	Why the outcome is stronger evidence
Tools installed	Lower toil, shorter lead time, or fewer handoff delays	Installation shows capability; the outcome shows whether delivery is improved.
Automated pipelines implemented	Faster feedback, fewer defects, and more reliable releases	A pipeline creates value only when it improves delivery outcomes.
Ownership and communication channels defined	Faster decisions, fewer coordination delays, and clearer accountability	Organizational structures matter only when they improve how work flows and decisions are made.
Dashboards created	Faster detection of issues and evidence-based decisions	Dashboards add value only when their information changes actions and outcomes.
Training completed or processes adopted	Safer delivery, measurable value, and continuous learning	Adoption is activity; repeatable outcomes demonstrate capability.



DECISION AND ACTION GUIDE

Executive decision: Business and technology leadership adopt the DEVOPS INSTITUTE® definition as the organizational reference, clarify where adjacent disciplines contribute, and assign an accountable owner for unresolved decision-right conflicts.

Role-specific actions: Product, engineering, operations and SRE, security, service management, and project or programme roles map their contributions to one value stream and remove competing ownership claims.

Evidence to review: An approved vocabulary, value-stream ownership map, decision-right matrix, shared outcome measures, and a short log of unresolved overlaps.

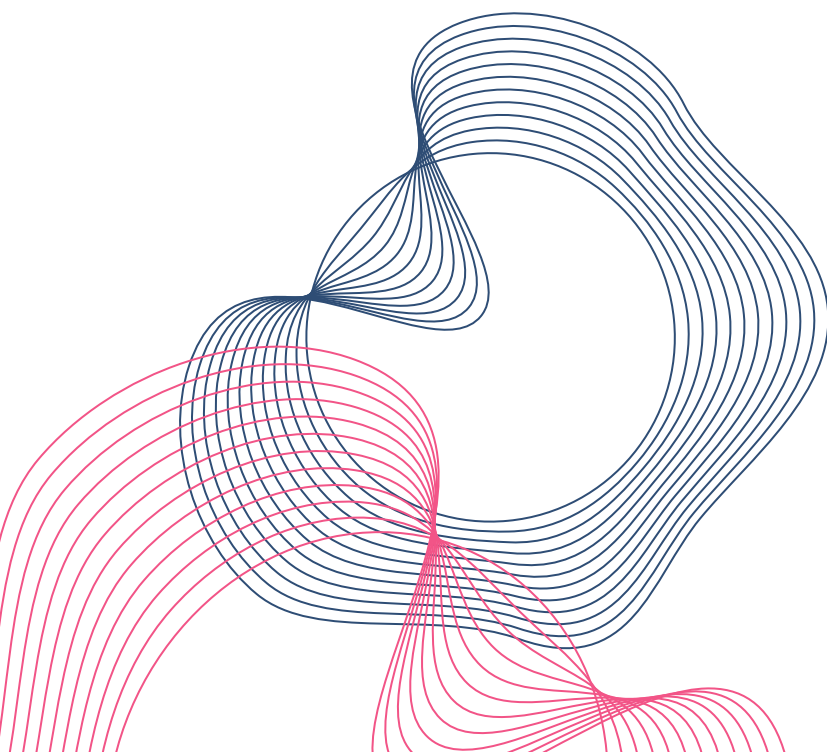
2.6 Chapter summary

DEVOPS INSTITUTE® defines DevOps as an organization-wide system for value delivery, not as a tool purchase or team label. DevOps connects product, engineering, operations, security, governance, measurement, and learning into one delivery system.

ITIL, PRINCE2, PRINCE2 Agile, and SRE contribute useful disciplines across the value stream. DevOps connects those disciplines to daily flow. Common misconceptions persist when visible activity is mistaken for capability. The test is to see whether outcomes improve.

The working habit to carry forward is clear-cut. When a new DevOps initiative is proposed, ask which part of the value delivery system it improves and what evidence will prove the improvement. This prevents the organization from mistaking installation, renaming, or local automation for real capability.

DevOps becomes useful when it changes how the organization makes decisions. It should help teams decide what to build, how to validate it, how to deploy and release it, how to operate it, and how to learn from it. That is the operating definition that the DevOps Standard uses in the chapters that follow.





03.

This chapter establishes AI as a structural participant in modern DevOps. AI does not replace DevOps; it raises the standard for DevOps. It explains how AI changes the rate, volume, and risk profile of delivery work, while keeping accountability within a governed operating model.

Chapter 1 made the case for a vendor-neutral standard. Chapter 2 established the organization-wide definition of DevOps. This chapter applies that definition to the reality of AI-assisted and AI-enabled work. The central point is practical: when AI increases the speed of creation, the delivery system must become stronger at validation, governance, feedback, and learning.

AI-native DevOps is DevOps in which AI participates in the delivery system by creating or modifying artifacts, supporting decisions, monitoring behaviour, or taking bounded actions. Validation, governance, observability, security, and human accountability remain explicit. AI may generate code, propose tests, summarize incidents, classify risk, recommend rollout patterns, or operate within tightly defined workflows. These capabilities are useful when they enter the same disciplined system as every other change, but become dangerous when they bypass review, evidence, security, reliability, or accountability.

The useful working habit is simple. By treating AI output as work entering the value stream, organizations can determine what evidence is required before that work can be trusted. Good engineering organizations already understand this discipline. AI makes it more visible, more urgent, and less forgiving.

Leaders should read this chapter as a governance and operating model chapter,

not as a tool adoption chapter. Practitioners should read it as guidance for keeping AI-assisted work within disciplined engineering practices. Assessors and educators should use it to clarify what must be validated, evidenced, and governed when AI becomes part of the delivery system.

3.1 How AI changes the DevOps system while the fundamentals still apply

AI changes DevOps first by changing load. More artifacts can be produced in less time. Not only code suggestions, test cases, and configuration fragments, but also deployment scripts, architecture notes, runbooks, incident summaries, and policy drafts can now appear faster than many teams can review them properly. The bottleneck moves from creation to confidence (DORA, 2025). This is familiar to anyone who has watched an organization automate without improving the system around the automation. Work arrives faster, but the queues move elsewhere, and pull requests wait for review. As test environments become unstable and security findings pile up, release decisions become harder because the available evidence is incomplete or inconsistent.

AI also changes the nature of some work. A human-authored change usually has a visible

AI-native DevOps

chain of intent, conversation, and context. AI-generated work may arrive with less explicit reasoning. It can look complete while carrying hidden assumptions, weak tests, licensing concerns, insecure patterns, or architectural shortcuts. The delivery system must expose those risks before production exposes them to the organization.

The fundamentals therefore become more important. Continuous Integration is essential to ensure that AI-generated code is integrated frequently and safely, while Continuous Testing must actively identify shallow or redundant generated tests. Continuous Security is also vital because AI may reproduce vulnerable patterns or mishandle sensitive data, and Observability still matters given that AI-enabled behaviour can degrade in production without a conventional code defect.

The same is true for governance. AI-assisted work needs policy boundaries, approved use cases, data-handling rules, evidence capture, and escalation paths. Governance should live inside the workflow where AI output is created, reviewed, validated, and released. A late committee review cannot keep up with a delivery system where generation happens continuously.

A useful operational example is a developer using AI to generate a database migration and related application code. The code may be compiled, and the migration may run in a test environment. However, this does not prove that rollback is safe, or that the migration can handle production data volume. It does not even prove that monitoring will detect partial failure, or that the change satisfies compliance requirements. The value lies in

the speed of the draft. Confidence comes from the DevOps system around it.

AI-native DevOps is therefore not a separate lane beside DevOps. It is the condition under which modern DevOps increasingly operates. The delivery system must remain coherent, regardless of how work originates. Some work is human-authored, some is AI-assisted, and some is executed by automated agents under policy control.

The lesson is a little uncomfortable and useful. AI can make a mature DevOps system better. It can also make an immature system louder, faster, and harder to control. The difference is usually found in the quality of validation, feedback, and ownership.

3.1.1 Three roles of AI in the delivery system

AI enters the delivery system in more than one form. Treating all AI as one capability leads to weak controls. A generated code suggestion, a predictive failure signal, and an automated remediation action carry different risks. The operating model should recognize these differences and apply controls that match the risks associated with each role (NIST, 2023a; Autio et al., 2024).

Machine learning detects patterns and supports decisions. It may classify deployment risk, identify anomalous telemetry, predict capacity saturation, detect suspicious behaviour, prioritize alerts, or recommend which tests to run. The main risk is confidence without enough context. The control pattern is calibration, monitoring, explainability where appropriate, and periodic review against actual outcomes.

Generative AI creates or modifies artifacts. These artifacts may include source code, unit tests, pipeline configuration, infrastructure definitions, documentation, user stories, security findings, runbooks, or incident summaries. The main risk is that the artifact appears plausible before it has earned trust. The control pattern is provenance, review, automated validation, security scanning, and evidence.

Agentic AI takes goal-directed action within defined boundaries. An agent may open a pull request, triage an incident, route a ticket, update a runbook, request additional tests, or initiate a rollback when a known safety condition is met. The main risk is action without adequate constraint. The control pattern is narrow scope, policy limits, reversible actions, audit trails, and clear human accountability.

Figure 3.1 outlines these three roles and shows how operational risk increases as AI moves from detecting patterns and generating artifacts to taking action within set guardrails.

These roles often work together. An incident workflow may use machine learning to detect an anomaly, generative AI to summarize likely

causes, and an agent to gather logs or open a change request. This can be useful, but it also means that the organization needs traceability across the entire workflow. Teams should be able to tell what observed the condition, what generated the recommendation, what action was taken, and who remained accountable.

Risk rises as AI moves closer to production action. A comment generated in a code review has low operational exposure. An AI-created infrastructure change has more exposure. An automated remediation action against a production service has much more. The control model should become stronger as the possible blast radius increases.

Organizations should authorize specific actions rather than approve a general label such as 'autonomous agent'. The authorization should define the tools, data, services, and environments the agent may access; the preconditions and postconditions for action; the evidence to retain; the recovery mechanism; and the accountable owner. Table 3.1 provides guidance for human involvement in different classes of agentic actions, with stronger controls applied as impact and irreversibility increase.

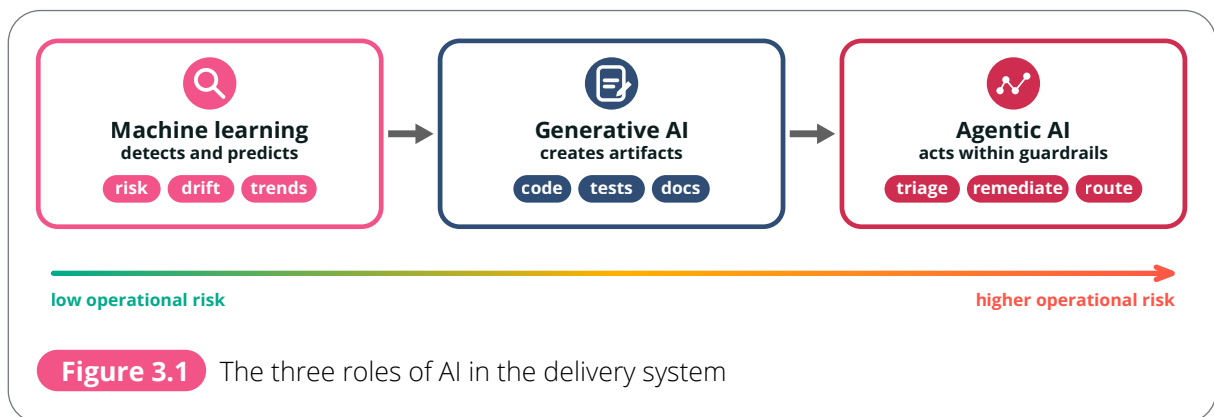


Table 3.1 Agent authorization practices⁷

Action class	Example	Human involvement	Minimum control
Advisory	Summarize telemetry, explain a failure, or propose a test.	A human interprets the output and makes the decision.	Read-only access, source links, provenance, uncertainty where relevant, and retained output.
Assisted	Draft a pull request, incident update, runbook change, or remediation plan.	A human reviews and authorizes the artifact before it changes the system.	Scoped access, validation, approval record, and no direct production action.
Bounded automated	Gather diagnostics, route work, restart an approved non-critical component, or open a rollback request.	A human pre-authorizes the action class and reviews exceptions and outcomes.	Pre-approved standard change pattern (validated prompt, model & parameters via CI/CD pipeline), least privilege, preconditions, post-action verification, rollback, monitoring, and audit trail.
High-impact or rights-affecting	Change entitlements, delete data, alter financial logic, release regulated decisions, or modify critical controls.	An accountable human gives explicit authorization; prohibits automation where consequences cannot be made acceptably reversible.	AI Asset Registration Audit (prompt, model, configuration as code), strong segregation, specialist review, complete evidence, tested recovery and formal risk acceptance where permitted.

As operational blast radius rises, authorization should become more explicit. Bounded automated actions should be least-privileged, observable, reversible, and tested with the same seriousness as other production automation. A kill switch is useful only when ownership, access, and recovery have been rehearsed.

3.1.2 Human-AI collaborative engineering: what changes, what stays the same

AI changes the shape of engineering work. Teams increasingly spend less time creating a first draft from nothing and more time defining intent, supplying context, reviewing options, validating outputs, and integrating changes into a working system. The craft moves closer to judgement, decomposition, and evidence-based acceptance.

A developer working with AI must describe the desired behaviour clearly enough for the tool to produce useful output. A tester must decide whether generated tests are meaningful rather than merely numerous. A site reliability engineer (SRE) must decide whether an AI incident summary matches the telemetry. A security engineer must decide whether the suggested remediation reduces risk. The work becomes more collaborative, but professional responsibility remains with the people and the organization.

The first practical change is context management. AI output depends heavily on the context provided. Poor prompts, stale documentation, incomplete architecture knowledge, weak examples, or missing operational constraints will produce weak output. Mature teams therefore treat context as an engineering asset. They maintain useful documentation, service catalogues, interface contracts, runbooks, and decision records because these resources improve both human and AI-assisted work (Autio et al., 2024; DORA, 2025).

The second change is review discipline. AI-generated artifacts can be persuasive. They often arrive in a polished format, and polished output tends to reduce skepticism. Teams need review practices that focus on behaviour, risk, maintainability, security, operability, and business intent. A code review that checks style while ignoring architecture and operational behaviour is inadequate in an AI-native system.

The third change is evidence capture. When AI contributes to work, teams should be able to trace what was generated, what was changed by humans, what checks were run, and what evidence supported release. This matters for learning, compliance, security, and incident investigation. It also helps teams improve their prompts, templates, validation rules, and use-case boundaries over time.

Some responsibilities remain unchanged even as AI reshapes engineering work. Humans still own architecture decisions and product trade-offs. They also own security expectations and compliance interpretation, and remain responsible for service reliability and customer impact. AI can assist with analysis and execution, but it cannot carry organizational accountability in any meaningful operational sense.

However, this is not an argument for slowing teams down. Good AI-native practice should remove repetitive work and improve feedback. It should also help teams find defects earlier, discover edge cases, compare options, and understand production behaviour. The discipline is therefore to make speed useful rather than decorative. For that reason, the human-AI collaboration loop should be explicit: a human sets intent, AI assists, and a human reviews and accepts responsibility.

A healthy human-AI engineering loop has five steps: human intent, AI assistance, human review, pipeline validation, and production learning. Pipeline validation and production telemetry provide evidence for improving the workflow, while organizational accountability remains with people and teams.

3.2 Governing AI-native delivery

AI-native delivery requires more than technical validation of individual outputs. It requires an operating model that defines permitted use, evidence, approval thresholds, monitoring, escalation, and human accountability across the value stream. The following sections explain how generated artifacts are validated and how AI governance is embedded in normal DevOps workflows rather than added as a late review.

3.2.1 Validating and governing AI-generated outputs

The core operating rule for AI-native DevOps is straightforward: AI-generated output enters the same governed delivery system as any other artifact. It should not receive special trust because it was generated quickly, and it should not bypass controls because the output appears complete. Production confidence still comes from evidence (Autio et al., 2024; NIST, 2024).

Validation begins with provenance. Teams should know whether an artifact was created by a person, assisted by AI, or produced by an automated workflow. The record does not need to become theatrical bureaucracy. It should be sufficient to answer operational questions later: what generated this, what sources or context were used, who reviewed it, and what validation evidence supported the decision?

Human review remains necessary because AI output can be correct in syntax and wrong in intent. Generated code can compile while violating an architectural boundary.



A generated test can pass while testing the wrong behaviour. A generated policy exception can sound reasonable while weakening regulatory control. Review should focus on the intended outcome, the risk of failure, and the evidence required for trust.

Automated validation becomes more important as artifact volume increases. Development controls include static analysis, unit tests, integration tests, and contract tests. Delivery controls include security scans, dependency checks, policy checks, performance tests, and deployment verification. Weak automation forces human reviewers to carry too much load. Overloaded reviewers eventually skim, defer, or accept work without enough confidence (NIST, 2022; NIST, 2024).

The amount of validation should be proportionate to risk. A generated internal note may need only human review, whereas a generated test may need review for relevance and maintainability. Likewise, a generated infrastructure change needs security, policy, and rollback evidence, while a generated production remediation workflow needs strong guardrails, rehearsal, auditability, and clear ownership.

Oversight also includes rejection. In practice, teams should be able to reject an AI output as unusable without treating that as a failure of adoption. Some outputs will be shallow, unsafe, irrelevant, or too expensive to correct.

In those cases, mature teams do not spend hours polishing a bad suggestion when a clean human-authored solution would be faster and safer.

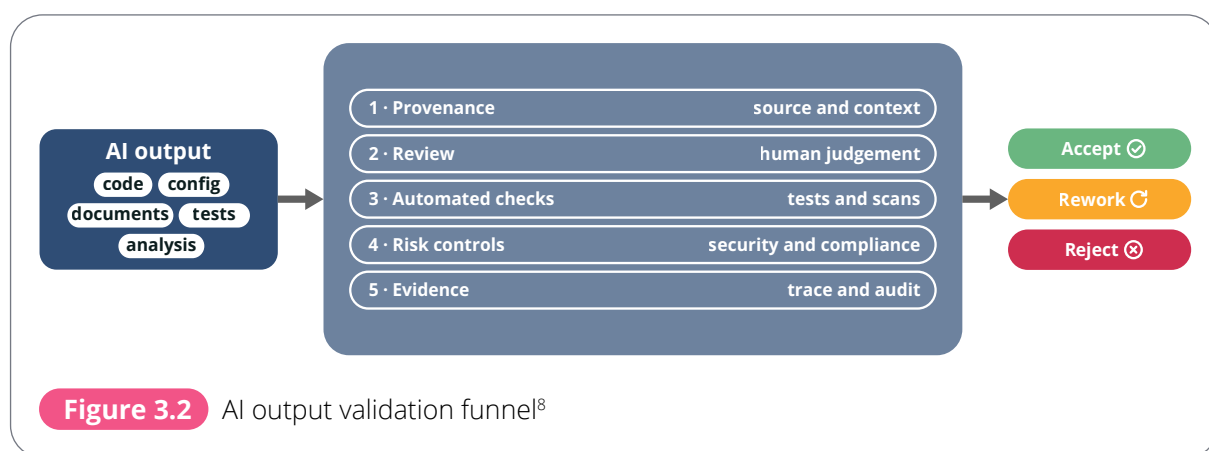
The most practical validation funnel has five layers:

1. Confirm provenance and context.
2. Perform human review.
3. Run automated checks.
4. Apply risk controls for security, compliance, reliability, and operational impact.
5. Capture evidence that supports the final decision.

The result may be acceptance, rework, or rejection.

This discipline is especially important for AI-generated tests. A generated test suite can increase coverage numbers without increasing real confidence. Teams should review whether tests assert meaningful behaviour, cover useful risks, and fail when the system is wrong. A green pipeline that proves little is an old software-engineering problem appearing in a new form.

Figure 3.2 shows AI output moving through provenance, review, automated checks, risk controls, and evidence before it is accepted, reworked, or rejected. It reinforces that generated code, configuration, or analysis is not production-ready until the evidence path has been completed.



Anti-pattern box: "Our AI writes the code, so we need fewer reviews"

This anti-pattern treats AI as a substitute for engineering judgement. It usually appears after a team sees fast productivity gains from code generation. The organization then assumes that review effort can be reduced because the tool appears to be competent.

The operational result is predictable. More code enters the system, while the review and validation capacity stays flat or shrinks. Defects move downstream.

Security findings appear late. Architecture drift increases. Teams spend more time explaining production behaviour that should have been challenged earlier.

The better pattern is risk-based review. Low-risk generated work can move quickly through automated checks and lightweight human review. Higher-risk changes need deeper review, stronger evidence, and clear approval thresholds. AI can reduce repetitive drafting, but it increases the need for disciplined acceptance.

KEY MESSAGE



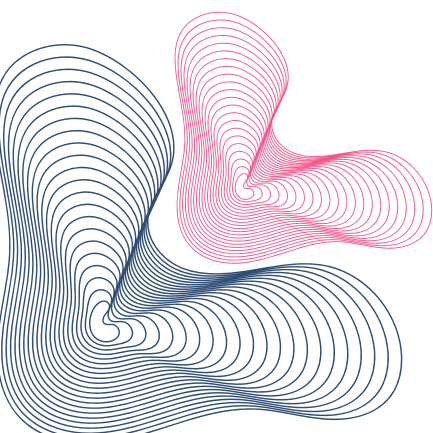
3.2.2 AI governance as a DevOps operating responsibility

AI governance is the system by which an organization directs, controls, and monitors AI use so that AI-enabled products, services, delivery work, and decisions create value while remaining ethical, secure, compliant, observable, and accountable. In DevOps, it is implemented through approved use cases, data rules, model and tool approval, policy, validation, evidence, monitoring, and human accountability. Because AI-enabled work occurs across the value stream, this governance belongs inside the DOI® Operating Model rather than in a disconnected late review (ISO/IEC, 2023; NIST, 2023a).

Effective AI governance begins by defining the conditions under which AI may be used. It starts with approved use cases. Teams need to know where AI may be used, what data may be shared, which tools or models are approved, and which outputs require review. These rules should be specific enough to guide daily work. A broad statement encouraging responsible AI will not help an engineer decide whether a production log sample can be included in a prompt.

Data handling is one of the most important control areas. Prompts, training data, retrieval sources, logs, traces, customer records, source code, and incident details may all contain sensitive information. Teams need clear rules for data classification, redaction, retention, access, and reuse. The safest pattern is to make correct behaviour easy through approved workflows and platform guardrails.

Governance should also define model and tool approval. The organization should know which AI services may be used for which purpose, how their outputs are validated, and what contractual, privacy, security, and operational constraints apply. This does not require every experiment to wait for a central board. It requires a clear path from experimentation o approved operational use.



Once governance rules are defined, they must be enforced consistently during delivery. This is where policy as code becomes useful. A pipeline can reject an unapproved dependency, require human review for high-risk AI-generated infrastructure changes, block prompts containing restricted data, or require evidence before deployment. These controls are more reliable when they are embedded in the workflow rather than remembered during a late review.

Enforcement alone is not enough. Organizations also need evidence that governance was applied. Evidence capture is the other side of governance. A regulated organization may need to show who approved a model-assisted decision, what tests were run, what data was used, and what monitoring was in place after release. A less regulated organization still benefits from the same evidence when an incident occurs. Evidence serves more than auditors. It also helps teams learn what happened.

Some decisions require additional human judgement because not all AI-assisted work carries the same operational risk. Human approval thresholds should be explicit. Some AI-assisted outputs can proceed after normal review and automated checks. Others require specialist review from security, architecture, compliance, legal, or operations.

The threshold should depend on user impact, data sensitivity, operational blast radius, regulatory exposure, and reversibility (NIST, 2023a; Autio et al., 2024).

Governance also needs feedback. If an AI-assisted deployment recommendation repeatedly overstates risk, the model, prompt, data, or workflow should be adjusted. If a generated remediation action works safely in one class of incident, teams may narrow the guardrails and automate more of the response. Governance should learn from evidence rather than freezing the first policy forever. Figure 3.3 summarizes how AI governance controls are embedded throughout the DevOps value stream rather than applied as a separate review activity.

Controls live where AI-enabled work happens. Guardrails make safe work easier and risky work visible. This is where the DOI® Operating Model earns its keep. It connects policy, workflow, platform, telemetry, and accountability. AI governance becomes practical when it is part of the same system that already manages change. The AI governance control plane places use cases, data rules, models and tools, policy, and evidence above the DevOps flow of plan, code, build, test, release, operate, and learn.

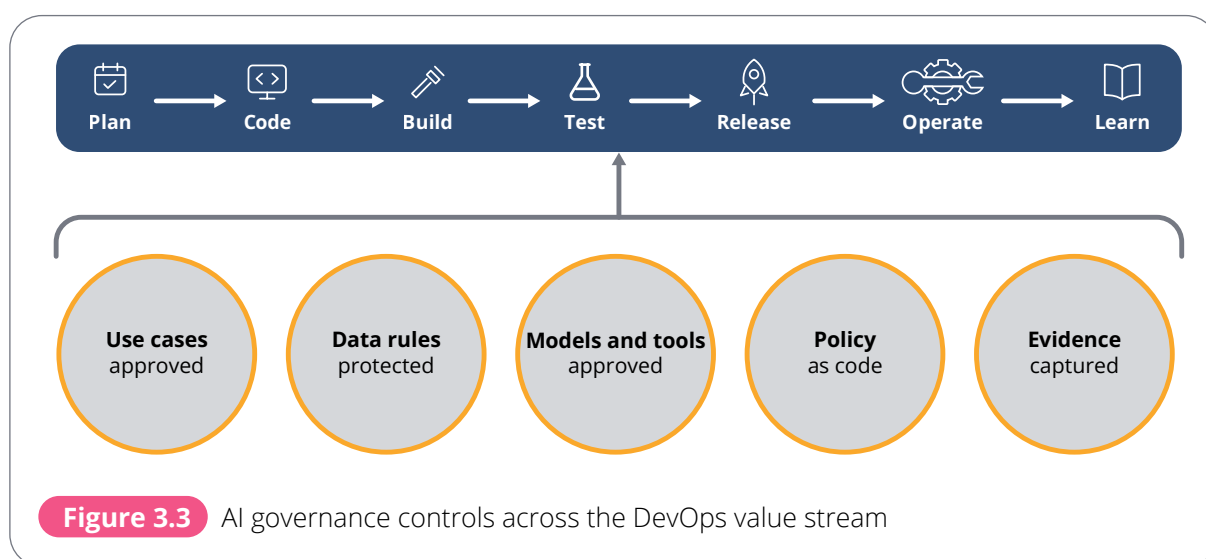


Figure 3.3 AI governance controls across the DevOps value stream

Case A – Northstar Financial Services (composite): AI-assisted incident triage under policy

Northstar introduced an AI assistant to reduce the time incident responders spent assembling change history, telemetry, affected services, and customer-impact context. The first pilot produced polished summaries, but it sometimes over-weighted noisy alerts and once missed a recent configuration deployment because deployment metadata was not included in its approved retrieval sources.

The AI governance response was to redesign the workflow, not to abandon the use case or grant the tool more authority. Approved data sources were defined, sensitive data was redacted, every statement had to link to source telemetry, and the summary displayed confidence and missing-context warnings.

The assistant could gather evidence and open an incident record, but it could not communicate externally, change severity, approve a deployment, or execute remediation.

During a later payment incident, the assistant correlated a latency increase with a recent configuration deployment and identified the affected customer path. The incident commander verified the evidence, selected rollback, and remained accountable for the decision. The audit trail preserved the sources, generated summary, human changes, decision, and outcome for the post-incident review.

Transferable lesson: AI becomes operationally useful when provenance, approved context, bounded permissions, human decision rights, and feedback are designed into the workflow. A capable assistant is not an accountable incident commander.

SCENARIO



3.3 AI implications for the DevOps Nine Pillars

AI-native DevOps governance is not a separate practice; it raises the standard for every DevOps practice. The DevOps Nine Pillars, which are classified and described in Chapter 4, provide a structure for understanding these practices, while AI changes the expectations within each practice area. The assessment question becomes more demanding: can each pillar of practice still operate safely when AI increases speed, volume, complexity, and uncertainty?

Leadership and culture must establish responsible AI use as an operating expectation. Leaders need to make the purpose clear, fund validation capability, and avoid rewarding raw output volume. Teams need psychological safety to challenge AI output, report weak results, and slow down when evidence is poor. A culture that punishes caution will eventually accept unsafe automation.

Design for DevOps should include AI validation, explainability, observability, data governance, and human oversight from the beginning. Teams need to decide how AI-enabled features will be tested, monitored, rolled back, and explained before the feature reaches production. Retrofitting these qualities later is expensive and often incomplete.

Continuous Integration must route both human-authored and AI-assisted artifacts through the same controlled path. The pipeline should capture provenance, run automated checks, and prevent shadow delivery. Continuous Testing must address deterministic behaviour, probabilistic behaviour, generated tests, evaluation datasets, drift, and human review where ground truth is delayed.

Elastic Infrastructure and Platform Engineering must support AI workloads without turning every product team into an AI infrastructure

specialist. Platforms may provide approved model access, prompt management, evaluation pipelines, GPU or accelerator capacity, telemetry patterns, and cost controls. These capabilities should reduce cognitive load while preserving team ownership of outcomes.

Continuous Security must address AI-specific risks. Prompt injection, data leakage, model tampering, unsafe tool use, malicious dependencies, exposed secrets, and vulnerable generated code all belong inside DevSecOps practice. Security controls should be embedded in design, pipelines, platforms, and production monitoring.

Continuous Delivery and Deployment must account for AI-assisted release decisions and AI-enabled system behaviour. Deployment risk analysis may be assisted by AI, but approval thresholds, rollback paths, progressive delivery, and production validation remain essential. A fast rollout without trustworthy telemetry is still a gamble.

Continuous Monitoring and Observability must expand beyond uptime and latency. AI-enabled systems also need monitoring for drift, degradation, quality, safety signals, bias indicators, token or compute cost, retrieval quality, and human escalation. Traditional service telemetry remains necessary, but it is no longer sufficient for systems whose behaviour depends on data, prompts, models, and context.

Measurement, value stream, and governance practices must also mature. AI may improve lead time, test generation, incident response, or knowledge work, but those gains should be evaluated against quality, stability, security, cost, and business value. A metric that counts generated output without measuring useful outcomes will mislead the organization.

The practical implication is simple: every DevOps practice area should be reviewed through an AI-native lens. What changes when some work is generated, some decisions are assisted, some workflows are agentic, and some production behaviour is probabilistic? That question prepares the reader for the operating model introduced in the next chapter.

DECISION AND ACTION GUIDE

Executive decision: The accountable AI sponsor approves use-case classes, data boundaries, permitted models and tools, human-approval thresholds, and the production actions that agents may take.

Role-specific actions: Engineering, platform, data, security, risk, legal, and operations roles implement provenance, validation, bounded permissions, monitoring, recovery, and escalation controls.

Evidence to review: An approved-use register, AI control matrix, provenance records, agent permission manifest, action log, approval and exception trails, recovery test, kill-switch test, and production-quality signals.

3.4 Chapter summary

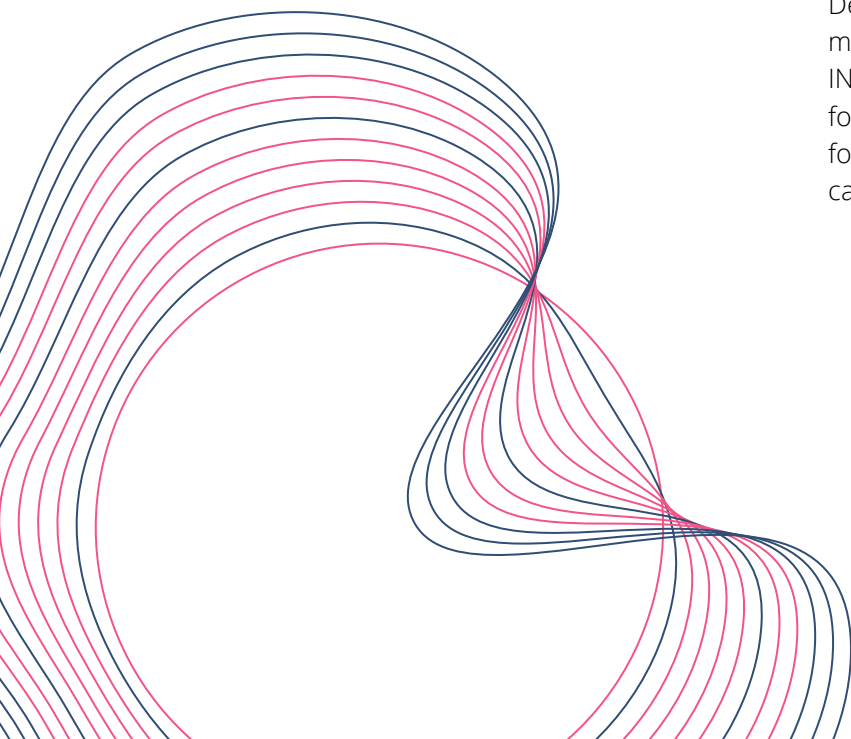
AI-native DevOps is the operating reality for modern delivery. AI can assist in creation, detection, prediction, orchestration, and operation. The DevOps system must still provide validation, governance, feedback, security, reliability, and accountability.

The central lesson of this chapter is operational rather than philosophical. AI changes the load on the delivery system before it changes the responsibility for outcomes. Every AI-assisted artifact and every AI-supported decision must be placed inside a system that can prove, monitor, and improve it.

AI raises the standard for DevOps because it increases the speed, volume, and ambiguity of delivery work. Machine learning, generative AI, and agentic AI play different roles; agentic authority should be granted by action class, with explicit permissions, evidence, recovery, and human accountability.

Human teams remain accountable for architecture, security, quality, operability, compliance, value, and production outcomes. AI-generated artifacts should enter the same governed pipeline as human-authored artifacts, with validation proportionate to risk.

AI governance becomes effective when it is embedded in workflow, platforms, policy, evidence, and feedback loops. AI-native DevOps is not a separate pillar. It raises the maturity expectation for every DEVOPS INSTITUTE® pillar. Chapter 16 extends this foundation with a technology-neutral method for evaluating and adopting emerging capabilities and high-stakes applications.





The DEVOPS INSTITUTE® Operating Model

PART 2

Chapter 4

The DEVOPS INSTITUTE®
Operating Model

Chapter 5

Leadership and
Collaborative Culture

Chapter 6

Design for DevOps

Chapter 7

Continuous Integration and
Continuous Testing

Chapter 8

Continuous Delivery and
Deployment

Chapter 9

Elastic Infrastructure and
Platform Engineering

Chapter 10

Continuous Security

Chapter 11

Continuous Monitoring and
Observability



This section introduces the DEVOPS INSTITUTE® Operating Model as an integrated capability system. The purpose is to give readers a working map of modern DevOps. The model does not treat DevOps as a list of disconnected practices. It connects the Nine Pillars with governance, measurement, learning, and value.

The central structure of **PART TWO** is the relationship between the DevOps Nine Pillars and the DevOps Blueprint. The Nine Pillars describe the practice domains that must mature. The Blueprint complements this capability view by showing how DevOps is implemented as a four-layer delivery architecture.

Chapter 4

Introduces the DOI® Operating Model itself. It explains how the Nine Pillars and the four-layer DevOps Blueprint work together, and why mature DevOps depends on both capability depth and system integration. This chapter is the map for the rest of PART TWO.

Chapter 5

Examines two Pillars: Leadership and Collaborative Culture. It treats leadership as the condition-setting function for DevOps, and culture as the daily pattern of behaviour that determines whether teams can raise risks, learn from incidents, share ownership, and use AI responsibly. It also introduces human-AI trust as a practical management concern for modern engineering teams.

Chapter 6

Focuses on another Pillar, Design for DevOps. It explains that architecture is one of the strongest inputs into delivery performance. Systems must be designed for observability, testability, deployability, security, operability, and AI validation before teams can expect reliable flow through the pipeline.

Chapter 7

Together, Chapters 7 and 8 define the delivery engine of the DOI® Operating Model. Chapter 7 covers the Pillars of Continuous Integration and Continuous Testing as disciplines of frequent integration, fast feedback, and meaningful validation.

Chapter 8

Presents the Pillar of Continuous Delivery and Deployment, showing how organizations keep software releasable while applying appropriate release decisions, deployment patterns, governance, and recovery mechanisms.

Chapter 9

Addresses the Pillars of Elastic Infrastructure and Platform Engineering. It positions infrastructure as a DevOps capability rather than a hosting concern, and Platform Engineering as the discipline that turns repeated engineering needs into governed, self-service capabilities. In AI-native environments, this includes workload elasticity, cost visibility, approved AI platform services, and reusable guardrails.

Chapter 10

Covers the Continuous Security Pillar. It explains how security becomes part of design, pipelines, platforms, release governance, operations, and feedback. It also expands the security model to include AI-specific risks such as prompt injection, model integrity, data exposure, generated code, agents, and AI supply chain concerns.

Chapter 11

Closes PART TWO with the Pillar of Continuous Monitoring and Observability. It treats production telemetry as the nervous system of DevOps. The chapter shows how logs, metrics, traces, service level objectives (SLOs), alerts, DevOps Research and Assessment (DORA) metrics, flow metrics, and AI behaviour signals turn production experience into evidence for better engineering, product, reliability, security, and architecture decisions.

04.

This chapter introduces the DEVOPS INSTITUTE® Operating Model as a complete, integrated system. The chapter explains how the DevOps Nine Pillars and the four-layer DevOps Blueprint work together, so readers can use the model as a map for assessment, governance, improvement, and AI-native delivery.

The DOI® Operating Model turns DevOps from a broad idea into an assessable, governable, and improvable system for value delivery.

The first three chapters established the need for a shared reference, clarified what DevOps means, and explained why modern DevOps must be AI-native. Chapter 4 provides the map that is used in the DevOps Standard book. It does not attempt to explain every practice in depth. It shows where each practice belongs and how the parts of the system affect one another.

A useful operating model shows not only what capabilities are needed, but also how they interact to support flow, control, evidence, and continuous learning. When organizations miss this systems view, they often improve isolated practices while the end-to-end value stream remains slow, fragile, or difficult to govern.

The chapter is most relevant to leaders, architects, transformation leads, platform leaders, assessors, and senior practitioners who need a shared view of how DevOps capabilities fit together. It gives leaders the structure for investment and governance. It gives practitioners a way to see how their local work connects to the whole delivery system.

4.1 The DOI® Operating Model as an integrated capability system

The DOI® Operating Model is presented in this book as an integrated capability system. It connects the DevOps Nine Pillars, which are practice domains: Leadership, Collaborative Culture, Design for DevOps, Continuous Integration, Continuous Testing, Elastic Infrastructure, Continuous Security, Continuous Delivery and Deployment, and Continuous Monitoring and Observability. People, process, and technology are the three dimensions through which each practice domain is realized. The DevOps Blueprint is the four-layer delivery architecture, comprising Elastic Infrastructure, the continuous integration and continuous delivery/deployment (CI/CD) pipeline, Application Release Orchestration, and Value Stream Management. Together, these layers support governed delivery, organizational learning, and delivered value. Each pillar contributes to the same objective: the reliable delivery of useful change through a controlled, observable, and continuously improving system (Forsgren et al., 2018; Hornbeek, 2019; Kim et al., 2021).

The DEVOPS INSTITUTE® Operating Model

This matters because many organizations still view DevOps as a collection of separate initiatives. One group improves pipelines, whereas another buys monitoring tools. A security team adds scanning, whereas a platform team builds templates. These activities can help, but they create limited value when the larger system is not designed as a coherent whole.

The operating model defined in this Official Book gives those practice domains a common structure. It allows an organization to ask practical questions: What capability are we improving? Where does it operate in the value stream? What evidence shows that it works? Which outcome should improve as it matures? These questions keep DevOps grounded in operations rather than slogans.

An organization with mature DevOps capabilities can replace or upgrade tools while preserving its established ways of working, controls, responsibilities, and decision logic.

Less mature organizations may mistake tool adoption for the development of underlying capability. The model provides leaders and practitioners with a stable framework for using appropriate technologies without allowing the technology stack to define the discipline.

The model can be used to:

- connect local practices to organizational outcomes
- assess capabilities through evidence and operational facts
- treat AI-generated work as part of the same governed delivery system
- measure improvement by flow, quality, reliability, security, and value.

The DOI® Operating Model describes how the organization organizes and enables DevOps work rather than prescribing a specific organizational structure. Figure 4.1 shows the practice domains, delivery architecture, governance, measurement, and feedback mechanisms as one connected operating system.



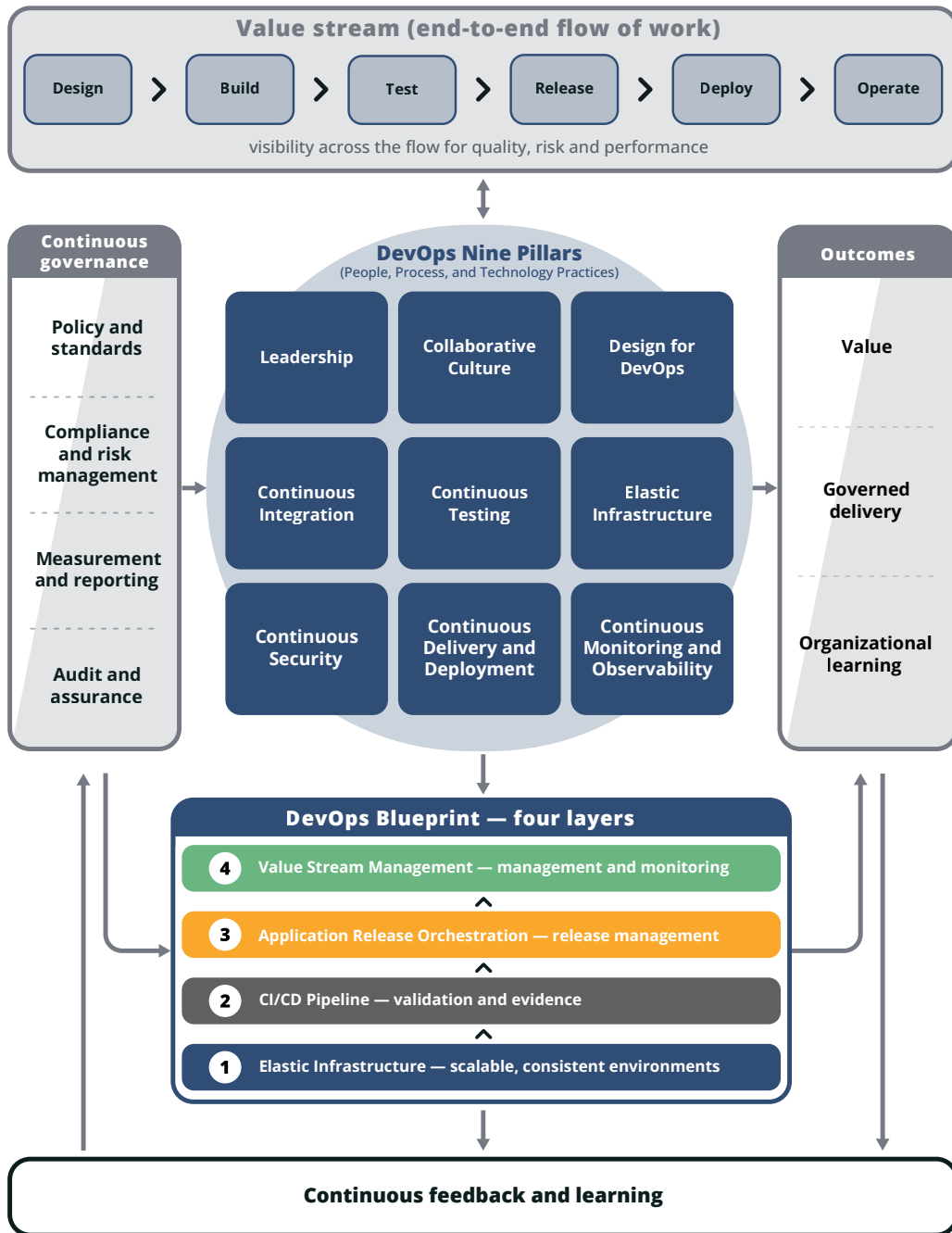
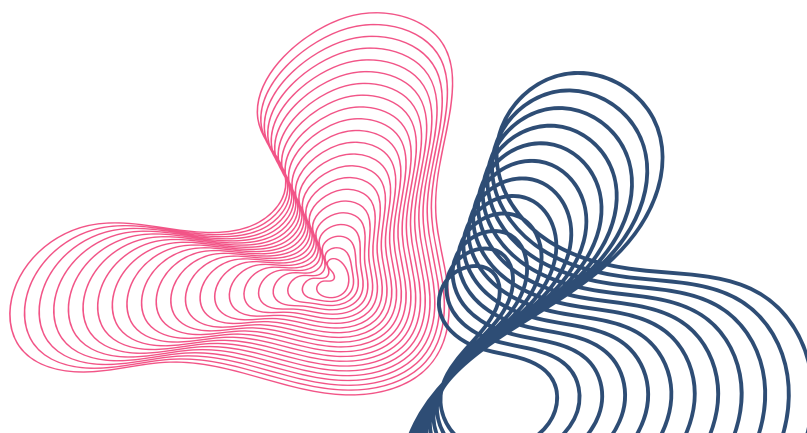


Figure 4.1 The DEVOPS INSTITUTE® Operating Model



4.2 The DevOps Nine Pillars as interdependent practice domains

The Nine Pillars define interdependent practice domains (see Figure 4.2), each inclusive of the three dimensions: people, process, and technology, which are required for mature DevOps. Each pillar must mature sufficiently to support safe flow, and the pillars must be interpreted as one system (Forsgren et al., 2018; Hornbeek, 2019; DORA, 2024).

The arrangement of the Nine Pillars does not represent a prescribed implementation or maturity sequence. The pillars are interdependent practice domains and should be strengthened according to the constraints and needs of the delivery system.

Weakness in one pillar creates friction in the whole model. For example, weak Continuous Testing reduces deployment confidence, while

weak Continuous Monitoring and Observability limits recovery and learning. Weak leadership can also leave teams constrained by outdated funding models, approval processes, and incentive structures that hinder collaboration, flow, and continual improvement. Similarly, weak Design for DevOps produces systems that resist automation, observation, validation, and independent change.

The DevOps Nine Pillars also prevent a narrow reading of DevOps. Fast pipelines alone do not indicate mature DevOps if reliability and security remain weak. Similarly, an organization with a strong culture but manual environments will still struggle to deliver reliably. On the other hand, a platform team can provide useful services, but product teams still need to address ownership, feedback, and responsibility for outcomes.

These interdependencies should also guide maturity assessment. DevOps maturity is assessed by reviewing the capabilities realized through an organization's implementation of the DOI® Operating Model. A maturity assessment that scores one pillar at a time is useful only when the results are interpreted as a system. The most urgent improvement may address the lowest score, but it may also address the pillar that constrains several others.

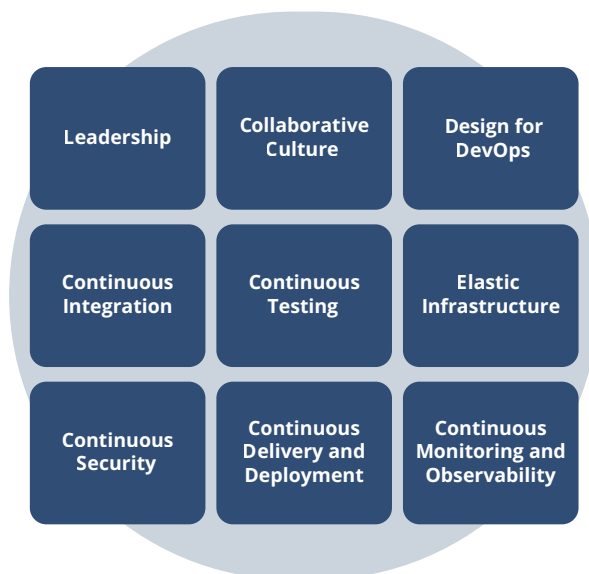


Figure 4.2 The DevOps Nine Pillars⁹

AI-native delivery does not create a separate operating model or an additional Pillar. It raises the capability expected across the existing DOI® Operating Model. Leadership sets acceptable use and accountability; Collaborative Culture supports critical review; Design for DevOps includes AI validation; Continuous Integration and Continuous Testing govern and validate AI-generated artifacts; Elastic Infrastructure provides scalable, governed AI environments; Continuous Security protects the AI supply chain and new attack paths; Continuous Delivery and Deployment preserves evidence, controlled exposure, and recovery; and Continuous Monitoring and Observability monitors AI behaviour, data, and drift. Chapter 3 provides the canonical AI governance model, while Chapters 5–11 develop these practice-specific implications.

**AI-RELATED
CONTENT**



4.3 The DevOps Blueprint as a four-layer delivery architecture

The DevOps Blueprint is the implementation architecture of the DOI® Operating Model. While the Nine Pillars define the practice domains required for mature DevOps, the

AI-native delivery increases the need for validation, governance, telemetry, security, and accountability. The DevOps Blueprint provides the structure for locating these requirements across Elastic Infrastructure, the CI/CD Pipeline, Application Release Orchestration, and Value Stream Management.

**AI-RELATED
CONTENT**



Blueprint describes the four operational layers through which those practices are implemented and connected in the delivery system (see Figure 4.3). The lower layers create the technical foundation for repeatable delivery, while the upper layers coordinate release decisions, expose flow, and connect delivery performance to business value. Each layer depends on the layers below it and provides feedback to adjacent layers. Together, the four layers show how delivery work moves from technical foundations through pipeline execution and release control to end-to-end alignment with outcomes.

The Blueprint is useful because organizations frequently improve the wrong layer for the problem they face. A team may try to increase deployment frequency when the real constraint is testing data. A leader may ask for better portfolio predictability when teams are blocked waiting for access to shared environments. A platform team may add self-service infrastructure while release approval remains manual and unclear.

The practical test is simple. Every capability should have a clear role within the Blueprint and a defined contribution to delivery. Metrics should be connected to relevant layers and outcomes to ensure that they support decisions and improvement rather than add reporting noise.

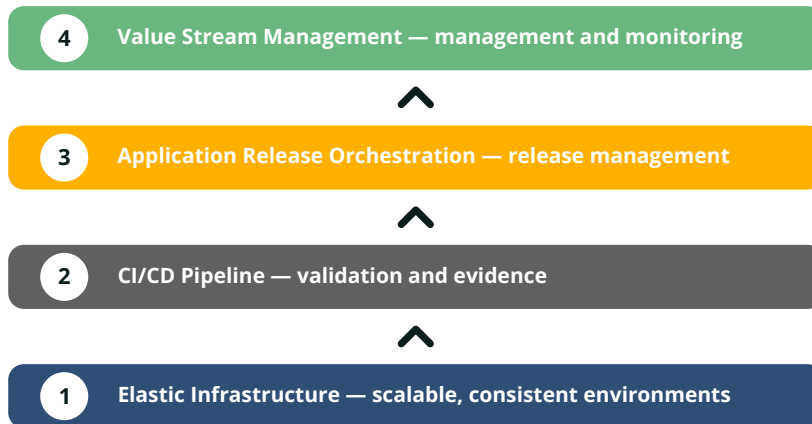


Figure 4.3 The DevOps Blueprint: four-layer delivery architecture¹⁰

4.3.1 Layer 1 Elastic Infrastructure: consistent, scalable environments

Elastic Infrastructure provides scalable, consistent, secure, and recoverable environments that can be provisioned when needed and adapted to changing workload demands. Infrastructure changes are managed through repeatable, governed practices, with security, observability, policy controls, and cost visibility built in. It provides the technical foundation for reliable development, testing, deployment, and operation.

It is a critical part of the delivery system. Teams cannot move quickly or safely when environments are manually created, inconsistently configured, or difficult to reproduce. Later, those weaknesses appear as test instability, release delays, incidents, security exposure, and cost surprises (Humble & Farley, 2010; Hornbeek, 2019; CNCF TAG App Delivery, 2023a).

The first layer includes:

- infrastructure as code (IaC)
- environment consistency
- automated provisioning
- secure configuration
- identity controls

- observability hooks
- cost visibility
- recovery patterns.

Together, these capabilities give teams dependable environments for development, testing, staging, production, and production-like validation. The goal is to ensure that environments remain consistent and reproducible as systems change.

Elasticity is more than scaling a server count. It includes the ability to provide environments on demand, absorb variable workloads, recover capacity after failure, and apply consistent policies across environments. Teams should be able to create approved infrastructure without waiting for a ticket queue, while the organization still maintains control over risk and cost.

Elastic Infrastructure

Notice that Elastic Infrastructure is both a Pillar and a layer in the DevOps Blueprint. It provides the common foundation for both because all other layers and pillars depend on infrastructure.

TIP



AI-native systems make infrastructure design more demanding. Depending on the use case and architecture, AI-native infrastructure may include burst capacity, accelerator scheduling, data movement, model-serving infrastructure, feature stores, vector indexes, and specialized monitoring. These capabilities should follow the same infrastructure governance principles as other production capabilities, including traceability, security, cost control, reliability, and clear operational ownership.

AI-RELATED CONTENT



A practical maturity signal is environment confidence. If a test failure regularly means 'the environment is strange', the foundation is weak. If teams cannot reproduce production-like behaviour before release, the pipeline will provide late and unreliable feedback. To improve environment consistency and confidence, organizations should:

- Codify environments and review infrastructure changes.
- Use self-service patterns with approved guardrails.
- Detect and correct drift early.
- Build cost and telemetry into the foundation.

Elastic Infrastructure appears in the full architecture in Figure 4.3 above and is developed further in Chapter 9.

4.3.2 Layer 2 CI/CD Pipeline: the delivery engine

The CI/CD Pipeline provides an automated and governed path for integrating, building, testing, scanning, packaging, and validating changes. It applies security and policy controls and produces traceable evidence to determine whether a change is ready for deployment and release. A pipeline is valuable when it provides fast feedback and trustworthy evidence.

Automating the movement of code through delivery stages is not sufficient for effective CI/CD. The pipeline must also validate changes, apply appropriate controls, and produce evidence that supports safe deployment and release decisions (Humble & Farley, 2010; Hornbeek, 2019; NIST, 2022).

At this layer, work is integrated, built, tested, scanned, packaged, signed, and prepared for deployment and release assessment. The pipeline should provide a clear record of:

- what changed
- who or what created the artifact
- which checks ran
- which policies applied
- what evidence supports readiness.

Those records matter for operations, security, audit, and incident learning.

Pipeline design should reflect the risk profile of a service or product change. A low-risk internal service may need a fast path with automated checks and automated promotion. A safety-critical or regulated service may need deeper verification, stronger segregation controls, and explicit review thresholds. Maturity comes from evidence-based flow, not from applying the same pipeline shape to every system.

The strongest pipelines give teams confidence while keeping feedback fast. That balance requires careful engineering. Checks should be organized to provide the most important feedback as early as possible. Unreliable or repetitive checks should be improved to reduce unnecessary alerts and delays. Findings should be prioritized according to their risk, impact, and relevance to the change.



Evidence should be visible to the people who make release, risk, and operational decisions. For AI-assisted work, the CI/CD Pipeline should apply Chapter 3's evidence principles: generated artifacts must be traceable and appropriately validated before progression. Where AI-assisted analysis supports release decisions, retain the source, relevant model/tool context, reviewer, and validation evidence.

The CI/CD Pipeline appears in the full architecture in Figure 4.3 above and is developed further in Chapter 7.

4.3.3 Layer 3 Application Release Orchestration: control and coordination

Application Release Orchestration coordinates deployment and release decisions across pipelines and delivery environments. It manages service dependencies, approvals, risk rules, evidence, exposure controls, and recovery paths. This capability is necessary because modern delivery often spans multiple services, teams, environments, and data changes, as well as feature flags, infrastructure changes, and compliance requirements. As delivery becomes more distributed, coordination becomes a capability.

Orchestration differs from the old release meeting. It should reduce manual coordination, make risk visible, and apply the right level of control based on the change. A small configuration update with all controls passed may move quickly. A change affecting regulated data, payment behaviour, identity and access management, AI-assisted decisioning, or critical infrastructure may trigger additional review.

The Application Release Orchestration layer should know:

- what is being deployed or released
- where it is going
- what dependencies exist
- what environments and users are affected
- what evidence is available
- what decision policy applies.

AI-assisted risk assessment may help summarize release evidence from tests, telemetry, dependencies, incidents, and security findings. However, release thresholds, human-review requirements, and final accountability remain organizational responsibilities.

AI-RELATED
CONTENT



It should also support rollback, roll-forward, feature disablement, canary decisions, controlled exposure, and post-deployment validation. Effective release control must include a clear recovery approach when a change does not perform as expected. Effective orchestration makes release activities routine and predictable. Teams spend less time coordinating release activities and more time improving the system that produces release evidence. Control becomes part of the delivery flow rather than being applied as a separate, late-stage activity before production.

Application Release Orchestration appears in the full architecture in Figure 4.3 above and is developed further in Chapter 8.

4.3.4 Layer 4 Value Stream Management: end-to-end alignment with outcomes

Value Stream Management connects delivery activity to business outcomes. It shows how work moves from demand and prioritization through development, validation, release, operation, and learning.

Without this layer, organizations may know that teams are busy, but not whether their work is delivering the intended outcomes and value (Kersten, 2018; Hornbeek, 2019; Pereira & Davis, 2024).

This layer helps leaders and teams see work in progress, wait time, rework, dependencies, risk, investment balance, and business impact. It also connects operational signals back to product and architecture decisions.

Value Stream Management should support decision-making, not simply provide another layer of reporting. Its purpose is to improve decisions. If flow time increases, leaders should ask where work is waiting and what constraints have appeared. If a feature consumes significant effort and produces weak adoption, product leaders should learn from the signal. Moreover, if reliability work is repeatedly deferred, the value stream view should make the resulting operational risk, delay, and future cost visible.

The main value of this layer is alignment. It gives product, engineering, operations, security, finance, and leadership a shared view of how work becomes an outcome. That shared view allows investment decisions to move from opinion to evidence.

Value Stream Management appears in the full architecture in Figure 4.3 above and is developed further in Chapters 13 and 15.

4.4 Connecting the DevOps Nine Pillars and the DevOps Blueprint

The DevOps Nine Pillars and the DevOps Blueprint are two views of the same operating system. The Pillars describe practice domains and what must mature. The Blueprint shows where those practices act inside the four-layer delivery architecture. Organizations need both views because improvement work can start from either direction.

The relationship is not one-to-one. A Pillar may operate across several Blueprint layers, and each Blueprint layer depends on practices from several Pillars.

For example, a maturity assessment may reveal that Continuous Testing is weak. In that case, the DevOps Blueprint shows that weak testing affects the CI/CD Pipeline, release readiness, and value stream predictability. Conversely, production incidents may reveal weaknesses in Continuous Monitoring and Observability. In turn, the DevOps Nine Pillars show that the issue may involve observability design, infrastructure telemetry, alert quality, SLIs, SLOs, and development ownership.

Table 4.1 Relationship between the DevOps Nine Pillars and the DevOps Blueprint

DevOps Nine Pillars	DevOps Blueprint
Define what must mature	Shows where practices operate
Nine interdependent practice domains	Four connected delivery layers
Include people, process, and technology	Connect infrastructure, pipeline, release orchestration, and value stream management
Assessed for capability and maturity	Used to design implementation and delivery capabilities
May operate across several Blueprint layers	Each layer may implement practices across several Pillars

This connection prevents shallow fixes. If releases are slow, the answer may involve test strategy, environment provisioning, approval policy, architecture coupling, or value stream prioritization. If security findings arrive late, the issue may involve design practices, pipeline controls, platform guardrails, and continuous governance. The model encourages teams to inspect the system rather than patch the symptoms. Accordingly, the same logic applies to AI-native issues: a hallucination, an unsafe recommendation, or a weak generated artifact should be traced across context, controls, telemetry, ownership, and product risk before it is assigned to one tool or team.

For leaders, the combined view supports investment. For practitioners, it supports design and daily decisions. For assessors and auditors, it provides a structured way to trace evidence from practice to capability to outcome.

4.4.1 Automation and orchestration: from task execution to system coordination

Automation executes a repeatable task. By contrast, orchestration coordinates multiple tasks, tools, policies, dependencies, decisions, evidence, failure handling, and feedback across the delivery system.

AI-assisted tools can generate artifacts, recommend actions, summarize evidence, and initiate selected tasks. These activities create value only when they are orchestrated. The organization still needs to know what was done, why it was permitted, which policy applied, what evidence was retained, and who remains accountable.

AI-RELATED CONTENT



This is why mature DevOps needs both: automation reduces manual effort and variation, while orchestration allows work to move safely through the larger operating model.

Build scripts, test runners, and infrastructure provisioning templates are examples of automation. By contrast, Application Release Orchestration decides how those automated steps fit together, when they may proceed, what evidence is required, what risk threshold applies, and what happens when a check fails. The distinction therefore becomes visible when the delivery system spans many teams and controls.

Table 4.2 Automation and orchestration in the delivery system

Dimension	Automation	Orchestration
Purpose	Executes a repeatable task.	Coordinates tasks, decisions, policies, dependencies, evidence, and feedback.
Scope	Usually a local step, tool, or service.	Spans tools, teams, environments, and the end-to-end workflow.
Examples	Build, test, scan, provision, package, or deploy.	Coordinate release evidence, approvals, dependencies, exposure, and recovery.
Policy handling	Runs a configured rule or action.	Applies risk thresholds and selects the permitted path.
Dependency coordination	Limited to the task being executed.	Makes cross-service and cross-environment dependencies explicit.
Failure handling	Reports, retries, or fails the task.	Stops unsafe progression, preserves evidence, and routes the right owner.
Evidence	Task result and execution log.	Trace of controls, decisions, approvals, exceptions, and outcomes.

Organizations sometimes automate fragments of an old process and expect system performance to improve. This produces faster local activity with the same queues, approval habits, rework loops, and unclear ownership. A poor process that runs faster can still produce poor outcomes faster.

A good practical test considers what happens when something fails. Automation should first report the failed task. Orchestration should then route the failure, preserve evidence, stop unsafe progression, notify the right owner, and create feedback for improvement. That is the difference between a faster toolchain and a controlled delivery system. In the same way, AI-assisted actions still require orchestration, evidence, and guardrails.

4.4.2 Continuous governance embedded in the workflow

Continuous governance is a defining feature of mature DevOps. Governance should operate inside the delivery workflow through policies, controls, evidence, approvals, exception paths, and audit trails. When governance appears only at the end of the work, it creates delay and encourages teams to treat control as an external obstacle (NIST, 2022; Hornbeek, 2024; PeopleCert, 2026d).

Embedded governance starts with clear policy and translates that policy into repeatable controls. Those controls may take the form of automated checks, platform guardrails, or required evidence. Other controls require human decisions because the risk or judgement requires accountable review. The goal is to make the decision path clear before the pressure of an approaching release rises.

AI-native delivery requires controls inside normal delivery workflows. Approved tools, prompt handling, sensitive data, model provenance, generated-code review, automated recommendations, and human-oversight thresholds should be governed as part of the operating model, not left in a separate policy document that teams consult too late.

AI-RELATED
CONTENT

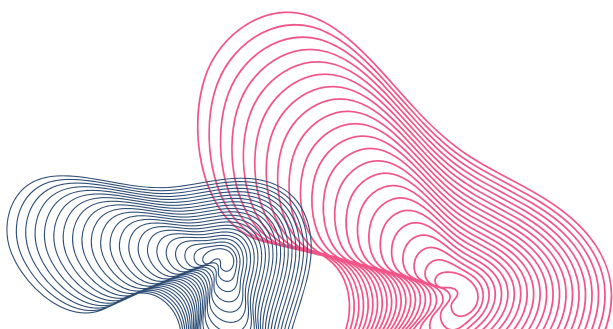


Governance should support speed and safety together. Low-risk changes should move through approved paths with minimal friction when checks pass. Higher-risk changes should receive additional reviews based on defined criteria. Exceptions should be documented, time-bound, owned, and visible.

Continuous governance also improves auditability because evidence is collected as work happens, reducing the scramble to reconstruct decisions later. As a result, audit, risk, compliance, and security teams can trust the system more when controls are visible, consistent, and measurable.

4.4.3 Anchor worked example | Case C: one change through the DOI® Operating Model

Context: Chapter 2 introduced CivicLink Public Services and its statutory eligibility change. This example uses the same change to show how the DevOps Nine Pillars and the DevOps Blueprint provide two complementary views of one delivery system. The case is useful because no single practice was sufficient:



citizen value depended on leadership decisions, engineering practice, service reliability, supplier coordination, security, and evidence¹¹.

The leadership team defined the outcome as a safer and more accessible eligibility journey delivered before the statutory date. It did not set deployment frequency or automation volume as the goal. The product, engineering, service-management, SRE, security, platform, supplier, and policy representatives worked as one value-stream group with explicit decision rights and escalation paths. Table 4.3 indicates how each of the Nine Pillars is applied and the evidence expected to demonstrate correct use.

The Blueprint located most of these capabilities in the delivery architecture, while Leadership and Collaborative Culture set the conditions across all four layers. Layer 1, Elastic Infrastructure, supplied repeatable environments, protected data, approved AI access, and telemetry foundations. Layer 2, the CI/CD Pipeline, integrated and validated human and AI-assisted artifacts while producing traceable evidence. Layer 3, Application Release Orchestration, coordinated supplier dependencies with risk thresholds, approvals, deployment, controlled release, and recovery. Layer 4, Value Stream Management, connected statutory demand with work in progress, wait time, operational risk, citizen outcomes, and learning.

Table 4.3 Applications and evidence for each of the DevOps Nine Pillars

Pillar	Application in the CivicLink change	Evidence expected
Leadership	Set the citizen outcome, risk appetite, funding boundary, statutory tolerance, and accountable decision owners.	Outcome statement, funding decision, risk thresholds, and decision log.
Collaborative Culture	Created a cross-supplier working group in which accessibility, operational, security, and policy concerns could be raised early.	Recorded concerns, resolved dependencies, rehearsal findings, and completed learning actions.
Design for DevOps	Separated eligibility rules from the user interface, designed a human fallback for AI classification, and made the citizen journey observable and reversible.	Architecture decisions, interface contracts, fallback path, telemetry design, and recovery design.
Continuous Integration	Versioned code, rules, infrastructure, model references, prompts, and documentation through one governed integration path.	Provenance, peer review, build records, policy checks, and a well-known mainline.
Continuous Testing	Combined unit, contract, accessibility, security, performance, continuity, and probabilistic AI evaluation.	Risk-linked test results, evaluation-set results, defect evidence, and accepted residual risk.
Elastic Infrastructure	Provided repeatable environments, protected test data, capacity for peak demand, and approved access to AI services.	Infrastructure definitions, drift results, capacity tests, access controls, cost signals, and recovery evidence.
Continuous Security	Applied threat modelling, data-handling rules, software supply-chain controls, and identity boundaries, and misuse monitoring.	Threat model, dependency and provenance results, data controls, access evidence, findings, and exception records.
Continuous Delivery and Deployment	Kept the change deployable, used controlled cohorts, separated deployment from release, and rehearsed rollback and feature disablement.	Release-readiness record, deployment record, release decision, cohort results, and recovery verification.
Continuous Monitoring and Observability	Measured successful citizen completion, service latency, manual-queue growth, supplier dependency behaviour, and AI classification quality.	SLI and SLO results, logs, metrics, traces, release context, AI-quality signals, and actionable alerts.

The two views exposed a constraint that a single-practice review would have missed. The pipeline was technically capable, but the manual fallback queue could not absorb a poor AI-classification result at full citizen volume. The pillar view showed weaknesses in Design for DevOps, Continuous Testing, and Continuous Monitoring and Observability. The blueprint showed where those weaknesses had to be corrected: design and evaluation in the pipeline, capacity and telemetry in infrastructure, and exposure control in Application Release Orchestration.

Outcome and trade-off. CivicLink deployed the full technical package but released it first to a controlled citizen cohort. A low-risk wording change followed the automated path, while a change to the AI confidence threshold required specialist review and explicit approval. The organization accepted a slower expansion of user exposure in exchange for stronger evidence and a dependable human fallback.

Transferable lesson. The DevOps Nine Pillars help identify weak practices, while the DevOps Blueprint shows where those practices must operate. The combined model prevents local fixes from hiding end-to-end constraints.

DECISION AND ACTION GUIDE

Executive decision: The executive sponsor selects one system constraint for improvement, defines the intended outcome and risk tolerance, allocates funding, and names an accountable improvement owner.

Role-specific actions: Product, architecture, engineering, platform, security, operations, and assessment roles map the constraint across the DevOps Nine Pillars and the DevOps Blueprint.

Evidence to review: A capability map, constraint hypothesis, dependency and evidence map, current baseline, target outcome, and investment decision.

4.5 Chapter summary

The DOI® Operating Model provides the structure needed to move from fragmented practice to coherent capability. The DevOps Nine Pillars define what must mature. The DevOps Blueprint shows how delivery work is organized. Continuous governance, measurement, and feedback make the system capable of improvement over time.

The model matters because DevOps now operates under greater load. Cloud platforms, distributed architectures, security obligations, regulatory expectations, and AI-generated work all increase the need for a disciplined operating model. Teams need autonomy, and the organization needs alignment. Those requirements can coexist only when the system is designed deliberately.

The chapters that follow expand the individual capability areas in depth. This chapter provides the map. The rest of PART TWO explains how each capability contributes to the whole system and how organizations can assess, govern, and improve it in practice.

An operating model is useful only when it helps teams deliver value more safely and visibly and learn more effectively from real work.



This chapter examines Leadership and Collaborative Culture, two interdependent practice domains in the DevOps Nine Pillars. Both are primary enablers of effective DevOps. Both are also common failure points when organizations change tools without changing the conditions in which teams work. DevOps is often introduced through automation, pipelines, cloud platforms, and delivery practices. These capabilities matter, as they give teams a technical advantage when the right operating conditions already exist. However, technical capability eventually reaches the boundary set by leadership behaviour and organizational culture.

At scale, the largest constraints often sit outside the pipeline. They appear in funding cycles, approval structures, conflicting incentives, ownership boundaries, risk tolerance, and the trust level between development, operations, security, product, and governance groups. A team can improve its build process and still be constrained by slow decision-making. A platform can provide excellent tooling and still fail when leaders reward local optimization.

AI-native engineering adds a leadership challenge because it increases the speed and volume of work that teams must trust, review, and govern. Leaders must create conditions for responsible use: clear accountability, practical guardrails, disciplined review, and a culture in which teams can challenge AI-assisted outputs without slowing valuable learning or delivery.

The chapter defines culture through observable behaviour, describes the leadership shift from task control to system

enablement, introduces human-AI trust as a management concern, and connects organizational design to flow, ownership, and accountability.

It is most relevant to leaders, transformation sponsors, product and technology executives, and senior practitioners, as well as platform leaders, security leaders, and anyone responsible for making DevOps work across organizational boundaries.

5.1 Leadership as a system enabler

DevOps transformations frequently struggle because they are treated as engineering initiatives rather than organizational change. Leaders approve new tools, fund automation programmes, and announce a new operating model while leaving the old decision system intact. The pipeline improves, yet delivery remains slow. Builds become automated, yet release approval remains manual. Teams talk about ownership, yet risk and funding decisions remain far from the actual work (Edmondson, 1999; Westrum, 2004).

Leadership defines the operating environment. In the DOI® Operating Model, Leadership is the practice domain concerned with creating and sustaining the organizational conditions in which DevOps can operate effectively. It shapes priorities, funding, governance, and decision rights. It also determines staffing, incentives, and tolerance for learning. Leaders are the people who exercise this responsibility

Leadership and Collaborative Culture

through their roles and decision authority; they are not limited to a single formal leadership team. Teams, on the other hand, operate inside those conditions every day. When leaders ask for faster flow while maintaining slow approval paths, teams receive mixed signals. Likewise, if leaders ask for shared ownership while measuring departments separately, local optimization becomes rational behaviour.

Leaders do not install DevOps, but they create the conditions in which DevOps becomes everyday work. This requires attention to necessary tasks that rarely appear in transformation announcements:

- Fund long-lived product and platform capability.
- Move governance from late approval to embedded control.
- Align security, operations, product, and engineering objectives.
- Give teams authority to make decisions close to work.
- Define clear boundaries for acceptable risk.

Together, these decisions create the operating environment in which teams work. Figure 5.1 shows how these system conditions influence team behaviours and, in turn, delivery outcomes.

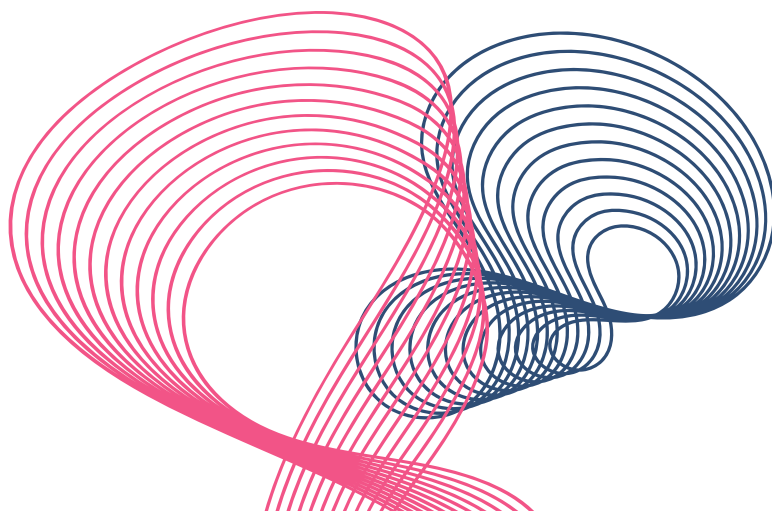
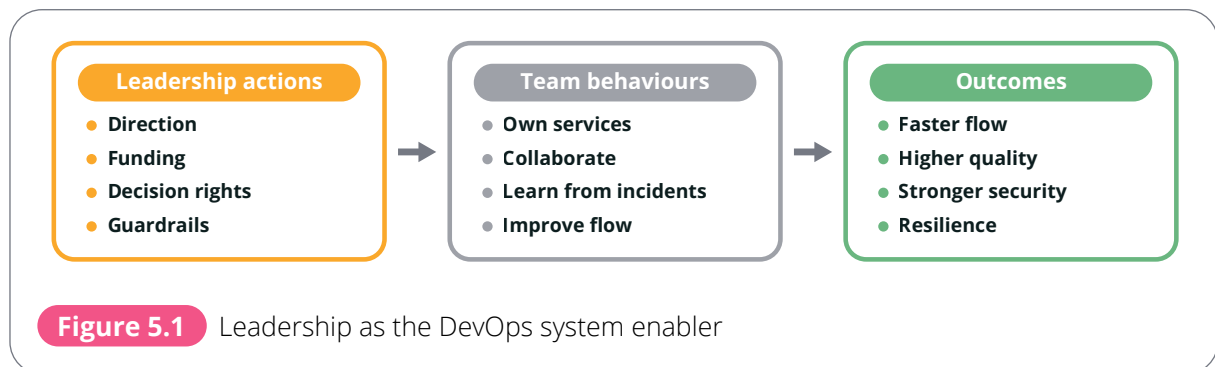


Table 5.1 Leadership responsibilities

Leadership responsibility	What leaders do	Expected outcome
Set direction	Define priorities, funding, and decision rights.	Teams work towards shared goals
Design the operating environment	Remove constraints and embed governance.	Faster, safer flow
Enable teams	Provide platforms, skills, authority, and incentives.	Greater autonomy and ownership
Establish governance	Define guardrails, accountability, and performance signals.	Safe, controlled delivery
Learn and improve	Review evidence, remove recurring blockers, and enable adjustments.	Continuous improvement

Managing constraints is one of the primary ways leaders continuously improve the operating environment. Leaders should know where work waits, where decisions queue, where teams depend on overloaded specialists, and where fear prevents honest reporting. A leader who cannot name the main constraints in the delivery system is managing through hope. However, hope is a poor operating model, especially when AI-assisted delivery increases the speed and volume of artifacts entering the system.

This does not mean leaders should become pipeline administrators. Their responsibility sits at the level of system design: clarifying intent, allocating investment, removing friction, and creating feedback loops. They also ensure teams have the skills, platforms, and authority to build and operate services responsibly. Just as importantly, they make trade-offs visible early enough to support informed decisions.

Leadership alignment matters because DevOps crosses functions. Technology leaders cannot change compliance policy alone; security leaders cannot change delivery ownership alone; product leaders cannot fix production reliability alone; and finance leaders cannot always see which funding choices create technical debt. Effective DevOps leadership requires alignment across products, technology, operations, security, risk, architecture, finance, and human resources. That alignment must be practical and visible, showing up in funding decisions, role expectations, measurement reviews, and the removal of blockers.

Leadership test

A leadership team can claim it supports DevOps when it can explain which constraints it has removed, which decisions it has moved closer to teams, which guardrails it has automated, and which outcomes have improved as a result.

KEY MESSAGE



Culture starts with what leaders consistently reward. Praising teams for hiding risk until the release date teaches the organization that appearances matter more than transparency. Punishing teams for reporting production defects honestly discourages openness and makes the system less safe, even if the metrics appear to improve. Likewise, funding choices send the same signal: prioritizing new features while neglecting platform, reliability, and security work allows technical debt to accumulate. Eventually, the organization has to repay it, and the invoice usually arrives during an incident.

The practical work of leadership is to make the desired behaviour easier than the old one by ensuring that:

- evidence is easier to produce
- secure defaults are easy to consume

- dependency blockers are visible
- incident learning is safe
- AI-assisted output is traceable.

This is the steady craft of leadership in DevOps. It is less dramatic than a transformation launch, and far more important.

5.1.1 From manager to enabler

Many managers were trained in operating models built around assignment, approval, escalation, and control. These behaviours can be useful in stable work with predictable tasks. DevOps work is different, as it crosses systems, disciplines, and sources of uncertainty. In this context, leadership shifts from directing individual activity to improving the conditions in which teams make responsible decisions.

This transition can be uncomfortable. For many leaders, control is often associated with personal approval: attending release meetings, reviewing status reports, and signing off on changes can make them feel close to the work. However, DevOps requires a more durable form of control. This form of control relies on clear intent, automated checks, observable signals, policy guardrails, and teams that can act responsibly within known boundaries. In this model, control moves into the system rather than sitting in the manager's calendar.

An enabling leader gives teams the context they need to make sound decisions. Teams should understand the business goal, risk appetite, customer impact, operational constraints, and boundaries for action. This requires more than a slogan about

Leader behaviour shift

The useful question moves from "Who is late?" to "What in the system is slowing responsible delivery?" That shift sounds small. In practice, it changes the entire management conversation.

KEY MESSAGE



empowerment. Decision rights must be explicit: teams need to know what they can decide independently, what requires consultation, what requires evidence, and what crosses a risk threshold. Table 5.2 summarizes the typical transition from older management patterns to more effective leadership patterns.

Removing blockers is core leadership behaviour. Blockers may include unstable environments, delayed security reviews, unclear product ownership, and brittle release approvals, as well as underfunded platform work, missing skills, and vendor constraints. They may also include architectural dependencies that force teams to coordinate with one another. These blockers are rarely solved by asking teams to work harder. Instead, they require leadership attention because they sit across organizational boundaries.

The leader also acts as a translator between organizational intent and team reality. Executives may care about market responsiveness, operational resilience, audit readiness, and cost control.

Table 5.2 Leadership shift from control to enablement

Old control pattern	Enabling leadership pattern
Assign tasks	Clarify intent, outcomes, and decision rights.
Approve every step	Set guardrails and risk thresholds.
Escalate answers	Remove blockers and support decisions close to the work.
Control the process	Coach capability and improve the system around the team.

Teams, on the other hand, experience those goals as work items, deployment constraints, incident load, tool friction, and dependency queues. A capable leader connects these perspectives without flattening either one. The team needs strategic context, and the organization needs truthful operational evidence.

Coaching becomes more important as autonomy increases. An enabling leader does not disappear; they shift their focus to developing judgement, strengthening team capability, and helping people understand trade-offs. This includes coaching teams to use metrics, interpret production signals, work with security partners, assess AI-generated outputs, and escalate risk without creating panic.

The transition also requires patience. Teams that have spent years waiting for approval will not become autonomous simply because an executive asks them to. They need practice, support, and repeated evidence that leaders will respond differently when uncertainty appears. If the first honest failure leads to blame, the old system will quickly return. People are practical: they believe more in repeated behaviour than leadership language.

A good DevOps leader remains accountable for outcomes. Enablement is not abdication. It requires sharper attention to the health of the system, stronger evidence, and better follow-through. Leaders should review outcomes, constraints, and learning with the same seriousness they once applied to task status. The work is more demanding because it requires judgement rather than escalation theatre.

AI changes the leadership transition by making activity easier to increase without necessarily improving the system. More generated code, tests, tickets, or incident summaries can create the appearance of progress. Enabling leaders therefore ask whether AI-assisted work improves flow, what evidence demonstrates its value, what controls validate it, and who remains accountable when the output is wrong. They also help teams develop the judgement needed to assess AI-generated work and escalate risk appropriately.

AI-RELATED CONTENT



5.1.2 Building and sustaining human-AI trust in engineering teams

AI is now part of the engineering operating model. It may assist with code generation, test creation, documentation, and incident analysis, as well as security triage, change risk assessment, and operational recommendations. These capabilities can reduce repetitive work and help teams reason across more information than they could manually process. They also introduce failure modes that are easy to underestimate when the output looks polished (NIST, 2023a; DORA, 2025).

Trust in AI should be calibrated, rather than blind or completely absent. In particular, blind trust creates risk while blanket distrust wastes capability and drives shadow adoption. By contrast, calibrated trust means teams understand where AI is useful and where it is weak, how outputs are validated, and who

remains accountable. This is a cultural requirement as much as a technical one.

A human-AI trust model starts with approved use. Teams need practical guidance about which AI tools and models they may use, what data can be entered, which work products may be generated, and which outputs require review. Vague permission leads to inconsistent practice. Conversely, vague prohibition often creates unobserved work outside the governed system.

The second component is provenance. A team should be able to understand whether code, a test, a runbook, a configuration change, or an incident summary was created by an individual, assisted by AI, or generated only by an AI workflow. Provenance does not need to be heavy for every low-risk artifact, but the delivery system should capture enough information to support review, traceability, and learning.

Human review remains central. Therefore, AI-generated output should enter normal engineering controls. For example, code requires review, while tests require a review of their intent and usefulness. In the same way, documentation requires accuracy checks, while incident summaries require comparison against evidence. Security recommendations also require validation against threat context. This matters because an AI-generated answer can be confident and wrong. In other words, confidence and correctness are different properties.

Validation should be proportionate to risk. For example, a generated unit test for a low-risk utility may require ordinary review. By contrast, a generated infrastructure policy, production remediation script, security exception, or deployment recommendation needs stronger evidence and approval. Review thresholds should therefore be defined before an incident forces decisions under pressure.

Operational feedback closes the trust loop. After AI-assisted work enters the system, teams should examine how it performs:

- Did AI-generated tests catch meaningful defects?
- Did AI incident triage reduce recovery time?

Human-AI trust questions

- What can the AI system do?
- How do we know whether its output is reliable?
- Who reviews the output?
- What evidence is required before use?

How do we improve the workflow when the output fails?

KEY MESSAGE



- Did a deployment risk score correctly identify dangerous changes?
- Did AI-generated documentation remain accurate?

These questions keep adoption tied to evidence rather than theatre.

Leaders must also account for privacy, intellectual property, security, and compliance. Data used in prompts or training workflows may contain sensitive information. In addition, generated output may reflect licensed material, outdated assumptions, or insecure patterns. For this reason, teams need clear policies and support for tooling. Asking engineers to make individual policy decisions at speed is unfair and unreliable.

A healthy culture treats AI errors as learning opportunities. If people are punished for admitting that an AI-generated artifact failed review or caused confusion, teams will hide problems. Similarly, if every AI issue becomes a reason to ban useful tools, teams will route around governance. The better pattern is disciplined use, visible evidence, and steady improvement of prompts, models, workflows, and controls.

Human-AI trust should be earned through evidence. Teams should know what the AI system is allowed to do, what data it may use, how outputs are reviewed, and when human approval is mandatory.

The core principle is straightforward. AI can assist engineering judgement, but it cannot own organizational accountability. People, teams, and leaders remain responsible for the systems they deliver and operate. That responsibility should be built into the culture before AI becomes routine work.

5.2 Collaborative Culture as a system enabler

In the DOI® Operating Model, Collaborative Culture is the practice domain concerned with the behaviours, norms, and working relationships that enable people to share ownership, collaborate across boundaries, learn from evidence and failure, and improve the delivery system together. Collaborative Culture is reinforced by organizational structure. Teams are more likely to adopt and sustain DevOps behaviours when the operating environment supports the desired way of working. This includes organizational design, interaction models, leadership, and measurement. Culture becomes durable when these elements consistently reinforce shared ownership, fast learning, and responsible decision-making. Team boundaries should support flow and accountability, while cultural indicators should expose friction, fear, dependency, and weak learning without turning measurement into surveillance. This section connects Collaborative Culture with organizational design and the evidence leaders can use to improve the operating environment.

5.2.1 Defining a DevOps culture

A DevOps culture is a pattern of behaviour that helps teams deliver useful change safely and repeatedly. It can be seen in how people raise risks, how teams handle incidents, how security participates in design, how leaders respond to bad news, and how product

decisions account for reliability and operational consequences. Culture is visible in ordinary work long before it is visible in a presentation (Edmondson, 1999; Westrum, 2004; Forsgren et al., 2018).

The foundational elements are familiar, though they are often underdeveloped:

- **Psychological safety** allows people to speak about uncertainty, weak assumptions, operational risk, and mistakes without fear of personal punishment.
- **Shared ownership** keeps teams accountable for the services they build, deploy, operate, secure, and improve.
- **Blameless learning** turns failure into a source of system improvement.
- **Transparency** makes work, risk, and trade-offs visible.
- **Continuous improvement** keeps the organization from treating today's process as a permanent process.

Psychological safety deserves careful treatment. It is sometimes mistaken for comfort. In engineering organizations, it can be understood as permission to raise concerns early. A team member should be able to say that the deployment plan is risky, the test data is poor, the architecture is too coupled, the AI-generated code needs review, or the deadline is forcing unsafe behaviour. Leaders who ask for transparency and punish discomfort will get silence.

Shared ownership changes the conversation around delivery. Development, operations, security, product, and platform groups cannot independently optimize their own narrow objectives and expect the whole system to improve. A developer who never sees production behaviour is missing part of the engineering problem. A security specialist who only appears at the end of the release process arrives too late. A product owner who ignores operational risk will eventually ship customer dissatisfaction.

Blameless learning does not mean absence of accountability. It means the organization first examines the conditions that made the failure possible while recognizing that people still have responsibilities, standards still matter, and poor practices still require correction.

Cultural indicator

A team has a DevOps culture when people can raise risks early, improve the system after incidents, build security into daily work, and use production evidence to guide the next change.

KEY
MESSAGE



The difference is that incident reviews begin with curiosity about the system rather than a search for someone to blame. Experienced operations professionals learn this lesson over time because complex systems rarely fail for a single reason. More often, failures emerge from a combination of weak signals, time pressure, assumptions, hidden dependencies, and missing feedback.

Transparency provides raw material for improvement. Teams need visibility into flow, quality, reliability, security, cost, and customer impact. They also need visibility into blocked work, work that is hidden, and work that is being repeated because the system did not learn the first time. In a healthy culture, transparency is useful rather than ceremonial. The numbers lead to better decisions, not to longer meetings.

A useful culture has both autonomy and alignment. Autonomy without alignment creates local variation and operational drift. Alignment without autonomy creates slow approval chains and learned helplessness. The practical balance comes from clear goals, explicit guardrails, shared metrics, and platforms that make good practices easy to adopt. When these conditions exist, teams can move quickly while staying connected to enterprise outcomes.

Culture must be reinforced through routines. Planning should include operational risk and dependency review. Code review should include maintainability, security, and operability. Incident reviews should create

action items that leaders fund and teams complete. Roadmap decisions should also consider reliability, technical debt, and platform capacity. If the routines do not change, the culture remains mostly language.

5.2.2 Organizational design for DevOps: team topologies and interaction modes

Culture is shaped by structure. Teams cannot behave with end-to-end ownership when the organization forces work through constant handoffs. They cannot move quickly when every meaningful change depends on overloaded shared teams. They cannot own reliability when operations knowledge sits downstream. In DevOps, the architecture of the organization becomes part of the architecture of delivery (Skelton & Pais, 2025).

Traditional functional structures often optimize specialization. Development writes code, operations runs systems, security reviews risk, quality assurance tests software, and release management coordinates deployment. This can provide deep expertise, but it also creates queues when work requires fast feedback across disciplines. DevOps therefore requires team boundaries and interaction models that support flow, shared ownership, and timely access to expertise.

Stream-aligned teams form the foundation. They own a product, service, or value stream from its conception through its operation and improvement. To do this well, they need product context, engineering skills, operational awareness, security expectations, and access to platform capabilities. They do not need every specialist embedded permanently. Yet they do need enough authority to avoid turning each change into a negotiation.

Platform teams support this ownership by making the safe path easier to follow. They provide reusable internal capabilities such as self-service environments, pipeline templates, deployment patterns, observability defaults, security controls, and operational tooling.

A good platform reduces cognitive load without taking accountability away from product teams. If it becomes a ticket queue, the organization has recreated the old bottleneck using new language.

Together, these team types create an ecosystem in which stream-aligned teams remain responsible for delivery while drawing on platform capabilities, specialist expertise, and temporary enablement when needed.

Enabling teams help other teams build capability. They may coach test automation, observability, cloud practices, DevSecOps, incident learning, or AI validation. Their success is visible when the teams they support become more capable over time. An enabling team that has become a permanent approval body has lost its purpose. Enablement should reduce dependency, not institutionalize it.

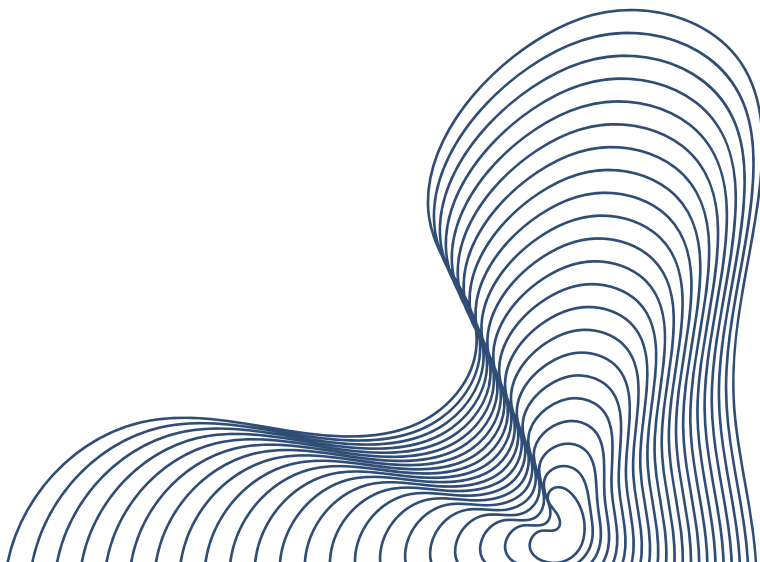
Some work still requires deep specialist expertise. Complicated subsystem teams may be necessary where that expertise is genuinely required. Examples include cryptography, high-performance data platforms, safety-critical control logic, model-serving infrastructure, or specialized hardware integration.

Their value depends on carefully managed interfaces. Without those boundaries, specialist knowledge becomes a constraint on ordinary delivery work.

Interaction modes matter as much as team types. Teams should know whether they are collaborating, consuming a platform as a service, or receiving facilitation from an enabling team. Informal dependencies are expensive because they remain hidden until planning fails. Explicit interaction modes make coordination manageable. They also make it easier to see when a dependency should be removed through architecture, self-service, automation, or clearer ownership. Figure 5.2 illustrates how stream-aligned teams interact with platform, enabling, and specialized subsystem teams to support delivery while reducing unnecessary coordination.

Reorganization alone rarely creates DevOps maturity. Moving boxes on an organization chart can clarify reporting lines, yet flow only improves when decision rights, funding, platforms, architecture, and measurement change with the structure. A team renamed as a product team will still struggle if it depends on five approval queues and cannot observe its service in production.

Organizational design should be reviewed as evidence emerges. Repeated blockers are design signals. An ignored platform may not be solving real user problems. Late security findings may indicate that security and delivery are interacting too late. DevOps culture becomes more durable when the structure supports the behaviour expected from teams.



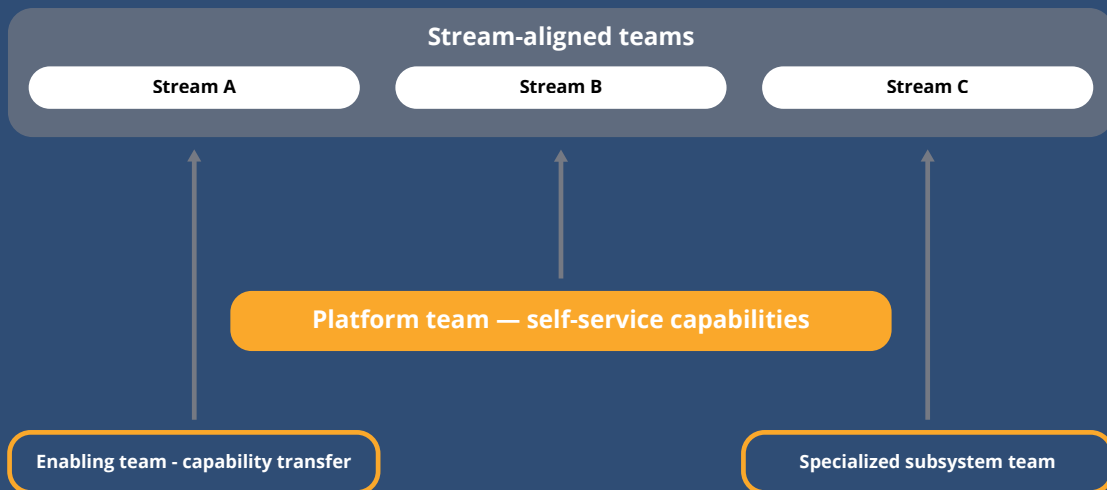


Figure 5.2 DevOps team topologies and interaction modes¹²

Design rule

Organize teams around the flow of value wherever possible. Use platform and enabling teams to reduce cognitive load. Keep specialist teams focused on genuinely specialized work with clear interfaces. Make team interactions and dependencies explicit, and redesign recurring dependencies that constrain flow through clearer ownership, self-service, automation, or architecture. Review organizational design as evidence emerges, rather than treating team structures as permanent.

TIP



AI-native work requires provenance, validation, evidence, and accountable oversight. In culture terms, this means that teams must be comfortable challenging AI outputs, admitting when generated work is weak, and using AI neither as magic nor as something forbidden by default. Mature teams treat AI as assistance within a controlled engineering system.

AI-native delivery also raises organizational-design questions: which AI capabilities belong to product teams, which should be provided by platforms, and which require central governance. As Chapter 3 explains, the aim is to preserve useful autonomy while reducing unmanaged variation.

AI-RELATED CONTENT



5.2.3 Measuring culture: leading indicators and system signals

Culture is difficult to measure directly, but it can be observed through behaviour. Leaders should be cautious with sentiment surveys as the only evidence. Surveys can reveal useful patterns, yet they rarely explain how work moves. A stronger view combines qualitative feedback, delivery data, operational signals, incident learning, team health, and observable leadership behaviour (Westrum, 2004; Forsgren et al., 2018).

The purpose of cultural measurement is not judgement but improvement. It should not rank teams or create a public scoreboard for psychological safety, but help leaders identify friction, fear, overload, or misalignment. When people believe the measurement system will be used against them, they manage the signal and the organization gets another dashboard, with less truth.

Leading indicators matter because lagging indicators arrive late. Attrition, major incidents, missed commitments, and audit failures usually appear after the system has been unhealthy for some time. Earlier signals include repeated escalations, blocked work, low participation in retrospectives, unresolved incident actions, delayed security engagement, hidden work, poor on-call experience, and teams that avoid raising uncomfortable information.

No single metric can describe organizational culture. Instead, leaders should look for patterns across multiple indicators: early risk reporting, incident follow-through, cross-functional participation, reduced handoffs, faster decisions, and the safe exposure of weak signals before they become failures. These indicators show whether the environment supports learning, collaboration, and continuous improvement.

Interpreting cultural indicators

Delivery metrics can reveal cultural issues when interpreted carefully. Long lead times may indicate approval queues, excessive dependencies, or unclear ownership.

Rising change failure may indicate weak testing, poor review, rushed work, or inadequate operational learning. Low deployment frequency may indicate technical constraints, risk aversion, or governance friction. Metrics rarely explain themselves. They should start conversations with the teams closest to work.

Incident learning is one of the clearest cultural signals. An organization that claims to be blameless should be able to show completed post-incident actions, improved runbooks, better alerting, revised tests, architecture changes, and clearer ownership. Incident review without action teaches cynicism. A postmortem that sits in a folder has not improved the system.

Human-AI trust needs its own evidence. Leaders should watch whether teams use approved tools, record provenance where needed, review generated outputs, report AI-related failures, and improve prompts or controls based on feedback. They should also check whether AI adoption reduces cognitive load or adds review confusion. Psychological safety can be observed in ordinary meetings. In practice, these behaviours become visible through questions such as:

- Do people raise risks before they occur in incidents?
- Do junior engineers challenge assumptions?
- Do security specialists feel invited into design discussions?
- Do operators feel heard when they describe production fragility?
- Do teams admit when AI-generated artifacts are weak or misleading?

Together, these behaviours show whether culture can process truth.

**AI-RELATED
CONTENT**



Measurement cadence matters. Cultural signals should be reviewed close to the work, not only in quarterly executive meetings. Team retrospectives, operational reviews, incident reviews, value stream reviews, and platform feedback sessions all provide evidence. One complaint may be noise, but the same complaint across teams is usually a system signal.

Leaders should respond to cultural indicators with visible action. When teams report dependency waits, leaders should reduce the dependency or fund the platform capability. When on-call load is damaging team health, they should adjust service ownership, alert quality, and staffing. Moreover, when AI use is inconsistent, they should provide practical guardrails. Measurement earns trust when people can see that truthful signals lead to better conditions.

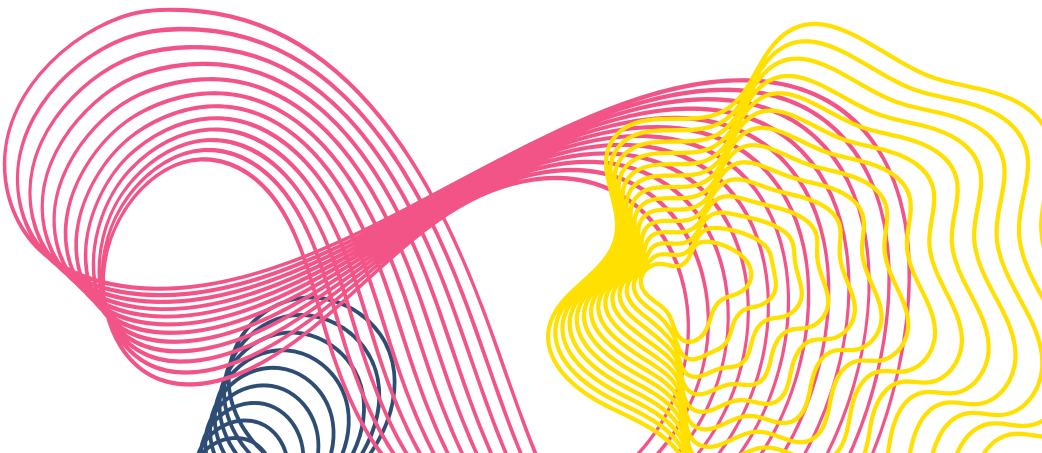
Warning sign

Culture is becoming unsafe when bad news arrives late, metrics look perfect, incident actions remain open, and teams describe problems privately that they will not raise in normal forums.



Table 5.3 Leading indicators of DevOps culture

Area	Leading indicators	Leadership response
Team behaviour	Retrospectives, escalation, collaboration	Support a blameless culture; remove blockers
Delivery flow	Lead time, deployment frequency	Value stream analysis
Incident learning	Closed actions, runbooks	Strengthen learning
AI trust	Review quality, provenance, approved tools	Improve AI governance



5.3 Leadership, security, and shared ownership

Security leadership is essential. The Chief Information Security Officer (CISO) and security organization provide risk strategy, policy direction, specialist expertise, threat awareness, and oversight. Those responsibilities remain important in DevOps. The failure occurs when DevSecOps is treated as security's responsibility alone. Secure delivery then becomes a gate owned by a separate function rather than a capability embedded across the delivery system.

When one silo owns DevSecOps, predictable behaviours appear. Engineering teams wait for security approval rather than building security into the workflow. Security teams become overloaded with reviews and exceptions. Product leaders treat security findings as external delays. Platform teams may implement controls without enough user feedback. Risk becomes visible late, when the cost of correction is high.

The better model is shared accountability with clear roles. Security leadership defines expectations, policies, risk thresholds, and specialist guidance. Engineering leaders make secure engineering capability part of team practice. Product leaders include security and resilience in prioritization. Platform teams provide secure defaults, reusable controls, policy as code, and automated evidence. Teams remain accountable for the security of the services they build and operate.

This model reduces bottlenecks by making security part of everyday delivery. Threat modelling informs design, pipeline checks identify dependency and secrets issues early, infrastructure policies apply automatically, and evidence is captured continuously. High-risk exceptions still receive human review, but ordinary secure delivery becomes part of normal work.

AI-native DevSecOps reinforces the need for shared ownership. AI-assisted artifacts, prompts, agents, datasets, and operational workflows require approved tools, validation pathways, escalation rules, and clear accountability. Security experts set the guardrails, while teams apply them in daily engineering work.

A security champion model can support this shift. Champions help teams understand secure coding, threat modelling, risk-based prioritization, and incident learning. They do not replace security specialists; they bring security knowledge into team routines. The role needs time, training, recognition, and a community of practice. Otherwise, it becomes another title added to an already overloaded engineer.

The leadership lesson is direct. DevSecOps should have executive sponsorship and specialist security leadership, while ownership of secure delivery must be distributed. The measure of success is simple. Teams should be able to move faster because the safest path is easier to follow, and security leaders should spend less time chasing preventable issues at the end of the process.

Anti-pattern box: "Our CISO owns DevSecOps."

The belief: This statement usually sounds responsible.

The reality: It often produces the opposite effect. It places secure delivery inside a separate silo and allows the rest of the organization to treat security as somebody else's work.

The better pattern: The CISO leads risk strategy and security expectations. Product, engineering, platform, operations, and security share responsibility for secure delivery outcomes.

KEY MESSAGE



Case A – Northstar Financial Services (composite): Leadership response after a failed deployment

A payment-service deployment failed during a high-pressure reporting period. An engineer had raised a concern about a configuration dependency in a team channel, but the concern was not escalated because the schedule was already under scrutiny. When the service degraded, the first management reaction was to ask who had approved the change.

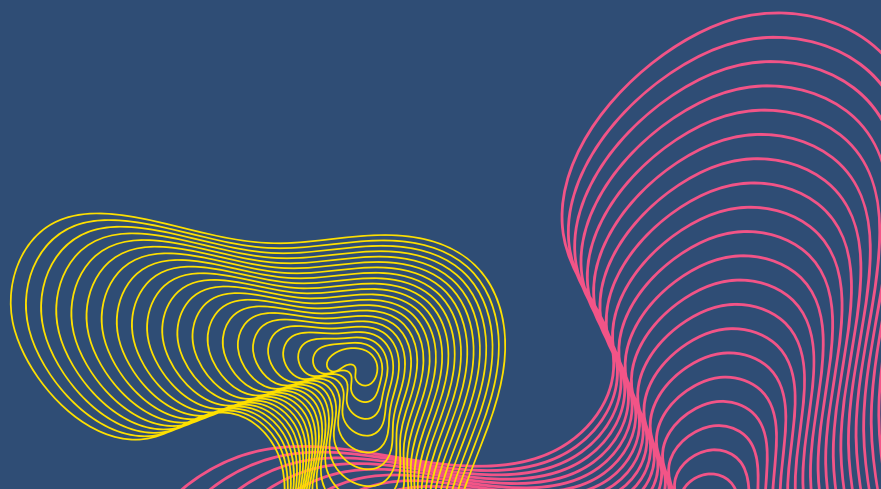
The accountable executive changed the tone of the response. She stated that the purpose of the review was to understand the conditions that made the failure likely, protected participants from retaliation, and paused discussions of individual performance. The review found an unclear deployment decision owner, an unrehearsed rollback path, a noisy alert, and an incentive that rewarded meeting the date more visibly than exposing risk.

Northstar named one accountable owner for the deployment decision, created an explicit escalation route for unresolved risk, rehearsed recovery for the service, and tracked learning actions to completion. The next release was delayed while the platform team corrected the configuration pattern. Leadership accepted the delay rather than asking the team to hide the remaining risk.

The important cultural signal was not the wording of the values statement. It was what leaders did when bad news became expensive. Engineers subsequently raised weak signals earlier because the organization had demonstrated that transparency would lead to system improvement rather than punishment.

Transferable lesson: Culture is observable in stressful decisions. Psychological safety, accountability, and learning become real when leaders protect early risk reporting, examine system conditions, and fund the corrective work.

SCENARIO



DECISION AND ACTION GUIDE

Executive decision: The executive team defines not only the leadership behaviours expected when bad news arrives, but also the decisions that move closer to teams, the incentives that must change, and the accountable owner for unresolved systemic constraints.

Role-specific actions: Managers create safe escalation paths and remove blockers; team leaders close incident-learning actions; people leaders review incentives; coaches support the behaviour change.

Evidence to review: Risks raised and acted on, decision latency, completed learning actions, team-health signals, removed dependencies, and repeated problems that remain open.

5.4 Chapter summary

Leadership and Collaborative Culture determine whether DevOps becomes a durable operating capability. Tools, automation, pipelines, and platforms provide important mechanisms, but they cannot compensate for fear, unclear ownership, conflicting incentives, late security engagement, unmanaged dependencies, or leadership behaviour that rewards local optimization. The operating environment determines what teams can sustain.

AI-native delivery raises the importance of both leadership and culture. On the one hand, leaders create the conditions for responsible AI use through clear accountability, practical guardrails, and evidence-based review. On the other hand, culture determines whether teams can challenge generated outputs and improve controls without fear.

The next chapters move deeper into technical and architectural capabilities. Those capabilities can only scale when leadership and collaborative culture support them. DevOps begins to take hold when leaders create conditions for flow, feedback, security, resilience, and learning, and then adjust those conditions when evidence shows the system needs to change.



This chapter is concerned with one of the most important elements of that environment: system design. It explains that DevOps capability is constrained or enabled by the design of the application being delivered. Application architecture shapes the constraints within which the DevOps delivery system operates. Applications cannot be made observable, testable, deployable, secure, or governable by goodwill after they are built. Those properties need to be designed into the application architecture while the important choices are still inexpensive to change.

Design for DevOps is therefore the practice domain concerned with designing applications so that the wider DevOps delivery system can build, test, secure, deploy, observe, operate, and improve them effectively.

DevOps performance is limited by application architecture choices before DevOps processes are automated. These choices influence delivery flow, operational resilience, and the ability to change the system safely. This becomes more important in AI-native delivery, where code, tests, configuration, prompts, and operational recommendations may be produced faster than the organization can review them by hand.

This chapter is especially relevant to application architects, senior engineers, platform teams, engineering leaders, security leaders, site reliability engineers (SREs), product owners, and anyone who participates in design review. The chapter also matters to executives because many expensive delivery constraints are created long before a programme asks for funding to remove them.

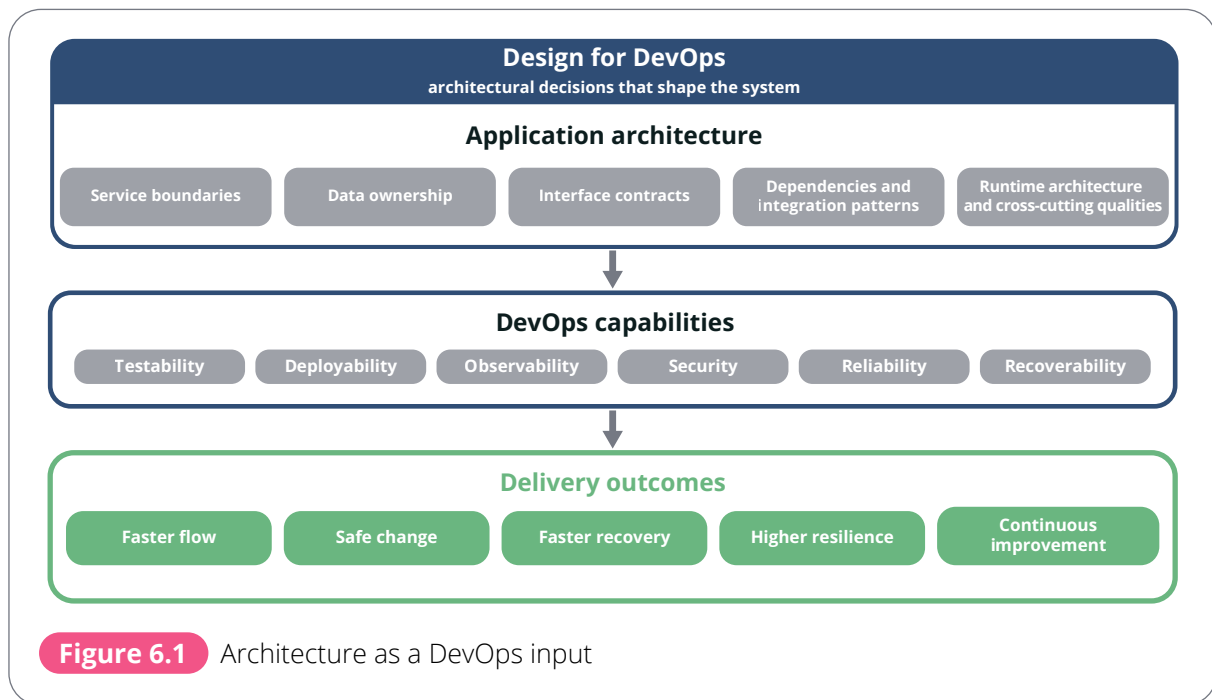
6.1 Architecture as a DevOps input: decisions that enable or constrain delivery

Architecture is one of the earliest inputs into DevOps performance. Tooling can automate a weak design, but it cannot remove the coordination cost created by poor boundaries, hidden dependencies, unclear data ownership, or runtime coupling. When teams struggle to release independently, the first useful question is often architectural, not procedural (Forsgren et al., 2018; Nygard, 2018).

Architecture is therefore both a technical and a leadership concern. Leaders influence delivery by funding, prioritizing, and governing the architectural decisions that determine how teams build, test, deploy, observe, and operate systems. Figure 6.1 illustrates how architectural inputs shape DevOps capabilities such as testability, deployability, observability, security, reliability, and recoverability, and how these capabilities support delivery outcomes.

A delivery pipeline moves work through a system that already has structure. In an organization where every small change

Design for DevOps



requires coordination across five teams, the pipeline simply automates the delay. With clear service boundaries, stable interfaces, owned data, and observable behaviour, the same pipeline becomes a reliable delivery engine. The difference is designed before the pipeline runs.

Architectural decisions affect build speed, testing scope, deployment risk, recovery options, and operational visibility. A shared database, for example, can create release coupling that no deployment tool can hide for too long. A synchronous dependency chain can turn one small service failure into a customer-visible outage. A poorly defined Application Programming Interface (API) can make contract testing unreliable and make every consumer team nervous about change.

Design for DevOps therefore starts by asking how the application or system will be changed, operated, and maintained. The design review should consider service ownership, deployability, testability, and observability, as well as security boundaries, reliability, data lifecycle, policy controls, and recovery behaviour. These topics are operational design concerns and they belong in the same conversation as scalability, performance, and functional correctness.

Good architecture gives teams room to move by reducing unnecessary negotiations, providing natural places to automate validation, and allowing failure to be detected, contained, and diagnosed. It also makes governance easier because controls can attach to known boundaries. A useful design can be imperfect, but teams still need to understand it well enough to own it.

Design review checkpoint

A useful DevOps design review asks whether a team can test a change, deploy it, observe it, secure it, and recover from failures without coordinating across half the organization. When the answer is no, the design has identified a delivery constraint, even if the system appears technically elegant on paper.

KEY MESSAGE



The following sections focus on three design qualities that have a particularly direct effect on delivery flow: observability, testability, and deployability. Security, reliability, operability, recoverability, data, and governance remain part of the broader design review and are addressed where relevant in this chapter and in the specialist chapters that follow.

6.1.1 Designing for observability: making systems understandable in production

Observability is a design requirement. Like service boundaries and interfaces, observability is an architectural decision rather than an operational enhancement. Teams should decide what the system must explain before the first serious incident asks the question. Logs, metrics, traces, events, release metadata, and business signals all become more useful when they are planned as part of the system design instead of scattered through the code after production pain appears (Majors et al., 2026; OpenTelemetry, 2026).

A production system needs to provide a clear and connected view of what is happening. A request should be traceable across service boundaries, with logs that connect errors to users, versions, feature flags, regions, and dependencies. Metrics should describe both technical behaviour and user impact. Events should show what and when changed and who was affected. Many organizations collect data but still lack observability. They have dashboards, but the dashboards do not answer the questions that matter during an incident. CPU, memory, and uptime may look normal while a particular customer segment cannot complete checkout. The issue may lie in a dependency, a data contract, a feature flag, or a release interaction. This is why correlation matters. Effective observability connects telemetry across services, deployments, infrastructure, and business events so teams can understand not only what failed, but also why it failed and who was affected.

Designing observability means deciding during design what signals the system must emit and how they will be correlated to explain behaviour in production. Teams should define which logs, metrics, traces, events, deployment metadata, and business signals will be collected, how trace and correlation identifiers will propagate across services, and which business events matter. They should also define what evidence will support service-level objectives (SLOs), incident response, security detection, product learning, and privacy requirements.

Good observability also supports design improvement. Production traces may reveal a dependency chain that needs simplification. Error data may show that one integration dominates customer pain. AI drift signals may show that a model requires retraining or stricter routing. Observability closes the design loop by turning production behaviour into evidence for the next architectural decision. Every deployment becomes an opportunity to improve the system rather than simply operate it. In this way, observability supports both reliable operations and continuous architectural improvement.

Operational checkpoint

Before a service is considered ready for production, the team should be able to answer three questions:

- How will we know it is healthy?
- How will we know users are harmed?
- How will we know what changed when behaviour shifts?

A design that cannot answer those questions is not ready to operate safely.

**KEY
MESSAGE**



Anti-pattern: “We’ll add observability later”

This anti-pattern appears when teams treat observability as cleanup work. The system is designed, built, and released; only after an outage or customer complaint does the organization ask for logs, metrics, traces, dashboards, and alerts. By then, the system may not expose the right boundaries, identifiers, state transitions, business events, or user context.

Retrofitted observability is often expensive and disappointing. Dashboards appear, but teams still cannot explain failures. Logs exist but they do not correlate. Metrics show resource health but miss user impact. Traces cross some services and then disappear at a critical dependency. The result is familiar to anyone who has worked on a serious incident with poor instrumentation: long calls, guesswork, repeated redeployments, and tired people staring at partial evidence.

The damage extends beyond incident response. Poor observability weakens product learning, security detection, reliability management, architecture improvement, and AI governance.

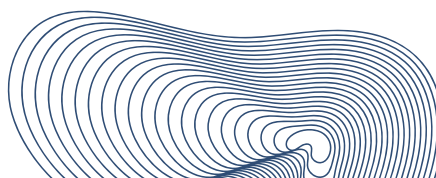
When a model, prompt, retrieval index, or agent workflow behaves unexpectedly, teams need telemetry that explains inputs, outputs, context, version, policy outcomes, and human intervention. Without that evidence, the organization has speed without understanding.

The better pattern is to define observability requirements during design. Services need telemetry expectations before production; critical user paths need signals for both technical behaviour and user impact. Releases need traceability, and AI-enabled features need behavioural, safety, cost, and quality signals. These decisions do not require a complex monitoring estate on day one. They require the system to emit the evidence needed to operate and improve.

Warning signs: The anti-pattern is present when teams say they will add logging after launch, when incident reviews repeatedly ask for data that was never captured, when dashboards multiply without improving diagnosis, or when business impact cannot be connected to technical behaviour. It is also present when AI features are released without monitoring for drift, refusal behaviour, grounding, cost, or misuse.

The better pattern: Design the signals with the system. Validate that they work before production exposure. Review them after incidents and releases. Improve them as the service evolves. Observability is one of the most practical ways to make design accountable to production reality.

**KEY
MESSAGE**



6.1.2 Designing for testability: enabling fast, reliable feedback

Testability is an architectural quality. It determines how quickly a team can learn whether a change is safe. When a system is difficult to isolate, difficult to configure, and dependent on fragile environments, testing becomes slow and untrusted. Slow feedback creates larger batches, delayed learning, and more expensive defects (Humble & Farley, 2010; Hornbeek, 2024). Like observability, testability must be designed into the system rather than added after implementation.

Designing for testability starts with boundaries. Components should expose interfaces that allow them to be tested independently, without recreating the entire production environment. Service contracts should be stable enough for consumer teams to validate expectations. Test data should be planned rather than improvised. External dependencies should be simulated or controlled where appropriate so that test failures reveal system behaviour rather than environmental noise.

A well-engineered test strategy uses different levels of validation for different kinds of

confidence and overall test coverage. Table 6.1 summarizes their primary purpose. Unit tests give fast feedback on local logic. Contract tests protect service expectations. Integration tests validate important paths across components. Performance and resilience tests expose scaling and failure behaviour. Security tests validate assumptions about identity, data, dependencies, and policy. Production validation then confirms behaviour under real conditions.

A well-designed system allows these tests to run without heroic setup. Teams need reproducible environments, clear configuration, documented dependencies, representative data, and observable test results. A test that fails for unclear reasons becomes a tax on delivery because it teaches teams to ignore the pipeline. Ignored pipelines eventually become expensive decoration.

A testable system reduces dependency on individual memory. As a result, engineers should be able to make a small change, run meaningful tests, and understand the result. In the same way, a security engineer should be able to validate controls without waiting for a quarterly review. At the platform level, teams should be able to offer approved patterns that make the common test path easy. In this way, testability gives the organization a practical way to meet test coverage goals and preserve confidence as speed increases.

Table 6.1 Different validation levels and their purposes

Validation level	Purpose
Unit tests	Provide fast feedback on code logic.
Contract tests	Protect service expectations between components.
Feature tests	Validate the feature works as expected.
Integration tests	Validate important paths across components.
Non-functional tests, including performance and resilience tests	Expose scaling and failure behaviour.
Regression tests	Validate existing capabilities have not been broken.
Security tests	Validate assumptions about identity, data, dependencies, and policy.
User acceptance testing	Validate the software change is acceptable to users.
Production validation	Confirm behaviour under real operating conditions.

Practical design questions

- Can a team test the component in isolation?
- Can it validate contracts with consumers and providers?
- Can the test data be rebuilt?
- Can a failed test be explained quickly?
- Can AI-generated artifacts pass through the same validation pathway as human-authored work?

These questions reveal testability better than a percentage on a coverage dashboard.

KEY MESSAGE



6.1.3 Designing for deployability: decoupling release from deployment

Deployability is an architectural quality that must be designed into the system through architecture, data strategy, runtime behaviour, and operational controls. A team cannot

achieve low-risk release if every deployment requires a large coordination event, manual configuration, a shared outage window, or synchronized change across many systems. The pipeline may be automated, but the release still behaves like a fragile ceremony (Humble & Farley, 2010; Nygard, 2018).

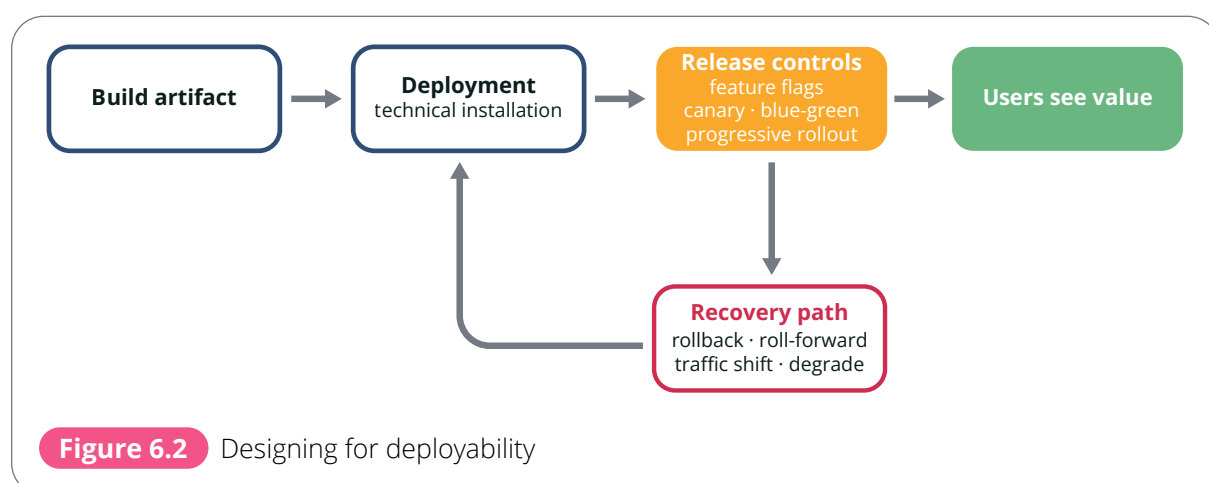
Small, reversible, independently deployable changes are easier to understand. They reduce the amount of new behaviour introduced at one time and make recovery decisions clearer.

Designing for deployability requires support for:

- backward compatibility
- versioned interfaces
- safe data migration
- configuration management
- traffic control.

These capabilities turn deployment into routine engineering work.

One of the most useful design disciplines is separating deployment from release. Deployment moves code, configuration, models, or infrastructure into an environment. Release exposes value or behaviour to users. A system that supports feature flags, entitlement controls, canary exposure, blue-green deployment, and progressive rollout allows teams to deploy safely while controlling when and how users experience the change. Figure 6.2 illustrates this distinction by showing how deployment, release controls, user exposure, and recovery work together to reduce delivery risk.



More broadly, deployability is the result of design choices such as small changes, independent deployment, feature flags, backward compatibility, progressive exposure, rollback, and recovery.

Every meaningful change should have a recovery path. In some cases, recovery means rollback; in other cases, it means rolling forward, disabling through a feature flag, shifting traffic, repairing data, or operating a degraded mode while the team resolves the issue. For this reason, the design must make those options available before the release is under pressure.

Good deployability helps governance as well as speed. When evidence is complete, low-risk changes can move through automated controls. By contrast, higher-risk changes can trigger human review with the relevant context already assembled. Consequently, the organization gains control through repeatable design rather than through late meetings and spreadsheet approvals.

Deployment readiness questions

- Can this change be deployed independently?
- Can it be hidden from users if needed?
- Can traffic be limited?
- Can the change be reversed or disabled quickly?
- Can the team see the effect on production?

A system that answers yes to these questions has the foundation for safer release.

KEY MESSAGE



AI-native delivery raises the cost of weak architecture. AI-assisted work increases the volume and variety of changes entering the delivery system, so designs need clear boundaries, approved patterns, and systems that remain observable, testable, secure, deployable, and operable.

AI-enabled systems extend observability requirements. A model, prompt, retrieval index, or agent workflow can be degraded without a code deployment. Observability therefore needs relevant signals about inputs, prompt and model versions, retrieval quality, confidence, safety outcomes, cost, latency, and human escalation so that teams can distinguish model drift, configuration changes, and software defects.

AI-native delivery also expands testing requirements. AI-generated code may look plausible and still violate hidden design assumptions, while AI-generated tests may increase coverage without improving confidence. AI-enabled product behaviour may also be probabilistic, so the design should define how deterministic and probabilistic behaviour will be tested, including appropriate ranges, thresholds, confidence, drift, bias, and unacceptable variation.

Finally, AI-native delivery expands what must be treated as a controlled deployment unit. Prompt templates, model versions, retrieval indexes, safety policies, routing rules, and tool permissions should be versioned, monitored, and reversible because each can change production behaviour.

AI-RELATED CONTENT



6.2 Design enablers and governed constraints

Architectural qualities enable effective DevOps. Those qualities become sustainable only when teams can apply them consistently. Architecture creates the conditions for change, while platforms, AI validation, and technical-debt governance provide the reusable enablers and visible constraints that help teams put those design decisions into practice. Platforms reduce repeated work, AI validation requirements define acceptable evidence, and technical-debt governance keeps known weaknesses from becoming hidden delivery risk. Together, these practices make safe design easier to use and harder to bypass.

6.2.1 Platform thinking and internal developer platforms

Chapter 9 provides the canonical definition and operating practices for Platform Engineering and internal developer platforms. The narrower purpose of this design section is to identify what a product or service must expose to platform capabilities before implementation begins (CNCF TAG App Delivery, 2023a; Fournier & Nowland, 2024; Skelton & Pais, 2025).

Design should explicitly define the platform contracts that every service is expected to consume:

- How the service will be built and deployed.
- Which infrastructure modules the service will consume.
- How identity and secrets will be supplied.
- What telemetry will be emitted.
- Which policy checks apply.
- How ownership will be registered.
- What recovery path is required.

AI-enabled services add requirements that must be designed into the architecture rather than improvised by each team. As explained in Chapter 3, controls for model access, versioning, evaluation, data handling, provenance, cost, fallback behaviour, and escalation should become reusable, platform-consumable contracts.

**AI-RELATED
CONTENT**

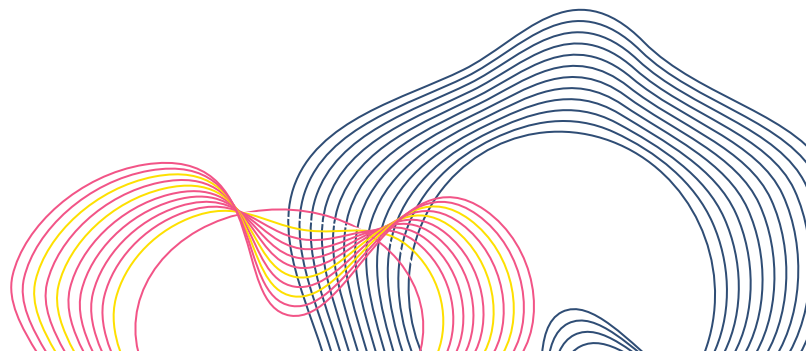


These decisions reduce late integration work and make the safe path reusable.

Platform thinking turns repeated engineering work into reusable internal capabilities. In doing so, Platform Engineering reinforces the operating environment described in Chapter 5 by making the safest and most effective engineering practices the easiest ones to adopt. The design test is whether teams can use those capabilities without losing ownership of the services they build and operate.

A team should also identify where the standard path is insufficient. An exception may be justified by workload characteristics, regulation, specialized infrastructure, or a genuine product constraint. The exception should have a named owner, evidence, an expiry or review point, and a path back to a supported platform capability where practical.

The platform does not own the service outcome. Product and service teams remain accountable for operability, security, reliability, and value. Platform capabilities reduce repeated work and cognitive load while preserving that ownership.



6.2.2 Designing for AI validation

AI validation is a design concern, not an operational afterthought. When AI is used to generate code, recommend changes, classify risk, write tests, summarize incidents, make predictions, or take operational action, the delivery system needs explicit validation rules. Those rules should exist before the team discovers that generated output has crossed a security, architectural, ethical, or compliance boundary (Autio et al., 2024; NIST, 2024). Validation should be proportionate to risk. A draft documentation summary may require only a light review, whereas a generated infrastructure policy, security rule, database migration, or autonomous remediation action demands stronger evidence and clearly defined approval thresholds. Those thresholds should be established during design, so teams are not forced to invent governance in the middle of an incident.

AI-enabled product behaviour also needs design-time validation. Recommendation models, fraud classifiers, service chatbots, and agentic workflows can behave differently across users, regions, time periods, inputs, and policy contexts. The design should therefore define how these systems will be evaluated, what acceptance criteria they must meet, when human review is required, which monitoring signals will be collected, how recovery will be handled, and how exceptions will be managed. These are core design decisions that enable safe and reliable delivery, not issues to be deferred to a governance committee after implementation.

Explainability is part of validation. The organization should be able to understand why an AI-supported decision was made at a level appropriate to the risk. A low-risk developer suggestion may require only ordinary code review. However, a customer-impacting decision may require reason codes, audit records, model version history, data lineage, and evidence of human oversight.

Security requires equal attention. Prompt injection, data leakage, unsafe tool use, model tampering, and supply chain issues can enter the system through AI features and AI-assisted delivery workflows. Design should define trusted model sources, approved data use, prompt management, access control, logging, and escalation paths. AI speed is useful only when the surrounding control model can keep pace.

AI-generated output should enter the same governed delivery system as any other artifact. It needs provenance, review, automated checks, evidence, and ownership. The system should record what was generated, what context was used, which model or tool produced it, who accepted it, and which controls validated it. This is basic engineering hygiene in AI-native delivery.

AI-RELATED CONTENT



Design standard

AI-generated output is not production-ready by default. Production readiness is earned through validation, governance, monitoring, and accountable human review. The design should make this path normal rather than exceptional.

KEY MESSAGE



Case B | Harbor Digital Commerce (composite): designing a customer capability for safe change

Harbor planned a saved-payment capability for its checkout service. The first design added fields directly to a shared database, called the payment provider synchronously, and assumed that deployment and release would happen together. The design could be built quickly, but it made rollback difficult, coupled several teams, and provided little evidence about the customer path after release.

The design review changed the shape of the work. The team adopted an additive schema with a compatibility period, defined an API contract, isolated provider failure behind a resilient interface, and placed user exposure behind a feature flag. The service emitted correlation identifiers and customer-facing outcome signals, and the team defined rollback, roll-forward, and feature disablement paths before implementation.

An AI-assisted fraud score was treated as a separate version of behaviour. The design included an approved model reference, evaluation data, confidence thresholds, explanation requirements, human fallback, cost telemetry, and monitoring for drift. These requirements entered the architecture decision and backlog rather than being added after the feature reached production.

The redesign required more effort before implementation, but it reduced coordination, simplified recovery, and enabled safer, incremental delivery once development began. It also allowed code, database, provider integration, and AI changes to be deployed independently and released gradually. The team exchanged apparent early speed for lower coupling and safer learning.

Transferable lesson: Design for DevOps means making validation, observability, security, deployability, recovery, and AI control properties explicit while important choices are still inexpensive to change.

SCENARIO



6.2.3 Technical debt as a DevOps risk

Technical debt is a DevOps risk because it slows safe change, hides defects, increases security exposure, makes recovery harder, weakens test confidence, and raises the cost of learning. Some debt is a deliberate trade-off, but unmanaged debt eventually becomes a constraint that shapes every future delivery conversation (Cunningham, 1992; Kruchten et al., 2012).

Debt appears in many forms. Architecture debt creates coupling and unclear ownership. Test debt produces slow or untrusted feedback. Security debt leaves unresolved vulnerabilities and weak controls. Data debt creates brittle integrations, undocumented meaning, and difficult migration paths. Platform debt forces teams into repeated manual work. Operational debt appears as outdated runbooks, excessive toil, and fragile recovery procedures. Regardless of its form, technical debt follows a common pattern: it reduces delivery capability until it is identified, managed, and treated as an operational risk.

Debt should be made visible in terms that leaders can understand. It is not enough to say the system is messy. Teams should connect debt to delivery delay, defect escape, security risk, incident frequency, audit effort, cost, and lost product capacity. The more clearly debt is tied to operational and business impact, the easier it becomes to fund the work needed to reduce it.

Prioritization matters because teams cannot pay back every debt item at once. Good debt management separates annoying debt from constraint debt. Constraint debt limits flow, creates unacceptable risk, or prevents the organization from meeting a strategic goal. That debt belongs to the product and transformation backlog, with ownership, dates, and visible trade-offs.

Governance should keep debt from becoming invisible again. Architecture reviews, platform standards, security controls, observability requirements, and delivery metrics can all expose debt early. The goal is continuous debt management, not a heroic cleanup project after the system becomes painful to change.

AI-native systems also create architectural debt when prompts, models, datasets, generated code, or evaluation assets change without versioning, review, or control. Chapter 3 discusses the importance of provenance and validation; in design work, the concern is whether these assets remain maintainable, operable, secure, and fit for future change.

AI-RELATED CONTENT



Practical debt categories

Classify debt by its effect on flow, quality, security, reliability, operability, cost, and AI governance. Then ask which items are blocking safe change. That answer is usually more useful than a large backlog of unlabelled cleanup tasks.

TIP



DECISION AND ACTION GUIDE

Executive decision: The technology executive accountable for architecture requires designs to address testability, observability, security, deployability, operability, recovery, and AI validation, and funds the technical-debt work needed to meet those expectations.

Role-specific actions: Architects, product owners, engineers, SREs, testers, and security specialists document quality attributes, contracts, failure modes, and validation responsibilities before implementation.

Evidence to review: Architecture decision records, non-functional requirements, interface contracts, recovery tests, telemetry acceptance criteria, and a prioritized technical-debt backlog.

6.3 Chapter summary

Design for DevOps is a practical discipline. It asks teams to treat operability, testability, deployability, security, governance, and observability as normal design concerns. These concerns do not slow good engineering; they reduce expensive surprises later. The distinction is important.

The strongest design decisions are often modest. A clear service boundary, a stable contract, a useful trace identifier, a versioned prompt, a reversible migration, or a platform template can remove weeks of coordination later.

Weak decisions compound in the opposite direction. They accumulate into delays, fragile releases, difficult audits, and incidents that take too long to understand.

AI-native delivery makes this chapter even more important. As AI assists with code, tests, documentation, configuration, risk analysis, and operations, the surrounding system must express intent clearly. Architecture, validation rules, telemetry, and guardrails become the shared language that enables both people and automated systems to operate safely within defined boundaries. Design is a primary enabler of flow, resilience, and continuous improvement. In summary:

- DevOps capability is enabled or constrained by design decisions made long before the pipeline runs.
- Architecture should be evaluated through ownership, testability, deployability, observability, security, recovery, and governance.
- Observability, testability, and deployability are design qualities that reduce operational risk and improve feedback speed.
- Internal developer platforms help scale good design by reducing cognitive load and making safe patterns easier to use.
- AI-native delivery requires validation, provenance, monitoring, and human accountability to be designed into the workflow.
- Technical debt should be governed as a delivery and operational risk rather than handled as an occasional cleanup task.

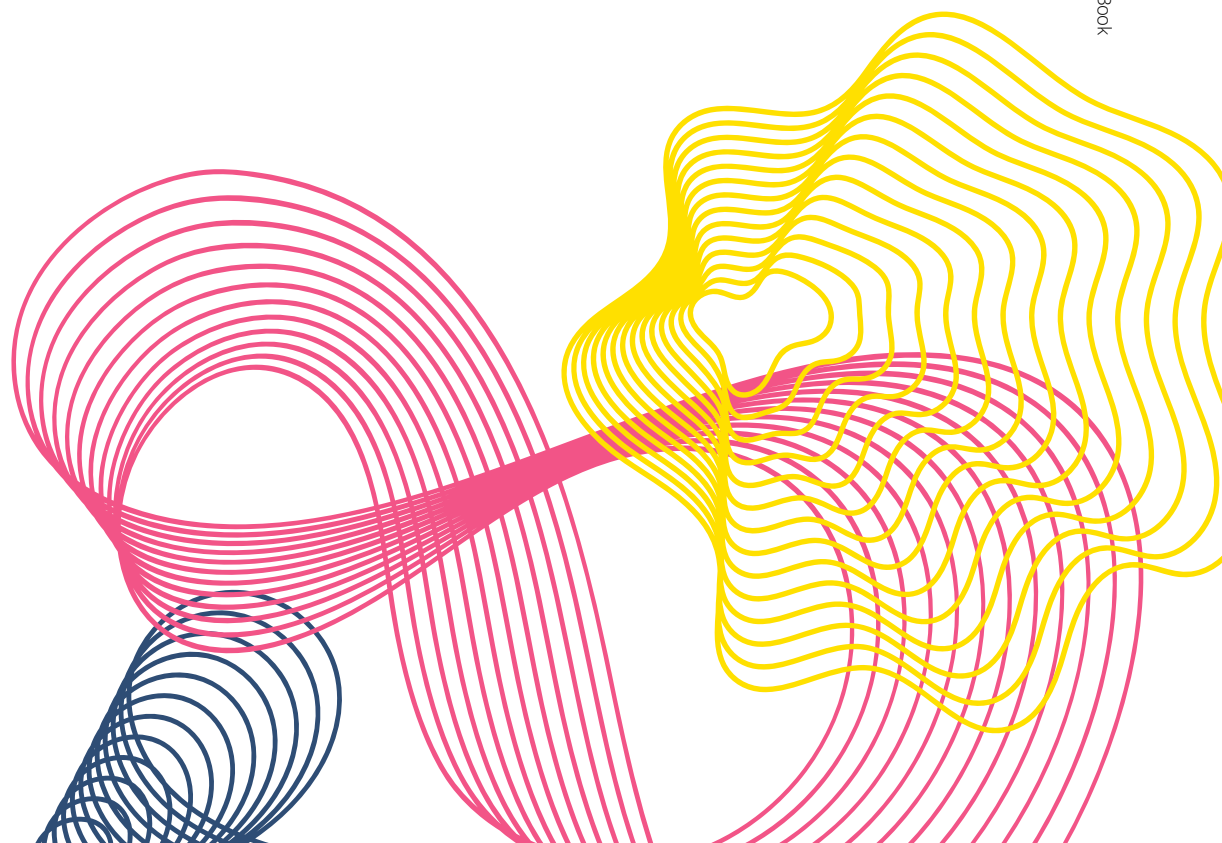
Adding observability after the system is built is costly. The strongest systems are designed to explain themselves in production.

Design review checklist

The following questions can be used in design review, architecture review, or readiness discussions. The questions are deliberately straightforward. A mature organization should be able to answer them without heroic investigation.

Table 6.2 Design review checklist

Design area	Question	Evidence to look for
Architecture	Can the owning team change the service without excessive coordination?	Boundaries, contracts, owned data, dependency map
Observability	Can the team explain production behaviour and user impact?	Logs, metrics, traces, events, release metadata
Testability	Can feedback be obtained quickly and trusted by engineers?	Automated tests, test data, environments, failure clarity
Deployability	Can the change be released gradually and recovered safely?	Flags, rollout controls, rollback or disablement path
Platform	Is the safest path easy for teams to use?	Golden paths, templates, self-service, support model
AI validation	Are AI-generated and AI-enabled changes governed by risk?	Provenance, checks, thresholds, audit trail, monitoring
Technical debt	Which debt items constrain safe change or create unacceptable risk?	Risk register, backlog, ownership, funded remediation



This chapter defines the delivery engine. Continuous Integration and Continuous Testing are treated here as operating disciplines that determine flow, quality, risk, and confidence in change. They allow teams to integrate change frequently, validate it continuously, and keep confidence intact as delivery speed increases. This becomes more important when AI-generated artifacts enter the same delivery stream as human-authored work (Hornbeek, 2024).

This chapter is most relevant to practitioners, engineering leaders, test leaders, platform teams, security teams, and anyone responsible for keeping the mainline healthy. Leaders should read it as an explanation of the conditions teams need to move quickly without sacrificing control.

The delivery engine of DevOps is often mistaken for a set of pipeline tools. That view is too narrow. Delivery pipelines matter significantly: they compile code, run tests, scan dependencies, package artifacts, and produce evidence. A pipeline cannot compensate for poor integration habits, long-lived branches, shallow tests, weak ownership, or unclear quality standards.

Continuous Integration and Continuous Testing are separate Pillars because they represent distinct practice domains: integration keeps changes small, frequent, and compatible in a shared mainline, while testing provides evidence that those changes behave as intended. They are covered together in this chapter because, in practice, frequent integration depends on continuous validation, and continuous testing gains much of its value by providing rapid feedback on integrated change.

They shape how teams make changes, how often those changes meet other changes, how quickly evidence is produced, and how confidently a release candidate can move forward. The value is not in the number of jobs configured in a build system, but in the speed and quality of feedback.

AI changes the pressure on this part of the system. Generated code, generated tests, generated configuration, and generated documentation can increase the amount of material entering the pipeline. That additional volume is useful only when integration, testing, security, and review practices are strong enough to absorb it. Otherwise, the organization simply creates more work faster than it can validate it.

7.1 Continuous Integration as a practice, not a pipeline

Continuous Integration begins with the way a team behaves around change. The CI server is an automation mechanism. The real practice is the frequent movement of small, reviewed, and validated changes into a shared mainline. A healthy team integrates often enough that defects, conflicts, and misunderstandings remain small (Fowler, 2006; Humble & Farley, 2010).

Continuous Integration and Continuous Testing

In weak CI environments, work can appear calm for days and then fail suddenly when branches, configuration changes, test data changes, and dependency changes meet. The integration event becomes a discovery event. Teams then spend time untangling conflicts that could have been detected while the work was still fresh. This is one of the oldest failures in software engineering, and the tools have never fully solved it.

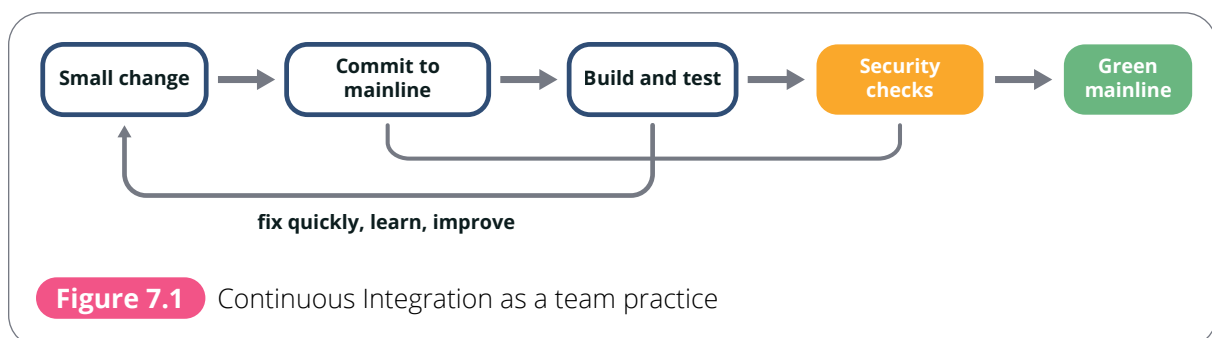
Mature CI practice has a few recognizable habits. Changes are small. Developers commit or merge frequently. Automated build, test, and security checks run on every meaningful change. The mainline remains in a known state. When the mainline breaks, the team treats it as shared work, restores it quickly, learns from the failure, and improves the integration process before continuing. Figure 7.1 illustrates how frequent integration, automated validation, and shared ownership work together to maintain a healthy (green) mainline: developers make small commits that integrate frequently into a shared mainline or trunk. Automated validation provides fast feedback through build, test, quality, security, and gate checks, while team behaviours such as shared ownership and continuous improvement support its reliability over time.

Small changes matter because they reduce the amount of unknown material entering the system. A one-line change can still be dangerous, but it is easier to reason about it than a large bundle of unrelated work. Small batches also improve review quality. Reviewers can examine intent, architecture, security, and test adequacy instead of scanning a large diff and hoping the important problems are visible.

Fast feedback matters because slow feedback changes behaviour:

- When builds take too long, developers stop waiting for the signal.
- When tests are unreliable, teams stop trusting the signal.
- When failures are hard to diagnose, teams work around the CI system.

CI becomes effective only when the feedback is fast enough and clear enough to influence decisions during the workday. The same CI discipline applies to AI-generated artifacts, which must enter the shared mainline through the same integration and validation process as human-authored work. Chapter 3 provides the broader requirements for AI validation, evidence, and accountability.



Operating guidance

Treat a broken mainline as a stop-the-line signal. Fix it before adding more work. A team that tolerates a red build is choosing uncertainty as its working state.

TIP



Healthy CI also requires stewardship of the delivery pipeline itself. Build time, test stability, failure diagnosis, and dependency behaviour should be treated as team concerns. When the pipeline becomes slow or noisy, teams begin to work around it. A CI system that people avoid has stopped serving its purpose.

Teams should review CI performance as part of normal improvement work. Useful questions include how long feedback takes, how often builds fail for environmental reasons, how often tests are quarantined, and how quickly a broken mainline is restored. These questions are operational, not administrative. They show whether the delivery engine is healthy enough to support faster flow.

7.1.1 Trunk-based development and feature branching: trade-offs

Branching strategy determines how integration happens in practice and, therefore, how quickly teams receive feedback. A team can have an impressive pipeline but still have weak CI if most work stays isolated for long periods. Integration frequency is a behaviour. The branch model either supports that behaviour or makes it harder (Humble & Farley, 2010; DORA, 2024).

Trunk-based branching keeps the shared mainline close to the current state of work.

Developers integrate small changes frequently, often using feature flags or other release controls to keep incomplete functionality hidden from users. This pattern supports fast feedback because each change meets the system early.

Feature branches can be useful. They support isolation, review, and experimentation. They become risky when they are long-lived, large, or treated as private workspaces. A branch that lives for weeks is a deferred integration problem. The team may feel productive while the branch grows, but risk is accumulating out of sight.

The issue is not whether feature branches exist, but how long work remains isolated from the shared mainline. The practical guidance is straightforward:

- Keep branches short-lived.
- Keep changes small.
- Use pull requests as a review mechanism, not as a storage area for unfinished design.
- Use feature flags when code must be integrated before functionality is exposed.
- Delete stale branches.
- Track branch age as an operational signal, especially in teams that struggle with late merge conflicts.

Some organizations cannot move immediately to pure trunk-based development. Large legacy systems, regulatory evidence requirements, or tightly coupled release trains may require an intermediate pattern. In those cases, the goal should be to shorten branch life, reduce batch size, increase automated validation, and move the integration point earlier. The branch model should evolve towards faster feedback rather than become an excuse for delayed learning. In fact, AI-assisted development makes frequent integration even more important: generated work may make a branch look ready before the system has proved it through build, test, dependency, and security checks.

Branch metrics should be interpreted carefully. For example, branch age, merge conflict frequency, pull request size, and review latency can expose integration problems. However, these measures should guide improvement, not punish developers.

Practical check

Ask how old the oldest active branch is. Then ask whether anyone would be surprised if it failed to merge cleanly. The answer usually tells you more than the branch policy document.

TIP



Therefore, the useful conversation is about why work is staying isolated and what the system needs to integrate earlier.

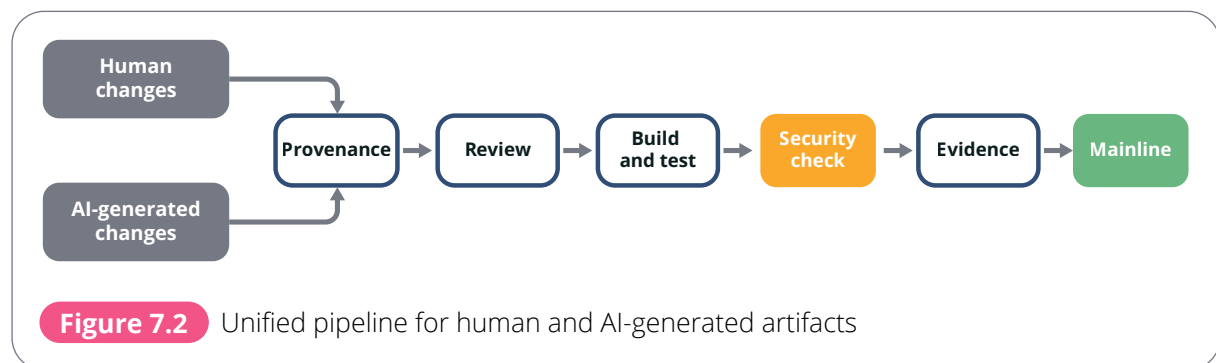
Release branches are a special case. They may still have a place in some environments, particularly where packaged software, regulated deployment windows, or long-lived customer versions are involved. Even in these environments, the same principles apply. In fact, release branches are most effective when they remain short-lived, rely on automated validation, have a clearly defined purpose, and do not evolve into parallel product lines.

7.1.2 Integrating human and AI-generated artifacts

Modern delivery systems now receive artifacts from multiple sources. People author code, tests, configuration, documentation, and infrastructure definitions while AI systems may generate or modify the same kinds of artifacts. Regardless of their origin, all changes should enter the same governed delivery pipeline. The pipeline must handle both sources without creating a second, weaker path for AI-assisted work (NIST, 2024; DORA, 2025).

An AI-generated artifact is not production-ready simply because it was generated. It becomes trustworthy only after meeting the same evidence requirements as any human-authored change. These include provenance, review, build and test (automated validation), security and policy checks, and evidence (traceability). Figure 7.2 depicts this unified approach.

Provenance is important. Teams should know whether an artifact was written by a person, generated by an AI system, or produced through a combination of both. They should also know the prompt or source context when that information is relevant and safe to retain.



Control point

Keep AI-generated work inside the normal delivery evidence trail. If the organization cannot explain where an artifact came from, it will struggle to explain why it should be trusted.

TIP



This does not require heavy ceremonies, but enough traceability to understand how a change was produced and who accepted responsibility for it.

Unified pipelines prevent shadow delivery. The latter occurs when generated artifacts are copied into systems, scripts, environments, or documentation without normal controls. It often starts with convenience. Someone uses AI to generate a deployment script or configuration change and applies it directly because the work seems small. Small, unreviewed changes can still create production risk.

The pipeline should therefore be designed as the common intake point. It should accept human-authored and AI-generated changes, attach context, run the required checks, produce evidence, and reject work that cannot be validated. This approach keeps the system understandable while ensuring accountability remains with the team rather than being pushed onto the tool that generated the artifact.

A unified delivery workflow does not need to be slow. Most controls should be automated, fast, and risk based. Low-risk changes can move the pipeline through quickly when the evidence is clean, whereas higher-risk changes can trigger deeper review. The goal is disciplined flow, not a pile of approval gates.

The metadata model should be practical. At a minimum, teams should capture the author or approver, artifact source, tool or model class

when relevant, change purpose, review status, test evidence, and approval state. However, sensitive prompts, customer data, and proprietary context may need to be excluded or redacted. In this way, traceability should strengthen control without creating new privacy or intellectual property risk.

A unified pipeline also supports learning. When a generated change fails, the team can see where it failed and why. That evidence can improve prompts, coding standards, test design, platform templates, and review guidance. Without a common pipeline, these lessons remain scattered across chat histories, local scripts, and private workstations instead of becoming shared organizational knowledge.

7.2 Continuous Testing: quality as a system property, not a gate

Continuous Testing is the practice of creating useful quality feedback throughout the delivery system. It is broader than test automation. Test automation is one mechanism. Continuous Testing includes test strategy, test data, environment quality, security validation, production monitoring, defect learning, and the way teams interpret evidence (Humble & Farley, 2010; Hornbeek, 2024).

Late testing is expensive because it finds problems after many decisions have already been made. The defect may be in code, design, data, configuration, dependency, or a misunderstood requirement. By the time a late test finds it, the team must reopen context, retest neighbouring behaviour, and often renegotiate delivery plans. The rework is visible, but the lost learning time is usually larger.

Quality should be treated as a property of the whole system. Design decisions affect testability, while branching strategy affects integration confidence. Infrastructure affects environmental reliability, while security controls affect whether a change can be released safely. Moreover, observability affects whether a team can detect quality issues after deployment. A test phase at the end cannot compensate for weak quality conditions upstream. Quality emerges from the combined effect of design, development, testing, deployment, operation, and learning rather than from a single testing phase. Figure 7.3 illustrates how quality is continuously created across the delivery lifecycle through validation and feedback at every stage. Design, code, review, automated tests, environments, pipeline evidence, observability, and production feedback all contribute to the quality signal.

Continuous Testing works best when every level of validation has a clear purpose. Unit tests provide rapid feedback that small pieces of logic behave as intended. Contract tests protect service boundaries before integration becomes expensive. Integration tests validate important interactions, while performance and resilience testing expose risk before users do. Production validation then confirms that the service remains healthy after release. On top of that, AI-enabled systems broaden the quality model because generated artifacts and probabilistic behaviour require forms of validation beyond conventional deterministic testing.

Engineering reminder

A test that no one trusts is operational noise. A smaller set of reliable, relevant tests will usually improve flow more than a large suite that fails unpredictably.

KEY MESSAGE



Teams should avoid treating testing as a gate owned by someone else. The team that changes the system needs access to quality feedback while the change is still small. Test specialists remain important, especially strategy, risk analysis, exploratory testing, automation design, and quality coaching. Their value increases when they help the whole system produce better evidence earlier.

The Definition of Done should include evidence, not just completed tasks. Test results, security findings, deployment readiness evidence, observability updates where appropriate, and a clear assessment of risk provide the confidence needed to move a change forward. This is particularly important for shared services, where one change may affect many teams.

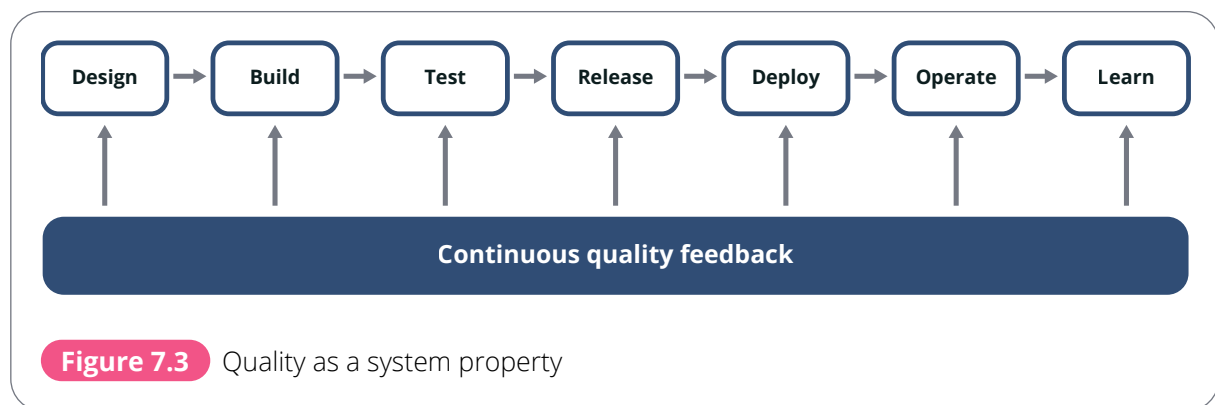


Figure 7.3 Quality as a system property

Ultimately, Continuous Testing depends on clean test data and reliable environments. Many teams blame automation for failures that are caused by stale data, unstable dependencies, or inconsistent configuration. Test reliability is an engineering problem. Reliable test environments are as important as reliable production environments because both determine how quickly and confidently teams can learn from change.

7.2.1 Deterministic and probabilistic testing: what changes when AI is in the loop

Traditional software testing assumes that software behaves deterministically. Given the same input and the same system state, the same output should occur. Much of the delivery pipeline is built on this assumption; a function returns the expected result, an API produces the expected response, and a configuration policy passes or fails. This model remains essential for most software components (Autio et al., 2024; NIST, 2024).

The distinction is straightforward. Deterministic testing confirms that software behaves as designed, while probabilistic testing evaluates whether AI-enabled behaviour remains acceptable under defined conditions and continues to perform safely in production.

This does not mean AI testing should be vague. Acceptance criteria still need to be explicit. On that account, teams should define what level of variation is acceptable, which errors are unacceptable, which populations require special attention, and what thresholds trigger human review. They should also define how the system will be monitored after release because offline evaluation rarely captures the full behaviour of production use.

A practical example is a support chatbot connected to a knowledge base. Deterministic checks can confirm that the service starts, routes requests, authenticates users, and logs interactions correctly. Probabilistic checks evaluate whether:

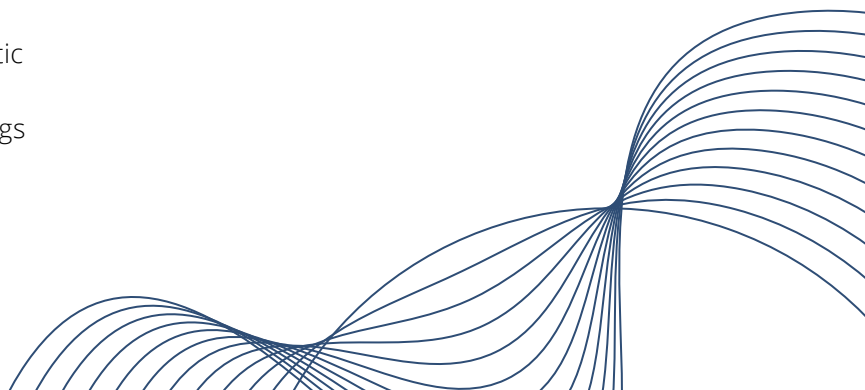
AI-enabled systems introduce different kinds of behaviour. Their outputs may be probabilistic, context-sensitive, or dependent on changing data, so the same input may produce acceptable variation rather than one fixed answer. Testing must therefore use broader evidence than a single expected output. Teams may evaluate distributions, confidence thresholds, drift, bias, edge cases, curated datasets, and human review samples. Pass-or-fail decisions still matter, but they are supported by a richer body of evidence.

AI-RELATED CONTENT



- Answers are grounded in approved or reliable sources.
- Citations are relevant to the answer.
- Safety rules are followed.
- Refusals are appropriate.
- Escalation occurs when confidence is low.

The objective is not to replace one testing model with another, but to apply each to where it is most appropriate. Components that should behave deterministically require deterministic validation, while AI-enabled components require probabilistic evaluation. Nevertheless, treating everything as probabilistic weakens control. In addition, treating AI behaviour as deterministic creates false assurance. Therefore, mature teams combine both approaches through clearly defined release criteria.



AI-native standard

AI behaviour must be evaluated before release and watched after release. Drift, prompt changes, data changes, and user behaviour can all change system quality without a traditional code defect.

**KEY
MESSAGE**



Evaluation datasets should be managed like important test assets. They require versioning, ownership, representativeness, and review. A stale evaluation set can make a model look stable long after real-world behaviour has moved on. For high-risk systems, evaluation should also include known failure modes, edge cases, and historically under-served user groups.

Production monitoring completes the test model. A release can pass offline evaluation and still fail when user behaviour changes, data distribution shifts, or retrieval content becomes stale. For this reason, teams should define post-release signals before deployment, including quality metrics, drift indicators, escalation thresholds, and human review paths where appropriate.

7.2.2 Test strategy across the pipeline

A strong test strategy decides which evidence belongs at each point in the pipeline. The usual mistake is to push every concern into one test suite or one stage. This creates slow feedback, unclear failures, and expensive maintenance. A better approach uses layered tests with different purposes, costs, and response times.

Each layer of testing serves a different purpose. Specifically, unit tests belong close to the developer. They provide fast feedback on small pieces of logic and support safe refactoring. Furthermore, component tests validate larger units of behaviour without requiring the full system. Accordingly, contract tests protect service boundaries by validating expectations between providers and consumers, while integration tests confirm that important parts of the system work together correctly.

Performance tests, resilience tests, security tests, and compliance checks often need different timing. Some should run early and often in lightweight form, whereas others should run on a schedule or before higher-risk releases because they are expensive or require production-like environments. The point is to match test depth to risk without making every change wait for every possible validation.

Security testing belongs in the test strategy, not outside it. Dependency scans, secrets detection, infrastructure checks, container scans, and policy validation should be part of the same evidence model that governs build and release. These checks need to be tuned so that they produce useful signals rather than long lists of unprioritized findings. The pipeline should therefore combine speed and depth, since fast checks provide immediate feedback on most changes and deeper checks provide confidence in larger or riskier changes. Production validation closes the loop by confirming that the system behaves correctly under real conditions. The strategy should be reviewed regularly because the system, risk profile, and team capability all change over time.

Design rule

Put the fastest reliable signal as close as possible to the point where the defect is created. Put deeper validation where it adds confidence rather than routine delay.

TIP



The familiar test pyramid remains useful as a thinking tool, provided teams do not treat it as a rigid formula. Some systems need more contract testing, whereas others need stronger simulation. Other systems need more production validation because pre-

production environments cannot reproduce real conditions. The right shape of the test strategy follows risk and architecture.

Very often, retiring weak tests is as important as adding new ones. For this reason, test suites should be curated rather than allowed to grow unchecked. Slow, redundant, flaky, or misleading tests consume attention and, over time, reduce trust in the evidence the suite provides. Periodic cleanup is therefore a quality practice: it keeps the evidence readable enough for teams to respond quickly when meaningful failures occur.

Table 7.1 illustrates how different types of testing contribute evidence at different stages of the delivery pipeline. Although the exact balance varies by architecture, risk, and delivery model, every layer has a distinct purpose; fast checks run early, whereas deeper checks run where risk and exposure warrant.

Table 7.1 Layered test strategy across the pipeline

Test layer	Local	CI	Pre-release	Production
Unit	Primary; fast	Required	Selective	Not applicable
API contract	Fast checks	Primary	Targeted	Synthetic where useful
Functional	Primary	Targeted	Limited	Limited
Integration	Limited	Integration	Deep, risk-based	Not applicable
Regression	Targeted	Limited	Extensive	Not needed
Security	Local scans	Full scans	Deep risk-based testing	Continuous signals
Non-Functional	Developer feedback	Targeted	Primary	Capacity and latency signals
User Acceptance	Limited	Limited	Primary	Limited
Smoke and health	Optional	Targeted	Deployment rehearsal	Primary after deployment
AI/probabilistic behaviour	Local evaluation	Automated evaluation	Representative data and human review	Drift, quality, and safety monitoring



AI-generated artifacts should be addressed through the test strategy as evidence-bearing delivery work, not as a separate checklist. Generated code should be tested like any other code, but the test plan should also account for why the AI output might be wrong. The test strategy should consider whether prompts provide sufficient context, generated tests genuinely validate behaviour rather than restate the implementation, generated configurations follow approved patterns, and changes introduce licensing, data handling, or security risks.

Detailed guidance on AI provenance, context, validation, risk, and accountability is provided in Chapter 3.

AI-RELATED CONTENT



7.2.3 Shift-left quality: moving validation earlier without slowing delivery

Shift-left quality means moving useful feedback closer to the point where work is created. It should not mean dumping more responsibility onto developers without support. When shift-left is done poorly, teams receive more tools, more alerts, and more obligations, while the underlying system remains difficult to change. That creates frustration and slower flow (NIST, 2022; Hornbeek, 2024).

Effective shift-left practices give teams feedback they can act on immediately. Local checks should be fast, standards should be clear, secure defaults should be available

through the platform, and test data should be easy to provision. Additionally, policy failures should explain what needs to change, and developers should not need to become experts in testing, security, compliance, and infrastructure simply to make a routine change safely.

Platform capabilities often determine whether shift-left succeeds or becomes an additional burden. Pipeline templates, local test harnesses, approved base images, and infrastructure modules, along with secrets management, observability defaults, and security policies allow teams to receive meaningful feedback earlier without inventing the supporting machinery themselves.

Earlier validation also changes the economics of defects. A defect found during local development is usually cheap to fix. By contrast, a defect found after integration costs more because other work may depend on it. Ultimately, a defect found after release can cost customer trust, incident response time, audit effort, and product confidence. The earlier signal is rarely perfect, but it often prevents more expensive discoveries later. The goal of shift-left is therefore to move useful validation earlier without replacing the need for production learning. Early feedback reduces the cost of change, while later feedback confirms how the system behaves under real operating conditions.

Shift-left should be balanced with shift-right learning. Some system qualities can only be confirmed in production. Real user behaviour, real traffic patterns, real data distributions, and real dependency failures often reveal issues that pre-production environments miss. Mature teams combine early validation with production monitoring, canary analysis, feature flags, SLOs, and incident learning.

Shift-left also needs clear routing for problems. If a security scan fails, the developer should know whether to fix the issue, request an exception, update a dependency, or consult a specialist. If a policy check fails, the message should explain the rules and the remediation path. A vague failure creates delay and teaches people to distrust control.

Practical caution

If shift-left increases local cognitive load faster than it improves feedback, the system will resist it. Improve the platform and the workflow before blaming the team.

Shift-left succeeds only when feedback is fast, relevant, and actionable. Moving a slow or noisy gate earlier does not improve delivery. It simply moves the bottleneck closer to the developer.

TIP



Effective shift-left systems remain quiet when work is healthy and precise when risk increases. They avoid overwhelming developers with unnecessary warnings. Instead they highlight the issues that matter for the change at hand, preserve evidence for later review, and allow deeper checks to run when the risk justifies the cost.

7.2.4 AI-assisted test generation: capability, limits, and governance

AI can help teams produce tests faster. It can suggest unit tests, identify edge cases, create sample data, draft regression scenarios, propose contract tests, and summarize expected behaviour from existing code. If used well, it reduces much of the routine effort that has historically left systems undertested (Autio et al., 2024; Hornbeek, 2024; NIST, 2024).

The risk is that AI-generated tests can look convincing while adding little confidence. An AI-generated test may follow the

implementation too closely, assert the wrong behaviour, miss the important boundary condition, or duplicate tests that already exist. It may improve the coverage percentage while doing little to reduce production risk. The underlying testing problem is familiar; AI simply accelerates both good and bad testing practices. Teams should treat AI-generated tests as candidate tests. They require review for relevance, correctness, maintainability, and risk coverage. Review should focus on the following three questions: What behaviour does the test protect? Which defect would it detect? Does the assertion provide meaningful evidence? Tests that cannot answer those questions should not become part of the permanent suite simply because they were AI-generated.

Governance for AI-assisted test generation should be light enough to preserve usefulness and clear enough to prevent sloppy practice. Teams should know which AI tools are approved, what data can be shared, how AI-generated tests are reviewed, who accepts responsibility, and what evidence is retained. Sensitive code, customer data, credentials, and proprietary designs need explicit handling rules.

The best use of AI in testing is often collaborative. A tester or developer asks for candidate scenarios, refines the output, adds missing risk context, and turns the useful pieces into maintainable tests. The AI system accelerates discovery, while the human team applies judgement. At the same time, the pipeline provides evidence, and production feedback then shows whether the test strategy is improving real quality.

AI can also help maintain the test suite. It can identify redundant tests, explain flaky failures, suggest missing assertions, and summarize test failures for faster diagnosis. However, those uses still require validation because the test suite is an engineering asset. For this reason, it needs design, ownership, maintenance, and periodic cleanup, like any other asset.

AI-generated tests should address risk as well as code structure. The quality of AI-generated tests depends heavily on the quality of the

prompt. Effective prompts describe expected behaviour, failure modes, boundary conditions, security concerns, and user impact rather than simply requesting tests for a function. Narrow prompts usually produce equally narrow tests. Teams should monitor whether AI-assisted test generation improves outcomes. Useful indicators include defects caught earlier, reduced regression effort, improved edge-case coverage, fewer flaky tests, and lower defect escape rates. If the only visible improvement is a higher coverage percentage, teams should slow down and reassess the practice.

Governance test

If an AI-generated test fails six months later, the team should still understand why it exists, what behaviour it protects, and who owns its maintenance.

KEY MESSAGE



Case B | Harbor Digital Commerce (composite): integrating human and AI-generated tests

The saved-payment change from Chapter 6 entered Harbor's shared mainline in small increments. An AI assistant generated a large set of unit tests and several data-migration tests. The volume looked impressive, but reviewers found duplicated assertions, weak negative cases, and tests that repeated the implementation rather than proving intended behaviour.

The team required every AI-generated test to state the risk or behaviour it covered. Human reviewers kept useful tests, rejected redundant ones, and added contract tests for the payment provider, compatibility tests for old and new schema versions, performance tests for peak checkout demand, and security tests for sensitive payment data.

A generated happy-path migration test passed while a manually designed partial-failure scenario exposed a rollback defect. The test strategy was changed so database migration, feature-flag behaviour, and recovery were validated together in a production-like environment. Provenance and review evidence for AI-generated tests remained in the same continuous integration and continuous delivery (CI/CD) pipeline as human-authored tests.

Harbor finished with fewer AI-generated tests than the initial assistant proposed, but stronger confidence about behaviour, integration, and recovery. Test count went down while the quality of evidence went up.

Transferable lesson: AI can accelerate test creation, but risk-based test design, human judgement, layered validation, and production feedback determine whether the tests create confidence.

SCENARIO



The belief: This statement often appears when a team wants simple proof of quality. It is understandable. Code coverage is easy to display, compare, and celebrate.

The reality: Code coverage is easy to misunderstand. Code coverage shows that code was executed during tests. It does not prove that the right assertions were made, that important risks were covered, that edge cases were examined, or that the system will behave correctly in production. A team can reach a high coverage number with shallow tests. AI-generated tests can make this easier by creating large volumes of low-value checks. The dashboard improves while the real confidence of the team stays flat.

KEY MESSAGE



The damage from this anti-pattern is subtle. Teams begin to optimize for a percentage rather than confidence. Reviewers approve weak tests because the metric improves. Leaders assume quality has improved because the number is high. Defects still escape, and the team becomes puzzled because the dashboard promised something the tests did not actually prove.

The better pattern is confidence-based quality measurement. Coverage can remain one input, but it should be read with defect escape rate, test effectiveness, mutation testing where practical, flaky test rate, production incidents, security findings, and customer-impact data.

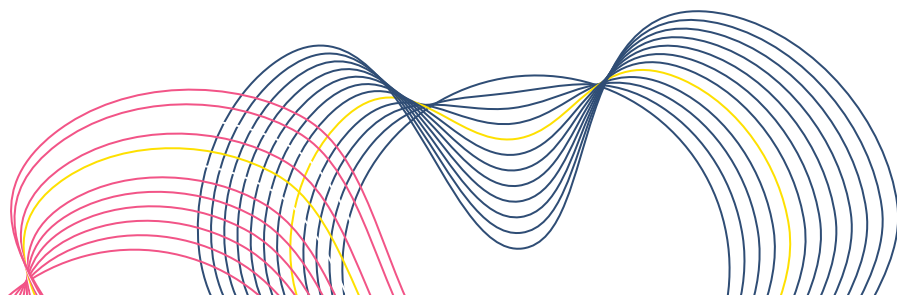
The question should be whether the test strategy helps the team make better release decisions.

A useful test suite has several qualities. It catches meaningful defects. It gives fast feedback. It is stable enough to trust. It is maintainable. It reflects real risk. It supports refactoring and change. It helps a new engineer understand expected behaviour. Those qualities do not fit neatly into a single percentage, which is precisely why quality work still requires engineering judgement. A stronger measure asks whether tests detect the kinds of defects the system suffers from. If production incidents repeatedly involve configuration errors, integration behaviour, performance degradation, or security findings, then unit coverage alone will not provide enough confidence. The test strategy should follow the defect history and the operational risk profile.

Leaders should be careful with quality targets. A target that sounds precise can distort behaviour when it becomes detached from engineering judgement. Coverage can be helpful when used by a team to find untested areas. It becomes harmful when used as a substitute for understanding what the tests are proving.

AI-generated outputs require meaningful validation. In this context, this means using AI to broaden test thinking, not simply to increase coverage numbers that may reproduce the implementation's assumptions.

AI-RELATED CONTENT



Corrective action

Keep coverage as a diagnostic signal. Pair it with outcome measures that show whether tests are catching real risk. Reward better confidence, not higher percentages.

TIP



DECISION AND ACTION GUIDE

Executive decision: The engineering executive sets expectations for frequent integration, risk-based testing, and acceptable feedback time, and funds dependable test infrastructure without imposing vanity targets.

Role-specific actions: Developers integrate small changes; test leaders maintain the risk-based strategy; platform teams keep pipelines fast and reliable; security and AI-assurance roles add proportionate checks.

Evidence to review: Integration delay, pipeline duration, flaky-test rate, escaped defects, coverage of material risks, and review evidence for AI-generated tests.

7.3 Chapter summary

Continuous Integration and Continuous Testing form the delivery engine of DevOps. Their value comes from disciplined engineering practices: frequent integration, meaningful validation, trusted evidence, and continuous

learning. Tooling supports those habits, but it does not create them on its own.

Branching strategy shapes integration risk. Long-lived branches delay learning and hide conflicts. Short-lived branches, trunk-based patterns, disciplined pull requests, and feature flags allow teams to integrate more often while controlling release exposure.

AI-generated artifacts belong inside the same governed delivery system as human-authored artifacts. They require provenance, review, validation, and evidence. As AI accelerates software delivery, disciplined CI and testing become even more important because more material can enter the pipeline in less time.

Continuous Testing treats quality as a system property. It combines design, review, automation, security checks, environment quality, test data, production monitoring, and incident learning. The test strategy should balance fast local feedback with deeper validation, where risk warrants it.

AI-enabled systems require both deterministic and probabilistic validation. Teams must test predictable components with precise checks while evaluating AI behaviour through datasets, thresholds, human review, drift monitoring, and production feedback.

Code coverage is a useful signal when interpreted carefully. It becomes dangerous when treated as proof of quality. The stronger goal is confidence: evidence that the system is fit for purpose, safe to change, and ready to operate.

Core principle

Integration and testing are learning disciplines. The goal is not to run more pipeline jobs, but to learn earlier, with enough evidence to make sound release decisions.

KEY MESSAGE



This chapter defines what it means to deploy and release safely, frequently, and confidently. It separates delivery capability from deployment and release decisions, showing how mature DevOps organizations keep every validated change in a deployable, releasable state while applying appropriate business, operational, security, and governance controls.

Continuous Delivery and Continuous Deployment are operating capabilities. They allow organizations to move changes safely through build, validation, deployment, release, monitoring, and recovery with the right balance of automation, governance, and human oversight.

This chapter is most useful for engineering leaders, release managers, platform teams, SREs, security and compliance partners, product owners, and senior practitioners responsible for reducing the cost and risk of change. It builds on Continuous Integration and Continuous Testing by asking what happens after a release candidate is created and how the organization proves that the change can operate safely in production.

A useful release system should feel routine on an ordinary day and reliable on a difficult one. The operational test is simple: when a serious release decision appears, the organization should already have evidence, ownership, recovery paths, and visibility. If those things must be invented during the release call, the delivery system is still immature.

8.1 Continuous Delivery versus Continuous Deployment

Continuous Delivery and Continuous Deployment are related capabilities, but they answer different operational questions.

Continuous Delivery asks whether every validated software or configuration change can be kept in a deployable, releasable state so it can be deployed to production on demand; the decision to expose the change to users remains separate and risk-based. **Continuous Deployment** automatically deploys every change that satisfies required automated tests, policies, and controls to production without a separate manual deployment authorization; user exposure may still be controlled separately (Humble & Farley, 2010; Hornbeek, 2019; PeopleCert, 2026d).

The distinction matters because release practice must fit the risk profile of the system. A consumer web service with strong automated testing, progressive exposure, and rapid rollback may be able to deploy many changes automatically. A payment platform, medical device back end, or public-sector eligibility system may require deliberate release timing even when the software is technically ready. Both environments can be mature. The measure is confidence, control, and recovery capability, rather than the removal of every human decision.

Continuous Delivery and Continuous Deployment

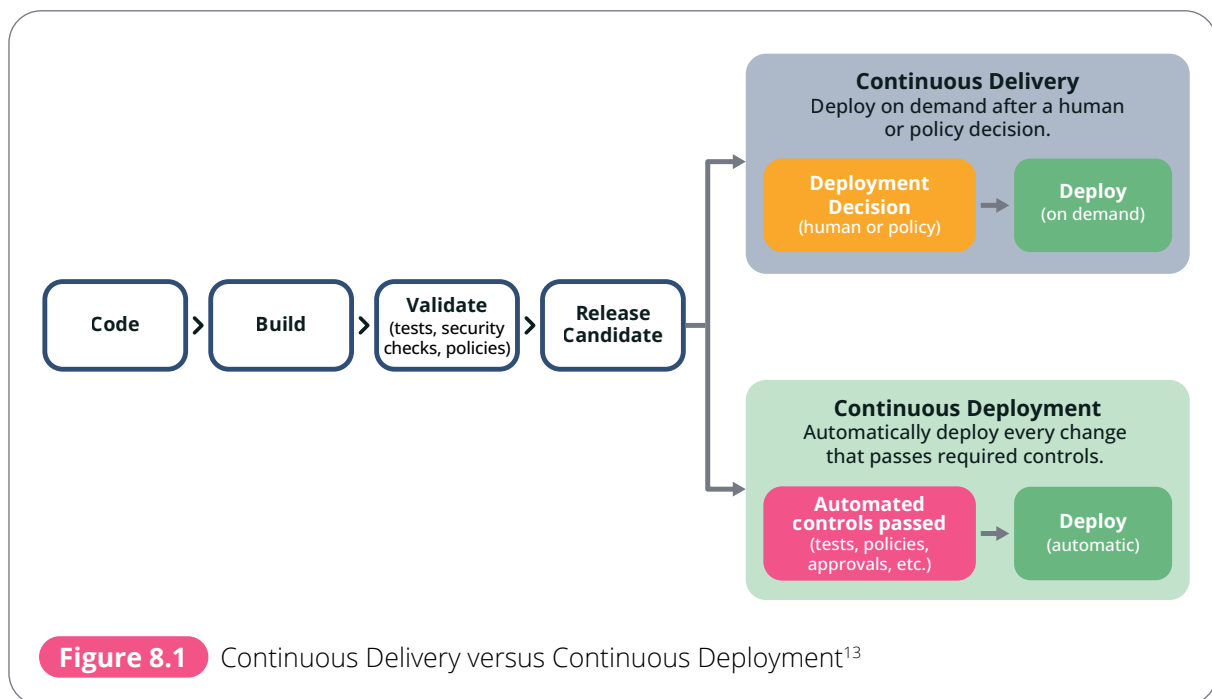


Figure 8.1 illustrates the common flow from code through build and validation to a release candidate, and the point at which Continuous Delivery and Continuous Deployment diverge. It highlights the different deployment decisions: on-demand deployment in Continuous Delivery and automatic deployment in Continuous Deployment. In both cases, release to users may still be controlled separately.

Continuous Delivery is often the more fundamental organizational discipline because it ensures that every validated change remains in a deployable and releasable state. Achieving this requires tests to provide reliable feedback, security checks to run continuously, and deployment and release evidence to be available before pressure builds.

AI-native delivery underlines the contrast between Continuous Delivery and Continuous Deployment. AI may increase the volume of candidate changes, but the delivery system must still determine what is releasable, what is safe to expose, and what requires human review, applying the validation and governance principles established in Chapter 3.

**AI-RELATED
CONTENT**



As a result, exposing a change becomes a choice based on business timing, risk, and user impact rather than a rescue operation at the end of a long development cycle.

Continuous Deployment works when the surrounding system is mature enough to absorb it. The pipeline must build repeatably, tests must be meaningful, deployment patterns must limit exposure, and telemetry must confirm production health quickly. Equally important is a well-defined recovery strategy, because automated deployment without fast detection and recovery simply accelerates the path to operational damage.

8.2 Release readiness and decision governance

A release decision should be based on current evidence and explicit risk, not calendar pressure or confidence in the room. Release readiness, orchestration, and AI-assisted risk analysis provide a connected decision system that shows what is ready, which controls have passed, which exceptions remain, and who is accountable. This section explains how governance can support faster flow by placing policy and evidence inside the delivery workflow.

8.2.1 Release readiness as evidence

Release readiness is the evidence-based state of a release candidate. It is the point at which the organization can say, with reasonable confidence, that a change is fit for deployment and that the team understands how to respond if production behaviour differs from expectation. This confidence should come from evidence collected through the delivery system, rather than from optimism, calendar pressure, or seniority in the room (Humble & Farley, 2010; Nygard, 2018).

A ready release candidate has several kinds of evidence. The build has been completed successfully. Automated tests have run and produced useful results. Security checks have been performed. Compliance evidence is available where the service or organization requires it. Performance and operational criteria have been considered. The deployment plan, rollback path, monitoring readiness, and approval status are known. In short, readiness means evidence exists, is current, and supports an accountable decision.

Readiness to deploy and readiness to release are separate states. A change may be safe to deploy into production while still hidden behind a feature flag. A capability may be technically ready and still await business timing, migration coordination, customer communication, or regulatory sign-off. Mature delivery systems make these states visible instead of blending them into one stressful release event.

Automated evidence improves release readiness because it reduces argument. Teams spend less time asking whether a control happened and more time interpreting the meaning of the signal. The evidence should be current, traceable to the change, and accessible to the people who approve, operate, audit, or support the release. Good evidence does not need to be ornamental. It needs to be trustworthy and usable under pressure.

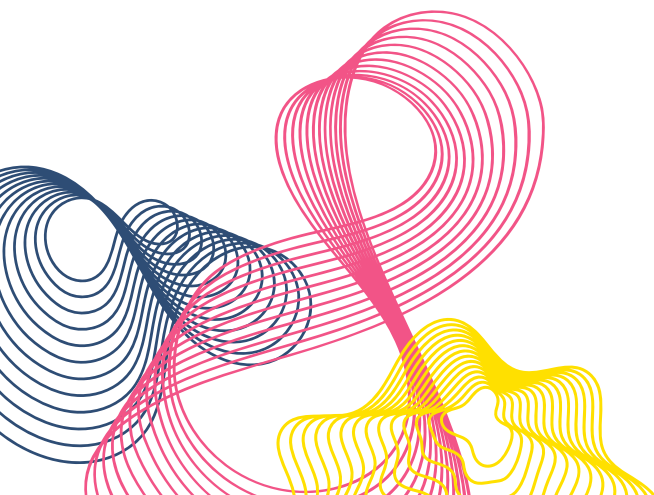


Table 8.1 Release readiness evidence checklist

Release readiness evidence category	Typical evidence	Decision supported	Named owner
Build and provenance	Versioned artifact, source, dependency inventory, signature	Is the release candidate traceable and immutable?	Delivery team
Quality	Automated tests, contract tests, performance results, AI evaluations	Does it meet acceptance and risk thresholds?	Engineering/product
Security and compliance	Scans, policy checks, approvals, recorded exceptions	Is the remaining exposure acceptable?	Security/control owner
Operational readiness	Telemetry, SLI/SLO coverage, runbooks, support readiness	Can the change be operated and detected?	Operations/SRE
Deployment and exposure	Target environment, rollout strategy, flags, release conditions	How will deployment and user exposure be controlled?	Release owner
Recovery	Rollback, roll-forward, disablement, traffic-shift evidence	Can failure be contained and service restored?	Service owner
Accountability	Decision record, exception owner, human-review threshold	Who answers for the deployment or release decision?	Named accountable owner

AI-generated outputs must be treated as delivery work that requires validation, evidence, and accountable ownership (Chapter 3). Regarding release readiness, this means that generated code, tests, infrastructure files, and configuration changes must meet the same security, operational, rollback, and evidence standards as any other change. AI assistance should not lower the release threshold; the level of review and evidence should remain proportionate to the risk and potential impact of the change. Release decisions should be based on the evidence produced by the delivery system, not on confidence in the AI tool that generated or assisted the work.

AI-RELATED CONTENT



8.2.2 Application Release Orchestration and embedded governance

Application Release Orchestration coordinates deployment and release decisions for services, pipelines, environments, dependencies, approvals, risk rules, evidence, exposure controls, and recovery paths. It provides control across the delivery system for a service without turning release into a hand-managed event. Coordination should reduce waiting, not become a new layer of procedural ceremony.

Modern changes often involve multiple services, shared environments, database changes, security controls, feature flags, and customer exposure rules. Teams need a way to understand which components are ready to deploy, which changes are ready to release, what is waiting for approval, and what is blocked by another dependency. The Application Release Orchestration layer makes these conditions visible and repeatable.

It also creates an evidence trail that can be reviewed later without reconstructing the event from chat messages and meeting notes.

Governance belongs inside this orchestration model. A low-risk change that passes automated controls may proceed with automatic approval. By contrast, a high-risk change may require a human review because it touches sensitive data, affects a critical service, changes infrastructure policy, or introduces a material compliance concern. The approval path should follow the risk of the change, not the habit of the organization.

Figure 8.2 illustrates how Application Release Orchestration coordinates these dependencies, environments, policy decisions, evidence, and risk into a single release process. It also shows movement through test, staging, and production environments while governance controls provide policy, evidence, and risk oversight.

Embedded governance depends on clear policy. Teams need to know which checks are mandatory, which are advisory, and which conditions require escalation. Policy as code can enforce repeatable rules for security scanning, test coverage thresholds, environment promotion, segregation of duties, evidence capture, and deployment approval. Human review remains valuable for judgement, context, and exception handling. It should be reserved for decisions where judgement is required.

AI-assisted outputs should not replace but support accountable human decisions, as discussed in Chapter 3. This means that AI may help summarize evidence, identify missing approvals, highlight dependency risk, or suggest rollout patterns, while release thresholds, approval authority, evidence controls, and audit trails remain governed by the organization's release model. Section 8.2.3 applies this principle to the selection and execution of deployment and rollout patterns.

AI-RELATED CONTENT



Practice note

Approvals should follow risk. A low-risk documentation update and a high-risk payment authorization change should not travel through the same approval path.

KEY MESSAGE

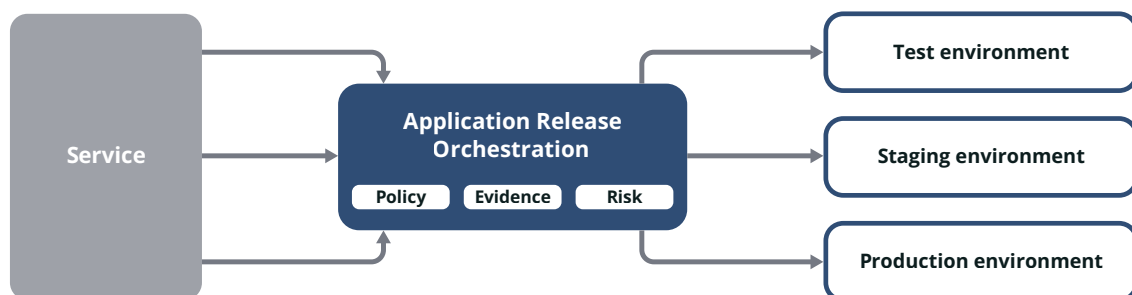


Figure 8.2 Application Release Orchestration and embedded governance¹⁴

8.2.3 AI-assisted risk-based deployment decisions

AI can help teams make better deployment decisions when the input signals are available and the decision model is governed. Release risk is rarely visible from a single check. It usually emerges from the relationship between change size, service criticality, test history, deployment history, security findings, dependency behaviour, production telemetry, and current business conditions. A well-designed AI assistant can help assemble those signals faster than a release manager working manually (NIST, 2023a; Hornbeek, 2024; DORA, 2025).

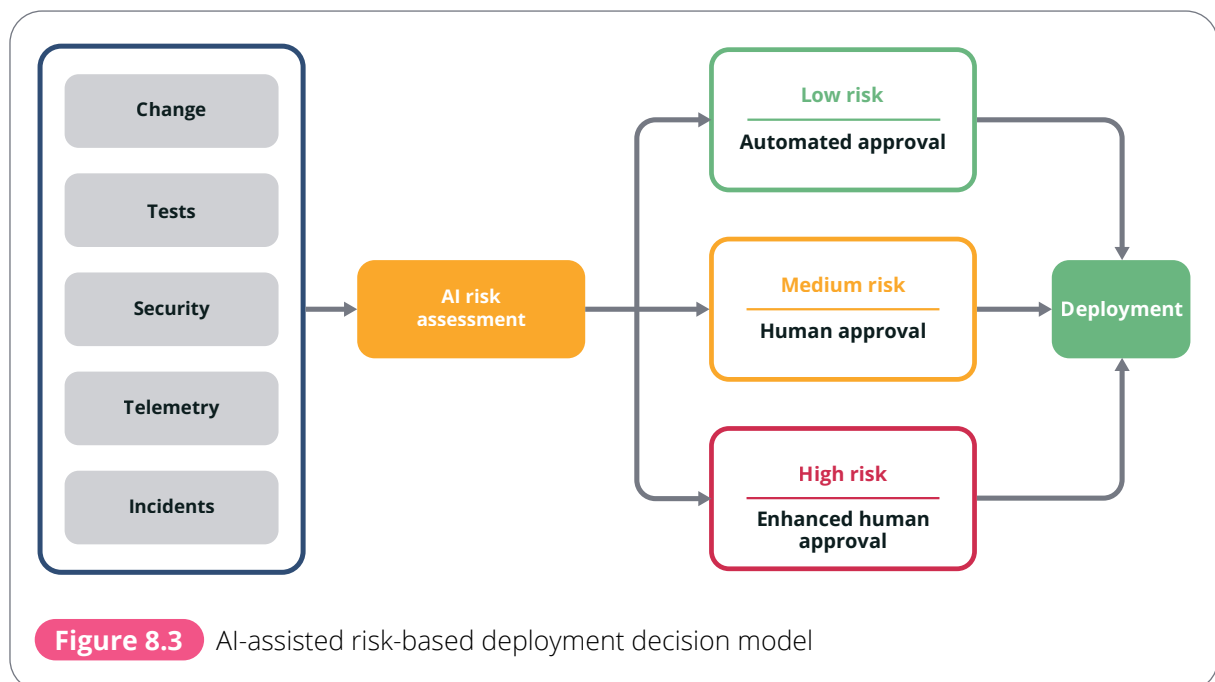
Useful inputs include code complexity, changed files, test outcomes, vulnerability findings, infrastructure changes, dependency updates, recent incidents, open problems, service-level objective status, on-call load, and customer impact. The output may be a risk score, a recommended rollout pattern, a proposed approval path, or a monitoring checklist. These outputs should be explainable enough for teams to understand the basis of the recommendation. Figure 8.3 outlines how these signals can be combined into an

AI-assisted risk assessment that recommends an approval path while preserving human accountability for material decisions.

AI should not become an unaccountable release authority. Some changes require human review regardless of a low predicted risk score. These include changes to regulated workflows, sensitive data handling, identity and access control, production infrastructure policy, safety-critical behaviour, financial transaction logic, or high-impact AI model behaviour. The organization must define these thresholds before the release system is under pressure.

The model itself requires monitoring. Teams should review how often AI recommendations align with actual outcomes. They should look for missed failures, unnecessary escalations, drift in prediction quality, and unexplained changes in risk scoring. When an AI-assisted recommendation contributes to a poor release decision, the post-incident review should examine the model, the prompt, the data, the governance rule, and the human decision path.

A practical starting point is modest. Use AI to summarize release evidence, identify missing controls, compare a change against historical failures, or propose a rollout plan. Keep the human decision clear. Over time, teams may automate low-risk paths when the evidence supports it.



AI-native control point

AI can help evaluate deployment risk, but the organization must define the evidence, thresholds, exception path, and human accountability model before relying on the recommendation.

**AI-RELATED
CONTENT**



Mature AI-assisted release management grows from proven signals, calibrated thresholds, and accountable practice.

8.3 Controlled deployment, exposure, and recovery

Deployment safety depends on controlling exposure and preserving practical recovery options. Progressive patterns, separation of deployment from release, rehearsed rollback or roll-forward, and production validation allow teams to learn from real behaviour without exposing every user at once. The following sections show how these controls turn failure from an unmanaged event into a bounded and observable condition.

8.3.1 Deployment patterns for controlled exposure

Deployment patterns reduce release risk by controlling where change runs, who can see it, and how quickly exposure increases. They are operational design choices. A team should choose a pattern based on the risk being managed, the service architecture, the available telemetry, and the recovery options. The pattern is only useful when the organization can observe behaviour and act quickly (Humble & Farley, 2010; Nygard, 2018; Hornbeek, 2019).

Blue-green deployment reduces cutover risk by maintaining two production-ready environments. Traffic moves from one environment to the other when the new version is validated. This pattern works well when environments can be kept consistent and data migration risk is manageable. It becomes harder when state, cost, or infrastructure complexity makes duplicate environments expensive or fragile. Yet, all patterns require monitoring, recovery, and explicit release controls.

Canary release reduces exposure risk by sending a small share of traffic or a small user population to the new version first. The team watches error rates, latency, business transactions, and user signals before expanding. Canary strategies require reliable segmentation, fast telemetry, and a clear stop condition. Without those controls, a canary becomes a small release that still depends on hope.

Feature flags decouple deployment from release by allowing code to exist in production while functionality remains hidden or limited. This supports gradual exposure, experimentation, emergency disablement, and business timing. Feature flags also create operational debt when they are not retired. Teams need ownership, naming discipline, visibility, and cleanup practices, or the flag system becomes another source of uncertainty.



Table 8.2 Deployment patterns for controlled exposure

Pattern	Best use	Exposure	Required signal	Recovery path	Release control
Blue-green	Environment replacement with rapid traffic switch	One full environment before switch	Health, functional, and data validation	Switch traffic back	Switch or flag controls availability
Canary	Limit risk while validating real behaviour	Small cohort, then expand	SLOs, errors, latency, and business signals	Halt expansion and route back	Staged, evidence-based exposure
Feature flags	Separate deployment from release	Selected users, segments, or features	Feature and user-outcome telemetry	Disable the flag	Explicit release by configuration
Progressive delivery	Automate staged exposure by policy	Increasing cohorts, regions, or traffic	Continuous technical and business signals	Pause, rollback, or disable	Policy-driven promotion

Progressive delivery combines gradual exposure, telemetry, and decision rules. The release expands only as confidence increases. AI-assisted systems can support this pattern by watching more signals than a human can reasonably inspect in real time, including latency shifts, error patterns, unusual user behaviour, and rollback indicators. Nevertheless, the decision rules still need boundaries set by people. Automated expansion should be tied to explicit health criteria, not general confidence in the tool.

8.3.2 Decoupling deployment from release

Deploying code and releasing value are different acts. Deployment moves a technical change into an environment. Release exposes functionality, behaviour, or value to users. Many organizations still treat these as one event, which increases pressure on release windows and makes technical deployment carry every business, operational, and customer risk at once (Humble & Farley, 2010; Hornbeek, 2019; PeopleCert, 2026d).

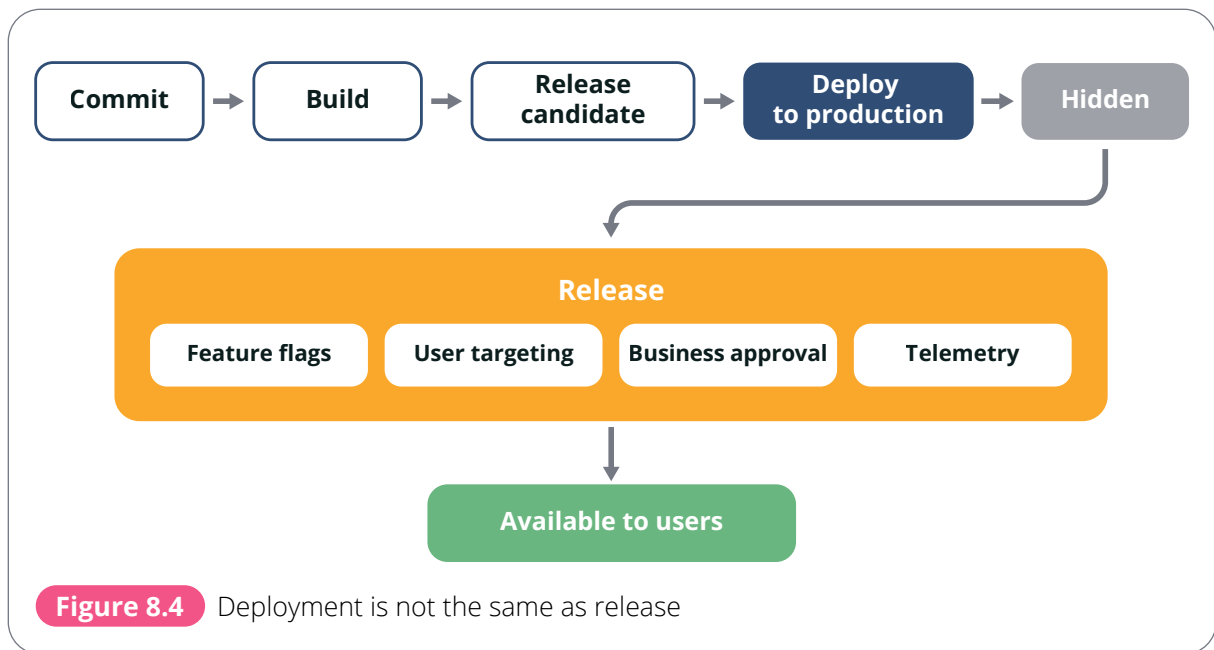
Decoupling deployment from release gives teams much greater control over both technical and business risk. Code can be deployed into production, verified under real infrastructure conditions, and remain hidden

until the organization is ready to expose it. This allows teams to separate technical risk from market timing, customer communication, migration readiness, and compliance review. As a result, the deployment event no longer carries the full weight of the launch, creating a calmer and more predictable release process. Release controls, such as feature flags, user targeting, business approvals, and production telemetry determine when and how functionality becomes available to users. Figure 8.4 illustrates this separation between technical deployment and controlled user exposure.

Feature flags, permissions, user targeting, configuration controls, dark launches, and staged exposure all support this separation. A team may deploy a new recommendation component into production and route only internal traffic to it. A public-sector service may deploy a new eligibility workflow while exposing it first to case workers in one region.

A commerce platform may release a payment integration to a small customer segment while monitoring abandonment and failure rates.

Practice requires discipline. Hidden code still needs ownership, documentation, monitoring, and cleanup. A flag left in place indefinitely can create hidden branches inside production. A partially released capability can confuse support teams if the operational state is not visible. Decoupling works when the release controls are designed, tracked, and reviewed as part of the same delivery system.



8.3.3 Rollback and recovery: designing for safe failure

Safe release depends on prevention and recovery. Prevention matters, but complex systems still fail. A release strategy that assumes every defect will be caught before production will disappoint the team eventually. A mature delivery system accepts that some changes will behave differently under real load, real data, real dependencies, and real users. It prepares for that moment before deployment begins (Beyer et al., 2018; Nygard, 2018; Hornbeek, 2019).

Rollback is one recovery strategy, but it is not the only one. The appropriate response depends on the nature of the failure and its impact on the system. Some issues can be corrected by rolling forward with a fix or disabling a feature flag, while others may require traffic to be redirected, configuration to be restored, or an earlier version to be reinstated. Data migration often demands a carefully prepared forward correction because reversing the schema could cause more damage. Recovery strategies should be designed and rehearsed before release.

Teams should know whether to roll back, roll forward, disable a feature flag, shift traffic, or restore data before production pressure arrives.

Database and data migration changes deserve particular attention. Schema changes, backfills, data transformations, and index changes often limit simple rollback. Teams should design migrations in small, compatible steps. They should test forward and backward compatibility, understand data repair options, and document the recovery path in operational language. A release plan that says rollback is available without explaining the implications for data recovery is incomplete.

Recovery also depends on observability and incident response. The team must know what signals indicate failure, who is responsible for the first action, how escalation works, and what customer communication may be needed. Recovery should be rehearsed for critical services. A rollback process that has never been used remains a theory, and theories have a habit of becoming expensive during an outage. Although AI may help detect anomalies, summarize recent changes, and suggest mitigation, recovery automation should remain bounded, observable, reversible, and subject to human override.

8.3.4 Deployment verification and production validation

A deployment can succeed technically while still failing operationally. The artifact may land on the correct server, containers may start, and health checks may pass while customers still experience failed transactions, slow responses, broken workflows, or incorrect AI outputs. Deployment verification confirms that the deployment was completed successfully. Production validation confirms that the service is behaving well enough to continue exposure.

Verification begins with simple checks.

- Did the expected version deploy?
- Are the required services running?
- Are dependencies reachable?
- Did the smoke tests pass?
- Are logs, metrics, and traces flowing?

These checks catch many common deployment problems and should run automatically after every deployment.

Production validation goes beyond deployment verification by examining the behaviour users experience. Synthetic checks confirm that critical journeys, such as login, search, checkout, claims submission, or payment authorization, continue to function correctly. Telemetry reveals latency shifts, error spikes, saturation, business transaction failures, and regional degradation. Business signals show whether conversion, task completion, or support contact rates have changed after exposure.

Production evidence should feed release decisions. A progressive rollout should pause when signals degrade. A feature flag should be disabled when critical thresholds are crossed. A successful deployment should improve the team's confidence in the release process. Over time, production validation becomes more than a release check; it becomes a continuous learning system that improves future deployment and release decisions.

Deployment reality

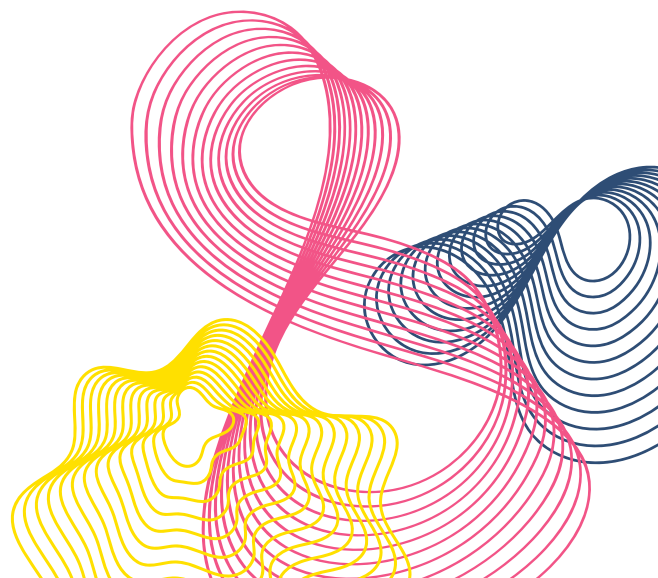
Deployment is not complete when the pipeline succeeds. It is complete when production signals confirm that the service is healthy, user impact is acceptable, and rollback or feature-disablement remains available.

KEY MESSAGE



AI-enabled services require production validation that goes beyond availability and latency. Where those systems affect users or business outcomes, production validation should include relevant model, prompt, retrieval, safety, and decision-quality signals. Chapter 11 provides further information on this matter.

AI-RELATED CONTENT



8.4 Deployment frequency, release flow, and capability maturity

Deployment frequency is useful when interpreted as a capability signal. Frequent deployments often indicate that teams work in small batches, integrate continuously, automate validation, trust their deployment process, and recover quickly. These are healthy signs. Deployment frequency is meaningful only when interpreted alongside the quality, stability, and value delivered by those deployments. A team can increase deployment frequency with trivial changes and still produce low value or high operational risk; release exposure should be assessed separately (Hornbeek, 2019; DORA, 2024; DORA, 2026a).

Frequency should be read with quality, stability, and value. Higher deployment frequency paired with lower change failure and faster recovery is encouraging. Higher frequency with growing defect escape, rising support calls, or worsening customer outcomes deserves careful attention.

The objective is to reduce the cost and risk of change, not simply to increase the number of deployments. Table 8.3 contrasts the characteristics of traditional release events with continuous release flow. The difference is not simply one of speed, but of how confidence, evidence, and coordination are created.

Many organizations begin with release events. These are large, scheduled, carefully coordinated efforts involving freeze windows, manual checklists, release meetings, approval chasing, and incident readiness. Release events usually exist because changes are large, and confidence is low. The event tries to compensate for uncertainty with coordination. It often makes the release more stressful without removing the underlying risk.

Release flow is different. Work moves in smaller batches. Evidence is collected continuously. Governance is embedded into the pipeline. Deployment is routine. Teams release when the change is approved, and the risk is understood. Product, operations, security, and support teams still coordinate, when necessary, but coordination is driven by actual risk rather than inherited ceremony. In addition, release flow becomes even more important, since AI-native delivery increases the volume of change entering the system. Therefore, the safest option is disciplined flow: smaller changes, stronger validation, observable production behaviour, and accountable release decisions.

Table 8.3 From release event to release flow

Dimension	Release event pattern	Release flow pattern
Batch size	Large batch	Small, validated batch
Coordination	Many meetings and release windows	Routine, automated orchestration
Evidence	Manual assembly near the event	Evidence produced continuously
Risk discovery	Late, after work has accumulated	Early, while change is small
Recovery	Complex rollback across a large change	Reversible patterns and limited exposure
Feedback	Weak or delayed	Continuous production and customer feedback

Case B | Harbor Digital Commerce (composite): from quarterly release events to daily deployment capability

Harbor began with a quarterly release train. Checkout, search, pricing, and account changes accumulated into one event. Testing and governance evidence arrived late, operations received incomplete change context, and the release calendar became the mechanism for creating temporary coordination. The ceremony was large because confidence was low.

The organization selected one checkout value stream and reduced batch size before pursuing more frequent deployment. Teams integrated earlier, strengthened contract and recovery tests, standardized release-readiness evidence, added deployment and release records, and instrumented the customer journey. Feature flags and canary exposure allowed deployment to production without immediately releasing every change to all users.

Harbor recognized that higher deployment frequency should be an outcome of capability, not a target to chase blindly. Teams earn frequency through small changes, reliable tests, automated evidence, observability, and safe recovery.

The first leadership target – daily deployments – produced undesirable behaviour. Teams split low-value changes to improve the number and avoid higher-risk work that needed deeper validation. Leadership replaced the target with evidence of on-demand deployability, lower wait time, dependable recovery, and improved customer outcomes.

Over several iterations, monthly and weekly release events became less dramatic. Some services became capable of daily deployment, while release remained a product and risk decision. The organization did not require every team to deploy every day; it required teams to maintain a safe path to production when change created value.

AI-assisted delivery increased artifact volume, but it did not change the acceptance rule. Generated code, tests, infrastructure changes, and release summaries followed the same governed pipeline. Harbor treated daily deployment capability as system confidence, not as permission to move unvalidated work faster.

Transferable lesson: Daily deployment capability should emerge from smaller batches, trustworthy evidence, embedded governance, controlled exposure, and rehearsed recovery. It is a capability to use when valuable, not a quota to optimize.

SCENARIO



DECISION AND ACTION GUIDE

Executive decision: Product and technology executives define deployment and release decision rights, risk tiers, approval thresholds, progressive-exposure rules, and recovery expectations, with a named accountable release decision owner for each product.

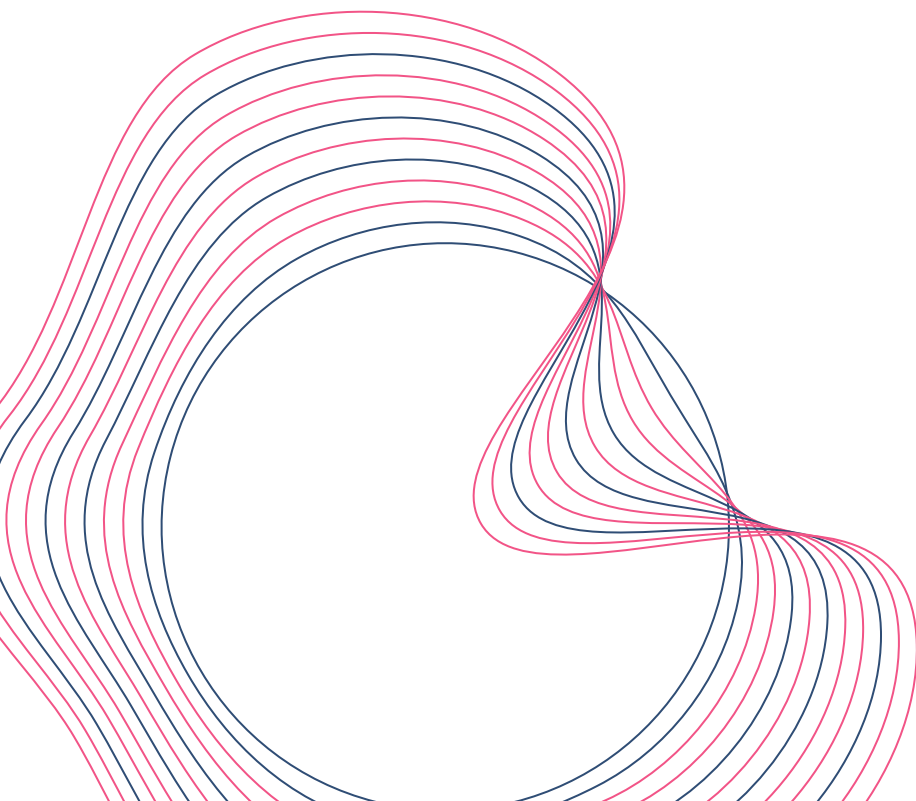
Role-specific actions: Engineering, operations, SRE, security, and product roles automate readiness evidence, controlled deployment, separate release records, production validation, and recovery.

Evidence to review: Release-readiness records, deployment and release records, exposure decisions, production-validation results, recovery tests, and approved exceptions.

8.5 Chapter summary

Continuous Delivery and Continuous Deployment are related but distinct capabilities. Continuous Delivery keeps each validated change in a deployable and releasable state, while Continuous Deployment automates deployment when the required controls pass. In both cases, readiness must be based on current evidence covering build provenance, quality, security, compliance, operational readiness, exposure, and recovery.

Governance belongs inside the delivery workflow. Risk-based approval paths, progressive exposure, reliable telemetry, and rehearsed recovery make release routine rather than ceremonial. AI can help assemble evidence and recommend rollout decisions, but its inputs, limits, audit trail, and human accountability must remain explicit.



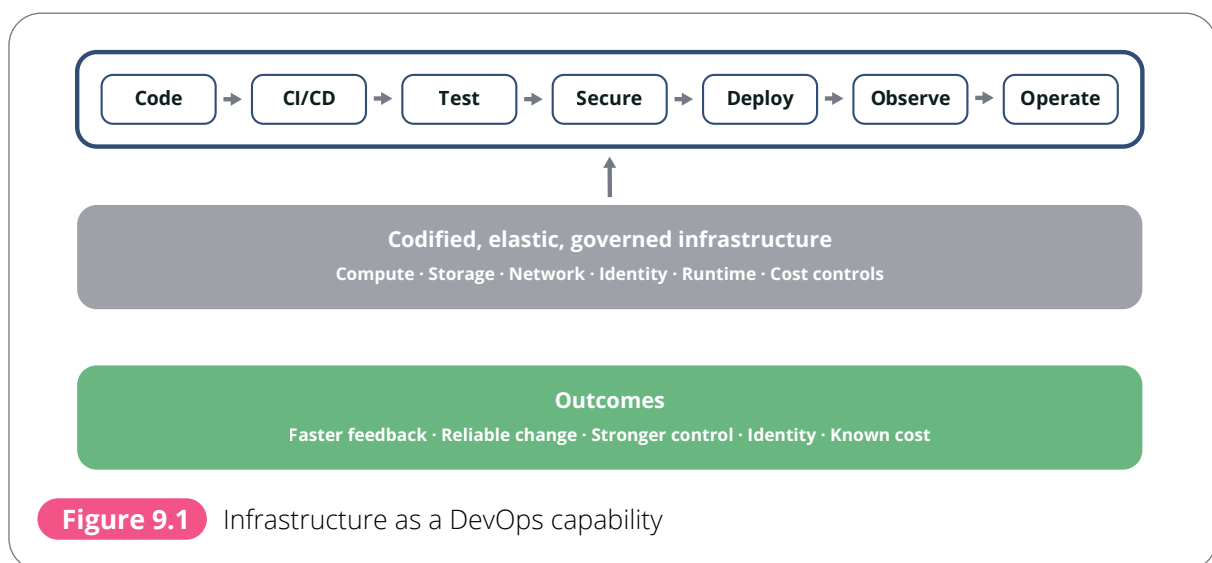


This chapter defines the infrastructure layer as an enabler of delivery speed, operational control, security, governance, and cost discipline at scale. It positions Platform Engineering as the maturity path for infrastructure, moving from manually requested environments to self-service capabilities that teams can use safely.

In this chapter, infrastructure refers to the compute, storage, networking, runtime, environment, identity, and related cloud or on-premises resources needed to build and operate services. Platform Engineering applies product thinking to reusable internal capabilities that help teams use this infrastructure and other delivery services safely and efficiently. Golden paths are recommended, supported ways for teams to use these capabilities for common delivery tasks without removing their ability to choose another approach when context requires it.

Infrastructure is now embedded in the delivery system. Mature DevOps organizations treat it as codified, elastic, observable, secure, cost-aware, and available through internal platforms. The most effective platform capabilities help teams move faster while preserving control, evidence, and accountability.

Figure 9.1 places infrastructure underneath the delivery capabilities that support value delivery, including code, continuous integration and continuous delivery (CI/CD), testing, security, deployment, observability, and operation. The upward arrow indicates that codified, elastic, and governed infrastructure enables and supports this delivery flow. The figure also shows how infrastructure maturity contributes to reduced friction and delay and to greater reliability, control, and speed, reflected in outcomes such as faster feedback, reliable change, stronger control, identity, and known cost.



Elastic Infrastructure and Platform Engineering

This chapter is most relevant to platform teams, infrastructure engineers, site reliability engineers (SREs), architects, security practitioners, finance partners, and leaders responsible for scaling delivery. Product teams should also read it, as infrastructure choices directly affect their daily work.

9.1 Infrastructure as a DevOps capability

Infrastructure quality determines how quickly teams can move, how confidently they can test, and how safely they can operate. A weak infrastructure foundation creates friction in every part of the value stream. Developers wait for test environments. Testers lose confidence when staging behaves differently from production. Operations teams inherit systems that are difficult to observe, support, and recover. Security teams find themselves reviewing exceptions rather than seeing approved controls applied consistently (Humble & Farley, 2010; Hornbeek, 2019; CNCF TAG App Delivery, 2023a).

In mature DevOps organizations, infrastructure is part of the product delivery system. It is versioned, reviewed, tested, deployed, monitored, and improved with the same discipline applied to application code. The practical outcome is straightforward. Teams can create and change environments without waiting on manual tickets, and leaders can still see evidence that the work followed policy.

Operational guidance

- Treat environment creation and infrastructure change as delivery work.
- Keep infrastructure definitions in version control and review them.
- Define approved patterns for common services instead of allowing every team to invent its own.
- Make cost, security, and reliability signals visible to teams before production.

TIP



Building on that, Chapter 9 applies the principle highlighted in Chapter 3: AI-native delivery increases the load on infrastructure, with a focus on repeatable patterns, validated modules, access controls, telemetry, and cost guardrails for AI-assisted work and model-serving workloads.

A mature infrastructure capability provides a predictable foundation for building, testing, deploying, and operating services. It embeds controls that reduce risk without forcing every decision into a manual approval queue, while giving teams and leaders visibility into the operational and financial consequences of engineering decisions.

The old infrastructure model was built around requests. Someone needed a server, a database, a firewall change, or a runtime dependency, and another team fulfilled the request. That model can work for slow change, but it struggles when software, data pipelines, AI services, and security controls change continuously. The modern model treats infrastructure as a product, providing reusable capabilities that teams can consume safely through self-service and governed automation.

9.1.1 Infrastructure as Code and environment lifecycle

Infrastructure as Code (IaC) changes infrastructure from a collection of manual actions into a governed engineering artifact. The definition of an environment becomes visible. It can be reviewed, tested, compared, reused, and restored. This provides the minimum foundation for reliable infrastructure at scale (Humble & Farley, 2010; Hornbeek, 2019; NIST, 2022).

The discipline is more than writing templates. A mature IaC lifecycle begins with clear definitions, continues through version control and peer review, applies automated validation, and ends with monitored environments. Drift detection closes the loop by identifying manual changes and either

reconciling them automatically or requiring a deliberate exception (Figure 9.2). The benefits become evident quickly: teams know which configuration was deployed, security can review reusable modules instead of individual console changes, and compliance gains evidence of approval and deployment. Operations can recreate environments during recovery, and developers can test against consistently provisioned environments. The same principle also applies to AI-generated infrastructure artifacts: they should follow the governed IaC lifecycle before production use, just as human-authored templates do.

Teams often underestimate the cultural change required. Manual changes often feel fast in the moment, but they are difficult to trace, reproduce, and explain during an incident. Codified infrastructure requires discipline initially, yet the investment pays off during the next release, recovery exercise, audit, or scaling event. Figure 9.2 illustrates the IaC lifecycle. It shows how version control, review, validation, automated provisioning and deployment (Apply), drift detection (Observe), and reconciliation keep running environments aligned with their approved definitions.

A reliable environment lifecycle should answer the following four questions without heroics: what changed, who approved it, how it was validated, and how the running environment aligns with the approved definition. If those questions are hard to answer, the organization still depends on memory and trust where it needs evidence.



Infrastructure changes follow the same control model as application changes
reviewable · testable · auditable · recoverable

Figure 9.2 Infrastructure as Code lifecycle

9.1.2 Cloud-native and elastic infrastructure for AI workloads

Cloud-native infrastructure is an operating approach, not a location. Its useful characteristics are automation, elasticity, resilience, API-driven provisioning, observability, and managed capabilities. These principles can be implemented across different environments. The choice of platforms is less important than adopting these operating principles consistently (NIST, 2024; FinOps Foundation, 2025).

Elastic infrastructure gives teams the ability to match resources to demand. While this benefits most applications, it is particularly important for AI-enabled systems, where AI training, inference, vector search, batch processing, and data pipelines generate highly variable workloads. Some activities require short bursts of computing, whereas others demand steady, low-latency service. A single static capacity model rarely satisfies both.

Elasticity should always operate within defined boundaries. Autoscaling without workload understanding increases cost without improving user experience. Although managed services reduce operational burden, they introduce portability, governance, skills, and cost considerations. Good engineering practice makes these trade-offs visible before the architecture is committed. Cloud-native design should keep the same practical question in view: what does this choice do to flow, reliability, security, cost, and recovery? The answer will vary by system. A citizen-facing service, a trading platform, and an internal analytics workload have different risk profiles. The principle remains stable. Infrastructure choices should make the system easier to change safely.

AI workloads need strong cost and operational signals. Token usage, accelerator consumption, model-serving latency, batch job duration, storage growth, and data movement all affect the economics of the platform. A team that cannot see those signals will eventually learn about them through a budget surprise or an operational failure. Neither is a good feedback mechanism.

AI-RELATED CONTENT



Design checks for elastic infrastructure

- Know which workloads are steady, bursty, training-heavy, inference-heavy, or data-heavy.
- Define scale thresholds and rollback thresholds before production use.
- Expose cost and performance telemetry to the team that owns the workload.
- Apply policy guardrails to expensive or sensitive compute classes.

TIP



9.1.3 Environment consistency and drift management

Environment consistency is one of the less visible but most important foundations of delivery confidence. Development, testing, staging, and production environments do not need the same size or capacity. They should, however, share consistent configuration, dependencies, deployment behaviour, security baselines, and policy rules. When those conditions diverge, test evidence becomes weaker.

Drift usually starts innocently. A patch is applied directly during an incident. A configuration setting is changed in staging to unblock testing. A temporary access exception becomes permanent. A dependency version changes in one environment and not another. Each change may have a reasonable explanation. Together they create a system where teams no longer know whether a successful test means much.

Drift management makes these differences visible through immutable infrastructure, automated reconciliation, configuration scanning, policy checks, and environment comparison. The goal is not to prevent every manual action but to ensure that any manual change is explicit, temporary, and reviewable. As explained in Chapter 3, AI-native delivery expands the scope of environment consistency; drift management must therefore include AI artifacts, runtime dependencies, data access policies, and model-serving conditions.

Production-like environments are essential because meaningful feedback depends on realistic operating conditions. A test environment that lacks realistic configuration, representative data patterns, or comparable network paths may provide comfort while missing the failure mode. Mature teams use scaled-down environments where appropriate, but they preserve the behaviour that matters for validation.

Reality check

Environment drift weakens every later signal. A test that passed in an unlike environment does not provide strong evidence for production readiness.

KEY MESSAGE



A practical test for environment maturity is to ask whether a new engineer can provision a compliant test environment without relying on the one person who knows the old scripts. If the answer is no, the organization still depends on tribal infrastructure knowledge hidden in the delivery path rather than a repeatable delivery capability.

9.2 Platform Engineering as a discipline

Platform Engineering is the discipline of designing and operating internal platforms that help delivery teams build, deploy, secure, observe, and operate software more effectively. The platform transforms repeated engineering work into reusable capabilities. It gives teams a golden path for common work while allowing specialized routes where the system genuinely requires them (CNCF TAG App Delivery, 2023a; Fournier & Nowland, 2024; Skelton & Pais, 2025).

This discipline grew out of infrastructure work, yet it is broader than infrastructure operations. Infrastructure operations tend to focus on running technical components.



Platform Engineering focuses on the experience of the teams consuming those capabilities. It asks whether the capability is discoverable, usable, documented, supported, reliable, secure, and improving. A platform that requires weeks of explanation before a team can use it is still carrying too much operational folklore.

The platform should reduce friction without removing accountability. Product teams still own the outcomes of the services they build and operate. The platform team provides common capabilities such as CI/CD templates, infrastructure modules, service catalogues, deployment workflows, secrets handling, observability patterns, and compliance evidence. The best platforms make the safe path the easiest path.

Platform Engineering provides a practical foundation for governing AI-enabled delivery. Approved model endpoints, prompt management patterns, AI workload environments, data access controls, policy checks, and observability conventions can be built into platform capabilities. Teams get usable paths. Governance teams get evidence. Security teams get consistency. Product teams get less accidental complexity.

A platform team should be careful about its own failure modes. It can become a ticket queue. It can become a centralized approval body. It can standardize the wrong abstraction and force teams into workarounds. These problems usually appear when the platform is treated as an internal mandate rather than an internal product.

Platform maturity typically progresses from manual support to reusable automation, and ultimately to governed self-service. Its success is measured by whether product teams can deliver faster, with lower cognitive load, stronger governance, and greater operational confidence.

9.2.1 Internal developer platforms with embedded governance

Internal Developer Platform (IDP)

A product-managed internal platform that provides self-service paths for building, testing, deploying, observing, securing, and governing delivery work.

DEFINITION



An Internal Developer Platform is the practical implementation of Platform Engineering. It provides self-service access to approved capabilities. A product team should be able to create a service, provision infrastructure, configure a pipeline, and deploy through approved workflows. It should also capture evidence, connect observability, and manage secrets without starting a chain of manual requests (CNCF TAG App Delivery, 2023a; Fournier & Nowland, 2024).

An important word is approved. Self-service without governance quickly becomes unmanaged variation. Embedded governance incorporates controls into templates, workflows, policies, service catalogues, and evidence capture. Teams do not need to remember every standard because the platform applies many of them by design.

Good Internal Developer Platforms usually include a service catalogue, reusable templates, infrastructure modules, deployment workflows, access patterns, documentation, support routes, and observability conventions.

AI-native capabilities should be delivered through the Internal Developer Platform model, with approved services, reusable patterns, clear access controls, and monitoring for model behaviour. When these capabilities are easy to discover and use, teams can adopt AI responsibly without creating fragmented or weakly governed local solutions.

AI-RELATED CONTENT



A platform capability is ready when:

- teams can find it without asking the platform team directly
- the usage path includes documentation and support expectations
- security, evidence, observability, and cost rules are embedded
- the platform team measures adoption and friction, then improves the service.

KEY MESSAGE



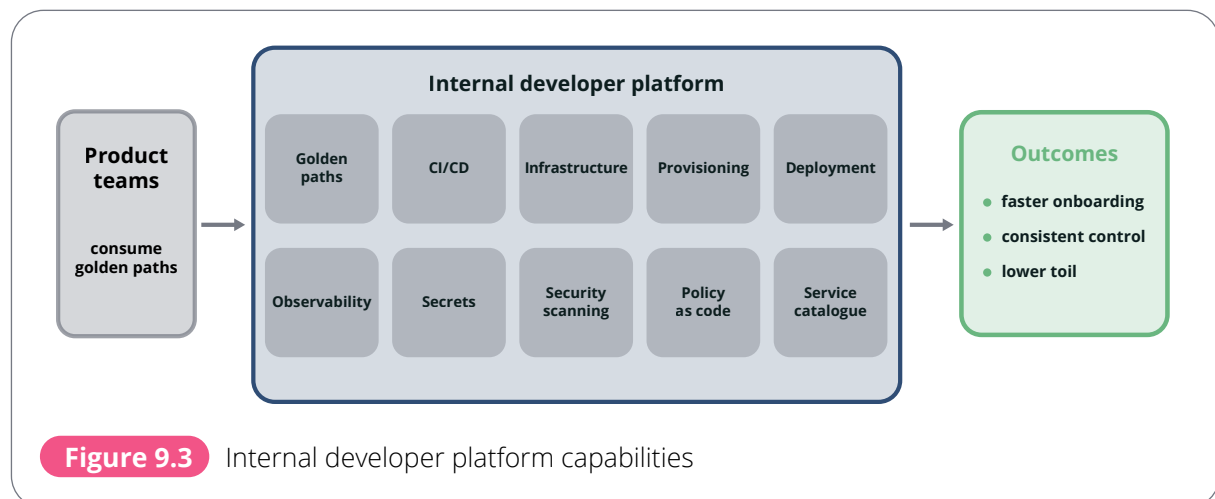
They also include feedback mechanisms. Teams need a way to report friction, request changes, and understand the platform roadmap. The platform should make onboarding faster. It should also make day-two operations better. A new service that is easy to deploy but hard to observe, patch, recover, or manage costs has only moved the problem. Mature Internal Developer Platforms include the operational details that teams tend to forget when under delivery pressure.

Figure 9.3 illustrates how an Internal Developer Platform packages reusable capabilities into a governed self-service experience, enabling product teams to consume approved services while reducing operational friction. The platform includes golden paths, CI/CD templates, infrastructure modules,

environment provisioning, observability patterns, secrets management, security scanning, policy as code, deployment automation, and related self-service capabilities.

9.2.2 The platform team as a product team

A platform team has customers. They are the internal product, service, data, security, and operations teams that depend on platform capabilities to do their work. This framing changes the behaviour of the platform team.



It must understand users, prioritize needs, manage a backlog, publish a roadmap, provide support, and measure whether its services are helping (Fournier & Nowland, 2024; Skelton & Pais, 2025).

Adoption is one useful signal, although it needs interpretation. Strong adoption may show that the platform provides real value. Weak adoption may show that the platform is hard to use, poorly documented, missing important capabilities, or slower than local workarounds. It may also show that leadership has mandated a platform before it has earned trust. The platform team should be curious about the reason, not defensive about the number.

Platform product thinking also changes how standards are introduced. A standard that improves security but creates a painful developer experience will generate exceptions. A standard built into a well-designed workflow can become normal practice. The platform team should work with security, compliance, architecture, finance, and product teams to turn required controls into usable capabilities.

Service management also matters. Platform services need service-level expectations, support channels, incident handling, change notices, documentation, and deprecation practices. Internal users build their own commitments on top of the platform. If the platform is unreliable or unpredictable, every service that depends on it inherits that uncertainty. Figure 9.4 shows the platform team operating as an internal product team, continuously learning from users, prioritizing improvements, providing support, and evolving the platform through ongoing feedback.

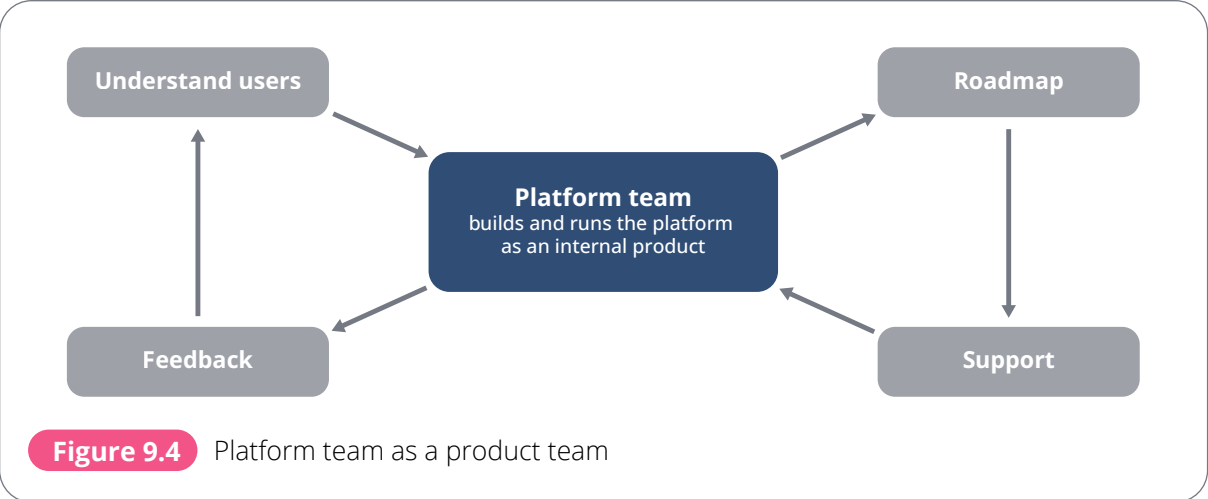
Responsible AI adoption also requires approved capabilities to be easier to discover and use than unmanaged alternatives. For platform teams, this means embedding AI services, validation hooks, data-handling controls, and monitoring patterns into the platform offering rather than relying on policy alone.

AI-RELATED
CONTENT



9.2.3 Cognitive load, governance, and cost awareness at scale

Platform investment pays back when it reduces unnecessary cognitive load. Product teams already carry domain complexity, user needs, operational behaviour, security requirements, and delivery pressure. Asking every team to master every infrastructure detail, policy rule, deployment pattern, runtime behaviour, and cost model creates overload. Overloaded teams make more mistakes, move more slowly, and rely on local shortcuts (FinOps Foundation, 2025; Skelton & Pais, 2025).



There is a difference between essential complexity and accidental complexity. Product-specific behaviour, regulatory requirements, and customer workflows may be essential. Rewriting common pipeline logic, recreating secrets patterns, or learning bespoke provisioning steps is usually accidental. Platforms should absorb repeated accidental complexity while leaving teams close to the parts of the system they must understand.

Governance and cost awareness should be built into this same operating model. Infrastructure governance becomes harder at scale because the number of teams, environments, services, data stores, regions, and policies increases. Manual review cannot keep up. Policy as code, automated tagging, approved modules, access controls, audit trails, budget thresholds, and evidence dashboards allow the organization to govern continuously.

Cost awareness is now an engineering concern. Elastic infrastructure gives teams the power to create resources quickly. AI workloads can increase spending quickly as well. Finance teams can report cost after the fact, but product teams and platform teams need cost signals while decisions are being made.

Tagging, allocation, forecasting, alerts, and budget guardrails should be part of daily engineering practices.

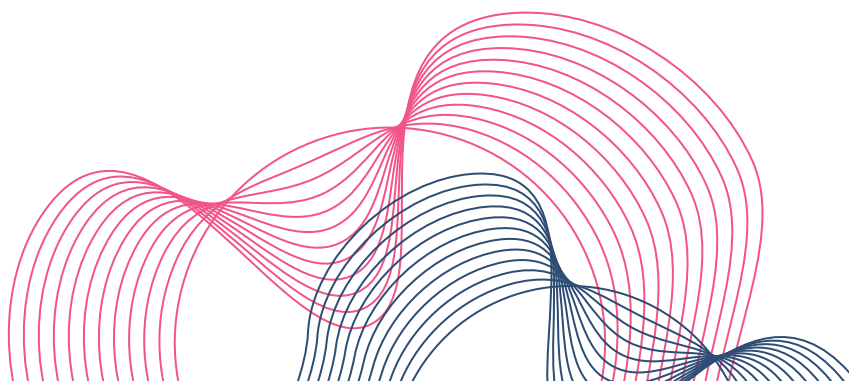
This is where platform, finance, and governance interests meet. The organization should not treat cost control as financial policing. It should treat cost as one of the operational signals of the system. Teams need to know whether a design choice increases cost in exchange for better performance, resilience, or value. Those trade-offs are often valid. Hidden trade-offs are a problem.

At scale, platform governance must include policy, security evidence, approved patterns, and cost visibility. Otherwise, self-service becomes another source of uncontrolled variation.

Useful controls include automatic expiry for unused development and test environments, cost-anomaly routing to the accountable team, and approved service or model tiers. AI workloads should be reviewed for token or accelerator cost, fallback and human-escalation cost, and cost per useful outcome. Shared visibility and showback should normally precede punitive chargeback, because the first objective is better decisions.

Table 9.1 Cost governance as an engineering feedback loop¹⁵

Step	Practical expectation
Attribute	Associate infrastructure, platform, environment, model, and service cost with a product, service, value stream, or accountable owner.
Express unit cost	Use meaningful units such as cost per successful transaction, environment hour, build, deployment, AI request, completed inference, or customer outcome.
Set guardrails	Apply budgets, quotas, approved service or model tiers, environment expiry, scaling boundaries, and cost-anomaly thresholds.
Review trade-offs	Examine cost alongside SLOs, latency, quality, security, resilience, support load, sustainability, and customer value.
Act and verify	Change architecture, scaling, model routing, caching, scheduling, environment lifecycle, or workload design, then verify that the whole outcome improved.



Case B | Harbor Digital Commerce (composite): turning a platform portal into a platform product

Harbor's first internal developer platform was a portal over existing infrastructure tickets. It standardized request forms but did not remove waiting, and teams continued to build local scripts because the platform path did not support their delivery needs. Adoption reports looked healthy because every request passed through the portal, while cognitive load remained high.

The platform team treated product teams as internal customers. It observed onboarding, interviewed teams about repeated work, and selected one high-friction journey: creating a production-ready service. The resulting golden path provided versioned infrastructure, a CI/CD pipeline template, identity and secrets integration, default telemetry, security policies, cost tags, and a documented exception path.

Adoption was not mandated. The platform team measured time to first deployment, repeat support demand, reliability of the platform services, user satisfaction, control failures, and cost visibility. Teams could deviate when the standard path did not fit, but exceptions were visible and reusable needs fed the platform's roadmap.

The trade-off was deliberate. A product-managed platform required sustained funding and support, and it could not satisfy every edge case. It reduced repeated work because teams chose it for usefulness rather than because a central function had renamed a ticket queue.

Transferable lesson: An internal developer platform is a product and self-service capability. Adoption, reduced cognitive load, dependable operation, and safer delivery are stronger evidence than the existence of a portal.

SCENARIO



AI-native infrastructure is subject to the same cost-governance logic. AI-related choices must be evaluated through evidence, not assumed value. A model-serving option may improve latency or output quality, but it should be assessed against cost, reliability, performance, security, support effort, and operational risk.

**AI-RELATED
CONTENT**



Anti-pattern: every team manages its own infrastructure

This anti-pattern usually starts with a reasonable intention. A team needs speed, and the shared process is too slow. Engineers create their own infrastructure scripts, choose their own runtime patterns, configure their own monitoring, and manage their own secrets. The team feels more autonomous at first. Over time, the organization pays for that autonomy through duplication, uneven security, rising cost, and unclear operational responsibility.

The damage appears gradually. One team patches a runtime dependency quickly. Another team remains exposed. One team has useful logs. Another team cannot diagnose incidents. One team has working cost tags. Another team creates unallocated spending. A third team uses an AI service without approved data controls because the official path was not ready. Each local decision has context. The overall system becomes difficult to govern.

The better pattern is shared platform capability with team autonomy inside clear guardrails. Teams should still own their services and make meaningful engineering decisions. The platform should provide reusable foundations, so teams do not need to rebuild common infrastructure, security, observability, and compliance patterns from scratch.

This distinction matters. Central control over every decision creates queues and resentment. Unmanaged variation creates operational risk. A mature platform model provides standards where consistency helps, and extension points where teams need local choice. It also treats exceptions as data. If many teams need the same exception, the platform likely needs to change.

Warning signs:

- Several teams maintain separate infrastructure templates for the same pattern.
- Security and compliance evidence is collected manually for each team.
- Environment drift is discovered during incidents or release windows.
- AI workload patterns appear outside approved data, security, and cost controls.

**KEY
MESSAGE**



9.3 Governed infrastructure and platform capability

Infrastructure maturity is measured by delivery confidence, not by the number of tools in use. A mature infrastructure model allows teams to move quickly because common capabilities are codified, tested, documented, observed, and governed. It does not depend on a small group of people remembering how the platform works.

The maturity path is practical. First, teams move away from manual provisioning and define environments as code. Next, they apply validation, review, and drift detection. Then

they design for elasticity, resilience, and cost awareness. As needs repeat across teams, those patterns become platform capabilities. Finally, the platform becomes a product that supports secure, reliable, AI-ready delivery at scale.

This progression also applies to AI-native environments, where infrastructure discipline must cover generated artifacts, model-serving workloads, data pipelines, and AI-assisted operations, ensuring they remain governed, observable, and reusable as delivery accelerates.

Leaders should fund platform work as product work. Engineering teams should treat infrastructure changes as code. Security and compliance teams should help define controls that platforms can apply automatically. Finance should provide cost signals that engineers can act on. When those responsibilities come together, infrastructure stops being a queue and becomes part of how the organization delivers value.

Infrastructure maturity is visible when environments are codified, secure, observable, cost-aware, recoverable, and delivered through governed self-service rather than ticket queues.

9.4 Chapter summary

Infrastructure is a DevOps capability because every delivery practice depends on consistent, secure, observable, recoverable, and cost-aware environments. IaC, automated validation, and drift management make those environments repeatable and auditable, while elasticity allows capacity to respond to changing demand, including AI workloads.

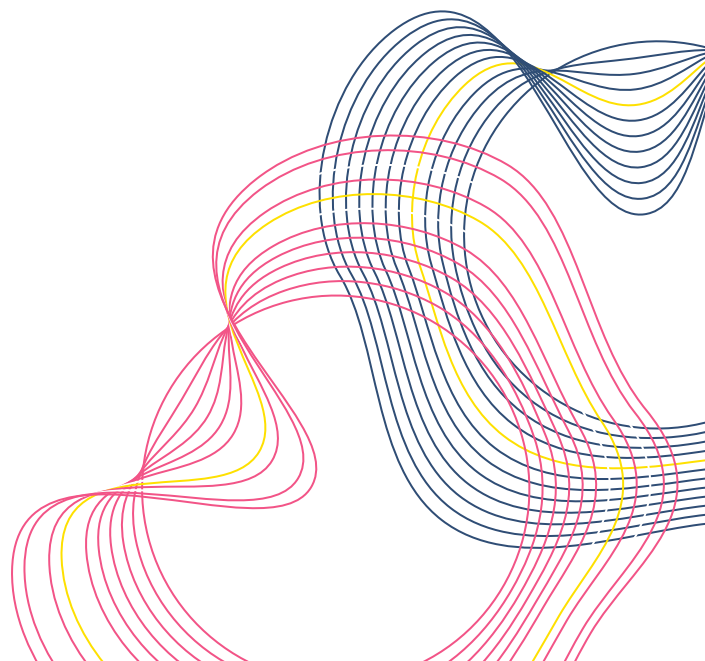
Platform Engineering turns repeated infrastructure and delivery needs into product-managed, governed self-service capabilities. Internal developer platforms should reduce cognitive load without removing product-team ownership. Their value should be demonstrated through adoption, usability, reliability, reduced repeated work, stronger controls, and visible cost outcomes.

DECISION AND ACTION GUIDE

Executive decision: Technology leadership funds the internal developer platform as a product, defines its target users, scope, service expectations, and acceptable variation, and names an accountable platform product manager.

Role-specific actions: The platform product manager conducts discovery and owns the roadmap; platform engineers provide governed self-service; security and finance embed guardrails; product teams adopt the paths or justify exceptions.

Evidence to review: Target-team adoption, self-service completion time, platform reliability, developer satisfaction, repeated-work reduction, cost-allocation coverage, unit-cost signals, budget variance, idle-resource cost, cost-anomaly response, and exception trends.



10.

This chapter establishes security as a first-class DevOps discipline and describes DevSecOps in practical operating terms. This chapter also addresses AI-specific security challenges that now sit inside the delivery system, including model integrity, prompt injection, data governance, and AI supply chain risk.

Continuous Security means security is designed, automated, measured, and improved throughout the delivery system. Specific security checks may occur at particular points, but security is a daily operating discipline that connects design, engineering, platform work, release governance, operations, and incident learning.

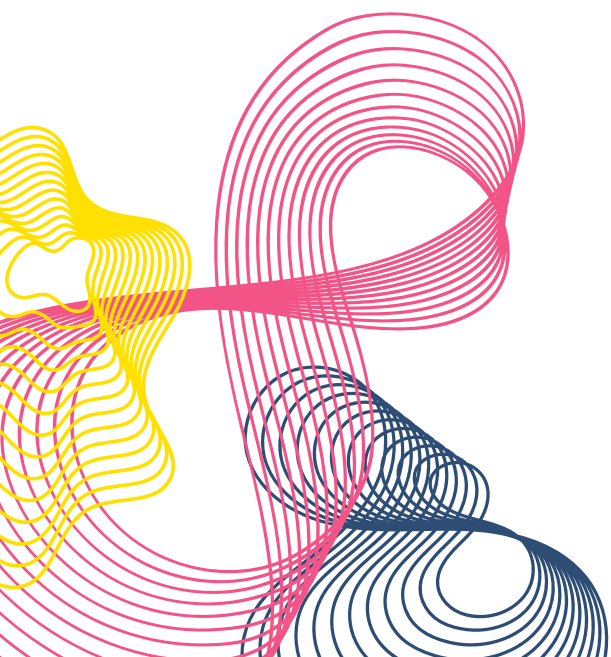
This chapter is most relevant to engineering leaders, security leaders, platform teams, SREs, release managers, architects, risk teams, and practitioners responsible for building secure delivery into everyday work.

10.1 DevSecOps: from security gate to system property

The old pattern treated security as a review at the end of the work. That pattern came from a slower delivery world, where large batches moved through stage gates, and security teams could inspect a release after most decisions had already hardened. Modern delivery moves too quickly for that model. Moreover, AI-assisted delivery increases the volume of artifacts entering the system, making late inspection even less reliable.

Security becomes stronger when it is distributed throughout the system with clear ownership. Engineering teams own the security of what they build and operate. Security specialists define standards, interpret risk, build patterns, support teams, and handle specialist analysis. Platform teams make approved controls easy to consume. Leaders fund the work and remove incentives that may encourage teams to bypass controls.

This does not mean developers are expected to become full-time security engineers. It means the delivery system gives them usable guardrails, fast feedback, secure defaults, and clear escalation paths. Security capability becomes embedded in the workflow, while expert support remains available when the risk or complexity justifies it (NIST, 2022; CISA et al., 2023; Hornbeek, 2025).



Continuous Security

DevSecOps: security embedded within each stage of the value stream



Traditional approach: security controls only at the end of the value stream

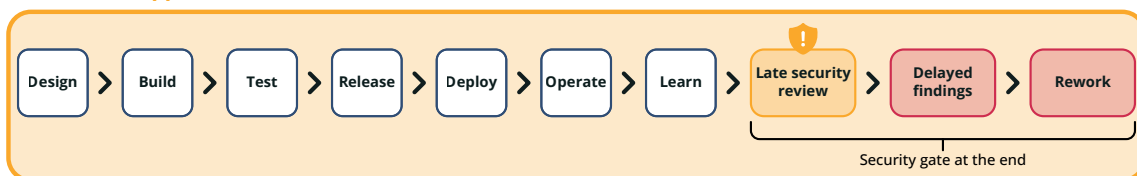


Figure 10.1 Contrasting DevSecOps with traditional late-stage security

Practice note

A team should be able to point to the controls that operate at each step of its delivery flow. If the answer is a meeting, a manual checklist, or a person who always remembers to ask the right question, the control is fragile. Start by making the most common controls visible, then automate those that repeat frequently and create little judgement value.

KEY MESSAGE



Figure 10.1 contrasts security embedded throughout the delivery system with security applied as a late gate. Security is shown across design, build, test, release, deploy, operate, and learn, reinforcing the chapter's treatment of DevSecOps as a system property.

10.1.1 Security requirements and threat modelling at design time

Security begins while the system is still being shaped. Architecture choices create trust boundaries, data flows, identity models, integration paths, and operational exposure.

Threat modelling gives teams a disciplined way to reason about those choices before code, pipelines, and infrastructure make them expensive to change (Deogun et al., 2019; NIST, 2022; Hornbeek, 2025).

The threat landscape is the set of threats, attack paths, vulnerabilities, assets, actors, and potential impacts relevant to the system and its operating environment. It changes as architecture, dependencies, data, users, infrastructure, and AI-enabled capabilities change. Understanding this landscape helps teams decide which risks require prevention, detection, mitigation, transfer, or explicit acceptance.

A useful threat model does not need to be a ceremony that slows delivery. For most teams, it is a structured conversation supported by lightweight artifacts. The team identifies what the system protects, who might misuse it, how data moves, where trust changes, and what controls should exist. It helps the team assess and prioritize security risks and select controls appropriate to the level of risk. The output should feed directly into engineering work: security requirements, abuse cases, logging needs, test cases, and pipeline checks.

Threat modelling feeds security risk management. Teams should assess identified threats according to likelihood, potential impact, exposure, and the effectiveness of existing controls. The response should then be explicit: reduce the risk through design or controls, avoid the risky activity, transfer the risk where appropriate, or accept the residual risk through an authorized decision. The level of evidence and approval should be proportionate to the risk.

Early security thinking matters most where the system handles sensitive data, regulated workflows, identity decisions, financial transactions, safety concerns, or AI-enabled decisions. AI-native systems add another layer of complexity. Teams must decide which prompts, models, datasets, tools, and generated outputs require additional controls. They must also define when human review is mandatory and what evidence should be retained.

Practice note

Threat modelling works best when it follows real design changes. A new API, a new data store, a new AI integration, or a new deployment path should trigger a short review of trust boundaries and failure modes. The output should be work the team can use, such as a test, a logging requirement, a control, or a backlog item.

KEY MESSAGE



10.1.2 Shift-left security and pipeline feedback

Shift-left security moves useful feedback closer to the point where work is created. It is most valuable when it helps teams fix issues while the context is fresh and the cost of correction is low. A secret detected in a local commit, a vulnerable dependency flagged during a pull request, or an unsafe infrastructure setting caught before deployment is easier to handle than the same issue discovered in production.

Common shift-left practices include:

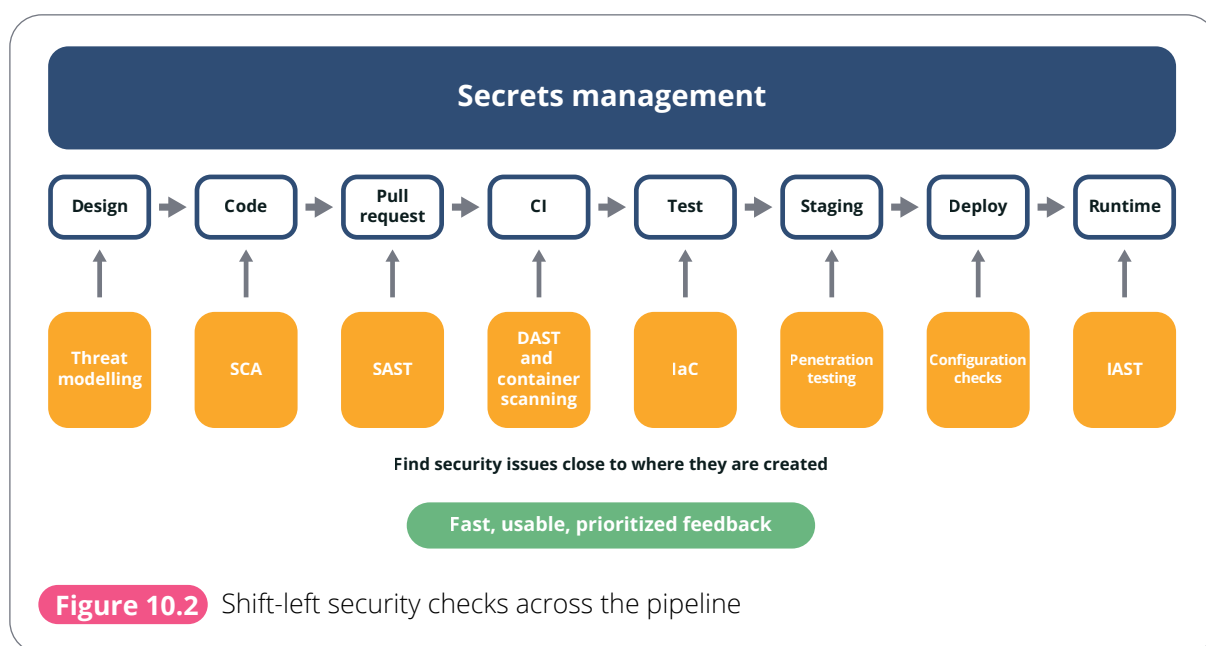
- secrets management
- threat modelling
- dependency scanning with Software Composition Analysis (SCA)
- Static Application Security Testing (SAST)
- Dynamic Application Security Testing (DAST)
- container scanning
- Infrastructure as Code (IaC) scanning
- penetration testing
- configuration checks
- Interactive Application Security Testing (IAST)
- security policy validation

Figure 10.2 illustrates how security checks can be distributed across the pipeline, from design through to runtime. These checks provide earlier and more continuous feedback by combining fast, local checks, such as secrets detection, SAST, and dependency scanning, with later-stage activities, such as Dynamic Application Security Testing (DAST), runtime protection, and penetration testing.

Dynamic testing and runtime security remain important, because some risks appear only when the system is assembled and exercised under realistic conditions.

Good security feedback spans the pipeline, with fast checks early and deeper checks where they add confidence.

The practical problem with shift-left security is noise. Tools that produce long lists of unprioritized findings train teams to ignore security feedback. A mature pipeline classifies findings by severity, exploitability, exposure, service criticality, and business context. It gives developers clear remediation guidance and it separates urgent defects from hygiene work that should enter the backlog.



AI-generated code and configuration make secure delivery controls essential, not optional. Generated artifacts should follow the same review, validation, provenance, and pipeline evidence requirements as human-authored work.

AI-RELATED CONTENT



Practice note

Early feedback must be fast enough to keep developers in the flow of work. A check that takes hours or produces hundreds of vague findings will be bypassed, delayed, or treated as background noise. Security teams should tune tools in collaboration with engineering teams and remove noisy rules that do not change decisions.

KEY MESSAGE



10.1.3 Security as code and continuous compliance

Security as code expresses policies, controls, compliance rules, and guardrails in versioned, reviewable, and testable forms. Policy as code makes the desired control behaviour explicit, and it allows the same rule to be applied consistently across infrastructure, pipelines, access, deployment approvals, and runtime configuration (NIST, 2022; SLSA, 2026).

Figure 10.3 shows how security and compliance controls are versioned, reviewed, tested, automated, and enforced. It maps these controls to infrastructure, CI/CD pipelines, access control, deployment, runtime operations, and audit evidence so that compliance evidence is generated through normal delivery work.

The value is consistency. Manual interpretation drifts as teams, environments, and deadlines change. Codified controls create a shared source of truth.

They can be reviewed like any other engineering artifact, tested before adoption, and improved as risk changes. They also create a better audit trail because policy decisions and evidence are captured as part of the normal workflow.

Continuous compliance applies the same logic to regulatory and governance needs. Evidence should be generated as work moves through

the system, rather than reconstructed later from screenshots, emails, meeting notes, and change tickets. A pipeline can record build results, test evidence, approval decisions, security scans, access checks, configuration baselines, exception approvals, and deployment records.

Control owners still matter. Some policy decisions require judgement, legal interpretation, or formal risk acceptance. Mature DevOps systems reserve human review for exceptional or high-risk conditions and automate the routine evidence collection that once consumed so much time. This approach protects both delivery flow and governance confidence.

Practice note

Compliance evidence has the most value when it is generated by the work itself. A build record, signed artifact, policy decision, access approval, test result, and deployment record can form a defensible evidence chain. This reduces audit effort and gives control owners greater confidence in the operating system.

KEY MESSAGE

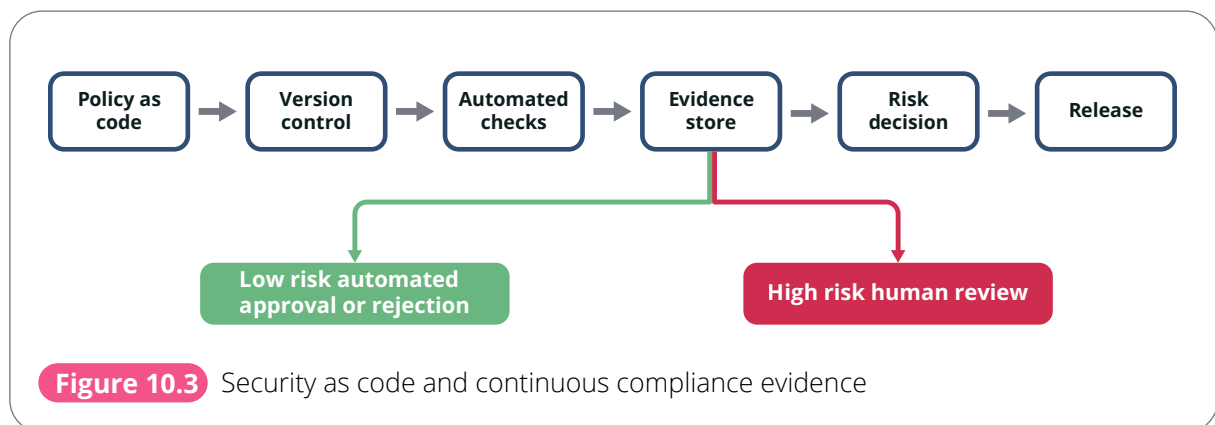


Figure 10.3 Security as code and continuous compliance evidence

10.2 Modern supply chain and AI security

Modern delivery depends on a broad supply chain of source code, dependencies, build systems, and artifacts, and also relies on models, prompts, datasets, plugins, and external services. Security must establish provenance, integrity, approved sources, access boundaries, and validation across this chain, including the new attack paths introduced by AI-enabled systems. This section extends established DevSecOps controls to the assets and dependencies that now shape production behaviour.

10.2.1 Software supply chain security

Software supply chain security protects the chain of artifacts, tools, services, and dependencies that feed the delivery system. Modern software depends not only on open-source libraries, container images, package registries, and build tools, but also on deployment automation, third-party services, and increasingly AI-generated code and AI-assisted tooling. Each component can introduce risk (NIST, 2022; Hornbeek, 2025; OpenSSF, 2026; SLSA, 2026).

A strong supply chain programme starts with provenance. Teams should know where dependencies come from, who approved them, how they were built, and whether they have known vulnerabilities. To protect the software supply chain, organizations must secure both their development assets and their release processes. On the asset side, this involves using software bills of materials (SBOMs), artifact signing, and trusted registries. Simultaneously, the release process must enforce dependency scanning, build isolation, and controlled package promotion.

The build pipeline deserves particular attention. Attackers often target the systems that build and deploy software because those systems are trusted by default. Build agents, credentials, signing keys, package publishing rights, and deployment tokens must be protected with the same seriousness as production systems. Access should be limited, logged, rotated, and reviewed. The control set should culminate in an evidence-based promotion decision. Table 10.1 provides a baseline rather than a universal assurance level; higher-risk products and services may require stronger provenance, independent verification, segregation, or formal approval.

AI expands the software supply chain beyond code and dependencies to include models, prompts, datasets, plugins, and third-party AI services. Continuous Security must therefore ensure that these assets are approved, traceable, scanned where possible, governed by clear use restrictions, and assigned to accountable owners.

AI-RELATED CONTENT

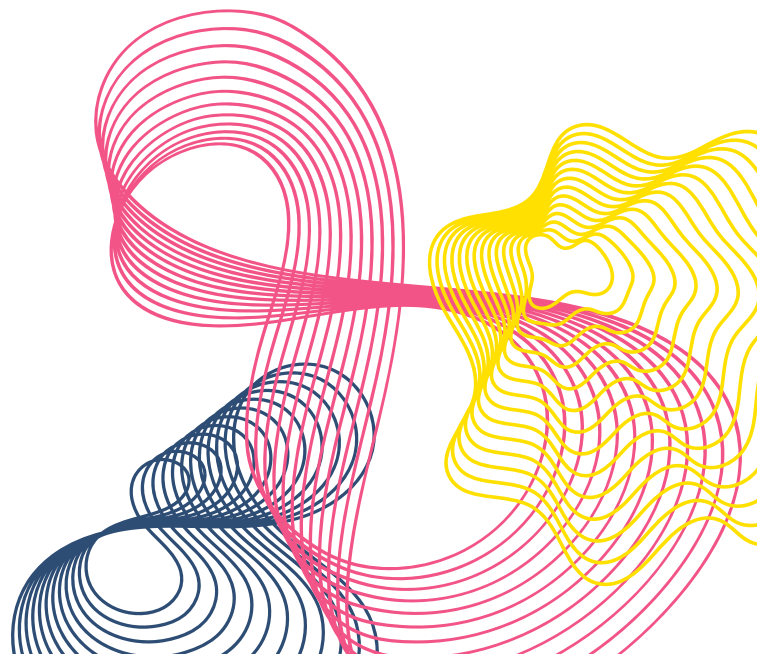


Table 10.1 Minimum software-supply-chain evidence before promotion¹⁶

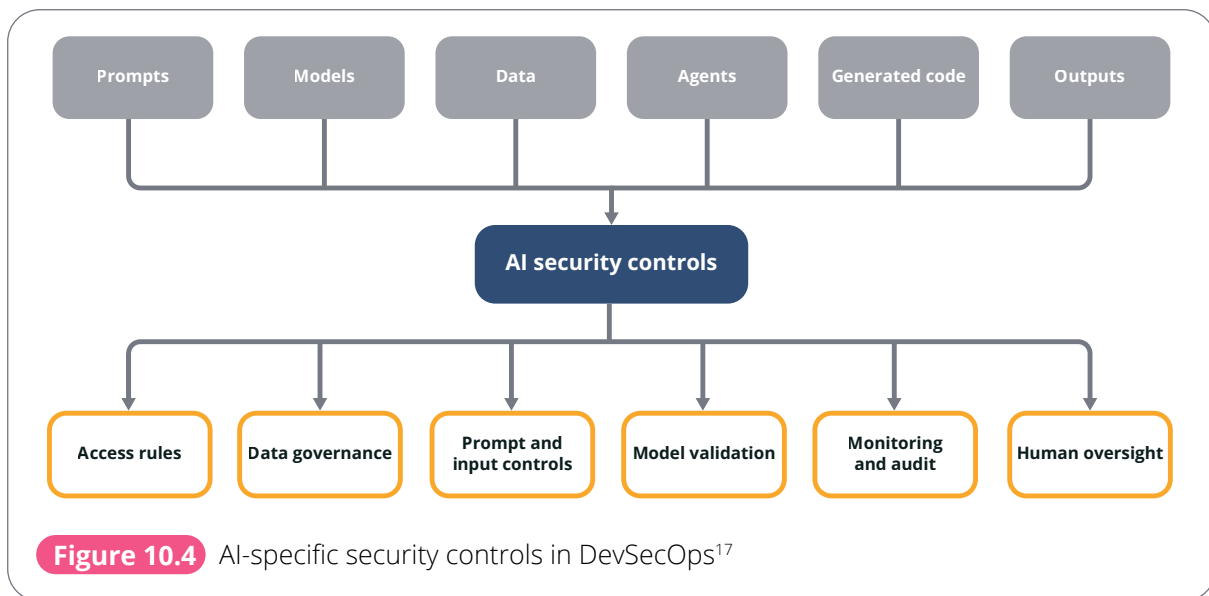
Control area	Baseline expectation	Evidence retained
Source and dependency control	Use approved sources, pinned versions, integrity checks, and dependency policies.	Dependency manifest, source record, integrity results, policy decisions, and Software Composition Analysis (SCA) results.
Build integrity	Use a controlled or isolated build, a least-privileged build identity, and protected credentials and signing keys.	Build record, builder identity, access log, and key-use record.
Component transparency	Generate and retain an SBOM for each release candidate.	Versioned SBOM linked to the artifact digest.
Provenance and attestation	Record source, inputs, builder, and build process, then verify the provenance against defined expectations.	Signed provenance or equivalent attestation and verification results.
Artifact integrity	Sign artifacts and verify signatures and digests before promotion.	Signature, digest, signer identity, and verification results.
Registry and promotion	Use trusted registries and approved, traceable promotion paths.	Registry history, access record, and promotion record.
Vulnerability and policy decision	Evaluate vulnerabilities, licensing obligations, policy requirements, and exceptions before promotion.	SAST, DAST, SCA, and policy results plus a named, time-bound exception where required.
AI-related assets	Record models, prompts, datasets, plugins, generated code, and third-party AI services that influence the release.	Asset version, provenance, approval, validation evidence, restrictions, and owner.

10.2.2 AI-specific security: model integrity, prompt injection, and data governance

AI-specific security is now part of **Continuous Security**. AI-assisted development and AI-enabled services introduce assets and attack paths that traditional application security programmes may not fully cover. Prompts, models, datasets, embeddings, and retrieval indexes, as well as agents, generated code, and tool connections all need controls. These controls extend existing security practice rather than creating a separate security lane (NIST, 2024; Hornbeek, 2025; OWASP Foundation, 2025a). Figure 10.4 summarizes the AI assets and outputs requiring protection and the principal security controls applied across them.

Prompt injection is one of the most visible examples. An attacker may try to manipulate an AI system into ignoring instructions, exposing hidden context, calling unauthorized tools, or producing unsafe outputs. Defensive design should include not only input handling, prompt filtering, and tool-use restrictions, but also retrieval controls, output validation, and monitoring for abuse patterns. Teams should also apply least-privilege access to AI tools and retrieval systems to limit the impact of compromised prompts or model behaviour.

Model integrity is another concern. Teams need approved model sources, controlled model versions, documented deployment records, and monitoring for unexpected behaviour. Training data and retrieval content require governance because they can affect decisions, recommendations, and generated responses long after code has been deployed. Changes to models, datasets, and retrieval content should therefore be traceable and subject to appropriate review before reaching production.



Practice note

AI controls should be designed around actual use cases. A coding assistant, a customer-facing chatbot, a fraud model, and an autonomous remediation agent do not carry the same risk. The control set should reflect the data involved, the actions the AI can influence, and the consequences of a wrong or manipulated output. Controls should become stronger as access, autonomy, and potential impact increase. Where AI can influence production systems or high-impact decisions, teams should define explicit authorization boundaries, monitoring, and recovery mechanisms.

KEY MESSAGE



Data governance is especially important. Sensitive information can enter prompts, logs, traces, training sets, embeddings, or human review queues if teams do not design controls deliberately. AI workflows should define:

- what data may be used
- what must be redacted
- where logs are retained
- who may access them
- how incidents are handled.

The practical rule is clear: AI changes the security surface, so **DevSecOps** must expand its control model accordingly.

10.3 Continuous Security operations and improvement

Security capability is incomplete when findings remain in scanners, dashboards, or incident reports. Vulnerability management, operational feedback, champion networks, and meaningful measures should convert security evidence into owned remediation, better tests, stronger platform guardrails, and improved design. This section focuses on the operating loop that allows security to learn and improve continuously.

10.3.1 Vulnerability management in a Continuous Delivery environment

Vulnerability management changes when teams release frequently, because periodic scanning and quarterly remediation planning cannot keep pace with systems that change every day. Vulnerabilities must be discovered, triaged, and assigned as part of the normal delivery flow. Continuous vulnerability management should therefore extend beyond discovery and triage to include remediation and verification. While this process also requires exception handling and learning, the weakness in many organizations is not scanning; it is the absence of accountable follow-through (NIST, 2022; Hornbeek, 2025; OWASP Foundation, 2025b).

The hardest part is prioritization. A raw count of vulnerabilities tells leaders very little. To prioritize effectively, teams need to analyse

factors across two dimensions. On the technical side, they need to understand exploitability, exposure, asset criticality, and compensating controls. From a business perspective, they need to evaluate service ownership, customer impact, and regulatory context. A low-severity issue on an internet-facing identity service may deserve faster action than a high-severity finding on an isolated development component.

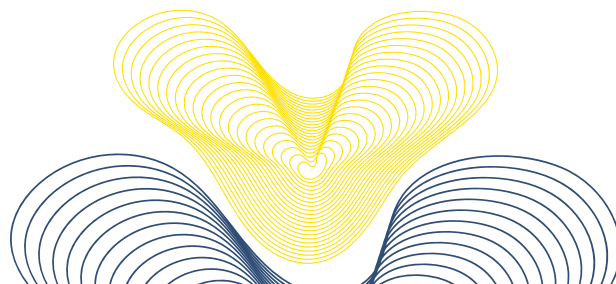
Remediation work should enter the same backlog and planning system as other product and reliability work. Emergency fixes need a fast path with clear authority. Less urgent findings need service-level expectations, ownership, and visibility. The security team can help classify risk, but the team that owns the service must be able to act.

Verification closes the loop. A vulnerability is not resolved simply because a ticket has changed state. The fix must be built, tested, deployed, and confirmed. In AI-enabled systems, teams may also need to verify that prompts, model versions, retrieval sources, or agent configurations have been updated safely. This is **the same** engineering discipline applied to security work.

Practice note

Vulnerability management improves when it is connected to ownership. Every finding should map to a service, a team, a risk decision, and a target response time. Findings without ownership become inventory, and inventory without action creates a false sense of control.

KEY MESSAGE



10.3.2 Security operations feedback into DevOps

Security operations produce signals that should improve engineering work. These signals include alerts, incidents, attack patterns, and penetration test findings. They also involve threat intelligence, vulnerability trends, and runtime telemetry. Together, they reveal how systems behave under pressure. However, they lose value when they remain inside dashboards or incident reports (Adkins et al., 2020; Hornbeek, 2025).

A mature feedback loop converts security operations findings into backlog items, test cases, and detection rules, as well as platform guardrails, design changes, and policy updates. If an incident reveals weak secrets handling, the result should include improved secret scanning, platform defaults, rotation procedures, and developer guidance. If attackers repeatedly target an API pattern, architecture and testing practices should change.

Practice note

Security operations should not be treated as a separate world that only responds to alerts. The best signals from operations become better defaults, better tests, better detections, and better architecture decisions. This is where security becomes a learning system rather than a permanent emergency response function.

KEY MESSAGE

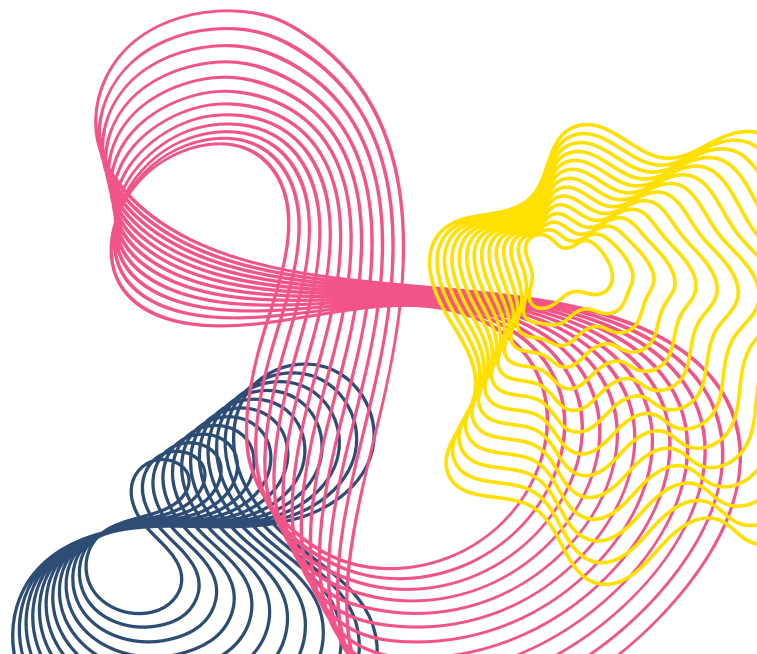


Security incidents should be reviewed with the same learning discipline as reliability incidents. Blame slows learning and hides the real causes.

The useful questions are about conditions:

- Why did the control fail?
- Why did detection arrive late?
- Why was recovery difficult?
- What should the delivery system do differently next time?

This feedback loop connects Continuous Security to Observability. Logs, metrics, traces, and deployment metadata help teams reconstruct what happened, while identity events, access records, and business impact signals help them decide which controls deserve investment. Production security telemetry should shape the next design and the next platform improvement. To achieve this, security operations should feed delivery decisions with evidence from incidents, detections, exploit attempts, vulnerabilities, misconfigurations, and control failures. The goal is to improve design, tests, platform guardrails, and runbooks.



Case A – Northstar Financial Services (composite): responding to software-supply-chain risk

A critical vulnerability was disclosed in a widely used open-source dependency. Northstar's nightly scanner found most affected services, but several teams had built artifacts from different registries and could not prove which dependency version had entered production. One high-value service also could not accept the recommended upgrade without a breaking interface change.

Northstar used the event to strengthen the delivery system. Teams routed dependencies through approved registries, generated software bills of materials, signed build artifacts, captured build provenance, and evaluated policy before deployment. Security findings entered the product backlog with service criticality, exploitability, exposure, and compensating controls rather than an undifferentiated list.

An AI assistant proposed version changes and summarized likely exposure, but the security and product teams reviewed the recommendation. Most services moved through the normal governed path. The incompatible service used an emergency change path with a time-bound exception, network restrictions, additional monitoring, and a named owner for the follow-up remediation.

The incident did not produce a claim that every vulnerability could be eliminated immediately. It produced better traceability, faster identification of affected artifacts, and a clearer decision path for risk that could not be removed at once.

Transferable lesson: Software-supply-chain security is an evidence and decision system. Provenance, approved sources, signed artifacts, risk context, exception ownership, and operational monitoring matter as much as the scanner.

SCENARIO



10.3.3 Security champions and metrics for continuous security

Security champions help scale security knowledge across teams. A champion is usually a practitioner embedded in a product, platform, data, or operations team who receives additional support, training, and access to security specialists. The role helps translate security expectations into everyday engineering decisions. Security champions should help teams interpret findings, improve patterns, and raise risks early rather than becoming informal gatekeepers.

A good champion model avoids two bad outcomes. It does not make the champion the only person responsible for security within the team. It also does not turn the central security group into a ticket queue for routine questions. Security champions help teams understand patterns, review designs, raise risks early, support threat modelling, and guide remediation. The central security team provides standards, expertise, coaching, and escalation.

Security measurements should support the same intent. Useful metrics show whether the system is preventing, detecting, responding, remediating, providing evidence, and learning effectively. Examples include time to remediate, vulnerability age, policy pass rate, secrets detection rate, dependency freshness, security test effectiveness, incident recurrence, and evidence of completeness.

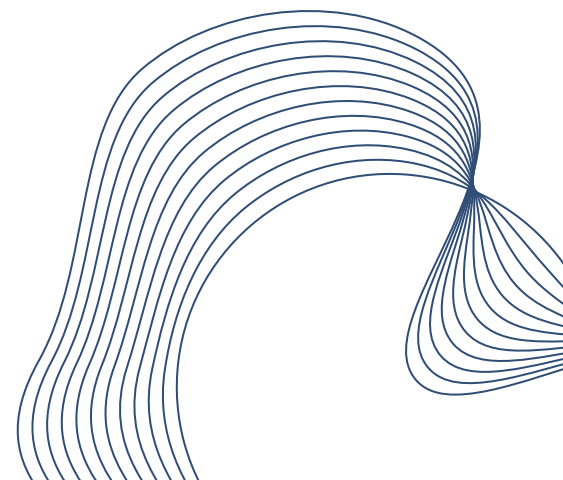
Practice note

A champion network needs time, recognition, and practical support. It fails when champions receive responsibility without authority or training. It succeeds when the central security team equips champions with patterns, examples, escalation paths, and a community where hard questions can be handled quickly.

KEY MESSAGE



Metrics can damage trust if leaders use them to rank or punish teams. Raw vulnerability counts are especially misleading because they reflect discovery rate, scanning coverage, asset count, and reporting culture as much as security posture. Metrics should guide conversations about risk and improvement. They should help leaders fund the right platform work, reduce noisy findings, and make secure delivery easier for teams.



Anti-pattern: “Security reviews the release every quarter”

The quarterly security review pattern belongs to a slow, batch-heavy delivery model. It assumes that security can inspect a large release after most decisions have been made and still provide meaningful control. In continuous delivery environments, that assumption breaks down quickly. Risk accumulates between reviews, findings arrive late, and remediation competes with release pressure.

The better pattern is continuous, evidence-based, and risk-aware security. Low-risk changes should pass through automated controls with evidence captured along the way. High-risk changes should trigger human review when the system detects conditions that justify it. This preserves judgement while reducing the queue for routine approvals.

This applies equally to AI-native changes, where prompts, models, retrieval sources, permissions, and generated artifacts may all affect production behaviour. Continuous Security keeps these changes visible, governed, and accountable as they move through normal delivery flow.

Security reviews the release every quarter. The organization treats security review as a scheduled inspection event rather than an embedded delivery capability. Teams wait for approval, security teams become overwhelmed by late findings, and leaders receive a false sense of confidence because the process exists even when the control is weak.

KEY MESSAGE



Practice note

The purpose of the anti-pattern is not to eliminate human judgement, but to reserve it for decisions that require it. Routine changes should produce evidence automatically, while high-risk changes should reach the right people with enough context to make a sound decision. This approach reduces unnecessary manual effort while keeping meaningful oversight in place. It also helps teams apply consistent controls without slowing down low-risk delivery.

KEY MESSAGE



DECISION AND ACTION GUIDE

Executive decision: Security and technology executives should set risk appetite, minimum software-supply-chain controls, exception authority, remediation expectations, and an accountable owner for each material control.

Role-specific actions: Security specialists should define threat models and guardrails; engineering teams should own secure implementation and remediation; platform teams should automate provenance, signing, software bills of materials, policy checks, and evidence collection.

Evidence to review: SBOM coverage, provenance attestations, signature and digest verification, builder identity, policy results, vulnerability age, dependency-exception age, remediation evidence, and incident-learning actions.

10.4 Chapter summary

Continuous Security turns security from a late inspection activity into an operating discipline. It connects design, engineering, and platform capabilities with release governance, production telemetry, and learning. This is how security keeps pace with delivery without becoming invisible or careless.

AI-native delivery broadens the security surface. DevSecOps must therefore extend normal controls, evidence, validation, and monitoring to AI-generated artifacts, AI-assisted decisions, and the models, prompts, datasets, agents, and toolchains that support them.

Security is strongest when it is built into everyday work. Threat modelling moves risk discovery earlier. Shift-left controls provide fast feedback. Security as code and continuous compliance make controls repeatable. Supply chain security and AI-specific security extend the control model to modern delivery. Metrics and champions help scale capability without turning security into a bottleneck.

This chapter defines observability as a feedback capability that helps the DevOps system detect, understand, and respond to changing conditions. It distinguishes monitoring from observability, explains the telemetry foundations that make diagnosis possible, and shows how production signals influence reliability, product, architecture, security, and AI governance decisions.

DevOps is incomplete when software merely reaches production. It becomes complete when production evidence informs the next decision.

Monitoring and observability are related but distinct capabilities. Monitoring tracks known conditions through predefined signals, thresholds, and expected states. Observability uses telemetry and system context to help teams investigate behaviour, including conditions they did not anticipate in advance. Monitoring helps teams detect that something is wrong; observability helps them understand what is happening and why. Together, they provide the operational visibility needed to detect, diagnose, learn from, and improve system behaviour.

This chapter is most relevant to engineering leaders, SRE teams, platform teams, product owners, operations teams, security leaders, and practitioners responsible for making production behaviour visible and useful.

11.1 Observability foundations

Observability begins with a clear distinction between detecting known conditions and explaining unexpected behaviour. It depends on deliberate instrumentation and correlated evidence across logs, metrics, traces, events, deployment context, business signals, and user impact. This section establishes the foundations for turning production evidence into trustworthy and useful information.

11.1.1 Observability as the DevOps nervous system

Production is the only environment where the whole system is present. The real users are there. The real networks are there. The real data shapes, traffic spikes, third-party dependencies, feature flags, security controls, and operational habits are present as well. A delivery system that stops learning when code reaches production is running with half its senses removed (Beyer et al., 2016; Hornbeek, 2024; Majors et al., 2026).

Observability is the mechanism by which production speaks back to the organization. It turns runtime evidence into learning. That evidence may arrive as logs, metrics, traces, events, deployment metadata, user behaviour, incident records, cost signals, security alerts, or AI behaviour indicators. The important question is whether the

Continuous Monitoring and Observability

organization can interpret these signals and use them to improve the next change.

A mature operating model behaves like a self-correcting control system. Production emits telemetry, teams interpret the signals, decisions lead to change, and validation checks whether the change achieved the intended outcome. The results then feed back into production, where new telemetry begins the next cycle of learning. The loop is not ceremonial. It is how the organization continually corrects itself, ensuring that production signals influence the next change. This is why observability belongs at the core of DevOps rather than in an operations appendix (see Figure 11.1).

In AI-enabled systems, a chatbot may remain available, fast, and cheap while answer quality degrades after a prompt change. A recommendation or fraud model may keep operating while user behaviour or attacker patterns shift. Therefore, AI broadens observability beyond availability and cost to include quality, drift, behaviour, and business impact.

AI-RELATED
CONTENT



Consider a checkout service that appears healthy on a traditional dashboard. CPU is normal. Uptime is green. Hosts are responding. Yet customers abandon carts at a higher rate after a release. The useful evidence sits across several signals. Traces show a slow call to payment integration. Logs show retries only for one region. Metrics show a rise in tail latency rather than average latency. Release metadata shows a feature flag change. Business telemetry shows checkout abandonment. None of those signals alone tells the full story. Together, they explain the issue.

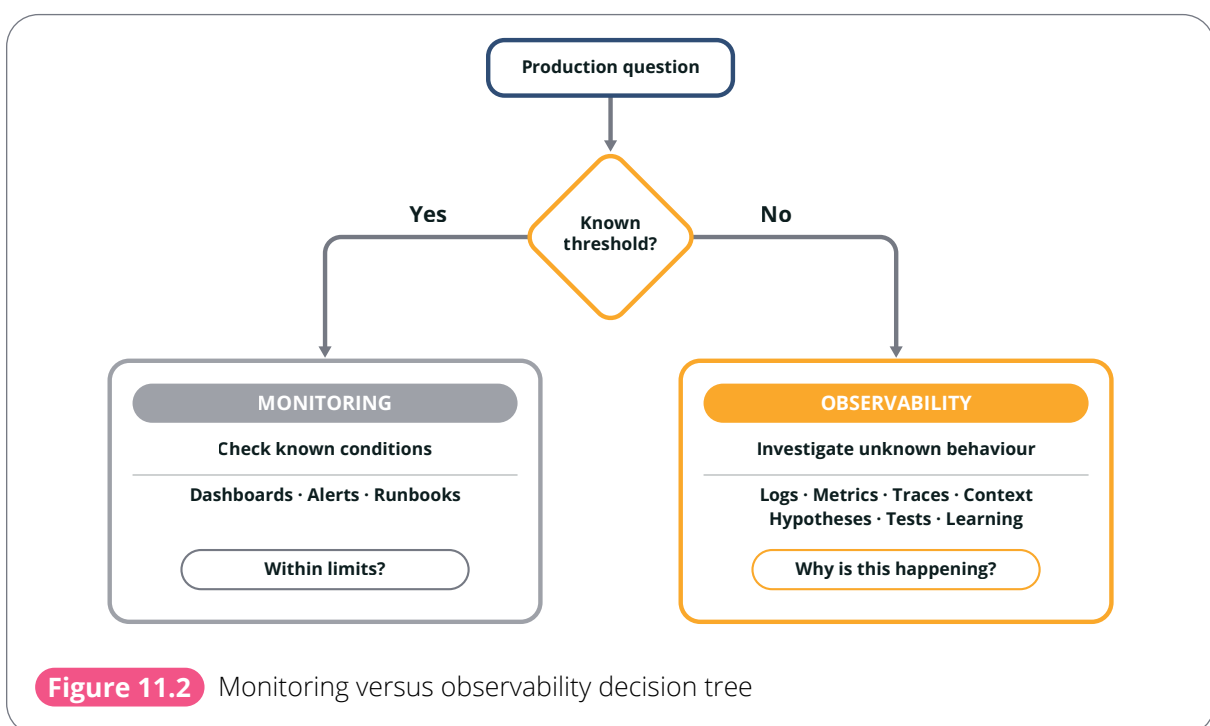
11.1.2 Monitoring and observability: known conditions and unknown behaviour

The distinction between monitoring and observability becomes practical when teams use production evidence to investigate system behaviour. Monitoring is the repeated observation of systems, services, practices, pipelines, or other entities to detect events and keep their current status known. It answers questions about known conditions:

Is the disk almost full? Has the error rate crossed the agreed threshold? Is request latency outside the expected range? These are valuable questions, and teams need them answered quickly (Majors et al., 2026; PeopleCert, 2026d). Monitoring therefore checks whether the system is operating within expected limits, using dashboards, alerts, and runbooks to identify and respond to predefined conditions.

Observability goes further by helping teams understand the internal state and behaviour of a system from its external outputs and signals, especially metrics, events, logs, traces, and release context. It supports investigation when the condition is unexpected and the team does not yet know which threshold matters. Observability asks why the behaviour is happening, where the failure path travels, which users are affected, and what changed before the symptoms appeared. This approach helps teams examine unexpected behaviour through hypotheses, tests, and correlated evidence, rather than simply detecting that something has gone wrong. Figure 11.2 summarizes this distinction as a simple decision tree.

The difference matters because modern systems fail in ways that do not respect organizational boundaries. A problem may appear only for users on one mobile application



version, in one region, after one feature flag change, when a third-party dependency is slow. A dashboard that shows service-level uptime may be correct and still miss the user-visible failure. In that situation, more dashboards do not create better observability. Better correlation, context, and evidence that teams can investigate are needed.

A simple disk space alert is monitoring. Disk usage reaches 90 per cent, the alert fires, and the runbook explains the response. A regional latency increase after a feature flag change is an observability problem. The team must correlate request traces, release metadata, logs, client versions, and regional infrastructure behaviour. The investigation begins with a question, then forms a hypothesis, tests it against evidence, and learns from the result.

DevOps teams cannot improve what they cannot see. They cannot move safely at higher speeds without trustworthy feedback. Production evidence must be close enough to development decisions to influence what the team builds, tests, refactors, automates, funds, or removes next.

11.1.3 Telemetry foundations: instrumenting the system

Observability is not magic. It depends on telemetry: metrics, events, logs, traces, and other signals emitted by systems, delivery pipelines, and AI-enabled components for monitoring, analysis, and decision support. Useful telemetry starts with deliberate instrumentation. Weak telemetry relies on whatever logs or counters happen to exist after the system has already failed (OpenTelemetry, 2026).

Telemetry moves from sources such as applications, application programming interfaces (APIs), infrastructure, CI/CD pipelines, and AI systems through instrumentation (including software development kits), collection, processing, and storage, before it is used for analysis and decision-making. To make

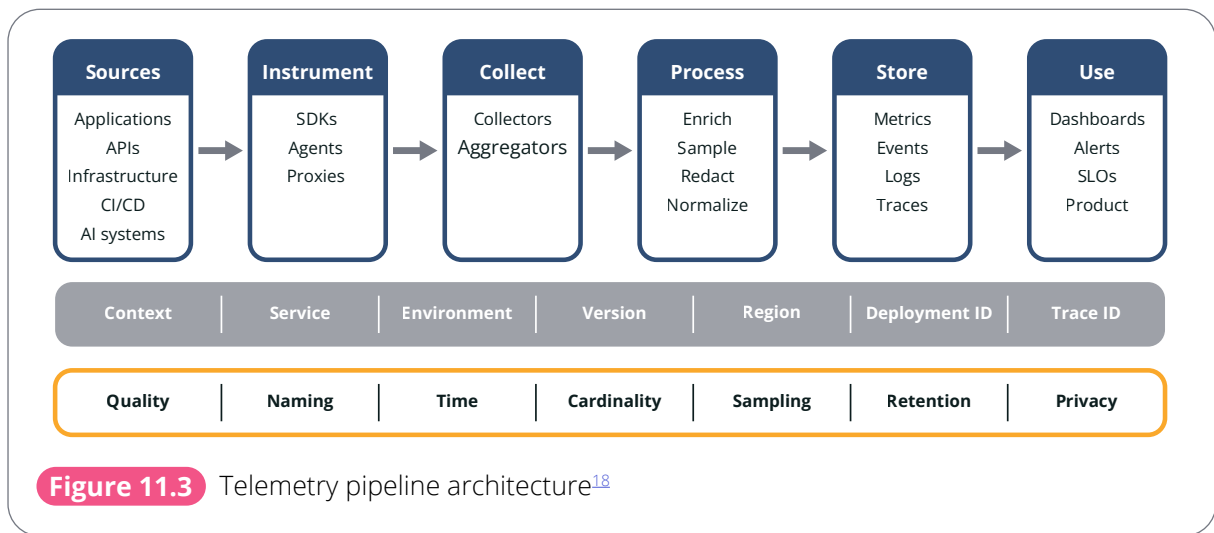
telemetry trustworthy and useful, signals should include consistent context such as service, environment, version, region, deployment ID, flags, and trace IDs. They should also be governed by quality controls covering naming, timestamps, cardinality, sampling, retention, and privacy. Figure 11.3 illustrates this telemetry pipeline and the context and quality controls that support it.

Instrumentation should cover more than the application's runtime to provide observability across the entire delivery and operating environment. At the application level, instrumentation records request paths, errors, business events, and domain behaviour, helping teams understand how the software functions in practice. Infrastructure instrumentation complements this by monitoring resource saturation, network behaviour, environment state, and overall platform health.

Observability is also needed throughout the software delivery lifecycle. CI/CD and deployment instrumentation tracks builds, tests, approvals, versions, deployment IDs, rollback events, and feature flag changes, creating a record of how changes move into production. User experience instrumentation focuses on what people encounter when using the system, capturing real user behaviour, client-side performance, abandonment, and task completion. Business event instrumentation links technical behaviour to value impact, connecting operational performance with business outcomes.

OpenTelemetry has become an important vendor-neutral framework for generating, collecting, processing, and exporting telemetry data such as traces, metrics, and logs. Its practical value is not that it solves every observability problem. Rather, it gives teams a common instrumentation model and reduces the cost of moving telemetry across tools and platforms.

Metadata turns raw signals into evidence. Every service should emit enough context to make investigation possible. Ordinary details, such as service name, environment, version, region, deployment ID, feature flag state, customer segment where privacy-safe, trace ID, and correlation ID, become essential during an incident.



A log line without a request ID is weak evidence. A metric without a service version cannot identify whether a deployment caused degradation. A trace without business context may reveal latency while hiding user impact.

Telemetry quality needs the same care as product quality. Therefore, teams should standardize naming, control cardinality, keep timestamps accurate, define sampling strategies, set retention policies, and redact sensitive data before it becomes a problem. Cardinality deserves special caution. A metric label that includes unbounded user IDs, session IDs, or free-form prompts can turn an observability platform into an expensive noise generator.

Telemetry should be treated as part of the product, since it is part of the production interface between the system and the people who operate, improve, secure, and govern it. Teams that defer telemetry design usually pay during incidents, when the missing signal cannot be reconstructed.

11.1.4 Logs, metrics, and traces: correlating the evidence

The three classic pillars of observability are logs, metrics, and traces. The phrase is useful as a teaching model, although observability is broader than these three data types. Events, profiles, deployment metadata, feature flag

records, user analytics, cost data, and AI-specific telemetry all add context. Still, logs, metrics, and traces remain the foundation for most production investigations (Majors et al., 2026; OpenTelemetry, 2026).

Logs are records of events. They explain what happened at a point in time. Structured logs are far more useful than free-form strings because they can be queried, filtered, correlated, and aggregated. A useful log event commonly includes timestamp, service, severity, request ID, account or session identifier where appropriate, error code, deployment version, and feature flag state. Logs are especially useful for discrete events, exception analysis, audit trails, and reconstructing the narrative of a failure.

Logs carry risk. They can be noisy, expensive, inconsistent, and dangerous when they contain sensitive data. Poorly structured logs create a kind of ‘string soup’. They may appear readable to a person skimming one line but fail as evidence because they cannot be queried reliably. Mature teams standardize their structure and apply privacy controls early.

Metrics are numeric measurements over time. They show trends, rates, distributions, and thresholds. Counters, gauges, histograms, and percentiles each answer different questions. The four golden signals from SRE practice, latency, traffic, errors, and saturation remain a useful starting point for user-facing systems. Tail latency deserves particular attention because averages often hide the users who have the worst experience.

Metrics should include business behaviour where possible. Conversion rate, checkout success, search relevance, support contacts, churn indicators, and AI escalation rate often explain why a technical change matters. A latency spike is useful. A latency spike that correlates with abandoned checkout attempts is more useful.

Traces show a request path through distributed systems. A trace is built from spans. Each span records one unit of work, with attributes, timing, status, and parent-child relationships. Traces are essential in microservices, serverless systems, event-driven systems, and AI pipelines because failures often emerge across service boundaries. A checkout trace may cross the API gateway, authentication service, recommendation service, payment service, database, and an external API. The trace shows where time was spent and how causality moved through the system.

Practical value comes from their correlation. Logs, metrics, and traces become powerful when they share correlation identifiers, service metadata, deployment versions, environment context, and business context. Without this shared context, the signals remain fragments. A trace identifies where a request slowed. Logs explain what happened inside the relevant span. Metrics show whether the behaviour is isolated or systemic. Business telemetry shows whether users were harmed. Together, these signals support a diagnosis that is operationally useful.

11.2 From telemetry to operational decisions

Telemetry creates value only when it changes an operational, product, or engineering decision. AIOps, feedback loops, SLIs, SLOs, and error budget practices help teams interpret signals, prioritize action, and balance change with reliability. This section explains how evidence moves from observation to accountable decisions and verified improvement.

11.2.1 AIOps: from signal reduction to bounded operational action

AIOps applies AI and machine learning to operational telemetry, service context, and workflows to reduce noise, detect and correlate conditions, support diagnosis and prediction, recommend responses, and execute bounded and reversible actions where explicitly authorized. It extends observability and operations; it does not replace SRE, incident management, or human accountability (Dang et al., 2019; Chen et al., 2025).

Table 11.1 AIOps use, practical purpose, and required control¹⁹

AIOps use	Practical purpose	Required control
Noise reduction and correlation	Group duplicate or related alerts and connect them to services, changes, dependencies, and user impact.	Preserve source signals and measure both noise reduction and missed-event risk.
Detection and prediction	Identify anomalies, saturation, degradation, or emerging failure conditions.	Monitor false positives, false negatives, drift, and changing baselines.
Triage and diagnosis support	Assemble deployment history, topology, telemetry, incidents, and plausible hypotheses.	Link conclusions to source evidence, expose alternatives, and label uncertainty.
Response recommendation	Suggest a runbook, rollback, escalation, capacity change, or diagnostic step.	Keep accountable human review for material operational decisions and record acceptance or override.
Bounded automated response	Execute predefined, low-impact, reversible action within an approved class.	Use allow-listed actions, least privilege, audit logs, monitoring, post-action verification, rollback, and a kill switch.

The control boundary changes as AIOps moves from analysis to action. Read-only correlation can normally operate with ordinary data, privacy, and quality controls. Recommendations require links to supporting evidence and, where relevant, indications of uncertainty. Production actions require allowlisted targets, least privilege, preconditions, post-action verification, recovery, monitoring, and an accountable owner. AIOps is therefore not a synonym for autonomous operations.

A small starter set of measures is sufficient: alert-volume reduction paired with missed-event rate; time to assemble useful incident context; recommendation acceptance and override rates; automated-action success and rollback rate; reduction in on-call toil; and cost per useful operational outcome. AIOps has failed if it suppresses noise while hiding incidents or automates work that responders must then repair.

Teams should begin with read-only correlation and diagnosis, validate performance against known incidents and controlled failure exercises, and expand authority only when accuracy, blast radius, recovery, and audit evidence are acceptable. The Harbor case later in this chapter applies the same principle: AI-assisted correlation supports responders, but evidence and accountable judgement govern mitigation.

11.2.2 Feedback loops: production signals to development decisions

Telemetry becomes valuable when it changes decisions. Passive visibility is the ability to look at the system. Active feedback is the ability to change the system because of what has been observed. Many organizations have the first and call it observability. The second is what DevOps needs.

Feedback loops operate at several levels. Immediate operational feedback supports alerting, incident response, rollback, and mitigation. Team feedback supports defect

analysis, retrospectives, backlog refinement, test improvement, and toil reduction. Product feedback supports decisions about adoption, abandonment, conversion, user paths, and feature value. Architecture feedback exposes recurring bottlenecks, service coupling, scaling constraints, and failure patterns. Business feedback connects reliability, speed, cost, trust, and customer experience. Together, these levels show how production evidence flows back into development and team learning, along with product decisions, architecture improvement, and business outcomes. Figure 11.1 above illustrates this production-to-development feedback path.

Production telemetry should become part of the ordinary work of the organization. Sprint planning should consider incidents, operational toil, recurring defects, and service-level risk. Architecture reviews should include traces, saturation trends, dependency behaviour, and deployment failures. Incident reviews should produce changes to code, tests, runbooks, dashboards, and platforms. Capacity planning should use real demand patterns rather than hopeful forecasts. Product roadmap conversations should include user behaviour and reliability evidence, not only feature requests.

The pattern is familiar: repeated database saturation becomes an architecture investment. Feature flag telemetry shows low adoption and a high error rate, so the roadmap changes. Trace data reveals excessive service hops, so a team simplifies the service boundary. Alerts show repeated manual remediation, so the platform team automates the recovery path. None of this happens because data exists. It happens because the organization has a disciplined path from evidence to decision.

Ownership is important. Developers own the operability of the code they build. Operations and SRE teams help provide platforms, observability patterns, guardrails, and reliability coaching. Product teams consume telemetry as evidence for product decisions. Security teams use production signals to improve controls and detection. Leadership funds the improvements that telemetry reveals.

When telemetry has no owner, it becomes another reporting artifact.

A useful feedback loop is short enough to matter and disciplined enough to avoid overreaction. Teams should not change direction because of every noisy signal. They should form hypotheses, check the evidence, make focused changes, and validate whether those changes helped.

11.2.3 SLI, SLO, and error budget

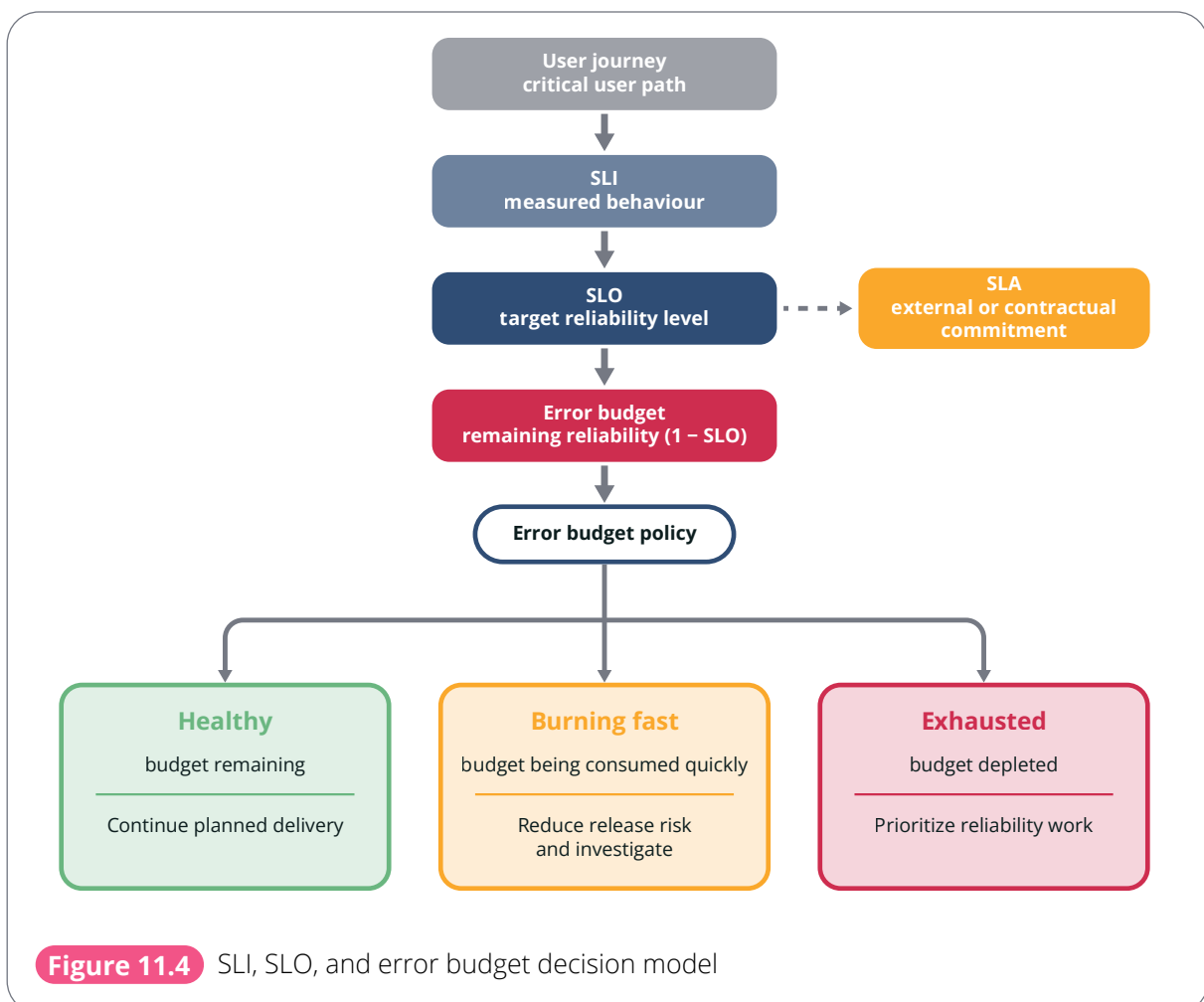
SRE practices turn observability into reliability management. The core contribution is a disciplined way to describe reliability in terms that users care about, then use that description to make delivery and risk decisions (Beyer et al., 2016; Beyer et al., 2018; Hornbeek, 2024).

Figure 11.4 depicts how user journeys are translated into SLIs, SLOs, and error budgets.

When the error budget is healthy, teams can continue with planned change; when it is consumed too quickly, reliability work, release restraint, or architecture investment becomes the responsible decision.

A **Service Level Indicator** (SLI) is a measurable indicator of service behaviour. Good SLIs describe user-relevant behaviour. Availability, latency, error rate, freshness, durability, correctness, and throughput are common categories. A database CPU metric may be useful to an operator, but it is rarely the best SLI for a user path. A checkout success rate, search response latency, or data freshness measure is closer to the experience that matters.

A **Service Level Objective** (SLO) is a target value or range for an SLI. An SLO may state that 99.9 per cent of checkout requests should succeed over a 30-day window. The exact target is a product and risk decision. It should reflect user expectations, business criticality, system cost, and operational reality.



A **Service Level Agreement (SLA)** is an external or contractual commitment. Teams should keep the distinction clear because contractual commitments carry obligations that internal objectives may not.

An **error budget** is the amount of unreliability allowed by the SLO. If the checkout SLO is 99.9 per cent of successful requests over 30 days, the error budget is 0.1 per cent of failed checkout requests during that window. That small number is more than a technical curiosity. It is a way to make trade-offs explicit.

An **error budget policy** defines how a team uses its allowable level of service unreliability to balance delivery speed with reliability, including what actions to take when the budget is at risk or exhausted.

One hundred per cent reliability is usually the wrong target. It may be impossible, economically irrational, or harmful because it slows learning and delivery without improving the user experience in proportion to its cost. Error budgets help teams have better conversations. When the budget is healthy, the team may ship faster within agreed guardrails. When the error budget is being consumed too quickly, the team reduces release risk and investigates. When the budget is exhausted, reliability work takes priority until the system returns within its agreed risk boundary.

Burn-rate alerting connects SLOs to operational alerting and response. Instead of paging because a server has crossed an arbitrary threshold, the team pages when the error budget for user-facing reliability is consumed too quickly. This does not remove the need for diagnostic metrics. It changes which signals trigger a response and which signals support investigation after paging.

SLOs are stronger when they are tied to user paths and reviewed with product owners. Reliability is a product decision as much as an infrastructure target. If reliability work has no product owner, it will compete poorly against visible feature work until an incident forces the issue.

11.3 Extending the scope of observability

Modern observability must cover more than application health. AI behaviour and the delivery system itself can degrade through drift, weak review, slow flow, unstable pipelines, or missing context, even when traditional infrastructure signals remain green. This section extends observability to the systems that create and change software, connecting operational evidence to user outcomes, governance, and learning.

11.3.1 Monitoring AI behaviour in production

AI-enabled systems add a different kind of production responsibility. Traditional software can fail after a code change, a configuration change, an infrastructure event, or a dependency failure. AI systems can degrade even when the code has not changed. Inputs shift. User behaviour changes. A prompt template evolves. A retrieval index becomes stale. A model behaves differently across regions, languages, customer segments, or over time (Autio et al., 2024; DORA, 2025).

Figure 11.5 illustrates how AI behaviour is monitored across the flow from live inputs, through AI services, including models, prompts, and retrieval, to outputs and resulting actions. It extends production observability to AI-specific signals such as model behaviour, prompt or retrieval quality, drift, decision accuracy, human escalation, latency, and cost.

Predictive AI systems need monitoring for data drift, concept drift, prediction drift, training-serving skew, model performance degradation, calibration, fairness indicators, feature quality,



and missingness. Offline validation remains necessary, but it cannot prove that the model will remain fit for purpose after deployment. Production behaviour must therefore be monitored continuously.

Generative AI systems need additional signals. Several signals matter: latency, token usage, cost per request, refusal rate, grounding indicators, hallucination indicators, safety policy violations, prompt injection attempts, retrieval quality for Retrieval-Augmented Generation (RAG) systems, tool-call failures, and human escalation rate. Some of those signals are operational. Some are quality signals. Some are governance signals. The point is that availability and latency are no longer enough.

Machine Learning Operations (MLOps) adds data validation, model validation, continuous training patterns, and production model monitoring to the normal CI/CD practices. In practice, a model deployment, prompt change, retrieval index update, feature pipeline change, safety policy update, or tool integration should be treated as a production change. Each needs appropriate post-deployment monitoring alongside provenance, validation, rollback or mitigation options.

Explainability and auditability become part of observability. Depending on risk, this may include feature attribution, reason codes, model cards, decision records, example-based explanations, prompt and model version traceability, dataset lineage, or human-readable audit trails. The level of evidence should match the risk of the decision. Higher-risk decisions affecting credit, employment, public benefits, or safety require a much stronger evidence model.

AI governance also belongs to the production feedback loop. Risk management guidance such as NIST's AI Risk Management Framework points organizations towards trustworthiness in design, development, use, and evaluation. DevOps provides the operating path for making that practical. It connects design controls, deployment evidence, production telemetry, incident learning, and remediation actions.

The examples are straightforward. A fraud model may maintain precision offline while recall drops in production because attacker behaviour changes. A recommendation model may increase click-through while weakening retention.

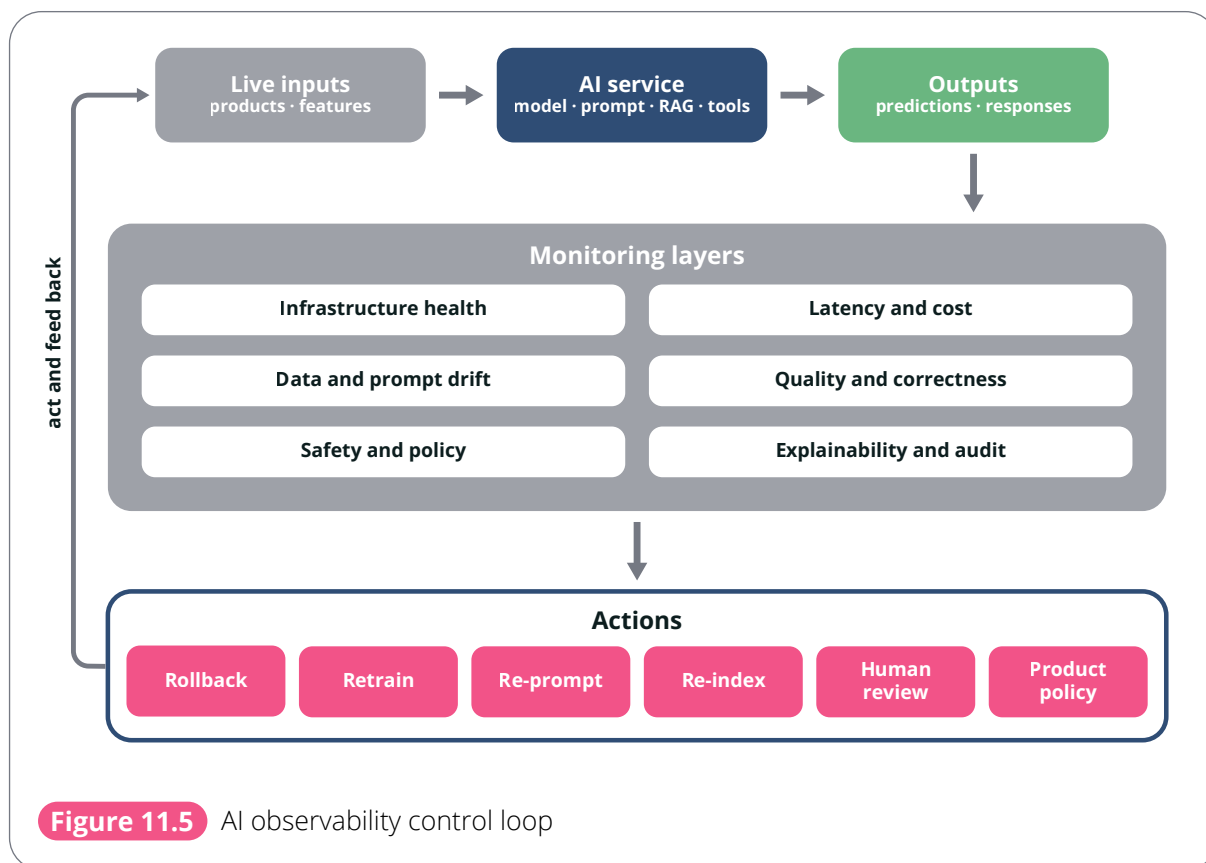


Figure 11.5 AI observability control loop

A chatbot may maintain acceptable latency while its hallucination rate rises after a prompt change. A RAG system may produce poor answers because the index is stale. Each problem is a production problem. Each requires observability.

11.3.2 Delivery-system observability: DORA metrics and flow metrics

Application observability explains the production behaviour of a digital product or service. Delivery-system observability explains how work moves through the engineering system that changes it. Both views are needed because a reliable service can still be constrained by long queues, unstable builds, slow reviews, weak release evidence, or excessive work in progress (Kersten, 2018; DORA, 2024; DORA, 2026a).

Delivery-system observability connects pipeline events, deployment records, incidents, work items, DORA metrics, and flow metrics so that teams can see how the delivery system itself behaves.

The delivery system view combines signals from version control, build, test, security, deployment, release, incident, service management, and value streams. The objective is to take traceability from a change through validation and deployment to production behaviour and user impact. This traceability helps teams locate constraints and determine whether an improvement changed the system.

DORA metrics and flow metrics are important inputs, but this chapter does not reteach their definitions or governance. Chapter 14 is the canonical narrative on balanced measurement, interpretation, data quality, and metric use. Appendix B provides the concise metric catalogue for implementation reference.

Within observability practice, the working requirement is therefore simple: instrument the delivery system well enough to answer where work waits, what changed, what evidence supported the decision, what happened in production, and what should improve next.

11.4 Alerting, on-call, and blameless learning

Alerting is a socio-technical system. It affects users, responders, sleep, trust, and recovery time. A good alert protects users and helps responders take useful action. A bad alert burns attention, trains people to ignore the pager, and slows recovery when a real incident arrives (Edmondson, 1999; Beyer et al., 2018).

The terms should be clear. An **alert** is a signal generated by a rule or detection mechanism. A **page** interrupts a person because urgent action is required. A **ticket** records work that should be handled but does not require immediate interruption. A **notification** informs an audience without necessarily demanding action. A dashboard is for inspection, whereas a report is for trend analysis. Confusing these channels creates noise.

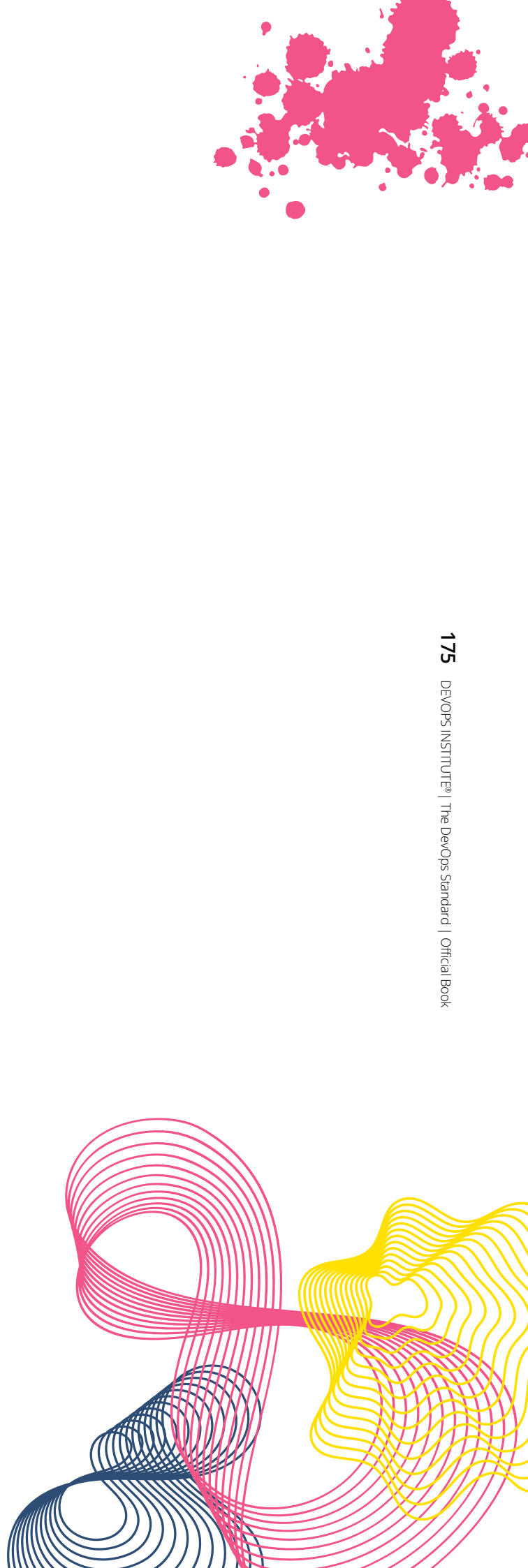
The practical paging standard is strict. A page should be urgent, actionable, and user-visible or imminently user-visible. It should have an owner, a clear statement of user impact, a runbook link, a severity level, recent deployment context, an SLO or business-impact reference, an expected first action, and an escalation path. If an alert lacks those properties, it may still be useful, but it probably should not wake someone.

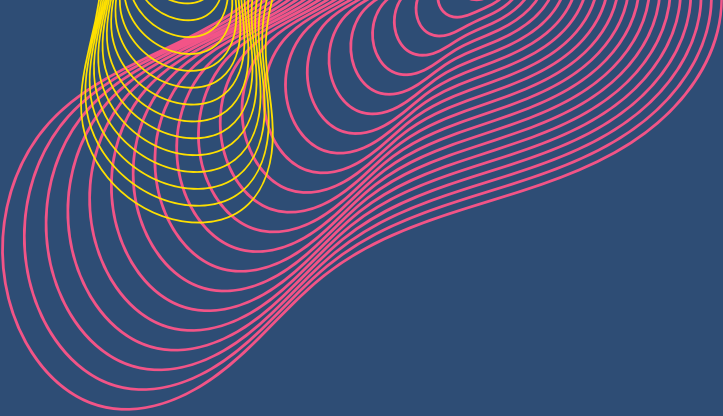
Symptom-based alerting usually protects users more effectively than cause-based alerting. Users do not care which internal component caused the failure. They care that checkout is failing, search is slow, or responses are unsafe. Cause-based alerts can support diagnosis or early warning, but pages should usually reflect user-visible symptoms. This keeps attention focused on the service experience and reduces noise from internal conditions that do not matter.

Sustainable on-call requires design. Fair rotations, training, shadowing, escalation support, incident command, psychological safety, and recognition all matter. Teams should track alert volume, sleep-impacting pages, noisy alerts, and alerts that are not actionable. Removing a bad alert is often reliability work. It improves the human system that protects the technical system.

Postmortems should be learning artifacts. A useful postmortem records the incident summary, timeline, user impact, detection path, response, contributing causes, what went well, what went poorly, where luck was involved, and action items with owners and dates, as well as alerting or observability gaps. The purpose is to improve the system. Blame narrows the search for causes. Learning widens it.

The best postmortems feed backlog and architecture decisions. They improve tests, runbooks, dashboards, alerts, deployment practices, service boundaries, capacity plans, security controls, and product expectations. A postmortem that produces no change is only a record. A postmortem that changes the system is observability doing its job.





Case B | Harbor Digital Commerce (composite): when green infrastructure hid a broken checkout

During a seasonal promotion, Harbor's infrastructure dashboards remained green while checkout completion fell. CPU, memory, and basic availability were within expected thresholds. Customers were abandoning payment because a downstream provider was intermittently slow, and the application retried in a way that amplified latency.

The team correlated traces, payment-provider timing, recent deployment context, customer-journey events, and support contacts. A checkout-completion SLI and SLO made the user impact visible. Error-budget burn changed the release decision: the product team paused a planned feature release and directed work towards the payment dependency and retry behaviour.

AI-assisted triage grouped related signals and highlighted the provider dependency, but responders verified the source traces before

acting. The first AI summary overweighted an infrastructure anomaly that was not causal, so the workflow was changed to show evidence links, alternative hypotheses, and missing context rather than a single confident explanation. Harbor then tracked time to assemble incident context, recommendation acceptance and override rates, and missed conditions. It did not authorize automated mitigation; that would require repeated evidence of accuracy, narrow blast radius, tested rollback, and complete auditability.

Production evidence then changed the roadmap. Harbor funded a payment abstraction and resilience work instead of another visible checkout feature. The decision combined observability with product judgement; telemetry did not make the decision automatically.

The experience also gave Harbor a practical observability maturity path. The levels below are useful only when they represent a stronger explanation, decision support, and learning, not a larger inventory of monitoring tools.

Continued on next page

Continued

Table 11.2 Observability maturity path

Level	Capability	Evidence expected	Next improvement
1	Basic uptime monitoring	Availability and basic health signals	Instrument critical user journeys and services.
2	Service dashboards and threshold alerts	Service status, latency, errors, and alert history	Correlate signals across the service.
3	Correlated logs, metrics, and traces	Cross-signal diagnosis with release context	Define SLIs, SLOs, and an actionable alert policy.
4	SLOs, error budgets, and actionable alerts	Error-budget decisions, on-call response, and learning actions	Connect telemetry to roadmap and architecture decisions.
5	Telemetry drives roadmap and architecture	Production evidence changes priorities and design	Extend observability to software delivery and AI behaviour.
6	Software, delivery, and AI observability	Traceability from change and model behaviour to user outcomes	Continuously validate automation, models, and guardrails.

Harbor used the maturity path to sequence improvement from basic detection to correlated explanation, reliability decisions, and product learning.

A practical checklist remained useful. Could the team detect user-impacting failure, explain why it happened, connect behaviour to a deployment or dependency, act on an SLO, complete learning actions, and monitor AI-enabled workflows after deployment?

Transferable lesson: Observability is an organizational learning capability. Green infrastructure does not prove that the user journey is healthy, and an AI-generated explanation does not remove the need for correlated evidence and accountable judgement.

SCENARIO



DECISION AND ACTION GUIDE

Executive decision: The accountable product or service executive should approve the SLO and error-budget policy for critical services, the required on-call investment, and how production evidence will influence product and release decisions.

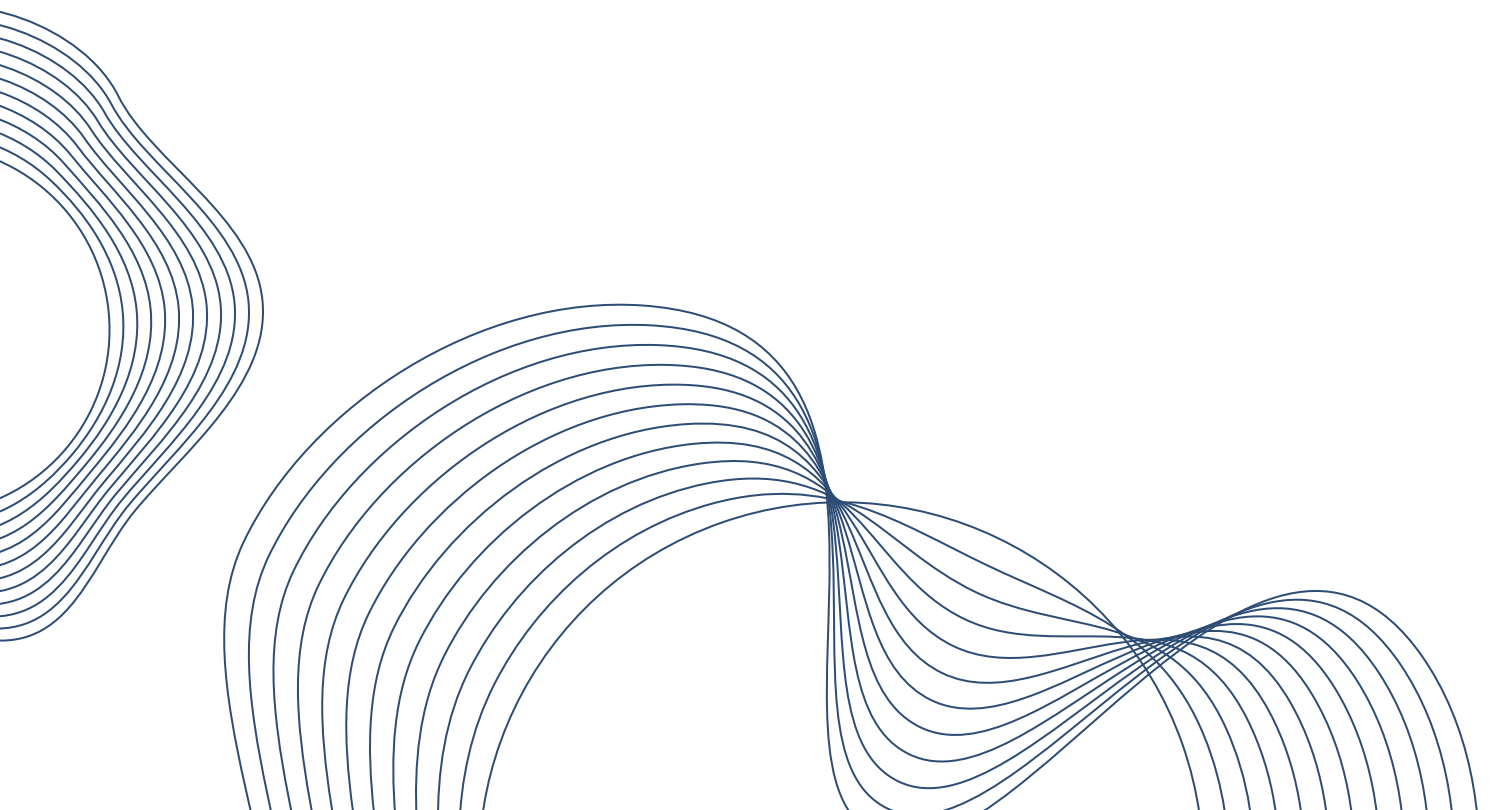
Role-specific actions: Product, SRE, engineering, operations, and security roles should define SLIs, instrument critical paths, tune alerts, review incidents, and monitor AI behaviour where applicable.

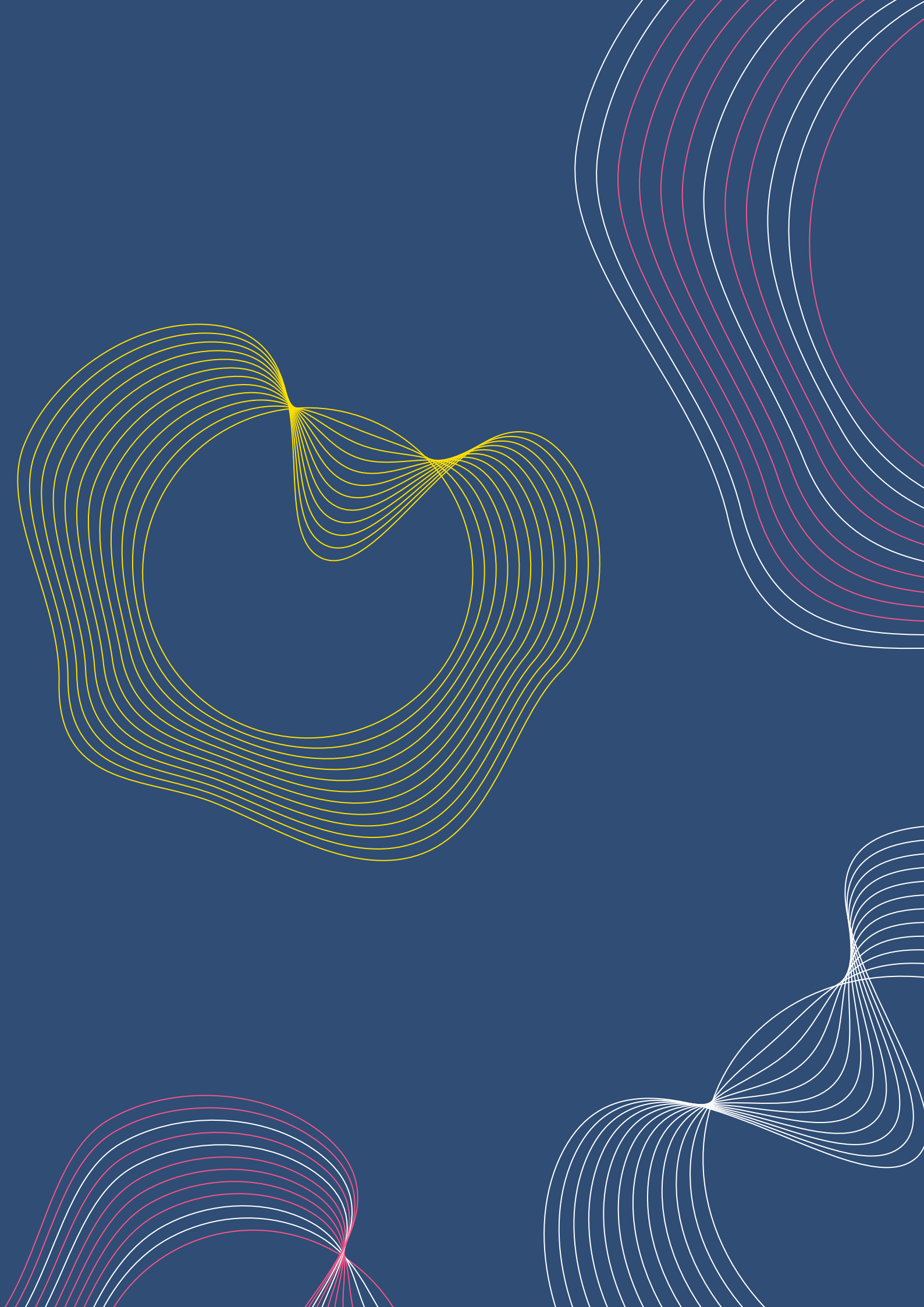
Evidence to review: SLI and SLO performance, error-budget decisions, alert actionability, telemetry coverage, AIOps recommendation accuracy and override rates, missed-event rate, automated-action success and rollback, completed incident actions, and AI drift or safety signals.

11.5 Chapter summary

Continuous Monitoring and Observability provide production feedback that allows the DevOps system to correct itself. Monitoring detects known conditions; Continuous Monitoring and Observability explains unexpected behaviour by correlating logs, metrics, traces, deployment context, business signals, and user impact. SLIs, SLOs, error budgets, and actionable alerts turn that evidence into reliability and release decisions.

AI-enabled systems expand the required telemetry to include quality, drift, safety, explainability, and cost, as well as prompts, models, retrieval, and human escalation. AIOps can reduce noise and support diagnosis, but automated action must remain bounded, reversible, evidenced, and accountable. The mature outcome is telemetry-driven learning in which production evidence changes architecture, roadmaps, platforms, security controls, and the next delivery decision.





Putting it into practice



Chapter 12

Assessing where you are

Chapter 13

The DEVOPS INSTITUTE®
Transformation Playbook

Chapter 14

Measuring what matters

Chapter 15

DevOps at scale



PART THREE puts the DOI® Operating Model into practice. This section is concerned with what organizations do after they understand the model: assess where they are, choose where to improve, build a transformation plan, measure the right signals, and scale without losing control.

The practical starting point is assessment. Many DevOps programmes begin with tools, automation, platforms, restructures, or training before the organization has understood its real constraints. That is usually expensive and rarely disciplined. A serious improvement effort begins with evidence. It asks what teams can do today, what slows flow, where reliability breaks down, and where governance creates friction. It also examines where security arrives late, where AI-enabled work is entering the system, and which constraints matter most.

PART THREE is written for leaders, transformation sponsors, senior practitioners, platform teams, assessors, coaches, and governance stakeholders who need to turn the DEVOPS INSTITUTE® model into a working improvement programme. The section is practical. It provides the bridge from framework understanding to operational change.

Chapter 12

Explains how to assess current maturity. It introduces the use of the DevOps Nine Pillars as an assessment structure, with Appendix A providing the practical assessment method and guidance. The chapter distinguishes between aspirational maturity, operational maturity, and evidence-based maturity. It also warns against using assessment as blame. A good assessment should reveal the condition of the system so that leaders can prioritize investment intelligently.

Chapter 13

Provides the transformation playbook. It explains why transformation should begin with strategy and goal alignment, followed by current-state assessment and value stream mapping. The subsequent stages are future-state mapping, solution options,

implementation planning, implementation verification, operationalization, expansion, and sustained improvement. It gives readers a sequence for the first 90 days, guidance for building the transformation case, and practical advice for scaling from team to programme to organization.

Chapter 14

Defines the measurement system. It treats metrics as feedback control, not reporting decoration. The chapter explains the balanced domains of Flow, Quality, Stability, and Value, and extends the measurement model to AI behaviour, drift, decision quality, explainability, and operational AI cost. It also places DORA metrics in context, explaining their value while warning against shallow comparisons, gaming, and false confidence.

Chapter 15

Addresses DevOps at scale. Scaling is where many DevOps efforts stall because the constraints move beyond individual team practices. Dependencies, architecture coupling, shared services, enterprise platforms, portfolio funding, compliance, governance, measurement, and coordination overhead become the main issues. This chapter explains how organization-wide DevOps requires a coherent operating model across teams, value streams, platforms, governance, architecture, funding, and metrics.

The chapters in PART THREE share a simple discipline. Do not improve randomly. Assess the current state, identify the constraint, choose a focused intervention, measure whether it helped, and use the evidence to decide the next investment. This discipline matters even more in AI-native delivery, where the speed of artifact generation can make a weak delivery system noisier and harder to control.

PART THREE moves the DevOps Standard book from definition to action. It gives organizations a way to use the DEVOPS INSTITUTE® model without turning it into a ceremony.

12.

This chapter gives organizations a practical way to understand their current DevOps maturity before they invest in improvement. Assessment is the prerequisite for targeted transformation because it separates belief from evidence and aspiration from operational reality.

The DevOps Nine Pillars (Chapter 4) provide the primary practice-domain lens for assessment. The complete DEVOPS INSTITUTE® Maturity self-assessment tool also evaluates four cross-cutting system capabilities that extend across the Operating Model. AI-native readiness is assessed as one of those cross-cutting capabilities and through its application within the relevant Pillars.

The chapter is written for leaders, transformation teams, assessors, and senior practitioners. It gives them a disciplined way to read the system, locate constraints, and convert findings into improvement investment. A maturity assessment should make the organization calmer and more precise. When it creates anxiety, rivalry, or defensive scoring, the organization has already started using the instrument incorrectly.

12.1 Assessment foundations

Effective assessment begins with a shared understanding of purpose, scope, maturity, and evidence. The objective is not to award a score or compare teams; it is to reveal the current capability of the system, the pattern of strengths and weaknesses, and the constraint that matters most. DevOps maturity assessment is a structured way to understand the system in order to improve it. This section establishes the assessment concepts needed before the process is conducted.

12.1.1 Assessment before improvement

Many DevOps initiatives begin with a solution before the organization has established the problem. A platform is purchased, a pipeline programme is announced, teams are reorganized, a new automation backlog is created. Yet no one has shown where the constraint sits. The organization may remain busy without addressing what is actually slowing delivery, increasing risk, or exhausting teams (Humble et al., 2014; DORA, 2024).

Assessment prevents random transformation. It gives leaders and teams a shared current-state picture before money, effort, and political capital are committed. Assessment asks a few hard questions: how work moves, where it waits, which controls are trusted, which practices are repeatable, and which outcomes can be shown with evidence.



Assessing where you are

Three views of maturity often coexist.

Aspirational maturity is what leaders believe or intend to be happening. **Operational maturity** is what teams experience in daily work. **Evidence-based maturity** is what can be demonstrated through practices, data, records, and outcomes. When these views disagree, the disagreement is useful. It tells the organization where perception and reality have drifted apart.

Consider a common scenario. An organization may believe that it has mature DevOps because it has CI/CD tooling, automated builds, and several cloud environments. Assessment may reveal a different picture: weak incident learning, fragile environments, manual approvals, late security evidence, and product feedback that rarely reaches the backlog. In an AI-enabled team, generated code may enter the pipeline quickly, while test-coverage reviews, provenance tracking, and model-risk checks remain inconsistent. The important outcome in this case is a clearer understanding of where improvement is required. Sometimes it is monitoring, leadership decision rights, platform self-service, or the way teams own production outcomes. Good assessment narrows the field of action.

Appendix A provides the practical assessment guidance for this book. Use it as a structured conversation supported by evidence, not as a survey completed in isolation. The assessment approach should help teams understand current capability, identify constraints, and choose the next responsible improvement.

12.1.2 Maturity levels: from initial to optimizing

A maturity level describes system capability, not the worth of a team or the competence of individuals. This distinction matters because teams can behave intelligently inside badly designed systems. A team may work hard, collaborate well, and still be trapped by manual approvals, brittle environments, missing telemetry, or governance rules that create unnecessary waiting. The five-level scale²⁰ used in this chapter gives organizations a common language for progression.

Level 1: Initial

Initial maturity describes work that is ad hoc and dependent on individuals. Deployments are irregular, documentation is thin, and incidents are handled reactively. Success often depends on a few people who know where the fragile parts are and how to work around them.

Level 2: Managed

Managed maturity means that some repeatable practices exist. Teams can follow known procedures, and local standards may be present. Basic automation may help with build, testing, or deployment work. The weakness is variation. One team may have a useful pipeline, and another may still rely on manual release coordination. In AI-native contexts, one team may review generated code carefully while another treats generated output as ordinary work with no provenance or special validation.

Level 3: Defined

Defined maturity means that shared standards exist across teams. The organization has common expectations for CI/CD, testing, monitoring, security, infrastructure, and delivery controls. At this level, people can explain what good practice looks like. The limitation is that outcomes may still be weakly measured. A defined process can look mature on paper while still producing long lead times, escaped defects, or noisy incidents.

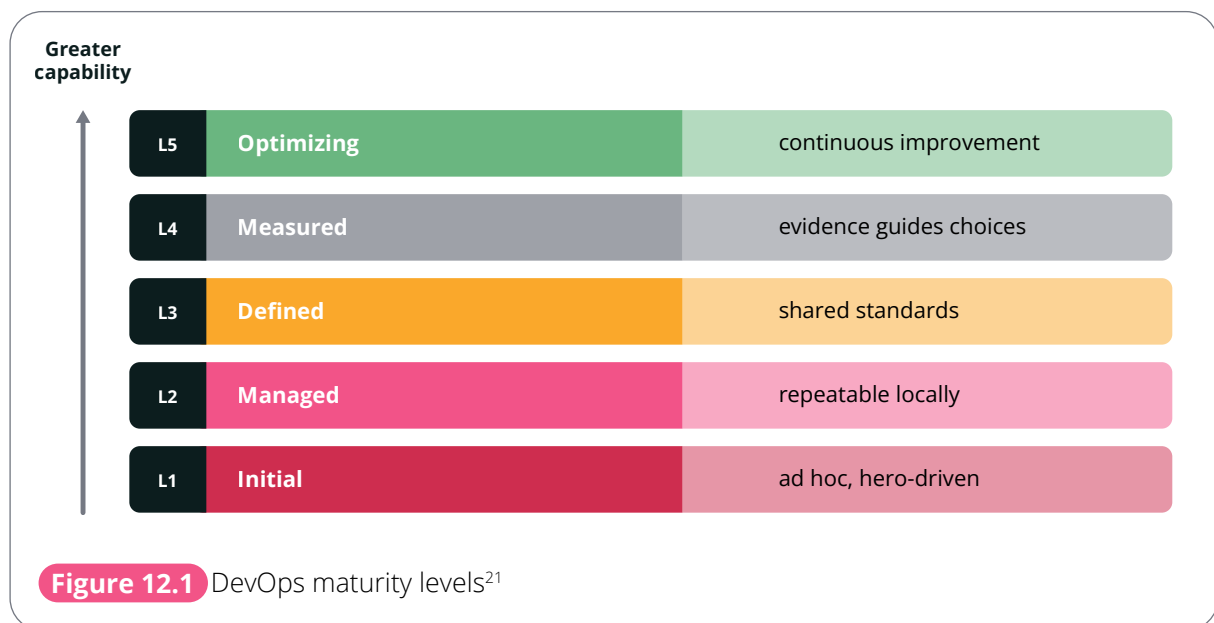
Level 4: Measured

Measured maturity means that practices are instrumented and evaluated. Teams use delivery, flow, reliability, incident, security, product, and AI behaviour signals to understand how the system is performing. Leaders can see bottlenecks and constraints. Measurement is used to improve the system rather than to punish individuals.

Level 5: Optimizing

Optimizing maturity means that continuous improvement is embedded in the operating model. Teams run controlled experiments, learn from production signals, improve the platform, complete post-incident actions, and adjust governance based on risk and evidence. AI-enabled delivery at this level includes validation, traceability, monitoring, and human accountability as normal operating practices.

The following figure shows maturity progressing from Initial to Managed, Defined, Measured, and Optimizing. It reinforces that maturity should be viewed as a progression in capability, rather than reduced to a single score.



12.1.3 Assessing the Nine Pillars and cross-cutting capabilities

The DevOps Nine Pillars prevent a narrow assessment. Without them, organizations tend to over-read the parts of DevOps they can see easily. Pipelines, tools, dashboards, and deployment frequency are visible. Leadership alignment, psychological safety, architecture fitness, test data, incident learning, and security ownership are often harder to see. They are also where many constraints exist.

A useful assessment reads **horizontally across the Pillars**, and **vertically within each Pillar**. Horizontal reading shows balance across capabilities. A team may be strong in Continuous Integration and weak in Continuous Monitoring and Observability,

which means it can integrate changes faster than it can understand them in production. Vertical reading shows depth and completeness of a capability. A team may have infrastructure automation yet still lack versioned modules, drift detection, recovery testing, or cost controls.

The Nine Pillars provide the practice-domain structure for assessment. Each Pillar is realized through the three enabling dimensions of people, process, and technology, as established in the DOI® Operating Model. Assessment considers how effectively these dimensions work together within each Pillar and extends the view to measurement and evidence. This helps determine not only whether relevant practices exist, but whether they are operating effectively and producing meaningful results. Table 12.1 summarizes the assessment criteria for each Pillar.

Table 12.1 Assessment criteria across the DevOps Nine Pillars

Pillar	People	Process	Technology	Measurement	Evidence
Leadership	Sponsorship	Vision and strategy	Decision support	Transformation progress and performance	Leadership decisions
Collaborative Culture	Ownership	Cross-team collaboration	Shared ways of working	Learning and team health	Autonomy and feedback
Design for DevOps	Operability mindset	Distributed architecture practices	Testable and observable services	Resilience	Architecture evidence (independence of modules, observable by-products)
Continuous Integration	Team ownership and short branches	Merges, builds, tests, and packaging	CI automation and fast feedback	Mainline health	Release candidate evidence
Continuous Testing	Development and quality assurance (QA) teams	Automated test generation, environment orchestration, test execution, and results analysis	Test harness and test data	Coverage and defects	Test results
Elastic Infrastructure	Skills in IaC, configuration as code, and containers	Self-service provisioning	GitOps, infrastructure and container orchestration, and configuration tools	Scale, flexibility, and resource utilization	Recovery and environment evidence (setup and recovery time, environment failures)

Continued on next page

Continued

Pillar	People	Process	Technology	Measurement	Evidence
Continuous Security	Development and security teams	Threat modelling, scans, and penetration testing	Security scanners (SCA, SAST, DAST, RASP, IAST) and penetration testing tools	Security posture	Policy compliance reports
Continuous Delivery and Deployment	Delivery and deployment ownership	Deployment strategies	Deployment tools	Failure rate, recovery time (DORA)	Release readiness and deployment health
Continuous Monitoring and Observability	DevOps and SRE teams	Monitoring and observability practices (SLOs, logs, metrics, traces, and events)	Alerts, dashboards	Operational performance, data and SLIs	Telemetry and AI signals

12.1.4 Reading maturity patterns

Assessment results usually form patterns. These patterns often matter more than the numeric scores. A simple average hides the shape of the system, and that shape helps leaders identify where to act. Mature assessors look for imbalances, constraints, dependencies, and contradictions.

One common pattern is **tool-rich but culture-poor**. The organization has many tools, but trust is low, approvals remain manual, teams remain siloed, and ownership is unclear. This usually means the transformation focused on technology while leaving the work system intact. AI tools can make this worse by increasing artifact volume while the review culture remains fragile.

A second pattern is **strong CI with weak release management**. Builds are automated, tests may run reliably, and integration looks healthy, yet deployments remain manual and infrequent. Change failure remains high because release coordination, environment readiness, rollback, and production validation have not matured alongside integration.

The evidence-based assessment logic across the DevOps Nine Pillars applies to AI-native delivery as well. AI-generated artifacts, AI-assisted decisions, and AI-enabled services must be reviewed, governed, monitored, and improved within the DOI® Operating Model, rather than managed through a separate checklist or side process.

**AI-RELATED
CONTENT**



A third pattern is **fast delivery with weak reliability**. Teams deploy often, but monitoring is thin, incident learning is weak, SLOs are unclear, and on-call load is rising. The organization has accelerated change without sufficient feedback and recovery capability. In AI-enabled services, the equivalent pattern appears when prompts, models, or retrieval indexes change faster than the monitoring of quality, drift, safety, and explainability can keep up.

Another pattern is **strong local teams with weak organizational alignment**. Some teams have mature practices, while others work in very different ways. Shared platforms are missing, governance varies, and common standards are thin. Maturity exists in pockets.

The organization has success stories, but it has not yet built the conditions that let good practice spread.

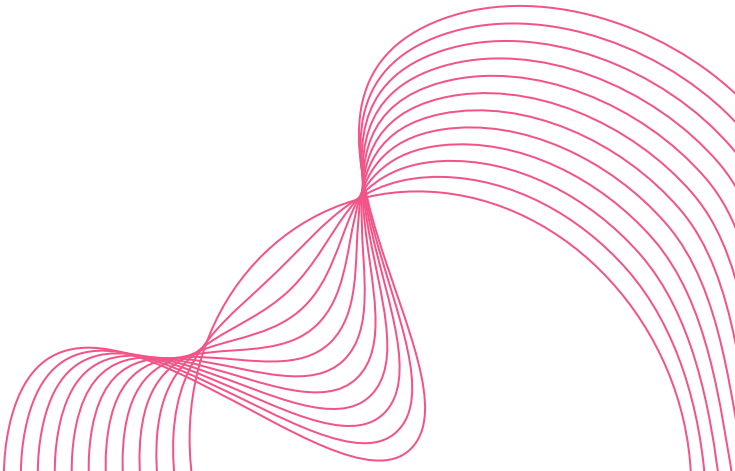
Security as a late gate remains a stubborn pattern. Reviews happen near release, findings cause delay, developers lack usable tooling, and security teams become bottlenecks.

Measurement without learning is equally common. Dashboards and metrics provide visibility, but the evidence does not lead to changed behaviour or improvement.

The shape of the maturity profile matters more than the average score. Tool-rich, culture-poor profiles require different actions from those needed for fast-but-fragile profiles or strong-local-team profiles.

Table 12.2 Interpreting common DevOps maturity patterns

Maturity pattern	Typical symptoms	Improvement focus
Tool-rich but culture-poor	Strong tooling, weak trust and ownership	Culture, collaboration, decision rights
Strong CI, weak release management	Fast integration, difficult releases	Release and deployment practices
Fast delivery, weak reliability	High speed, weak feedback and recovery	Observability, reliability, recovery
Strong local teams, weak alignment	Mature teams, inconsistent organization-wide practices	Platforms, standards, governance
Security as a late gate	Late reviews create delays and bottlenecks	Embedded security and controls
Measurement without learning	Visibility does not lead to improvement	Learning, decisions, action



12.2 Conducting and interpreting the assessment

An assessment becomes useful through disciplined participation, evidence, and interpretation. Teams should apply a consistent process, record the basis and confidence for each rating, and read the results as a maturity pattern rather than as one average score. This section explains how to use the assessment instrument to produce a credible current-state view and practical findings.

12.2.1 Conducting the internal assessment

The internal assessment should be scoped clearly. A team, value stream, platform group, business unit, and an organization do not share the same boundaries, evidence, or decision rights. The scope must be stated before scores are discussed so local excellence is not averaged into an enterprise claim and organization-wide constraints are not assigned to one team (Hornbeek, 2019; DORA, 2024; PeopleCert, 2026d).

Participants should represent the system being assessed and include people who understand

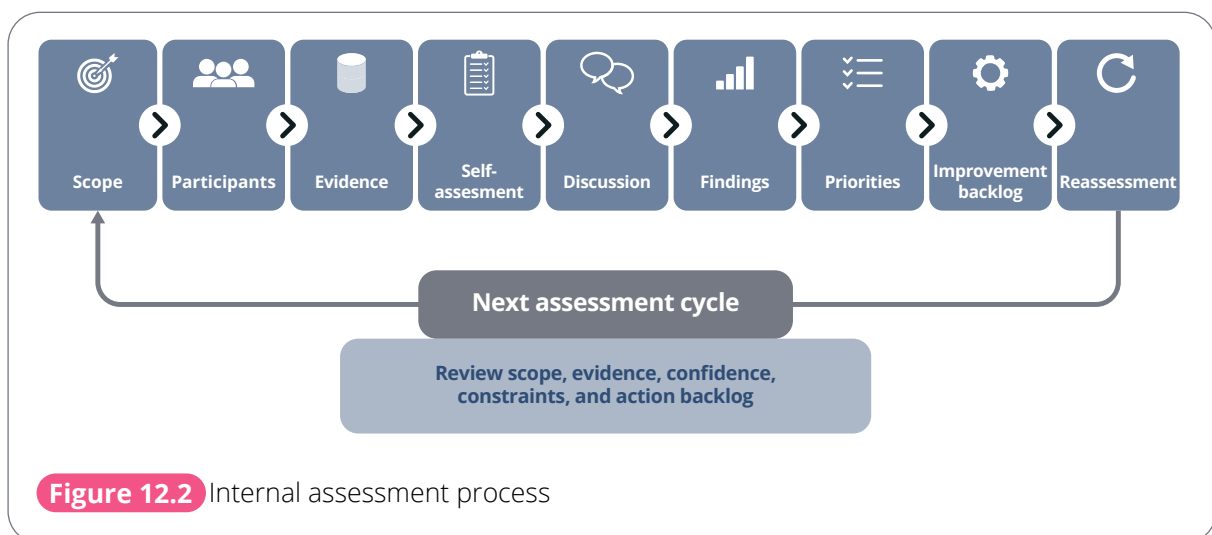
how work moves. The evidence should combine operational records with informed judgement. Toolchain data, pipeline results, incidents, architecture, security findings, service evidence, product outcomes, and AI-governance records are more reliable than a consensus reached in a workshop unsupported by facts.

Appendix A provides the canonical assessment method and guidance for the DEVOPS INSTITUTE® Maturity self-assessment tool.

It defines the fields, scoring rules, confidence rating, participant guidance, evidence expectations, and interpretation approach. This chapter retains the principles so leaders can govern the assessment without duplicating the detailed assessment materials.

Figure 12.2 depicts the internal assessment workflow and emphasizes an evidence-based assessment process and reassessment rather than a one-time maturity exercise.

The assessment should have a named accountable owner, such as the value stream owner, transformation lead, or executive sponsor, who is responsible for reviewing the findings, tracking improvement actions, and commissioning reassessment. During active improvement, assessment evidence should be reviewed monthly and a structured reassessment should be conducted after each 60-to-90-day improvement cycle. Once a stable improvement cadence has been established, a full reassessment should be conducted quarterly or semi-annually, according to the scope, risk, and rate of change.



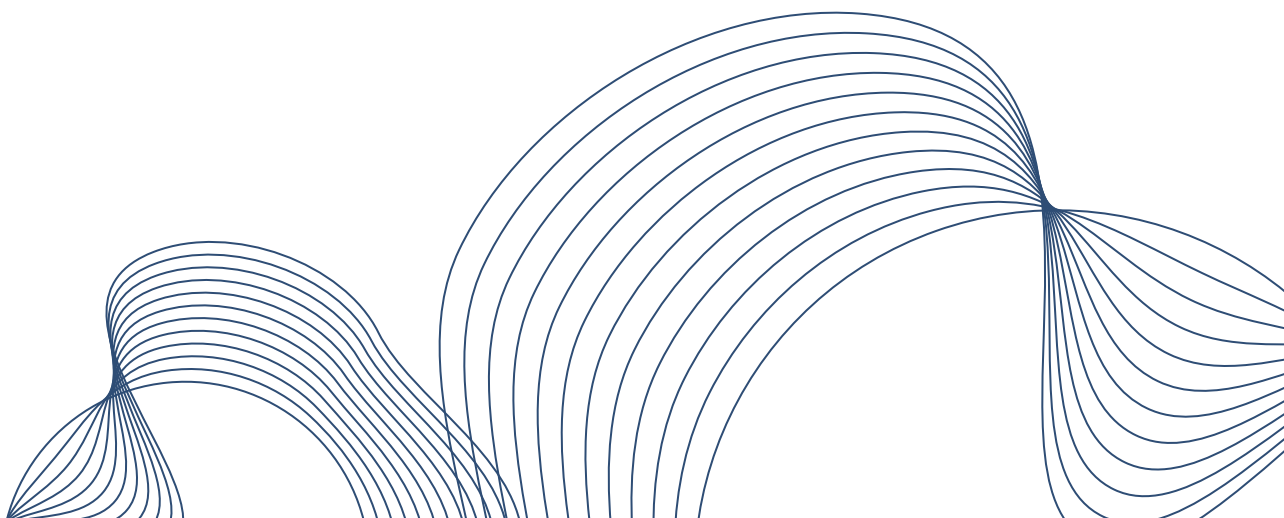
12.2.2 Using Appendix A to interpret results

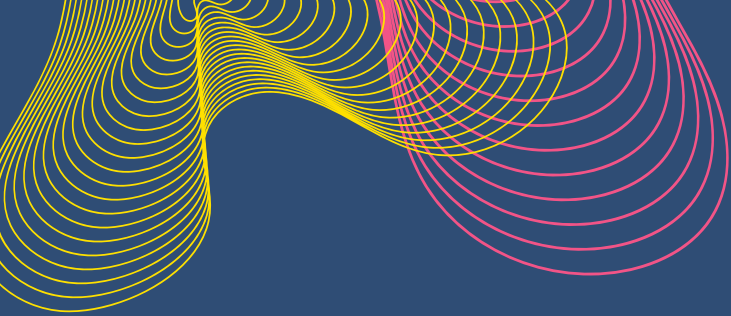
The first output is a current-state profile, not a roadmap. Read the distribution of scores, the strength of evidence, the confidence rating, and the dependencies among pillars. Averages can hide the constraint: a mid-level capability may be the area that limits several stronger capabilities, while the lowest score may not yet be the active bottleneck.

Interpret disagreement as evidence about the system. Leaders may see a documented process, while practitioners experience unreliable execution. Security, operations, product, platform, and delivery teams may hold different parts of the record. The facilitator should make those differences visible and resolve them through evidence rather than hierarchy.

Appendix A explains the maturity model, scoring approach, confidence guidance, and interpretation of results. The chapter-level interpretation should summarize the maturity profile, evidence quality, material risks, principal constraints, and a small number of improvement themes. Section 12.3.1 then turns those themes into investment choices.

Interpretation should compare scores, evidence quality, confidence rating, and operating outcomes. A low score with strong evidence is more useful than a high score based on aspiration alone.





Case A – Northstar Financial Services (composite): reading an evidence-based maturity profile

Northstar assessed the payments value stream before selecting another automation programme. Leaders initially described the value stream as highly mature because it had a modern pipeline and frequent test execution. Engineers scored the same value stream lower because test data was unreliable, environments drifted, release evidence was assembled manually, and production alerts did not describe customer impact.

The assessment team scored individually, then compared evidence. Continuous Integration was relatively strong, but confidence in the score was high only for build and merge practices. Continuous Testing and Continuous Monitoring and Observability were weaker. Security controls existed, yet evidence quality varied by service. The most important constraint was not simply the lowest score:

shared test data and weak customer-journey telemetry constrained testing, release readiness, recovery, and learning at the same time.

The team recorded confidence alongside maturity. Where evidence was incomplete, it did not average opinions into a false sense of precision. The resulting improvement theme was to strengthen test data, service-level telemetry, and release-readiness evidence before investing in a new pipeline product.

The assessment changed the investment conversation. It replaced a tool request with a system problem supported by operational facts and a small number of testable improvement hypotheses.

Transferable lesson: A maturity score is useful only when it is supported by evidence and interpreted as a system. Prioritize the capability that constrains several outcomes, not automatically the lowest number or the most visible tool gap.

SCENARIO



12.3 Improvement investment, benchmarking, and governance

Assessment creates value when its findings change priorities, funding, and control decisions. Improvement investment should address the constraint, benchmarking should add context rather than competition, and governance should use defensible evidence while preserving a learning mindset. This section connects assessment results to action, accountability, and reassessment.

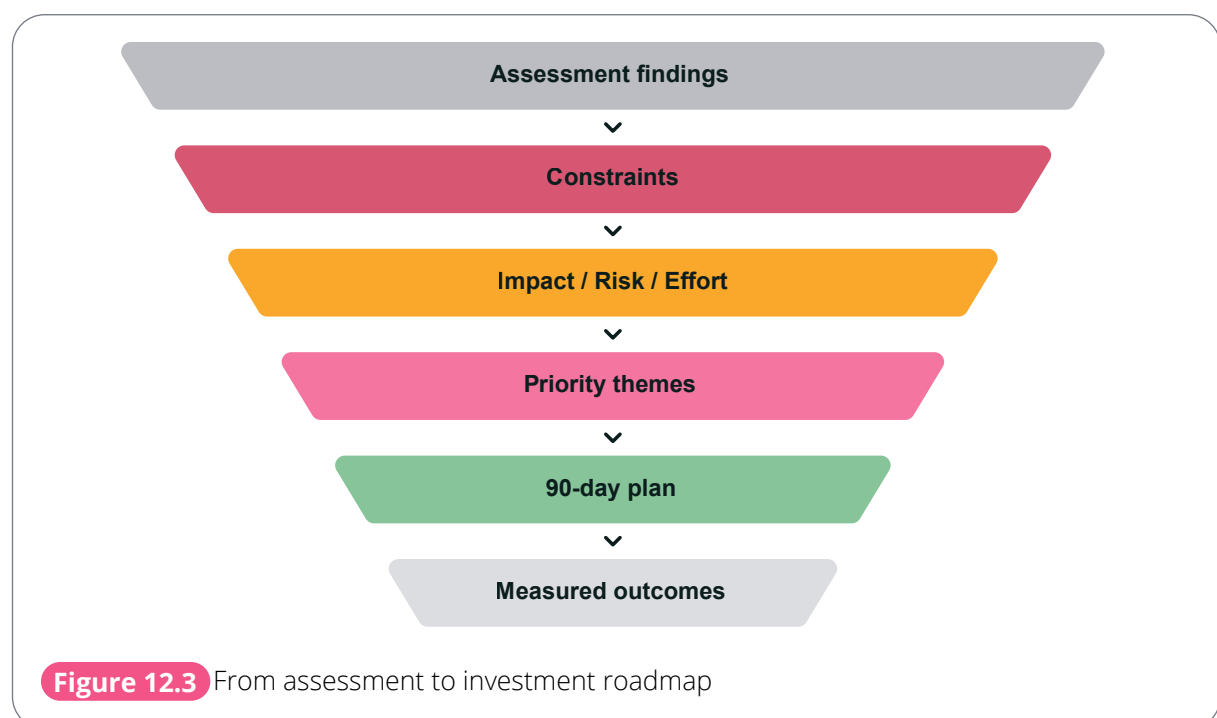
12.3.1 Prioritizing improvement investment

Assessment without prioritization becomes reporting theatre. A maturity profile may be accurate and still be useless if it does not change investment. The organization should use assessment results to choose a small number of high-value improvements and fund

them properly. Trying to raise every pillar one level at the same time usually creates scattered activity and weak outcomes (Humble et al., 2014; Kersten, 2018).

Prioritization should consider the following criteria: user impact, business value, risk reduction, removal of delivery bottlenecks, reliability improvement, security exposure, cost of delay, implementation effort, and dependencies. Together, they keep the discussion grounded, helping leaders see why invisible work matters. Test data management, environment consistency, observability, platform reliability, security automation, and technical debt reduction often look less exciting than feature work. Yet they may be the work that makes feature delivery safe and predictable.

The assessment should lead to improvement themes rather than a long, unranked backlog. Foundational fixes may address environment consistency, source control hygiene, build reliability, or access management. Flow improvements may reduce approval queues, large batches, and dependency delays. Reliability improvements may introduce SLOs, incident learning, and production validation. Security integration may move controls into pipelines. Platform enablement may reduce repeated work and cognitive load.



A practical 90-day improvement plan selects two or three high-leverage constraints. It defines the outcome, leading indicators, owner, funding model, and review cadence. For example, if Continuous Testing is weak and deployments are slow, the first investment may be test automation quality, test data, and contract testing. If Continuous Monitoring is weak and incidents are frequent, the first investment may be service telemetry, SLOs, and alert cleanup.

Leadership matters at this point. If Leadership scores low, beginning with tooling rarely helps. The organization may need clear sponsorship, decision rights, funding, and accountability before technical improvement can take hold. If Continuous Security scores low, the first investment may be pipeline controls, threat modelling, secrets management, and security champion support. If AI-native governance is weak, the organization may need provenance, validation, and monitoring controls before scaling AI-assisted delivery more broadly.

The following figure illustrates how assessment findings become constraints. These are then filtered through impact, risk, and effort to produce priority themes, a 90-day improvement plan, and measured outcomes. It reinforces that assessment has value only when it changes investment decisions.

12.3.2 Benchmarking responsibly

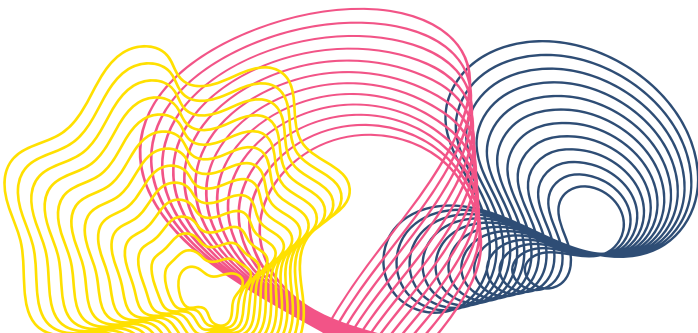
Benchmarking compares assessment results or performance data with external reference points. It can calibrate ambition, reveal blind spots, support investment cases, and help organizations learn from comparable peers. If used well, benchmarking sharpens questions. If used badly, it becomes a ranking exercise with misleading precision (DORA, 2024; Hornbeek, 2024).

The main risk is comparing unlike systems. A small digital product team, a regulated payment platform, a public-sector case-management system, and an AI-enabled clinical decision-support workflow may all use DevOps practices. Even so, their regulatory exposure, architecture, team size, release risk, data sensitivity, and customer impact differ substantially. A single benchmark number cannot capture all that context.

Responsible benchmarking compares similar contexts and uses ranges rather than exact rankings. It focuses on trends more than snapshots. It combines external data with internal evidence. A team that is slower than a benchmark may be blocked by approval queues, architecture coupling, manual evidence gathering, or missing platform support. The benchmark points to a question, while the internal assessment helps explain the likely constraint.

DORA metrics and flow metrics can support benchmarking when definitions are clear. The DevOps Standard book uses four DORA measures: deployment frequency, lead time for changes, change failure rate, and failed deployment recovery time. Teams should document scope, calculation rules, start and end points, exclusions, and data sources before comparing results.

Flow metrics add a value-stream view. They help leaders see work movement, load, distribution, and waiting time. Maturity assessment explains capability. Benchmarking compares outcomes. Both should be read together. Moreover, one team may have respectable DORA numbers and still struggle with customer value, product feedback, or team sustainability. Conversely, another team may look slower because it handles high-risk work with more extensive compliance evidence. Benchmarks should be used to ask better questions, not to rank unlike teams. Compare similar contexts, use ranges rather than exact targets, and combine external data with internal evidence.



In AI-enabled contexts, delivery benchmarks should be interpreted alongside the behaviour, governance, and risk evidence introduced in earlier chapters; otherwise, speed comparisons may overlook changes in quality, accountability, or exposure.

**AI-RELATED
CONTENT**



12.3.3 Audit, governance, and reassessment

An assessment used for improvement has a different purpose from an assessment used for audit or governance. An improvement assessment focuses on learning, prioritization, team engagement, capability growth, and bottleneck removal. The tone should be collaborative and evidence-seeking. The best result is a clearer understanding of the system and a funded improvement backlog (DORA, 2024; PeopleCert, 2026d).

An audit or governance assessment focuses on control effectiveness, risk exposure, policy adherence, regulatory obligations, traceability, and accountability. The tone is more formal because the output may need to be independently reviewed. Evidence standards are higher. Scoring must be defensible. Findings may require remediation tracking, documented exceptions, named risk owners, and mappings to control requirements.

The difference in purpose should not remove the learning mindset. Governance should strengthen DevOps rather than turn assessment into punishment. Good governance uses evidence, avoids blame, tracks remediation, and connects findings to improvement. It also rewards transparent reporting. Teams should not learn that hiding problems protects them.

Audit evidence may include change records, approval history, deployment logs, access control records, security scan results, incident reports, postmortem actions, SLO reports, pipeline configuration, model approval records, prompt version history, and AI decision logs. In mature systems, much of this evidence is created automatically by pipelines, platforms, observability systems, and policy as code controls.

Reassessment completes the learning cycle. Active transformations may be reassessed quarterly. Stable organizations may be reassessed semi-annually. Major platform changes, organizational changes, significant incidents, delivery failures, or AI governance events may also justify reassessment. Compare Pillar scores, evidence quality, outcome metrics, completed actions, and new constraints.

Reassessment can be gamed if leaders reward inflated scores or punish honest low scores. Require evidence. Review outcomes, not practices alone. Keep the assessment connected to real work.

Reassessment should update the roadmap, funding, governance attention, and improvement backlog. If the organization is trying to win against the model rather than improve the system, the model has stopped helping.

12.4 Honest assessment as a leadership practice

Assessment reveals current capability. The DevOps Nine Pillars prevent a tooling-only view. Maturity levels describe progression. Patterns reveal system constraints. Appendix A turns the model into a practical instrument. Prioritization converts findings into investment. Benchmarking adds an external perspective when context is respected. Audit use requires stronger evidence and clearer governance discipline.

Principle

DevOps maturity assessment is not a judgement of people. It is a structured way to understand the system so the organization can improve it.

KEY MESSAGE



DevOps maturity is not a badge. Treating it that way encourages inflated scoring, defensive workshops, and weak improvement plans. Honest assessment requires leadership courage. Leaders must be willing to hear uncomfortable truths, fund invisible work, and improve systems rather than blame teams. This is where assessment becomes more than analysis. It becomes leadership practice.

The organizations that benefit most from assessment are usually the ones that make it safe to report reality. They ask teams what blocks flow and check whether postmortem actions are completed. They fund reliability when it is the constraint, while investing in platforms when repeated work is consuming delivery capacity. They strengthen AI validation when AI-generated artifacts and AI-enabled services move faster than governance and monitoring.

The practical test is simple. After the assessment:

- Does the organization understand itself better?
- Did the conversation reveal the real constraints?
- Did leaders fund the work that teams need to do?
- Did measurements improve decisions?
- Did reassessment show that the system changed?

A maturity model earns its place only when it helps answer those questions.

DECISION AND ACTION GUIDE

Executive decision: The executive sponsor authorizes the assessment scope, evidence access, participation, and non-punitive use of results, and assigns an accountable owner to act on the findings.

Role-specific actions: Assessors facilitate evidence-based scoring; practitioners validate evidence and confidence; leaders identify the constraining capability and convert the result into funded improvement work.

Evidence to review: The maturity profile, evidence notes, confidence rating, constraint hypothesis, prioritized backlog, decision record, and reassessment date.

12.5 Chapter summary

Assessment should begin before transformation investment. Read maturity across the DevOps Nine Pillars, look for patterns, and prioritize the constraint that limits system performance. Use benchmarking carefully, adapt evidence standards for audit, and reassess after real change. In AI-native delivery, include AI-generated artifacts, AI-enabled services, model behaviour, prompt changes, and governance evidence in the assessment scope.

Assessment results should lead to specific improvement actions rather than become an end in themselves. Reassessment should then provide evidence that those actions have strengthened system capability.



This chapter provides a practical, sequenced guide to DevOps transformation from starting point to organizational scale. It helps leaders and practitioners move from assessment to action, with enough structure to guide investment and enough humility to adjust when evidence changes.

DevOps transformation succeeds when an organization improves its organization-wide delivery framework in deliberate steps. The work starts with strategy and evidence, proves value in a real value stream, then expands through reusable capabilities, lightweight governance, and continuous learning.

Chapter 12 explained how to understand the current state. This chapter explains how to act on that understanding. The practical problem is familiar: leaders approve a transformation, teams receive new tooling, ceremonies multiply, and six months later the same approvals, dependencies, fragile releases, and incident patterns remain. Activity increased. Capability did not.

A better transformation begins with a business problem that matters. It selects a scope small enough to learn from, funds the invisible work that blocks flow, and measures outcomes before declaring progress. AI-native delivery raises the stakes, requiring the surrounding delivery system to provide stronger validation, governance, observability, and accountability.

This chapter focuses on how transformation is sequenced and governed as it scales; Chapter 15 describes the enduring organization-wide operating design that supports DevOps at scale.

13.1 Transformation sequence: the first 90 days

A DevOps transformation should begin with strategy and goal alignment. Before teams change tools or processes, leaders need to agree on which outcomes matter. These may include faster product delivery, safer releases, lower operational risk, better customer experience, improved audit evidence, or more sustainable engineering work.

Strategy gives improvement work a reason to exist. Without that reason, transformation becomes a collection of activities searching for justification (Humble et al., 2014; Hornbeek, 2019; Kim et al., 2021).

The next step is an honest diagnosis. The evidence-based assessment from Chapter 12, combined with current-state value stream mapping, provides a view of the principal constraints, wait states, control friction, and improvement opportunities. In AI-native environments, this view also includes how AI-generated artifacts, model changes, prompt updates, and automated recommendations are reviewed and controlled.

Future-state value stream mapping then gives the organization a practical direction. It should describe how work will move with fewer handoffs and better ownership. It should also define approaches to achieving earlier validation, clearer controls, and faster feedback. The future state is not a fantasy architecture. It is a concrete operating model that teams can test in a real value stream.

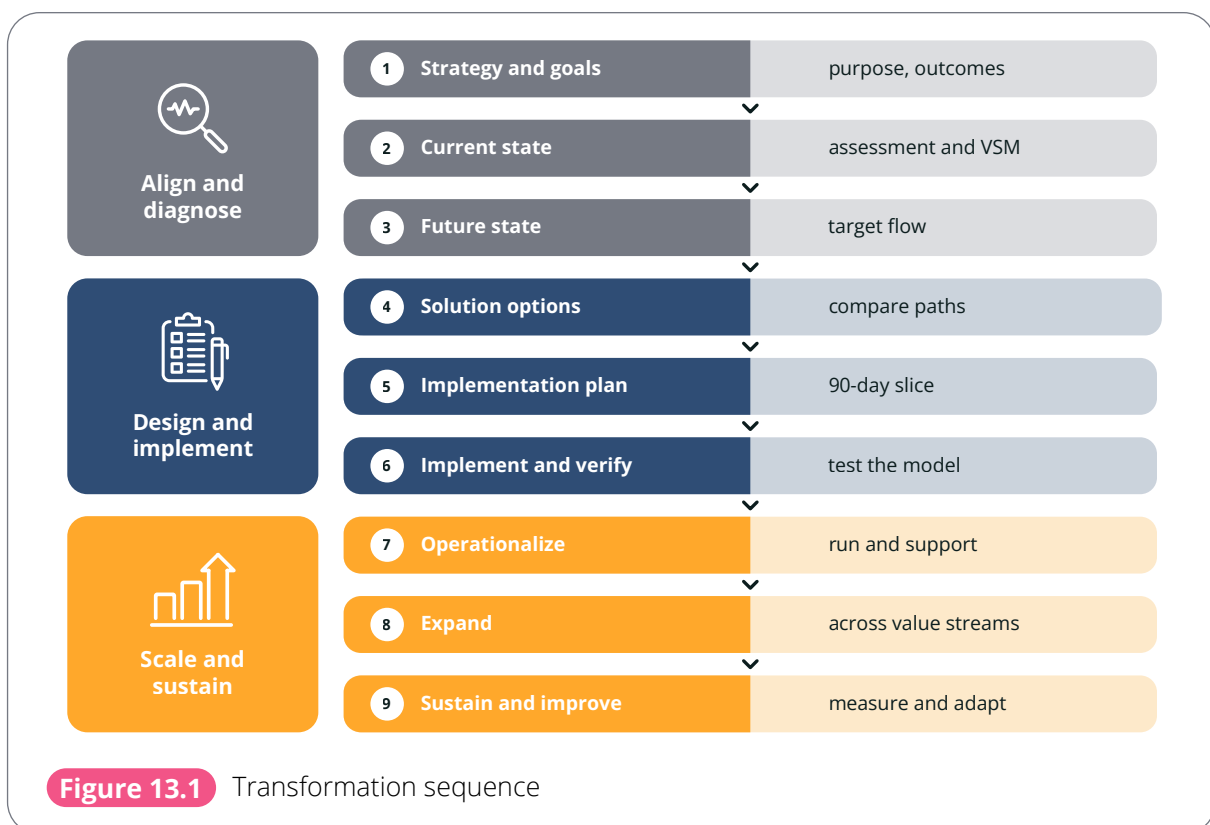
The DEVOPS INSTITUTE® Transformation Playbook

Solution options should be compared before implementation planning begins. A team may improve flow through test automation, platform self-service, policy as code, observability, deployment patterns, or changes to decision rights. The right option depends on the constraint. A weak testing system calls for different investments than weak leadership alignment or fragile environments.

Readiness means the organization has enough sponsorship, team participation, data access, and scope control to make progress. It does not require perfect maturity. It requires a starting point where the pain is visible, the business impact is credible, the team can participate across functions, and the dependencies are manageable enough to enable learning within a bounded period.

Start with a value stream that matters. Avoid a politically impossible area, a low-value pilot that nobody cares about, or a platform-only effort disconnected from product outcomes. The first scope should have enough business relevance to justify attention and enough operational friction to reveal what the organization must fix next.

Figure 13.1 outlines the full transformation sequence: strategy and goal alignment, current-state assessment and value stream mapping, future-state value stream mapping, solution options, implementation planning, implementation and verification, operationalization, expansion, and the combined step of sustainment and improvement. It positions transformation as phased, evidence-driven adoption rather than a big-bang change.



13.1.1 Starting the first 90 days

A 90-day period is recommended because it is long enough to mobilize teams, implement a focused set of improvements, and gather evidence of progress, while remaining short enough to sustain urgency and enable rapid learning. It is a practical planning horizon, not a fixed requirement. A shorter or longer period may be appropriate depending on the scope, organizational readiness, dependencies, risks, and regulatory constraints. Whatever period is selected, it should have clear priorities, measurable outcomes, and a defined review point.

The first 90 days should prove that targeted improvement is possible. They should not promise organizational transformation. The work is to establish credibility, create shared understanding, reduce a meaningful constraint, and produce evidence that leaders can use for the next investment decision (Humble et al., 2014; Hornbeek, 2019).

Figure 13.2 divides the first 90 days into three phases: Days 1-30 align, assess, and map; Days 31-60 improve, automate, and instrument; Days 61-90 measure, learn, and decide how to scale. It also highlights the need to prove improvements in flow, risk reduction, and feedback, create evidence and momentum, and avoid big-bang and tool-first behaviour.

Days 1-30: Align – Assess – Map

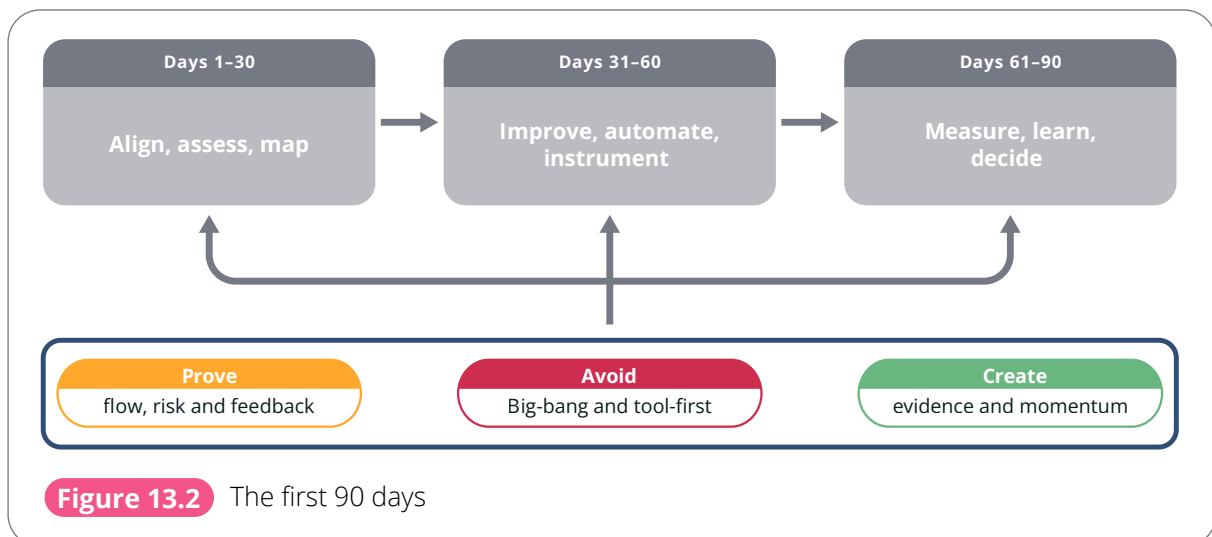
During the first 30 days, the team aligns on the business problem, completes a focused assessment, maps the current value stream, and selects a small measurement set. This is where many organizations rush. They want to do something quickly, so they skip the map and start automating a process they do not understand. This usually makes the wrong work faster.

Days 31-60: Improve – Automate – Instrument

Days 31 through 60 should concentrate on two or three improvements with visible effect. Typical examples include removing a manual approval queue, stabilizing test data, adding deployment telemetry, improving rollback capability, integrating a security check into the pipeline, or creating a self-service environment pattern. AI may help summarize incidents, generate test ideas, or assess change risk, but those suggestions need review and evidence before they affect release decisions.

Days 61-90: Measure – Learn – Decide

Days 61 through 90 should stabilize the change and measure the results. The team should ask whether lead time improved, deployment risk declined, feedback accelerated, quality improved, or on-call load reduced. The organization should also document what blocked the work. Those blockers are often more valuable than the success story because they reveal the next scaling constraint.



The first 90 days should avoid too many metrics. Five good signals are usually enough: one flow metric, one quality metric, one stability metric, one customer or business signal, and one team health indicator. If the pilot involves AI-enabled delivery, add a signal for validation quality, exception rate, or human review burden.

Leadership discipline matters during this period. Leaders should protect the scope, fund the improvement work, and prevent the pilot team from becoming a special exception. A pilot that succeeds only because it receives unusual privileges, extra staffing, or exemption from normal controls gives the organization a story, not a model.

By the end of the first 90 days, the organization should have evidence that it can reduce at least one real constraint without weakening security, reliability, or governance. A team can own build, run, and improve responsibilities with proper support. Evidence can guide investment better than opinion, reporting pressure, or tool preference.

13.1.2 Building the transformation case and defining outcomes

A credible transformation case starts with current pain. Slow lead time, fragile deployments, production incidents, manual compliance work, high rework, and team overload are not only technology problems. They affect revenue and service quality, while also impacting customer trust, employee retention, audit cost, policy delivery, and risk exposure (Hornbeek, 2019; DORA, 2024; PeopleCert, 2025c).

The case should use evidence from assessment, value stream mapping, incident data, flow metrics, reliability signals, security findings, and team interviews. A maturity score alone is rarely persuasive. Leaders need to see what the score means in operational terms. For example, weak Continuous Testing may explain why deployments are slow.

Weak Continuous Monitoring and Observability may explain why deployment frequency is low. Weak Leadership may explain why teams cannot remove cross-functional blockers.

Buy-in is not a single approval. It must be maintained through visible progress and honest reporting. Product leaders need to understand how flow affects customer learning. Security and risk leaders need confidence that controls are improving rather than being bypassed. Finance needs to see why platform, reliability, and technical debt work deserve funding even when the work does not appear as a feature on a roadmap.

Outcomes should be defined before improvement work begins:

- Activities count effort: teams trained, tools installed, dashboards created, workshops completed.
- Capabilities describe what the organization can now do: automated testing, deployment automation, observability, policy as code, platform self-service.
- Outcomes show whether the system improved: shorter lead time, fewer severe incidents, faster recovery, lower defect escape, stronger security posture, better customer experience, or lower operational cost.

A balanced set of measurements protects the transformation from distortion. A speed metric should be paired with a safety metric. Lead time should be read with change failure rate. Throughput should be read with defect escape. Deployment frequency should be read with recovery and customer impact. AI-assisted delivery should be read with validation quality, model or prompt change traceability, and human accountability.

The transformation case should acknowledge trade-offs. Some reliability work will delay a feature. Some platform work will require investment before teams feel the benefit. Some governance automation will require risk and audit teams to participate in design. Honest trade-offs build trust because they show that transformation is being managed as a system, not sold as a slogan.

Table 13.1 Transformation cases

Technical pain	Business translation	Likely investment
Slow lead time	Delayed market or mission response	Flow redesign and test data
High change failure rate	Customer trust and support cost risk	Release safety and observability
Manual compliance	Audit cost and release delay	Policy as code and evidence automation
Fragile environments	Unpredictable delivery	Platform self-service and IaC

Finally, a credible transformation case connects current pain and assessment evidence to its business impact. It links target outcomes and investment to the measured benefit. A maturity score without an improvement decision is only reporting.

13.1.3 Sequencing improvement across the DevOps Nine Pillars

The DevOps Nine Pillars should guide sequencing, not create a checklist race. Trying to raise every pillar equally at the same time usually creates shallow activity across the organization. Mature transformation focuses on the minimum practice set needed to improve flow safely.

Leadership and Collaborative Culture create conditions for change. They provide sponsorship, decision rights, funding, psychological safety, and shared ownership. If these are weak, tooling initiatives tend to stall because teams cannot change the system around the tool. Leaders must remove constraints, not only approve work.

Design for DevOps comes early because architecture determines what delivery can realistically do. Systems with tight coupling, shared databases, unclear ownership, and poor operability resist automation. Continuous

Integration and Continuous Testing follow because fast feedback is the engine of safe flow. A team with weak tests should be cautious about increasing deployment frequency.

The Elastic Infrastructure and Continuous Delivery and Deployment pillars reduce friction. They make environments repeatable, deployments less manual, releases more controlled, and recovery more predictable. Continuous Monitoring and Observability protect reliability, while Continuous Security manages security risk. Observability gives teams confidence that production behaviour is understood. Security embedded in pipelines and platforms prevents late-stage gates from becoming the default control mechanism. Sequencing also depends on context. A regulated organization may need automated evidence and DevSecOps controls earlier. A product team with severe production instability may need observability and Service Level Objective (SLO) work before release acceleration. A platform-heavy organization may need infrastructure self-service before multi-team flow can improve. The sequence should follow the constraint, not fashion.

Build enough capability across the relevant pillars to make the next improvement safe. Do not accelerate a system that cannot detect failure, recover quickly, or explain user impact. Treat AI-generated change as an amplifier: it increases the need for sequencing discipline because it increases the volume and speed of artifacts entering the delivery system.



AI-native delivery connects all Nine Pillars, but in different ways. Leadership defines accountability, Collaborative Culture supports critical review, and Design for DevOps establishes validation and observability requirements. The technical and delivery Pillars ensure that AI-related work is integrated, tested, supported by appropriate infrastructure, secured, monitored, and released under control.

AI-RELATED
CONTENT



13.1.4 Value Stream Mapping and pilot design

Value Stream Mapping turns transformation from general ambition into visible work. It shows how an idea becomes delivered value. A good map includes work steps, wait states, handoffs, approvals, rework, tool transitions, defects, environment delays, information gaps, and production feedback (Kersten, 2018; Hornbeek, 2024; Pereira & Davis, 2024).

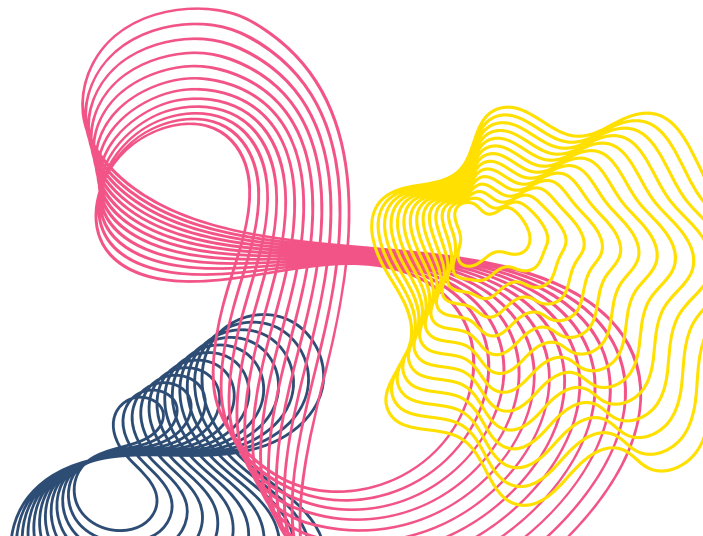
The map should distinguish process time from wait time. Many delivery systems spend surprisingly little time doing active engineering work compared with the time spent waiting for approval, clarification, environment access, test data, security review, release windows, or dependency resolution. That waiting is where many transformation opportunities live.

Current-state mapping should be blunt, evidence-based, and respectful. It should not blame teams for workarounds they use to survive the system. If an operations team manually approves every release, ask what risk that approval is trying to manage. If security review happens late, ask which evidence is missing earlier. If developers avoid a platform, ask what makes the platform difficult to use.

Future-state mapping should remove unnecessary handoffs, shift feedback earlier, and automate controls where appropriate, while reducing batch size, clarifying ownership, and improving visibility. In an AI-native context, future-state maps should also show where AI is permitted, where AI outputs are validated, where human approval is required, and where evidence is captured.

The pilot tests the future-state model. A useful pilot uses a real value stream with real users, measurable outcomes, and cross-functional participation. It should have a clear baseline, a bounded improvement backlog, leadership support, and a plan to capture learning. It should also include normal security, compliance, and operational constraints unless alternative controls have been designed and approved.

Pilots fail as learning mechanisms when they become special cases. Innovation labs disconnected from production often produce polished demonstrations and weak scaling evidence. Hero teams produce morale-boosting stories but not repeatable capability. A pilot should produce improved metrics, reusable practices, platform requirements, governance lessons, and clear evidence for the next scale decision.



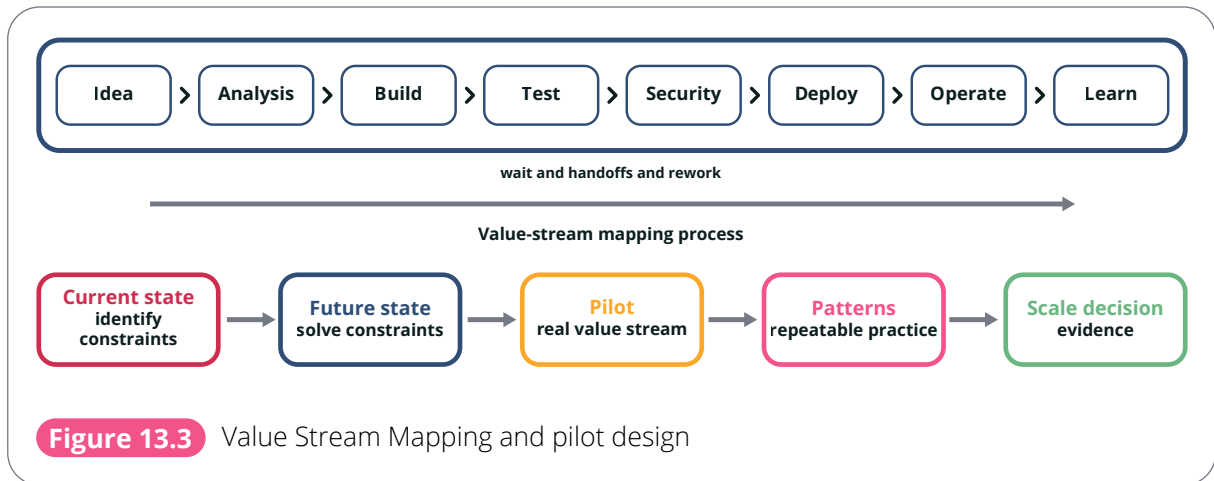


Figure 13.3 connects value stream mapping to pilot design. It depicts the current flow of work, including feedback loops, and highlights the need to identify wait time, handoffs, and rework. It then shows how the identified constraints inform a future state, a pilot in a real value stream, reusable patterns, and an evidence-based scale decision.

13.1.5 Anchor case

Case A: the first 90 days of a regulated value stream transformation

Starting point: Northstar used the Chapter 12 assessment to select one payments value stream. The system already had a pipeline, but test data, customer-journey telemetry, release evidence, and dependency coordination were weak. The sponsor defined a practical outcome: to reduce the wait between an approved payment change and a safe release while improving reliability and audit confidence. The programme deliberately avoided an organization-wide tool rollout.

The first month produced an uncomfortable result. The proposed transformation backlog was too large and mixed product features with platform work, security remediation, training, and reporting. The sponsor narrowed the scope to one customer journey and protected capacity for test data, observability, and release-evidence work. A cross-functional value-stream group met to discuss decisions and evidence rather than status reporting.

The second month exposed a failed intervention. A central platform team supplied a mandatory pipeline template and expected teams to migrate quickly. Product teams bypassed parts of it because it did not support their mainframe dependency, generated slow feedback, and duplicated existing controls.

Pilot design checks

- Choose a real product, service, platform capability, or value stream. It should have visible delivery or reliability pain and a sponsor who can remove blockers.
- Define boundaries clearly. Include what is inside the pilot, what is outside, what dependencies will be accepted, and what decisions the team can make without escalation.
- Treat AI-assisted work as part of the scope where it already exists. Generated code, generated tests, AI-assisted risk scoring, and model-related changes need the same discipline as other production changes.

TIP



Table 13.2 Transformation 90-day startup activities

Period	Operating decisions	Evidence expected
Days 1-30	Confirm the value-stream boundary, sponsor, accountable owners, baseline, principal constraints, risk tiers, and outcome hypothesis. Map work, wait time, evidence gaps, and dependencies.	Current-state map, baseline measures, decision rights, risk and evidence gaps, and a small, prioritized backlog.
Days 31-60	Strengthen test data, customer-journey telemetry, release-readiness evidence, and the platform path. Define approved AI uses and provenance requirements.	Repeatable test data, SLI and SLO, release evidence record, platform adoption feedback, and AI-control evidence.
Days 61-90	Deliver small changes, use controlled deployment and release, rehearse recovery, review outcomes, and adjust the intervention.	Flow and wait-time changes, quality and stability results, release decisions, recovery results, team feedback, and audit evidence.
After day 90	Reassess the constraint, stop activities that do not improve the system, scale reusable patterns, and fund the next capability.	Updated maturity profile, decision record, reusable platform or policy pattern, and the next improvement hypothesis.

Northstar stopped treating non-adoption as resistance. Platform engineers worked with the pilot team, separated mandatory evidence contracts from optional implementation choices, and improved the golden path around the real workflow.

During the third month, the team delivered small payment changes through the revised path. Deployment of production and release to customers was recorded separately. A canary cohort, service-level telemetry, automated policy evidence, and rehearsed recovery supported the decision. AI-assisted code and incident summaries were allowed only through approved tools and the same provenance and review path as other artifacts.

Outcome and trade-off: Northstar did not declare the value stream transformed after 90 days. It reduced avoidable waiting, improved release and audit evidence, and made customer-impact telemetry part of decision-making. One planned feature was deferred, and the platform path still required work for legacy services. The review nevertheless produced enough evidence to continue funding the capability and scale the reusable evidence contract to a second value stream.

Transferable lessons: A first-90-days plan should create evidence and learning, not a miniature organization rollout.

Select one meaningful value stream, protect enabling work, test the operating model through real change, and be willing to stop or redesign an intervention that does not help teams deliver safely.

DECISION AND ACTION GUIDE

Executive decision: The sponsor selects the pilot value stream, names the accountable value-stream owner, approves the 90-day outcome, funding, and risk boundaries, and defines the stop, continue, or scale criteria.

Role-specific actions: Product, engineering, platform, security, finance, risk, and governance roles execute the sequence; the transformation lead removes dependencies and protects enabling work.

Evidence to review: A pilot charter, baseline and target, funded backlog, dependency log, review cadence, 30-, 60-, and 90-day evidence, and the resulting decision.

13.2 Scaling and governing the transformation

A successful pilot does not scale by copying every local practice across the organization. Scale requires reusable patterns, product-managed platforms, clear funding, embedded governance, and decision rights that preserve local accountability while reducing repeated work. This section explains how transformation expands from one value stream to broader organizational capability without recreating central queues.

13.2.1 Scaling from team to programme to organization

Scaling begins when the pilot stops being only a local improvement and starts revealing system constraints. At team level, the focus is on local flow, ownership, testing, deployment, observability, and feedback. At programme level, the main work becomes dependency management, shared environments, release coordination, architecture, funding, and platform support (Humble et al., 2014; Hornbeek, 2019; Skelton & Pais, 2025).

Programme scaling should not copy the pilot everywhere. It should extract reusable patterns, identify systemic constraints, and create enabling capabilities for several teams. These capabilities may include shared backlog visibility, programme-level value stream mapping, reusable pipeline templates, common observability patterns, platform services, security guardrails, and programme-level metrics.

The organization should be careful about what it scales. Reusable patterns are worth scaling. Local workarounds and hero behaviour, along with manual approvals, tool sprawl, and metrics without context should be retired.

AI changes scaling pressure by increasing the volume of work entering the DevOps system. The challenge is not simply to accommodate greater output, but to help teams absorb the additional work safely through shared patterns while preserving local ownership and accountability.

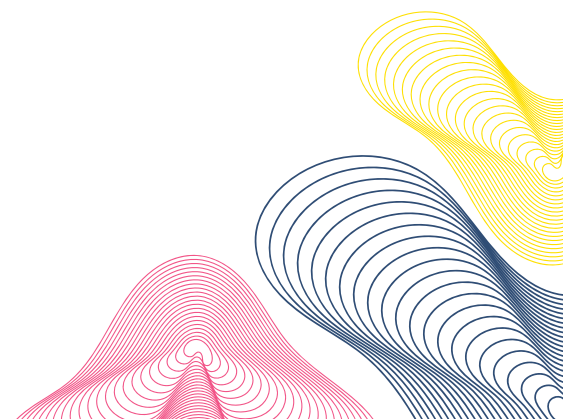
**AI-RELATED
CONTENT**



A workaround that was tolerable in a pilot can become a serious drag when fifty teams depend on it.

Organization scaling changes the problem again. The work expands into operating models, portfolio funding, governance, architecture, risk, compliance, talent, outsourcing, and platform strategy. Organization-wide DevOps requires standardization where it reduces friction and local autonomy where teams need context-sensitive decisions.

Architecture becomes a scaling discipline. Repeated service coupling, shared data dependencies, brittle integrations, and environment conflicts create coordination costs. Organization-wide architecture should help reduce dependency load through clear service ownership, Application Programming Interface (API) contracts, contract testing, platform guardrails, deployability patterns, and observability requirements. Architecture that only reviews designs after decisions have already hardened usually arrives too late.



13.2.2 Governance, platform enablement, and funding

Transformation governance should maintain alignment, remove blockers, prioritize investment, manage risk, track outcomes, and share learning without becoming another approval queue. The governance forum should focus on constraints and decisions, using guardrails rather than gates, automating evidence collection where possible, reviewing outcomes rather than activity volume, and escalating dependencies that teams cannot resolve locally. Status reporting is useful only when it helps someone act (Humble et al., 2014; FinOps Foundation, 2025).

Risk review still matters, and material policy decisions, regulatory interpretation, and high-risk exceptions may require human judgement. The difference is that evidence should travel with the work rather than be reconstructed at the end.

Platform enablement provides the reusable capability that lets improvement scale.

Product teams should not have to invent their own pipeline patterns, environment provisioning, secrets management, observability setup, policy checks, and deployment automation each time. A good internal platform reduces cognitive load and makes the safest path the easiest path.

The platform team should operate as a product team. Its users are internal engineering teams. Adoption, developer satisfaction, time saved, lead time reduction, compliance friction, operational reliability, and support quality are meaningful signals. Mandating a weak platform usually creates avoidance behaviour. Building a useful platform creates adoption through value.

Sustained funding is essential. Project-only funding creates predictable damage: teams disband, knowledge is lost, platform work is postponed, reliability work competes badly with features, and improvement becomes optional. Product and value-stream funding, platform product funding, risk-based modernization funding, and dedicated transformation investment all have a place when used deliberately.

Table 13.3 Transformation governance practices and pitfalls

Governance should provide	Governance should avoid
Investment decisions based on evidence	Extra approval gates with no risk reduction
Dependency escalation and blocker removal	Status theatre and reporting burden
Risk-based guardrails and exception paths	Punitive metric reviews
Learning shared across teams	Central command and control



AI-native transformation requires explicit funding. Teams need protected capacity to adapt policies, strengthen validation, maintain oversight, and build the skills needed to use AI responsibly. Treating AI adoption as cost-free productivity only shifts effort into hidden review work, quality exposure, and unclear accountability.

**AI-RELATED
CONTENT**



13.3 Adapting and sustaining the transformation

Transformation must remain responsive to evidence because constraints, risks, technologies, and business priorities change. Anti-patterns and context-specific starting points help leaders recognize when the chosen approach is distorting the system, while an adaptive roadmap keeps funding, capability building, and reassessment connected. This section focuses on sustaining improvement after the initial programme energy has passed.

13.3.1 Anti-patterns and context field notes

Transformation anti-patterns usually announce themselves early. The organization installs tools and flow does not improve. Dashboards multiply, but decisions stay the same. Reporting lines change and dependencies remain untouched. Automation makes a bad process faster. A central DevOps group owns the label while teams continue to wait for enablement. Security remains a late gate. The pilot succeeds and never spreads. Metrics become a weapon, so teams hide problems.

The big-bang anti-pattern deserves special attention. It usually begins with good intent and poor sequencing. Leaders launch organization-wide changes before local proof, restructure teams before capability exists, position tool rollout as transformation, and set aggressive timelines without understanding system constraints. Teams experience confusion, overload, and compliance pressure. Leaders may see activity, while the delivery system barely changes.

The better pattern is phased and evidence-driven. Start with one value stream, prove measurable improvement, capture reusable patterns, and scale deliberately. Reassess as the system changes. This approach may look less dramatic from the outside. It is usually more durable because it creates learning before it creates scale.



Table 13.4 Transformation anti-patterns and corrections

Anti-pattern	Early signal	Correction
Tool-first	Tools are installed, but flow does not improve.	Map the value stream, target the constraint, and measure outcomes.
Big bang	Everything changes at once; teams experience overload.	Pilot one value stream, learn, and scale deliberately.
Reorganization only	Reporting boxes change while dependencies remain.	Redesign ownership, interfaces, and decision rights around value.
Automate the old process	Queues and rework move faster but remain.	Simplify the process and remove waste before automating it.
Central DevOps ownership	Teams wait for a central group to enable delivery.	Use central teams for platforms and coaching; keep outcome ownership with delivery teams.
Late security	Security becomes a release blocker.	Build threat modelling, policy, and evidence into design and pipelines.
Punitive metrics	Problems are hidden, and local optimization increases.	Use metrics for learning and system improvement, not ranking.
Pilot trap	The pilot succeeds, but reusable patterns do not spread.	Capture patterns, fund adoption, and reassess as the system changes.

Context matters. A technology organization with strong engineering talent and existing tools may begin with observability, SLOs, platform enablement, and delivery consistency. Its constraints may be fragmentation, cost, architecture, and reliability at scale. The transformation emphasis is on standardizing golden paths without slowing high-performing teams.

A regulated organization has different constraints. Legacy systems, approval gates, segregation of duties, and audit expectations shape the operating model. Its first move may be value stream mapping, automated evidence, DevSecOps controls, and deployment risk reduction. The transformation emphasis is on replacing manual gates with automated controls while building trust with risk and audit teams.

A public sector body often works under policy-driven outcomes, procurement limits, funding cycles, legacy contracts, public accountability, and skills constraints. A sensible starting point may be one citizen-facing service or an internal value stream. Incident learning, environment consistency, and delivery transparency can improve service outcomes and public trust without pretending governance constraints will disappear.

AI adoption does not follow a single path. The same playbook should be applied differently depending on organizational purpose, risk, regulation, and public impact. What stays constant is the evidence-based approach; what changes is the order of implementation, the level of assurance required, and the emphasis placed on certain controls.

AI-RELATED CONTENT



Table 13.5 Context-specific starting points

Context	Typical constraints	Sensible starting point	Control emphasis
Technology organization	Fragmentation, cost, architecture, reliability at scale	Observability, SLOs, platform enablement, delivery consistency	Standardize golden paths and AI platform governance without slowing strong teams.
Regulated organization	Legacy systems, approval gates, segregation of duties, audit expectations	Value Stream Mapping, automated evidence, DevSecOps controls, lower-risk deployment	Replace manual gates with automated controls; strengthen explainability and traceability.
Public sector body	Policy outcomes, procurement and funding cycles, legacy contracts, public accountability, skills constraints	One citizen-facing or internal value stream; incident learning, environment consistency, delivery transparency	Protect accessibility, fairness, public trust, and procurement obligations.

13.3.2 Roadmap, funding, and continuous capability building

The transformation roadmap converts assessment findings, value stream maps, and pilot evidence into an adaptive plan informed by DORA and flow metrics, reliability data, security and compliance requirements, and product priorities. It should not read like a fixed multi-year installation schedule. It should show how the organization will build capability over time and how evidence will change the plan (Hornbeek, 2019; DORA, 2024; Pereira & Davis, 2024).

A practical roadmap uses horizons. The first 90 days prove value and establish baseline metrics. The next three to six months stabilize and extend patterns to adjacent teams or services. The six-to-twelve-month horizon scales platform and governance capabilities. The twelve-month-plus horizon improves the organization's operating model, including funding, architecture, measurement, and leadership routines.

The roadmap should remain evidence-driven and name capability themes, target outcomes, and investment needs. To ensure effective execution, it should also identify owners, dependencies, and risks, and define measures and reassessment points.

Importantly, it should protect invisible work. Platform development, reliability, security automation, technical debt reduction, observability, training, and coaching often determine whether visible delivery improves. They need funding because they rarely survive when treated as spare-time work.

Funding should follow the operating model the organization wants. Long-lived product or value-stream funding supports durable ownership. Platform product funding supports reusable capability. A transformation investment fund can help remove early constraints. Risk-based modernization funding helps address legacy systems and compliance exposure. Leaders should expect improvement work to compete with feature work and should make that trade-off explicit.

Return should be demonstrated through measurable outcomes including:

- reduced lead time
- lower incident impact
- less rework
- improved audit evidence
- better team productivity
- stronger reliability
- better service outcomes.

The evidence should also include team health. A transformation that improves a dashboard while exhausting responders is creating another problem.

The closing principle is practical. A successful DevOps transformation does not make change look dramatic. It makes better delivery, safer operations, stronger feedback, and continuous learning part of normal work. The organization still has constraints, but it becomes better at seeing them, funding the right work, and improving the system without waiting for a crisis.

DevOps transformation is continuous capability building: evidence informs scope, improvement proves value, reusable patterns enable scale, and reassessment keeps the transformation aligned with changing constraints. Continuous capability building becomes even more important in AI-native delivery, where increased speed and output must still be matched by validation, governance, observability, and human accountability.

13.4 Chapter summary

DevOps transformation should begin with strategy, evidence, and a clearly selected value stream. Assessment and current-state value stream mapping reveal where work waits, where risk accumulates, and which capabilities constrain the system. The first 90 days should test a focused outcome hypothesis, deliver a measurable improvement, expose blockers, and create evidence for the next investment decision.

The DevOps Nine Pillars should be sequenced according to constraints rather than implemented as a checklist. Scale should follow proven patterns through platforms, guardrails, architecture, sustainable funding, and balanced measurement. The roadmap must remain adaptive, with periodic reassessment to reflect new evidence, technology, risk, and business priorities.

This chapter defines measurement as a feedback control mechanism for the DOI® Operating Model. The goal is to help teams, leaders, and stakeholders see the delivery system clearly enough to improve it. Good measurement makes constraints visible, connects technical performance to outcomes, and creates conditions for better decisions.

The chapter uses a balanced model across flow, quality, stability, and value. These four core measurement domains provide different views of system performance and should be interpreted together rather than as isolated measures. AI behaviour extends these core domains with additional measures for AI-native delivery, where models, prompts, retrieval systems, data pipelines, and AI-assisted workflows become part of production responsibility. A modern DevOps measurement programme must measure both software delivery and the behaviour of the systems that delivery now enables.

The practical discipline is to start small, define metrics carefully, review them in context, and protect them from gaming. Metrics become useful when they change understanding or action. They should help teams identify trends, test hypotheses, locate constraints, assess risk, or determine whether

an intervention is producing the expected outcome. The task is not simply to report a number, but to understand what may explain it, what other evidence supports that interpretation, and what decision should follow. Metrics should guide learning and investment, not become a substitute for judgement. The experienced practitioner knows this well: a clean dashboard can still hide a messy system.

14.1 Measurement system foundations

A trustworthy measurement system begins with a clear purpose and a reliable path from operational events to decision-ready evidence. It should define the domains being measured, the data sources, ownership, lineage, quality, privacy, and interpretation rules before numbers reach a dashboard. This section establishes measurement as part of the DevOps architecture rather than as a reporting layer added after the work.

Table 14.1 Types and use cases of measurement

Measurement area	What to check
Flow	How work moves from request to delivered value
Quality	Whether work is correct, secure, maintainable, and fit for use
Stability	Whether change is safe and recovery is fast
Value	Whether delivery improves customer, mission, risk, or business outcomes
AI-specific measures	Whether models, prompts, retrieval, and agents behave as intended in production

Measuring what matters

14.1.1 The instrument panel of the DevOps operating system

Measurement is the instrument panel of the DevOps operating system. A pilot would be reckless to fly by altitude alone, and a delivery organization is just as exposed when it steers by a single number. Teams need signals about flow, quality, stability, value, supplemented where relevant by AI behaviour and risk signals, because each tells part of the truth. The hard work is learning how to read those signals together before they become expensive surprises (Forsgren et al., 2018; DORA, 2024).

Reporting and **measurement** are often confused. Reporting usually looks upward and backward. It summarizes what happened to people who were not close to the work. Measurement, when used well, sits close to the work and helps people decide what to do next. In fact, a metric is useful when it informs a decision; if it never alters a choice, it is usually just noise, regardless of how polished the dashboard looks. The continuous working loop is to measure, interpret, decide, improve, verify, and measure again.

The purpose of measurement in DevOps is to create feedback. First, teams measure the system to interpret the signal, which then allows them to decide what constraint is visible. Once identified, they can improve the system and then verify whether the change helped. This is the same engineering instinct used in troubleshooting: first observe carefully, then form a hypothesis, make a controlled change, and finally, check the result.

The difference at organizational scale is that the system being measured includes people, process, architecture, automation, governance, and production behaviour.

A mature measurement system supports decisions at several levels:

- A team may use lead time and defect escape data to improve its delivery workflow.
- A product leader may use telemetry and adoption data to decide whether a feature should receive more investment.
- A Site Reliability Engineering (SRE) team may use error budget burn to advise on release risk.
- A security leader may use vulnerability age and policy failure trends to fund control automation.
- An executive may use flow, reliability, cost, and customer-impact data to decide where invisible platform work deserves real funding.

Single-metric management is dangerous because it rewards local optimization. Deployment frequency can rise while customer value stays flat. Test coverage can improve while defect escape remains stubborn. Similarly, incident counts can fall because teams stop reporting smaller failures, just as cycle time can improve because work is split into artificial fragments. The metric may look healthier while the system quietly becomes worse. This is old operational wisdom: once a measure becomes a target without context, the signal starts to degrade.

Measurement also depends on trust. If teams believe metrics will be used to rank them, shame them, or punish them, they will protect themselves.

They may argue about definitions, avoid difficult work, hide bad news, or optimize the number instead of the system. Leaders who want honest measurement must protect honesty. They need to ask what the signal reveals about the system before asking who can be held responsible. The practical rule is simple enough to remember: measure close to the work, review metrics with the people who can act, and connect every important metric to an improvement decision. Metrics should help the organization see clearly rather than decorate dashboards, comfort leaders, or create theatre. Good measurement makes reality harder to ignore and easier to improve.

The operating habit matters as much as the metric. A weekly review that only asks whether the number is green teaches teams to defend the number. A useful review asks what the signal suggests, which constraint is visible, what evidence is missing, and what decision should change. This is where experienced leaders earn trust. They separate noise from signal, then fund the work needed to improve the system.

Measurement should also be located at the right level. Some signals belong to the team because the team can act immediately. Other signals belong to the value stream because the constraint crosses teams, environments, release practices, or architecture boundaries. Organization-level metrics should be reserved for investment and operating-model decisions. When every metric is pushed upward, teams lose the local feedback they need. When every metric stays local, leaders miss the systemic problems that only investment and policy changes can fix.

In an AI-native environment, measurement must extend beyond traditional delivery and operational indicators. It should show whether AI-assisted work and AI-enabled products are producing reliable, explainable, and accountable outcomes. The organization remains responsible for those outcomes, even when AI contributes to recommendations, classifications, or decisions.

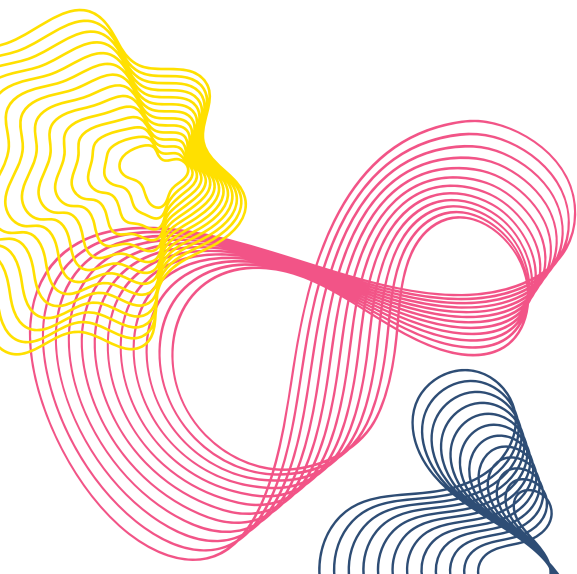
AI-RELATED
CONTENT

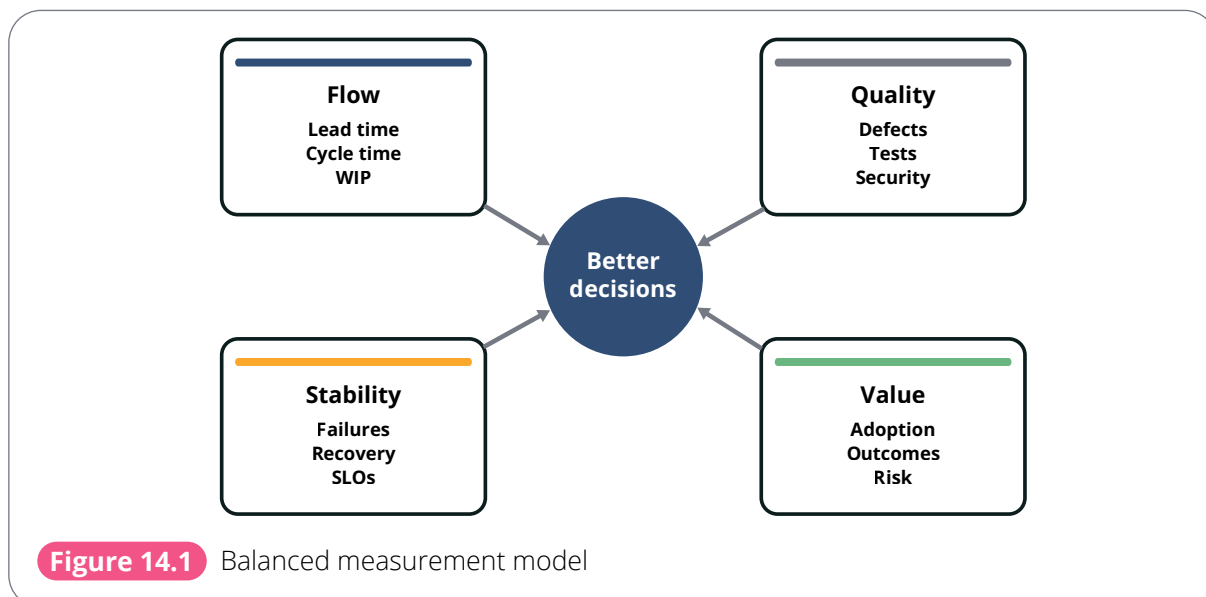


14.1.2 Measurement domains and the data pipeline

A useful DevOps measurement model needs balance. This chapter uses four core domains (DORA, 2024; Hornbeek, 2024; PeopleCert, 2026d):

- **Flow:** shows how work moves through the delivery system. It includes lead time, cycle time, and work in progress.
- **Quality:** shows whether the work is correct, secure, maintainable, and fit for use. It includes defects, tests, and security.
- **Stability:** shows whether change can be absorbed safely and whether the service can recover quickly. It includes failures, recovery, and service level objective (SLO) performance.
- **Value:** shows whether delivery produces outcomes that matter to customers, citizens, employees, regulators, or the business. It includes adoption, outcomes, and risk.





These domains must be read together. Specifically, fast flow with poor quality creates rework, support load, and loss of trust. High stability with poor flow may indicate excessive control, large queues, and delayed learning. Strong delivery metrics with weak value metrics may mean the organization is efficiently producing output that does not matter much. High value with poor stability can work for a short period, then fatigue, incidents, and customer harm catch up. Balanced measurement prevents one part of the system from declaring victory while another part pays the bill.

Flow metrics include lead time, cycle time, and throughput, alongside work in progress, queue time, and flow efficiency. These metrics show whether value can move through the organization without constant waiting, rework, dependency negotiation, and handoffs. Quality metrics include defect escape rate, test effectiveness, code health posture, rework, and customer-reported defects. They show whether the delivery system is producing confidence or simply moving faster. Stability metrics include the DORA stability measures (change failure rate and failed deployment recovery time) plus incident frequency, service restoration time, SLO attainment, deployment health, and alert quality. Operational signals such as latency, traffic, errors, saturation, and availability provide additional evidence of service behaviour.

Value metrics include adoption, retention, conversion, customer satisfaction, task completion, cost reduction, risk reduction, and policy delivery. They help close the gap between engineering activity and useful outcomes.

The measurement domains are only as reliable as the data underneath them. Metrics do not appear by magic. They are calculated from toolchain events, version control, issue trackers, and CI/CD systems. They are also derived from deployment tools, incident records, and observability platforms. Furthermore, they incorporate data from security scanners, product analytics, customer support systems, and financial systems. Each source has assumptions and can be incomplete. Manual status updates, missing timestamps, duplicate work items, inconsistent severity definitions, and fragmented toolchains can corrupt the signal before anyone sees a dashboard. A measurement data pipeline usually follows a predictable path: raw event collection, normalization, context enrichment, metric calculation, interpretation, and decision. However, the context step is where many organizations fail because a timestamp by itself is weak. For instance, a deployment event becomes more useful when it includes the service name, version, environment, region, feature flag state, and incident link. Other useful context includes the work item type, customer impact, and business priority.

Similarly, a defect count becomes more useful when defects are classified by severity, source, escape point, and affected user path.

Definitions matter. If one team counts lead time from ticket creation and another counts from code commit, comparison is meaningless. If one service counts a rollback as a change failure and another counts only customer-visible incidents, the numbers will mislead. To prevent these discrepancies, metric definitions should be documented. Each definition should specify the metric's purpose, formula, data source, scope, owner, review cadence, and known limitations. This work is dull, which is often why it is skipped. It is also why many measurement programmes become arguments about arithmetic instead of conversations about improvement.

The same need for definition applies to AI-native measurements: model, prompt, retrieval, and data updates should be visible as operational events, so teams can link performance changes to their cause.

The best measurement systems make uncertainty visible. Confidence rating should be part of the review when data is incomplete, definitions are new, or toolchain integration is weak. A metric with low confidence can still be useful if it triggers better data collection and better questions. It only becomes dangerous when leaders treat it as precise. Precision in presentation is not the same as accuracy in measurement.

Measurement architecture should be treated as part of the DevOps architecture. Data capture, identity, metadata, retention, privacy, and access controls need design attention. Teams should know which signals are authoritative, how corrections are made, and how sensitive data is protected. This is especially important for AI-native systems because prompts, user inputs, model outputs, and evaluation labels may contain sensitive information. The measurement system must inform decisions without creating new privacy or security exposure.

14.2 Core delivery and outcome measures

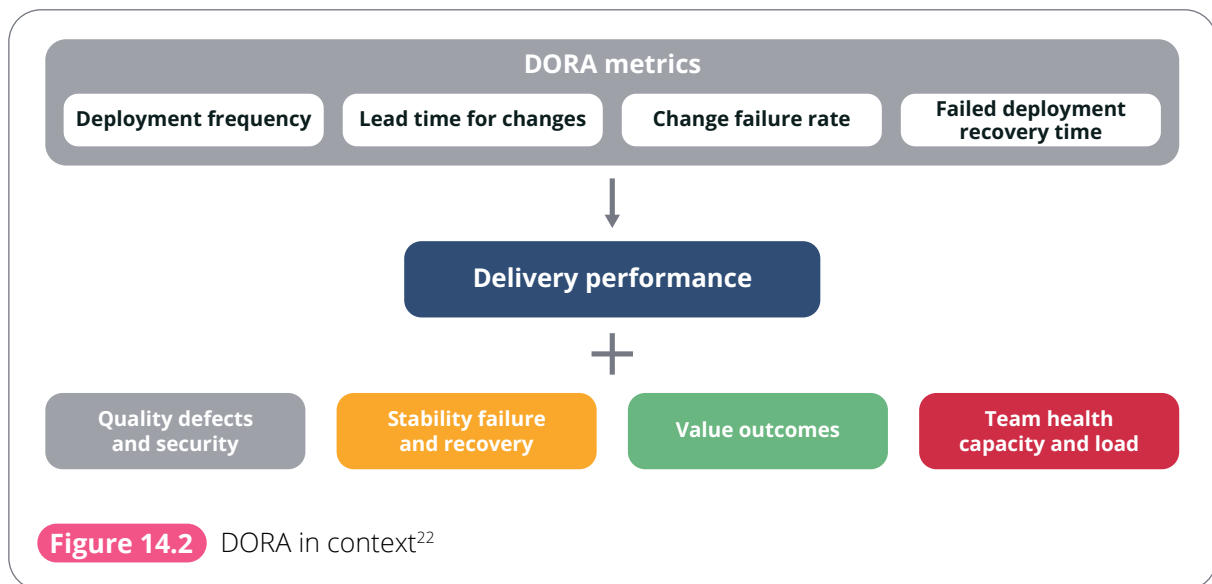
No single metric can describe the health of a DevOps system. DORA, flow, quality, stability, and value measures should be read together so that improvements in speed are tested against safety, reliability, and useful outcomes. This section explains the core measures and the context required to use them without rewarding local optimization or metric gaming.

14.2.1 DORA metrics in context

DORA metrics are among the most useful software delivery measures because they combine speed and stability. They help an organization understand whether its delivery system can move change into production with reasonable frequency, acceptable risk, and recoverability. They are strong signals, and they have earned their place in modern DevOps measurement. However, they are incomplete if treated as the entire measurement system (DORA, 2024; DORA, 2026a; DORA, 2026b).

Figure 14.2 places DORA metrics inside a broader measurement system. Lead time for changes, deployment frequency, failed deployment recovery time, and change failure rate inform delivery performance, but they must be interpreted alongside quality, stability, value, and team sustainability signals.





This book uses four DORA software delivery performance measures for continuity with the DEVOPS INSTITUTE® measurement model. Deployment frequency is how often deployments occur. Lead time for changes is the time from committed change to production. Change failure rate is the proportion of deployments that require immediate intervention. Failed deployment recovery time is the time required to recover from a failed deployment that needs intervention (DORA, 2024; DORA, 2026a; DORA, 2026b).

Definition discipline matters more than labels. Organizations should document the scope, start and end points, denominators, exclusions, severity treatment, incident linkage, and data sources for each measure. A review that mixes different calculation rules without making them explicit creates confusion during interpretation and benchmarking.

DORA metrics measure delivery system performance well. They show deployment cadence, change speed, deployment stability, recovery capability, and delivery health trends. They are especially useful when reviewed over time for a product, service, or value stream. A team that steadily reduces lead time for changes while not worsening its change failure rate or failed deployment recovery time is usually improving its ability to deliver safely. A value stream that rarely deploys and

recovers slowly probably has batch-size, automation, testing, deployment, architecture, or ownership problems that deserve attention.

DORA metrics do not tell the whole story. They do not prove that delivered work was valuable, that customers benefited, or that quality improved, security risk decreased, architecture became easier to change, or teams are working sustainably. A service can have strong DORA results while producing low-value changes, accumulating technical debt, or burning out responders. DORA must be read with quality, stability, value, and team-health signals.

Honest use of DORA starts with context. Measure teams, services, or value streams in ways that reflect how work reaches production. Avoid comparing unrelated teams as if the work were equivalent. A team maintaining a safety-critical payments service, a team operating an internal reporting tool, and a team experimenting with a customer-facing recommendation engine may have very different risk profiles and delivery patterns. Ranking them by deployment frequency is poor measurement practice.

DORA also needs countermeasures and careful interpretation. Deployment frequency should be read with change failure rate, failed deployment recovery time, customer impact, and deployment rework rate. Lead time for changes should be paired with defect escape, team load, and value outcomes.

Table 14.2 DORA metrics use

DORA definition discipline	What to check
Document the model	Four measures with documented local calculation rules
Document the scope	Team, service, product, or value stream
Pair the metric	Speed signal plus quality, stability, and value signal
Review the trend	Use movement over time, not isolated ranking

Other rework and repeat-incident measures remain useful quality signals, but they should not be mislabelled as DORA metrics. If a metric improves while its countermeasure worsens, the system is probably being distorted (DORA, 2026a; DORA, 2026b).

Sudden elite performance should be reviewed carefully. It may reflect real improvement, and teams deserve credit when the system gets better. It may also reflect definition changes, selective measurement, missing incident links, or gaming. Mature leaders do not celebrate the dashboard until they understand what changed in the system. The most appropriate question is not whether the number looks impressive, but what decisions the metric helped improve.

A practical DORA review often starts with a simple question: what would we do differently if this signal is true? If lead time for changes is rising, the answer may involve smaller batch sizes, less approval delay, better test automation, or clearer product slicing. If change failure rate is rising, the answer may involve deployment patterns, production validation, incident analysis, or architectural simplification. If defect escape or rework is rising, the organization may need to examine test effectiveness, operational readiness, and the pressure to ship incomplete work.

DORA metrics are stronger when paired with real examples from the system. A chart showing recovery time is helpful, but a review of the last three failed deployments provides much deeper insight. Teams should investigate what failed, how the failure was detected, and how long mitigation took. They also need to analyse what slowed recovery and verify whether the same pattern appears elsewhere. Numbers provide the entry point; engineering judgement turns them into improvement work.

14.2.2 Flow metrics across the value stream

Flow is the movement of work from request to delivered value. Flow metrics help teams understand where work moves, waits, accumulates, or is repeatedly reworked. Their value lies in identifying constraints and determining whether changes to the delivery system improve movement towards value. In many organizations, the greatest source of delay is not the coding time but the wait time. Work waits for planning and prioritization, architecture and security reviews, environment and test-data provisioning, deployment approvals, and business sign-off. Flow metrics help reveal those hidden queues (Kersten, 2018; Hornbeek, 2019; Pereira & Davis, 2024).

Lead time is the elapsed time from one defined point in a value stream to a later defined point in that same value stream. It includes all cycle times and wait times for the steps within that measurement scope.

The time from request, commitment, or code change to delivery is an example of lead time. This is useful for understanding how long customers, users, regulators, or business stakeholders wait for change.

Lead time = Σ cycle times + Σ wait times, for all steps between the selected start and end points.

Cycle time is the active time required to transit one step in the value stream, excluding wait time before or after that step. Cycle time equals the active processing time for one value stream step.

It is useful for understanding delivery execution once the team has begun work. The distinction matters because each metric points

to a different constraint. High lead time with low cycle time often means work is waiting before the team starts. By contrast, high cycle time usually means the work is too large, too dependent, too unclear, or too burdened by rework.

Wait time measures time spent waiting between steps. The waiting between value stream steps is not cycle time. It is wait time, sometimes referred to as queue time, delay time, or handoff time.

In mature value stream analysis, wait time often explains more delay than active work time. A ticket may spend two days in development and three weeks waiting for dependency clarification, test data, approval, and a release slot.

Flow efficiency compares active work time to total elapsed time. Low flow efficiency is not a reason to shame a team. It is a signal that the system contains avoidable waiting.

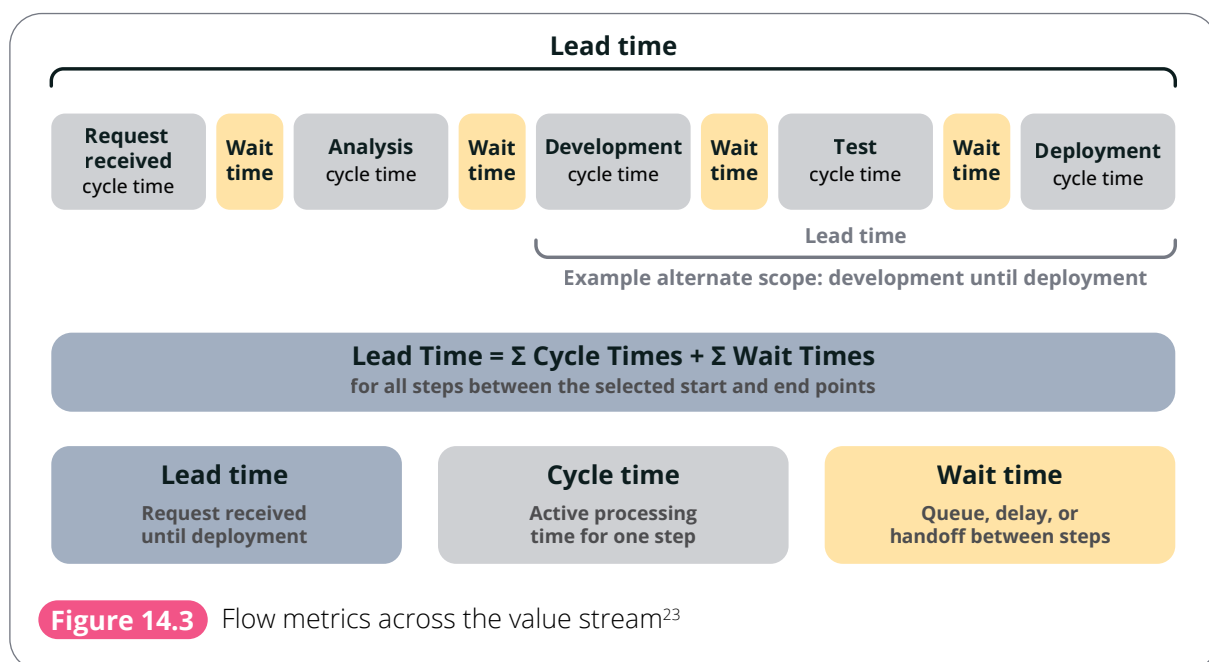
Figure 14.3 traces work moving from request received through start work, build, test, deploy, and done. Lead time spans the selected start and end points, and the figure also shows an example alternate scope from the start of development to deployment. Cycle time is the active work inside a step, and wait or queue time sits between steps.

Throughput measures the number of completed work items over a period. It should be interpreted with item size, item type, and

quality signals. A team can increase throughput by splitting work into smaller items, which may be healthy. A team can also increase throughput by closing thin slices of low-value work, which may be misleading. Throughput without value and quality context becomes another way to count activity.

Work in Progress (WIP) measures the amount of work started but unfinished. High WIP is one of the quiet killers of delivery performance. It increases context switching, hides blockers, stretches cycle time, and makes forecasting unreliable. Leaders often ask teams to start more work because everything is urgent. The system then slows because too much work is competing for the same attention, environments, and reviewers. This overload also affects test capacity, security support, and deployment paths.

Flow item types improve interpretation. Features create new customer, business, or mission capabilities. Defect fixes repair quality problems. Risk work reduces security, compliance, operational, or regulatory exposure. Debt work improves future delivery by reducing friction, fragility, and maintainability problems. A value stream that spends most of its capacity on defects and emergency risk work has a different problem than a value stream investing heavily in features. The mix tells leaders whether the system is creating value, repairing itself, or paying for old decisions.



Value Stream Mapping supports flow measurement by exposing the work steps, wait states, handoffs, approvals, rework loops, tool transitions, and information delays through which work moves. Teams should resist the temptation to draw up the official process. The useful map is the one that shows what really happens on a Tuesday afternoon, when a production defect, a security finding, and a product deadline are all competing for attention.

Flow metrics should guide system improvement. If WIP rises while throughput falls, the system is overloaded. If cycle time is high, work may need smaller slices, clearer ownership, better architecture, or fewer dependencies. If queue time dominates lead time, approvals, handoffs, or shared services may be the constraint. If throughput rises while defects rise, speed is damaging to quality. Flow measurement is useful because it makes these trade-offs visible before the organization simply asks people to work harder.

WIP limits are one of the most practical interventions flow metrics can support. A WIP limit does not make work less important. It forces the organization to confront capacity, priority, and sequencing. When leaders demand that every request start immediately, they create the illusion of progress and the reality of delay. A visible WIP limit helps teams finish work, shorten feedback loops, and expose decisions that need leadership attention.

Flow metrics should also be segmented by work type. Feature work, defect work, risk work, and debt work often move through different paths and experience different delays. Combining them into one average can hide the real constraint. For example, feature flow may look acceptable while security risk work waits for weeks. Debt work may never start because visible feature demand consumes all capacity. Flow measurements should help the organization see the trade-offs it is already making.

14.2.3 Quality metrics: correctness, security, and rework

Quality in DevOps is broader than a green test run. It includes product correctness, customer experience, reliability, security, maintainability, operability, and 'fit for purpose'. A delivery system that moves quickly while producing avoidable defects is consuming its own capacity. The organization may feel fast for a while, then spend more time on rework, support, incidents, and emergency fixes (Beyer et al., 2018; DORA, 2024).

Quality metrics help teams understand whether delivered work is correct, secure, maintainable, and fit for purpose. Defect escapes, rework, security findings, and test effectiveness show whether speed is producing quality outcomes or creating defects, rework, and hidden risk elsewhere in the system.

Defect escape rate

Defect escape rate is one of the most useful quality metrics because it asks where defects are found. Defects found before release are less expensive and less damaging than defects found by customers, regulators, or production systems. A rising escape rate usually means the test strategy, review process, requirements clarity, environment quality, or observability is not catching meaningful problems early enough. The point is not to count defects for blame, but to improve the system that allows defects to escape.

Test effectiveness

Test effectiveness is different from test coverage. Coverage shows that code was executed by tests. Effectiveness asks whether the tests detect meaningful problems. Useful indicators include defects caught by automated tests, flaky test rate, test execution time, quality of failure signals, mutation testing results where appropriate, and production defects linked to missing tests. High coverage with weak assertions is a fine way to fool yourself. Experienced teams learn to ask whether their tests would fail if the system behaved incorrectly.



Security posture

Security posture is also a quality signal. Open vulnerabilities by severity, time to remediate, dependency freshness, and secrets exposure reveal whether security is integrated into delivery. Security posture is further indicated by policy violations and security defects found before and after release, alongside container or infrastructure misconfiguration findings.

Security metrics should be reviewed with engineering teams, product owners, and risk leaders. A vulnerability list that remains in a separate dashboard until release week is still evidence of late security integration.

Rework

Rework links quality directly to flow and value. When teams spend a high percentage of capacity fixing preventable defects, investigating flaky tests, repairing brittle deployments, or remediating repeat security findings, the delivery system is paying interest on poor quality. Rework is often invisible in portfolio discussions because it is spread across tickets, incidents, support requests, and unplanned deployments. Measuring it gives leaders a practical reason to fund test data, automation, observability, refactoring, and security enablement.

Together, these metrics describe whether the delivery system is producing quality outcomes. They should be read alongside stability metrics, which show whether the organization can introduce change safely, recover quickly, and sustain reliable service over time.

14.2.4 Stability metrics: safe change, recovery, and operational health

Stability measures the ability to change safely and recover quickly. This does not mean avoiding change. In strong DevOps systems, stability comes from smaller changes, better automation, observability, fast rollback, clear ownership, and incident learning. A system that deploys once a quarter may look stable until the

release finally happens. Large batches often hide risk until they arrive at once.

Stability metrics help teams understand whether delivery speed is supported by reliable operation and effective recovery. They show whether the organization can introduce change without unacceptable disruption and restore service effectively when failures occur.

Core stability metrics include the DORA stability measures (change failure rate and failed deployment recovery time) plus incident volume and severity, service restoration time, SLO attainment, deployment health, and alert quality. Teams should distinguish failed deployment recovery time from broader service restoration or incident-resolution clocks. Deployment frequency becomes meaningful only when read beside failure and recovery signals. Frequent deployments can be safe when changes are small and recovery is fast; infrequent deployments can be risky when they accumulate change, coordination, and uncertainty.

Operational signals

Operational signals provide a production view. Latency, traffic, errors, saturation, and availability help teams understand how the service behaves under real conditions. Tail latency deserves particular attention because averages hide pain. A checkout system with acceptable average latency may still fail to satisfy high-value customers during traffic spikes.

Stability metrics should feed release decisions and error-budget policy, as well as incident review, architecture investment, and platform improvement. They are not just operations numbers, but signals about whether the organization can change safely.

Team sustainability

Quality and stability also influence team sustainability. Repeated production defects increase interruptions, create emotional strain, and reduce the time available for thoughtful engineering. Noisy alerts teach responders to distrust the alerting system, and slow recovery erodes confidence. These are measurable system effects, even when they are not captured in the incident ticket. A delivery system that exhausts the people operating it is not stable in any meaningful sense.

Alert quality

Alert quality deserves special attention because it shapes the human response to instability. A page should be urgent, actionable, and tied to user-visible or soon-to-be user-visible impact. Tickets can handle non-urgent work. Dashboards can support inspection, and reports can show trends. Mixing these channels creates noise. Teams should measure how many alerts wake people, how many require action, how many are duplicates, and how many are removed after review. A quiet, trusted alerting system is an operational asset.

14.2.5 Value metrics: connecting delivery performance to outcomes

DevOps measurement is incomplete if it stops at engineering performance. The purpose of faster delivery is not to produce more releases, but to deliver valuable, reliable, secure, and adaptable software-enabled services. Value metrics connect delivery capability to outcomes that matter outside the engineering organization.

Value takes different forms in different contexts:

- In a **commercial product**, value may appear as revenue influenced, activation, retention, conversion, checkout completion, search success, support contact reduction, or customer satisfaction.
- In an **internal platform**, value may appear as faster onboarding, fewer support tickets, lower cognitive load, reduced duplicated work, or higher developer satisfaction.
- In a **public-sector service**, value may appear as citizen service completion, processing time, accessibility, policy delivery, and trust in the service.
- In a **regulated organization**, value may include risk reduction, audit effort reduction, control effectiveness, and operational resilience.

Output is not an outcome. Features shipped do not guarantee delivered value. Story points

do not equal business impact. Deployment count does not equal customer benefits. Even adoption can mislead if the product change creates long-term retention problems, support burden, or risk exposure. Value measurement requires the discipline to ask what behaviour or result should change because software changes.

Technical metrics can still connect strongly to value. Reduced lead time can allow a product team to run smaller experiments and learn faster. Better reliability can protect customer trust and reduce support demand. Better security can reduce exposure and regulatory risk. Lower rework can free capacity for value-producing work. Faster recovery can reduce incident impact and preserve confidence during change. The connection is rarely automatic, so teams should make it explicit.

Hypotheses are a practical way to connect delivery and value metrics. A team might say, if we reduce payment latency, checkout completion should improve. Another team might say, if we reduce deployment lead time, product managers can run more controlled experiments per quarter. A platform team might say, if we improve self-service environment provisioning, onboarding time for new services should fall. These statements create a measurable link between capability and outcome.

Attribution must be handled with care. Business outcomes are influenced by commercial, technical, operational, regulatory, and market factors. A DevOps metric rarely deserves all the credit for a business result. It can still be useful. The question is whether the metric improves decisions, helps test hypotheses, and reveals constraints that would otherwise remain hidden.

Value metrics should be close enough to the delivery system that teams can learn from them. Product analytics, support contact trends, and user path telemetry should not live only in separate business dashboards. The same isolation should be avoided for conversion data, operational cost data, and customer sentiment. Engineering teams need enough context to understand whether their work has improved real outcomes. Product

teams need enough operational data to understand whether value was delivered safely. The discipline here is uncomfortable but necessary. Some delivery improvements will not produce the expected value. Some beloved features will show weak adoption. Some reliability investments will protect value that is hard to see until an incident does not happen. Mature organizations accept this ambiguity. They use value metrics to ask better questions and make better investment decisions, not to manufacture certainty.

Value measurement often requires product, engineering, operations, support, security, finance, and business stakeholders to sit with the same evidence. This can be uncomfortable because each group sees a different part of reality. Product teams may see adoption. Engineering may see rework. Operations may see incident loads. Finance may see costs. Security may see exposure. The value conversation improves when these views are reconciled rather than averaged away.

The most useful value metrics are usually tied to user paths. Checkout completion, claim submission, citizen application completion, report generation, onboarding completion,

and support deflection are more concrete than broad satisfaction scores. They also create better engineering questions like the following:

- Which step fails most often?
- Which dependency adds to latency?
- Which user group experiences the poorest result?

Value becomes actionable when it can be traced back to the work system.

14.3 AI-specific metrics

AI-enabled products and AI-assisted delivery systems require additional measurement alongside the four core measurement domains because their behaviour can change without a conventional code change. Input data changes. User behaviour changes. Prompts are adjusted. Retrieval indexes become stale. External context shifts. A model that performed acceptably during evaluation can degrade in production because the world around it has moved (Autio et al., 2024; DORA, 2025).

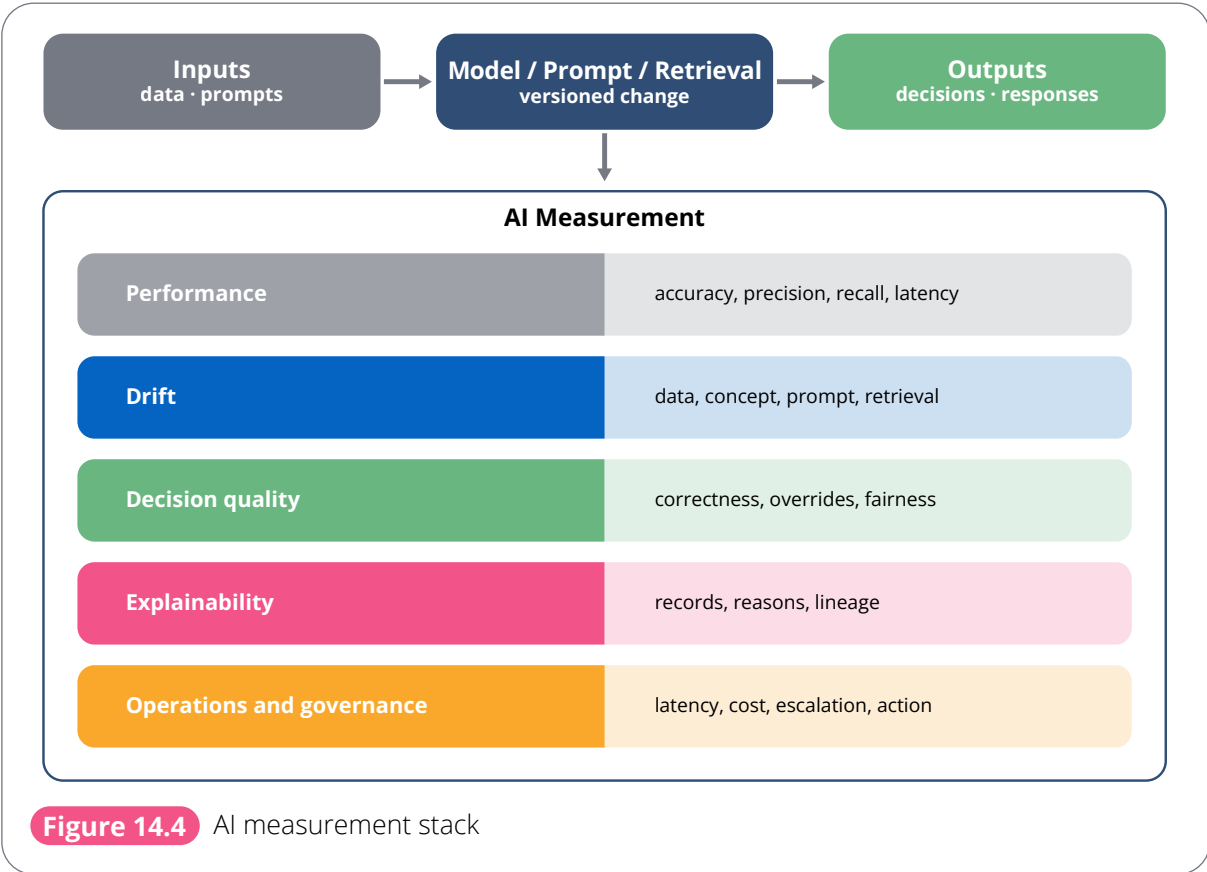


Figure 14.4 illustrates the AI measurement stack from inputs through model, prompt, and retrieval changes to outputs, with measures for performance, drift, decision quality, explainability, and operations and governance. It covers both product quality signals such as accuracy, groundedness, and hallucination rate, and operational signals such as latency, token or compute cost, tool success, and human escalation.

Predictive AI systems should measure model accuracy, classification quality (precision, recall, F1), calibration, decision latency, and error rates. These metrics must be interpreted against the business or mission context. A fraud model with high precision and falling recall may look efficient while allowing more fraudulent transactions through. A clinical or safety-related model may require a different tolerance for false positives and false negatives than a marketing model. The metric is only meaningful when it reflects the cost of being wrong.

Drift metrics help teams detect changes in the relationship between data, model, and outcome. Data drift occurs when input distributions change. Concept drift occurs when the relationship between inputs and labels changes. Prediction drift occurs when model outputs shift over time. Prompt drift occurs when prompt templates, system instructions, or user prompt patterns change behaviour. Embedding drift and retrieval corpus drift are especially important in retrieval-augmented systems, where stale or poorly indexed knowledge can degrade answer quality even when the model itself is unchanged.

Decision quality metrics focus on whether AI-supported decisions are correct, compliant, fair, and operationally useful. They may include correctness, business rule compliance, human override rate, appeal or dispute rate, downstream error rate, and bias or fairness indicators. Human override rates are worth watching carefully. Too many overrides may indicate weak model quality, whereas too few overrides may indicate over-trust, poor review design, or ineffective escalation paths.

Generative AI introduces additional quality metrics, including groundedness, hallucination rate, retrieval quality, citation accuracy, safety policy violations, and human escalation. A chatbot may have good latency and high user engagement while quietly giving unsupported answers. A code assistant may produce plausible output that introduces insecure patterns. A summarization system may omit material context in ways that look harmless until a decision depends on it.

Explainability and auditability also require measurable evidence when AI influences decisions. Depending on risk, this may include feature attribution, reason codes, decision records, model cards, prompt and model version traceability, dataset lineage, and human-review records. Higher-risk or regulated use cases require stronger evidence, defined before deployment rather than after a complaint or audit request.

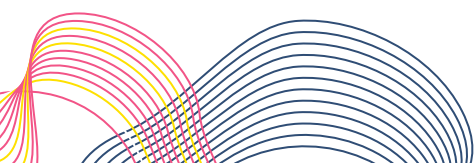
Operational AI metrics are also part of DevOps measurement. Latency, cost per request, token usage, timeout rate, model availability, fallback rate, and vendor or model dependency health influence customer experience and operating cost. A prompt that grows over time can raise costs and latency. A retrieval system with poor cache behaviour can create performance problems. A model fallback path that is rarely tested may fail exactly when traffic spikes.

AI changes should be treated as production changes. These include:

- model deployment
- prompt modification
- retrieval index refresh
- data pipeline adjustment
- agent tool permission changes.

The measurement system must capture these events with traceability, context, and ownership. Offline evaluation is necessary, but it is not sufficient. AI behaviour must be measured after deployment because that is where users, data, risk, and operational reality meet.

Accordingly, AI measurement must also include a review of the control model. A human-in-the-loop process that is overloaded will fail quietly. A human-on-the-loop process with poor escalation thresholds may miss a harmful drift.



Automated actions without clear rollback and audit trails can create accountability gaps. Metrics should therefore include not only model behaviour, but also human review capacity, escalation quality, override handling, and the timeliness of governance responses.

Retraining is not automatically the right answer when an AI metric worsens. Drift may indicate a need for retraining, but it may also indicate stale retrieval content, poor feature quality, a broken data pipeline, changing business rules, or misuse of the system. Good AI measurement separates symptoms from intervention. The same engineering discipline used for production incidents applies here: first collect evidence, form a hypothesis, make a controlled change, and verify the result.

14.4 Building a measurement programme

Most measurement programmes fail because they try to measure everything at once. The dashboards multiply, definitions drift, ownership becomes unclear, and teams stop trusting the purpose. A better starting point is modest and disciplined: a small metric set tied to the current constraint, such as one flow metric, one quality metric, one stability metric, one value metric, and one team health or sustainability signal. The set should pair speed with safety, have clear ownership and documented definitions, support regular trend reviews, and retire metrics that do not inform decisions. It should be small enough for real conversation and broad enough to prevent a single metric from dominating behaviour (DORA, 2024; PeopleCert, 2026d).

A practical starting set might include lead time, defect escape rate, change failure rate, a customer-impact indicator, and on-call burden. Another organization may choose flow time, vulnerability remediation age, SLO attainment, service completion rate, and team load. The right set depends on the

transformation goal and the visible constraint. If the main problem is frequent incidents, the starting set should include stability and recovery. If the main problem is slow delivery, flow and queue time matter. If the main problem is poor customer adoption, value signals need more attention.

Each metric needs an owner responsible for its definition, data quality, interpretation, and review. A data owner must understand the integrity of the underlying source. A review forum must exist where the metric is interpreted and acted upon. A decision record should capture what changed because of the signal. Without these responsibilities, measurement becomes dashboard sprawl with better graphics.

Review cadence should match the signal. Operational signals may need daily or weekly review. Team improvement metrics often fit an iteration cadence. Value-stream trends may be reviewed monthly. Strategic investment signals may be reviewed quarterly. Audit and governance evidence may follow a separate cadence. The cadence matters because stale metrics create stale decisions, and over-frequent reviews create noise.

Measurement governance should be lightweight and real. A definition registry prevents each team from inventing its own calculation. Data-quality checks expose missing timestamps, inconsistent labels, and duplicate records. Ownership clarifies who can explain the metric. Review forums ensure that signals are interpreted. Metric refinement allows the organization to retire measures that are no longer useful, duplicated, not actionable, or causing poor behaviour. Metrics should have a lifecycle and some should be retired.

Gaming is the predictable failure mode when metrics become targets without trust. Teams naturally optimize for whatever leadership measures. Without balanced metrics, this can produce behaviours such as meaningless deployments, artificially split work items, hidden incidents, premature ticket closure, or low-value tests. These behaviours are rational responses to a poorly designed measurement system.

The anti-pattern is familiar: “Our DORA metrics are all elite.” The dashboard shows success, yet customers see little value, quality is uneven, incidents still hurt, and teams are tired. The numbers may have improved because work was reclassified, incidents were disconnected from deployments, clocks were started late, or teams learned which numbers leadership wanted. The damage is serious. Trust falls, risk hides, investment decisions weaken, and the organization gains false confidence.

Preventing metric gaming requires balanced measurement rather than additional control. Pair every speed metric with a safety metric. Pair every engineering metric with a value signal where possible. Review trends rather than rankings. Combine quantitative and qualitative evidence. Measure systems rather than individuals. Ask teams to explain signals before judging them. Audit a sample of metric data when results look suspicious or unusually good.

Interpreting metrics well is a skill. Every review should ask:

- What has changed?
- Is this a trend or noise?
- What denominator is being used?
- What work is included (or excluded)?
- Who can act?
- Which countermeasure should be tested?

Averages should not be trusted blindly because distributions and outliers often tell the real story. Correlation is not the same as causation. A metric should open a conversation, support a hypothesis, and lead to one or two actions that can be verified later.

The chapter principle is worth stating plainly: the purpose of DevOps measurement is not to prove that the organization is performing well. It is to help the organization see clearly enough to improve. Measurement is one of the ways DevOps becomes self-correcting. When leaders refuse to weaponize metrics, fund the improvements revealed by evidence, and reward transparency, the system gets smarter. When metrics are treated as decoration, surveillance, or competition, the system learns to hide.

Corrective action for metric gaming should start with the measurement system rather than accusation. This process begins by reconfirming the definitions and checking whether teams understand the purpose of the metric. Additionally, it involves adding countermeasures and reviewing a sample of source data. It is also important to evaluate whether the metric changed a real decision. If it did not, it should be retired or redesigned. A metric that creates fear or confusion deserves as much engineering attention as a flaky test or noisy alert.

The final responsibility sits with leadership. Leaders decide whether metrics become learning instruments or weapons. They decide whether improvement work receives funding when the evidence shows a constraint. They decide whether teams are rewarded for transparency or punished for surfacing problems. The measurement system will eventually reflect those choices. Teams listen to what leaders celebrate, what they ignore, and what they punish.

DECISION AND ACTION GUIDE

Executive decision: Leadership approves a balanced measurement model, metric owners, the review cadence, safeguards against individual ranking or gaming, and an accountable owner for measurement data.

Role-specific actions: Product, engineering, data, operations, security, and finance roles define measures, lineage, thresholds, and interpretation rules; leaders record the decisions that follow.

Evidence to review: A metric dictionary, data lineage, balanced measurement view, data-quality checks, decision log, and records of measures that were changed or retired.

Case A – Northstar Financial Services (composite): when balanced measures changed the investment decision

Northstar translated the payments transformation into a measurement chain. The customer and business outcome was fewer failed payment journeys and support contacts while maintaining fraud and compliance controls. Flow measures included lead time, work age, and wait time. Quality and stability measures included defect escape, change failure rate, failed deployment recovery time, and the payment-journey SLO.

The team also measured platform adoption, time to provision a compliant environment, on-call burden, and the review and rework associated with AI-assisted artifacts. These measures were not combined into one score. They were reviewed together as evidence for the outcome hypothesis.

An early dashboard showed that deployment frequency had increased. A narrow interpretation would have declared success. The same review showed rising change failure, brittle test-data work, and growing review load. Leadership stopped pushing frequency as a target and funded the test data and platform improvements identified in the maturity assessment.

Later reviews showed more stable change and lower wait time, but customer abandonment had not yet improved enough. Product and engineering then examined provider latency and user experience rather than claiming that delivery performance automatically proved customer value.

Transferable lesson: Metrics matter when they change a decision. Pair flow with quality and stability, connect engineering evidence to customer outcomes, and treat measures as signals for learning rather than targets for ranking teams.

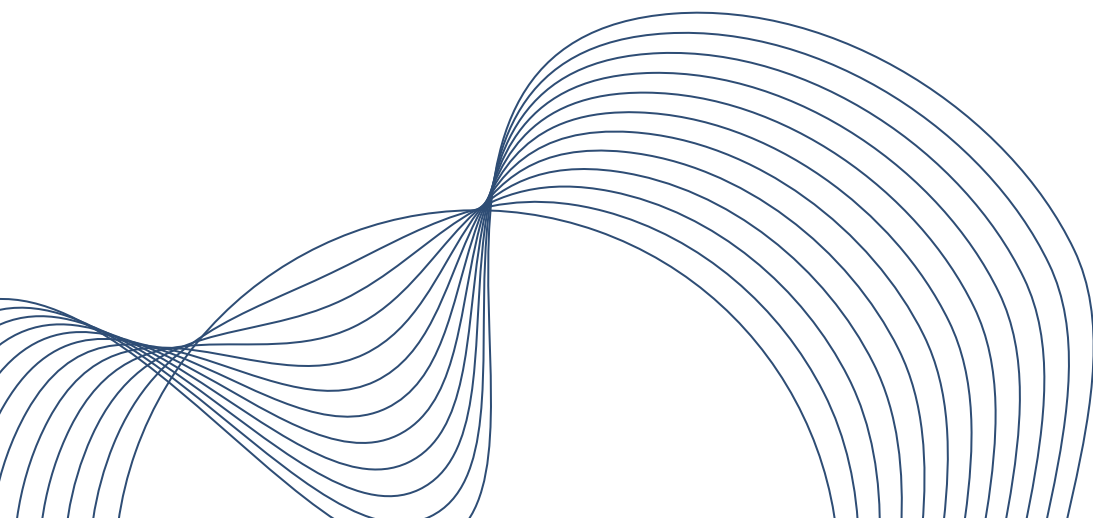
SCENARIO



14.5 Chapter summary

Measurement is the feedback mechanism of the DevOps operating system. A balanced view combines flow, quality, stability, and value so that faster delivery is not mistaken for better delivery. DORA metrics are useful signals, but they need clear definitions, reliable data, context, and countermeasures to prevent any one measure from becoming the target.

AI-native delivery extends these core measurement domains with additional measures for performance, drift, decision quality, explainability, latency, cost, safety, and human oversight. A measurement programme should start with a small set of owned measures, document their lineage and interpretation, and review them where action can be taken. The purpose is not to rank teams; it is to help the organization see clearly enough to improve the system.





Chapter 13 explains how DevOps transformation is sequenced, governed, and scaled. Chapter 15 focuses on the enduring organization-wide operating design that supports DevOps during and after scale, building on the DOI® Operating Model introduced in Chapter 4. This chapter addresses the design changes required when DevOps extends across many teams, value streams, platforms, regulatory contexts, and technology domains. Chapters 5, 9, and 14 provide canonical treatments of team topology, Platform Engineering, and measurement. Chapter 15 does not reteach them; it explains what the organization must add when local capability is no longer enough.

At scale, the constraint usually moves beyond individual team practices to shared organizational systems, such as portfolio decisions, cross-value-stream dependencies, platform boundaries, and architecture, along with policy, evidence, funding, procurement, and coordination. DevOps at scale succeeds when autonomy and alignment are designed into that system rather than negotiated through recurring escalation.

15.1 Why DevOps at scale is different

A high-performing team can integrate, test, deploy, operate, and learn effectively while the organization remains slow. Shared environments may create queues.

A dependent service may change at a different cadence. Portfolio funding may interrupt long-lived ownership, and common controls may still require manual reconstruction. Local maturity therefore does not automatically create end-to-end flow (Kersten, 2018; Skelton & Pais, 2025).

The organizational design problem is to decide which capabilities remain local, which become shared services, which policies must be common, and where variation is necessary. Those decisions should reduce coordination costs without creating central ownership of product outcomes.



Autonomy and alignment

Figure 15.1 From team DevOps to organization-wide DevOps

DevOps at scale

Figure 15.1 illustrates the scaling path from team flow through multi-team dependencies, programme coordination, and platforms and governance to an organization-wide DOI® Operating Model. It also shows that scaled DevOps requires autonomy and alignment to be managed together.

Table 15.1 contrasts team-level concerns with the broader design concerns introduced by DevOps at scale.

The comparison is not a maturity ladder. Team capability remains essential. Scale adds another layer of design around the team so that local flow is not defeated by organizational queues, conflicting standards, or invisible dependencies.

15.2 Design for scale

Organization-wide DevOps requires deliberate design around teams, not the centralization of their work. Organization boundaries, value streams, platform portfolios, governance, architecture, funding, and dependencies must operate as a connected system that preserves autonomy while reducing coordination costs. This section examines the mechanisms that allow local capability to become reliable, governable flow at scale.

Table 15.1 Team-level and organization-wide design concerns

Team DevOps	DevOps at scale
Team autonomy	Organization-wide operating model
Local pipelines	End-to-end value flow
Team ownership	Portfolio and value stream ownership
Engineering practices	Organization design
Local metrics	Organization-wide measurement
Team coordination	Platform-enabled coordination
Team governance	Continuous governance
Software delivery	Software, data, and AI delivery

15.2.1 Organizational design across a portfolio of teams

Section 5.2.1 and Figure 5.2 provide the canonical team-topology model. At scale, the question is not whether stream-aligned, platform, enabling, and complicated-subsystem teams exist. The question is whether the organization manages their boundaries and interaction modes as an intentional portfolio (Skelton & Pais, 2025).

Every persistent dependency should have an explicit interaction model. Teams may consume a stable service, collaborate for a defined period, or receive temporary enabling support. Without a clearly defined interaction model, collaboration becomes a permanent queue, and specialist knowledge becomes an informal approval gate.

Leaders should review the portfolio for ownership gaps, duplicated capabilities, overloaded shared teams, and interactions that never evolve into self-service.

Reorganization is justified only when it changes decision rights, accountability, service boundaries, or the flow of work. Simply moving boxes on an organization chart without changing those conditions preserves the same system under new labels.

AI-enabled delivery adds one further consideration, which Chapter 3 addresses through its detailed governance model. At scale, the focus is on embedding those boundaries consistently across teams, platforms, and value streams through reusable guardrails and approved service patterns, while product teams remain accountable for outcomes.

**AI-RELATED
CONTENT**



15.2.2 Organization-wide Value Stream Management

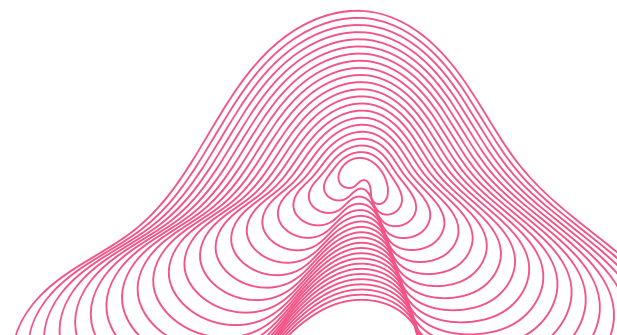
Value Stream Management makes constraints visible across team and organizational boundaries. It should show where work waits between demand, funding, design, delivery, control, operation, and learning, not merely aggregate local team activity into a portfolio dashboard (Kersten, 2018; Pereira & Davis, 2024). It should also reveal work in progress, dependencies, queue time, and flow time, as well as rework, risk, and outcome signals across teams and governance points.

The objective is not simply to observe these constraints, but to change the decisions that create them. Visibility depends on signals that expose constraints. These include shared queues, dependency age, and cross-value stream work in progress. They also involve blocked time, approval latency, platform demand, unplanned risk work, and the delay between production evidence and portfolio decisions. These signals help leaders distinguish a local improvement opportunity from an organization-wide constraint that teams cannot remove themselves.

Portfolio action should reduce or remove the constraint. This may involve:

- funding a shared capability
- changing a policy
- retiring a dependency
- separating a coupled service
- narrowing work in progress
- revising investment priorities.

Visibility that does not change a decision is reporting, not Value Stream Management.



15.2.3 Platform portfolios and federation

Chapter 9 defined Platform Engineering and the internal developer platform. At scale, organizations must also manage a platform portfolio by deciding which capabilities should be common across the organization, which belong to domain platforms, which can be extended locally, and how these layers interoperate without creating competing golden paths (CNCF TAG App Delivery, 2023a; Fournier & Nowland, 2024).

A federated model is often more practical than one universal platform. A shared foundation can provide identity, evidence, policy, service registration, and common delivery interfaces. Domain platforms can add capabilities for data, AI, regulated workloads, or specialized infrastructure. Local extensions remain possible when they conform to published interfaces and ownership rules.

Mandated adoption is not sufficient evidence. The organization should measure whether platform products reduce cognitive load, waiting, operational risk, and delivery variability while preserving team ownership. A platform that becomes a compulsory ticket queue has scaled centralization, not DevOps.

15.2.4 Continuous governance and compliance at scale

Governance at scale must be consistent enough to be trusted and adaptable enough to reflect risk. The organization should define not only common policy intent, risk tiers, and evidence schemas, but also decision rights, exception rules, and control ownership. Teams and platforms then implement those requirements in the normal delivery workflow (NIST, 2022; ISO/IEC, 2023; PeopleCert, 2026d).

Chapter 4 establishes continuous governance as governance embedded in the delivery workflow. At scale, shared policy and evidence services prevent every value stream from interpreting the same obligation independently. On the one hand, low-risk changes can move through pre-approved paths when controls pass. On the other hand, higher-risk changes can invoke accountable human review based on explicit criteria. Therefore, exceptions should be named, time-bound, evidenced, and visible across the relevant governance layers.

Table 15.2 Platform-portfolio decisions at scale

Scale concern	Design decision	Evidence that the decision works
Shared foundation	Define the minimum common services and interfaces used across value streams.	Adoption, reliability, support demand, policy coverage, and reduced duplicate implementation.
Domain capability	Assign product ownership for specialized data, AI, security, or infrastructure services.	Clear service levels, bounded queues, user feedback, and independent domain evolution.
Local extension	Provide supported extension points and an exception path rather than forcing one implementation.	Documented ownership, compatibility evidence, review dates, and low reintegration cost.
Federation	Use common identity, metadata, evidence, telemetry, and lifecycle rules across platform products.	Interoperability, traceability, consistent controls, and reduced handoffs between platforms.
Lifecycle and funding	Fund platform products for operation, improvement, migration, and retirement, not only initial build.	Roadmaps, adoption decisions, deprecation evidence, cost transparency, and completed migrations.

Federation is important. Central functions define obligations and assurance expectations; platform and delivery teams implement reusable controls; product and service owners remain accountable for the risk of their outcomes. Audit and regulatory review should consume evidence generated by the system rather than require teams to reconstruct the history after release.

Chapter 3 defines the AI-native governance model. At scale, the focus is on applying it consistently across teams, platforms, and value streams through the same risk, evidence, and accountability architecture used for other delivery controls.

AI-RELATED
CONTENT



15.2.5 Dependency economics, architecture, and coordination overhead

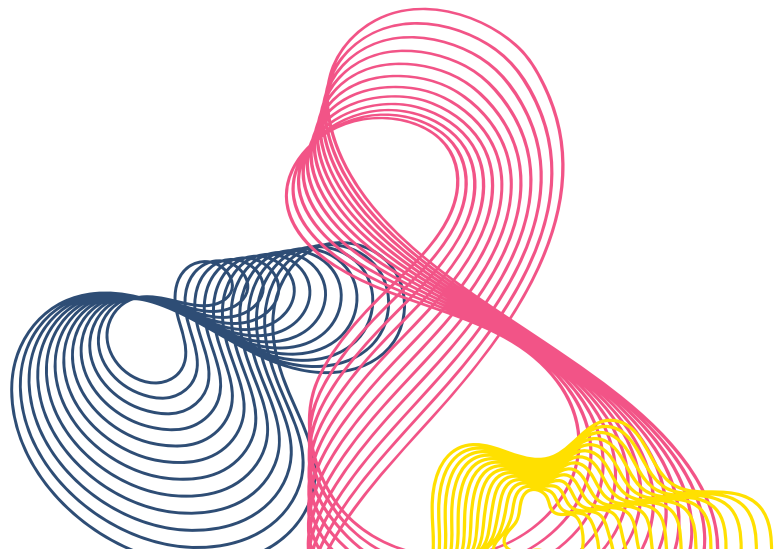
Dependencies create an economic cost each time work waits, plans are synchronized, tests are repeated, or releases are coordinated. Mature organizations therefore manage dependency reduction as an investment portfolio that reduces recurring dependencies through clearer ownership, stable API contracts, platform self-service, decoupled architecture, shared standards, and funding for structural fixes rather than merely tracking them in plans and meetings (Conway, 1968; Skelton & Pais, 2025).

Recurring dependencies can be classified by cause:

- unclear ownership
- unstable interfaces
- shared data
- coupled release
- scarce specialist capacity
- platform gaps
- procurement constraints
- policy design.

Each recurring dependency should have an owner and a reduction strategy. Some dependencies should become stable services whereas others should be removed through architecture. Some of them require platform capability, and some must be accepted with explicit service expectations.

Architecture standards should favour independent change and backward compatibility. They should also ensure observable interfaces, contract testing, recoverability, and clear data ownership. Organization-wide architecture creates the most value when it reduces future coordination cost and provides usable guardrails, not when it becomes another late review queue. Finally, AI systems add dependencies, which should be governed through contracts that define versioning, data freshness, latency, cost, quality, fallback behaviour, monitoring, and accountability.



15.3 The organization-wide DOI® Operating Model

The organization-wide DOI® Operating Model combines the design decisions in this chapter and builds on the model introduced in Chapter 4 and the transformation approach described in Chapter 13. Stream-aligned ownership remains at the centre, and platform products reduce repeated work. Organization-wide Value Stream Management exposes cross-boundary constraints, and continuous governance provides common policy and evidence. Architecture reduces dependency costs, while long-lived funding sustains products, platforms, reliability, security, and modernization.

Figure 15.2 identifies the principal elements of the scaled operating model. These elements form a connected system of value streams, teams, Platform Engineering, governance, architecture, measurement, and funding. The bidirectional relationships show that organization-wide DevOps scale depends on coordinated operating mechanisms, not copied local practice.

Measurement at this level should support organization-wide decisions rather than rank teams. Useful evidence includes cross-value-stream wait time, dependency age, platform adoption and reliability, control automation, exception age, portfolio work distribution, operational risk, customer or mission outcomes, and the time required to remove a systemic constraint. Chapter 14 remains the canonical measurement guidance.

A practical field pattern is to scale from a small number of value streams. This can be accomplished by capturing the reusable platform, policy, evidence, architecture, and funding decisions that made the improvement possible.

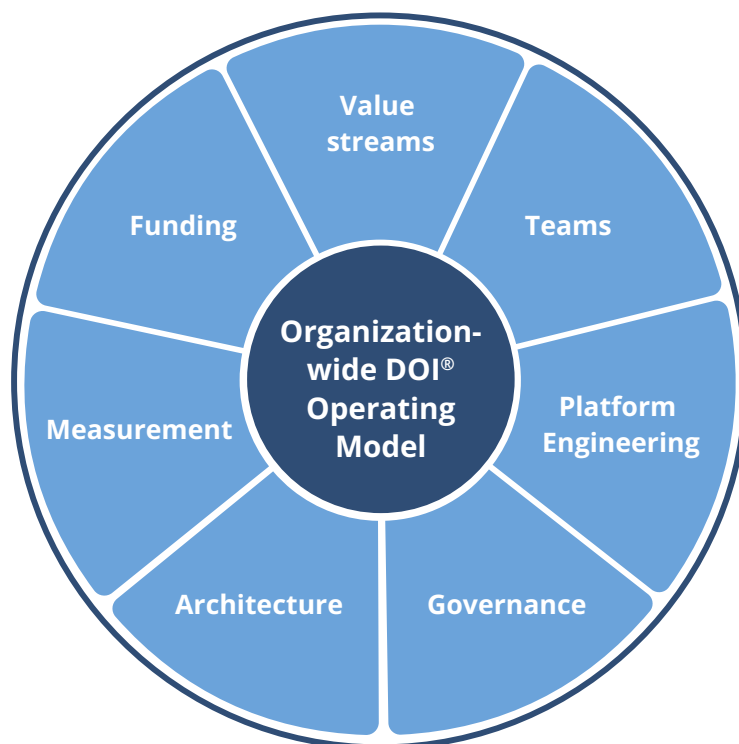


Figure 15.2 The organization-wide DOI® Operating Model: core elements and relationships²⁴

Table 15.3 Scaling DevOps from teams to organizational capability

Scale concern	Pattern	Evidence used to govern the pattern
Team boundaries and ownership	Organize around long-lived products and value streams, with explicit platform, enabling, and complicated subsystem responsibilities.	Dependency wait time, ownership gaps, incident handoffs, and team interaction feedback.
Platform portfolio	Federate common identity, policy, evidence, and telemetry services with domain platforms for payments, data, and customer channels.	Adoption, reliability, support demand, time to use, exception patterns, satisfaction, and cost.
Governance and evidence	Use common risk tiers and evidence contracts while allowing different tools and implementation paths.	Automated control results, exception age, audit confidence, release decisions, and policy effectiveness.
Architecture and dependencies	Define compatibility, observability, recovery, and data-ownership expectations; fund removal of high-cost shared dependencies.	Dependency age, wait time, contract failures, change coupling, recovery results, and service-level evidence.
Funding and portfolio decisions	Fund long-lived products, platforms, and capability improvements rather than temporary tool installation projects.	Outcome hypotheses, value-stream performance, platform-product evidence, risk reduction, and learning.
AI-enabled work	Maintain an approved-use-case registry, common data and provenance controls, bounded permissions, and production monitoring.	Use-case approval, validation evidence, human decision records, drift and quality signals, incidents, and cost.

The organization can then extend those capabilities deliberately. The organization scales not by copying one team's ceremonies or tool configuration, but by productizing proven conditions for flow.

The health test asks:

- whether teams can move independently within clear boundaries
- whether shared capabilities are adopted because they help
- whether dependencies are decreasing
- whether leaders can identify and fund the largest organizational constraint.

15.3.1 Anchor case - Case A: scaling from successful teams to organization-wide capability

Organization context: After the payments pilot, Northstar had evidence that a better operating model could improve flow and control. Scaling was harder. More than one hundred teams worked across customer products, shared platforms, mainframe services, data products, and regional obligations. A uniform toolchain would not fit, but unrestricted local variation preserved hidden dependencies and inconsistent evidence²⁵.



Northstar first attempted to scale through a central migration plan. Teams were assigned dates to adopt one pipeline and one platform template. The plan produced visible progress reports, but teams with legacy, regional, and specialist constraints created workarounds. The organization was standardizing the appearance of delivery rather than the evidence and outcomes that mattered.

The revised model separated what had to be common from what could remain contextual. Common elements included terminology, identity, risk tiers, release-readiness evidence, deployment and release events, software supply-chain provenance, AI-use controls, and cross-value-stream telemetry. Teams could choose tools and local practices when they met those contracts. Platform teams were funded as products and federated around shared services instead of duplicating every capability centrally.

Organization-wide Value Stream Management focused on dependency economics rather than team rankings. Portfolio reviews examined where work waited across value streams, which shared services created coordination cost, which exceptions became permanent, and whether platform products were useful. The organization did not require every team to reach the same maturity level; it required each value stream to understand its risk, evidence, dependencies, and next constraint.

Outcome and trade-off: Northstar accepted visible, governed variation instead of pretending that one implementation would fit every context. The federated model required stronger product management for platforms and more disciplined evidence contracts. It also reduced hidden variation, made dependencies discussable, and allowed reusable patterns from one value stream to be adopted elsewhere without forcing a release event for the entire organization.

Transferable lessons: Scale through shared outcomes, evidence, platforms, and boundaries. Uniform tools can simplify reporting while increasing local workarounds; federated capability creates autonomy only when the organization is clear about what must remain common.

DECISION AND ACTION GUIDE

Executive decision: The executive committee decides which capabilities remain local, which become shared, how platform services are federated, which guardrails apply, and who is accountable for portfolio and value-stream outcomes.

Role-specific actions: Portfolio, value-stream, platform, architecture, governance, security, and product leaders manage dependencies and shared services while preserving local autonomy inside agreed boundaries.

Evidence to review: A dependency map, platform portfolio, policy catalogue, shared evidence services, funding decisions, cross-value-stream outcomes, and governed variation or exception records.

15.4 Chapter summary

DevOps at scale changes the location of the constraint. Local practices remain important, but organization-wide performance depends increasingly on team boundaries, cross-value-stream dependencies, platform portfolios, architecture, funding, procurement, governance, and shared evidence. The organization must decide which capabilities remain local, which are federated, and which require common policy.

Organization-wide Value Stream Management should expose waiting, dependency economics, and coordination overhead across portfolios. Platforms and governance should provide reusable capabilities, risk tiers, exception paths, and decision rights without taking ownership away from product teams. AI-native scale adds shared responsibilities for models, data, prompts, evaluation, monitoring, and human oversight.

Future Horizons and next steps

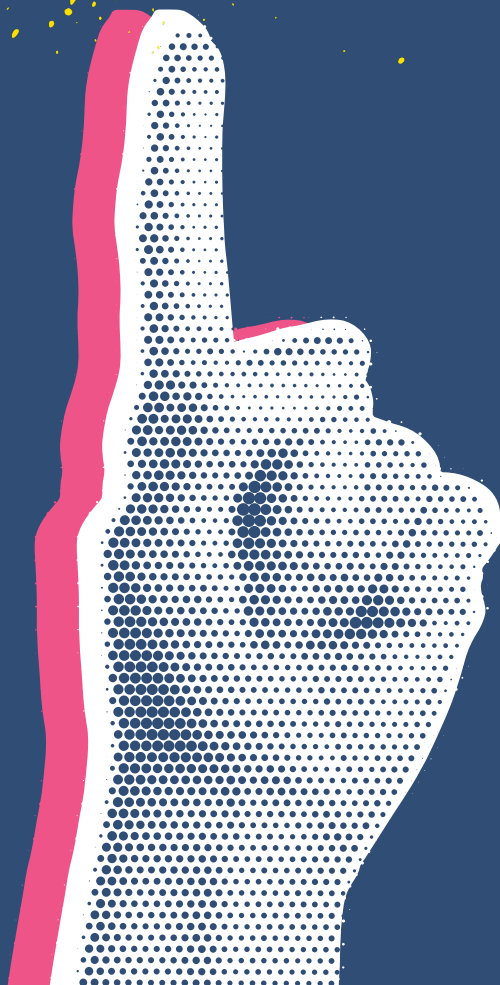
PART 4

Chapter 16

Future horizons: applying
DevOps principles to emerging
change

Chapter 17

Your next steps



PART FOUR closes the DevOps Standard book by looking forward and then returning the reader to practical action. That sequence matters. A standard must be stable enough to guide current practice, but it must also help organizations absorb change without losing control of the delivery system.

DevOps will continue to evolve. AI will change how work is created, reviewed, tested, secured, deployed, operated, and learned from. Platform Engineering will continue to mature from tooling support into an organizational capability. Regulation and compliance expectations will become more continuous, evidence-based, and risk-aware. DevOps principles will also extend further into AI systems, hardware-adjacent systems, operational technology, and other environments where software behaviour affects physical, social, financial, or public outcomes.

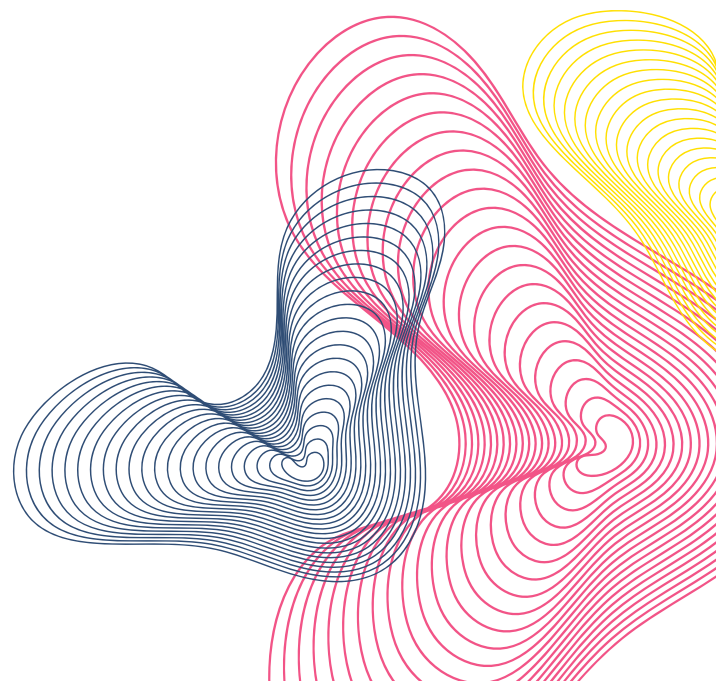
PART FOUR is deliberately grounded in durable principles rather than short-lived predictions. The future of DevOps should not be read as a catalogue of products or vendor announcements. It should be considered through the operating questions addressed throughout the book:

- Does the change improve flow?
- Does it strengthen feedback?
- Does it reduce risk?
- Does it improve quality and security?
- Does it create trustworthy evidence?
- Does it preserve human accountability?
- Does it help the organization learn?

Emerging capabilities should be evaluated according to outcome, risk, authority, evidence, reversibility, accountability, and production behaviour rather than novelty or technical possibility alone. High-stakes and broader delivery domains may require stronger assurance, different recovery assumptions, and careful human oversight, but the underlying DevOps disciplines still apply. For leaders, the next step is to commission an honest assessment, build a transformation case from evidence, fund the first meaningful improvement, and protect the learning required for change. For practitioners, the next step is to use the DevOps Nine Pillars as both a team-improvement lens and a personal development framework. For multipliers, the next step is to use the DevOps Standard book as an engagement framework for workshops, assessments, and value stream maps, alongside transformation backlogs, coaching plans, and learning pathways.

PART FOUR closes with a practical point. DevOps maturity is not a badge, and the future of DevOps is not a destination. The work continues through assessment, action, measurement, learning, and deliberate adaptation. The value of this DEVOPS INSTITUTE® reference is that it gives organizations a stable DOI® Operating Model for doing that work with discipline.

A DevOps reference is useful only when it changes how people deliver, operate, secure, govern, and learn from real systems. That is the responsibility the reader carries forward from the DevOps Standard book.



16.

This chapter examines emerging changes without turning the DevOps Standard into a catalogue of predictions. Its purpose is to show how organizations can apply the DOI® Operating Model when technologies, risks, and delivery domains change.

The future of DevOps will include increasingly autonomous AI-assisted workflows, context-aware and federated platforms, and more delivery work in high-stakes digital and cyber-physical environments. These developments matter because they can change who or what makes decisions, the blast radius of change, the evidence required before release, and the meaning of recovery.

The durable response is not to create a separate operating model for every new technology. Organizations should begin with the same questions addressed throughout the DevOps Standard:

- What outcomes should improve?
- What flow and feedback will change?
- What risks are introduced?
- What evidence creates confidence?
- Who remains accountable?
- How will the system be observed and recovered?

This chapter therefore focuses on organizational adoption decisions. It does not predict specific products. It provides a disciplined way to adopt, adapt, constrain, defer, or reject an emerging capability based on evidence.

Chapter 3 defines the operating controls for AI in today's delivery system, including authority, guardrails, evidence, human oversight, and recovery. This chapter builds on those foundations by providing a technology-neutral method for evaluating and adopting new capabilities, particularly where risk, uncertainty, or potential impact is high.

16.1 Durable DevOps principles under emerging pressure

Emerging technologies and delivery domains may change the speed, authority, and consequence of change. However, they do not remove the need for disciplined flow, feedback, security, governance, recovery, and accountability. The important task is to identify which assumptions are under pressure and which operating principles must remain stable. This section separates durable DevOps principles from short-lived technology predictions.

Future horizons: applying DevOps principles to emerging change

16.1.1 Emerging pressures on the DOI® Operating Model

Three pressures are likely to test the assumptions of existing delivery systems. The first is the movement from AI assistance and recommendations towards increasingly autonomous decisions and actions. Chapter 3 defines the different roles of AI and the controls needed for agentic action, while Chapter 11 addresses AIOps. The future pressure is whether those controls remain sufficient as systems move closer to taking action in production (NIST, 2023a; DORA, 2025).

Automation executes a defined action when a known condition occurs. Autonomy selects or initiates an action inside a decision space.

As the possible operational effect increases, the organization needs stronger evidence and narrower permissions. Additionally, it needs clearer escalation, reliable recovery, and explicit human accountability.

Figure 16.1 illustrates the progression from AI assistance to AI recommendations, AI-orchestrated workflows, guarded autonomy, and human accountability. Human-defined guardrails such as telemetry, policy, tests, an audit trail, and rollback remain present as autonomy increases.

The second pressure is the emergence of context-aware and federated platforms. Platforms may assemble delivery paths according to service criticality, data sensitivity, and regulatory exposure, as well as cost, architecture, and the type of change being made. They may support both human and agentic users across several domain platforms.

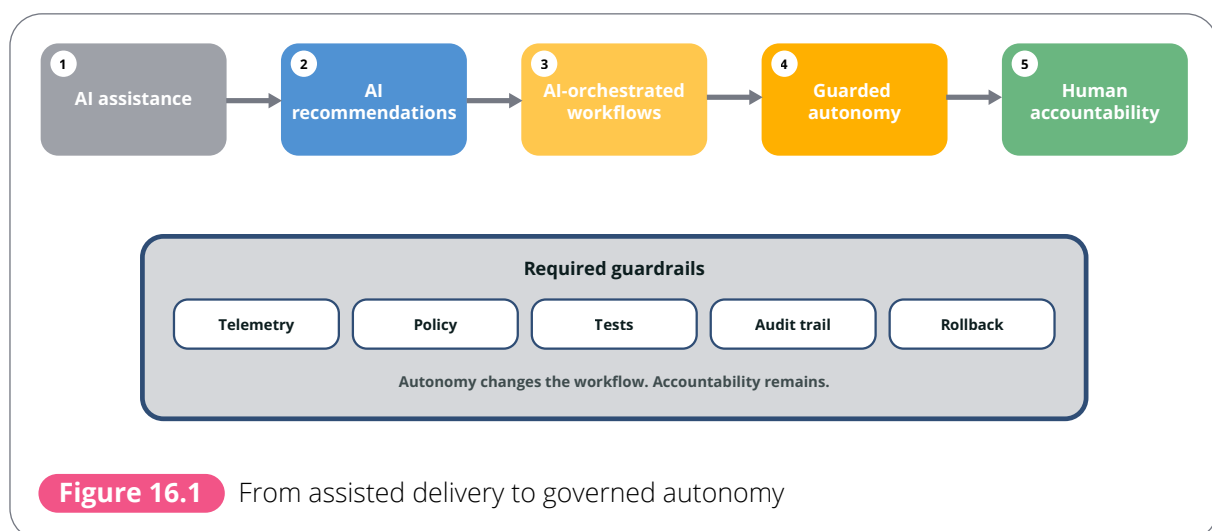


Figure 16.1 From assisted delivery to governed autonomy

This increases the importance of machine-usable operating contracts, bounded permissions, open interfaces, explainable policy decisions, and clear lifecycle ownership (CNCF TAG App Delivery, 2023a; DORA, 2024).

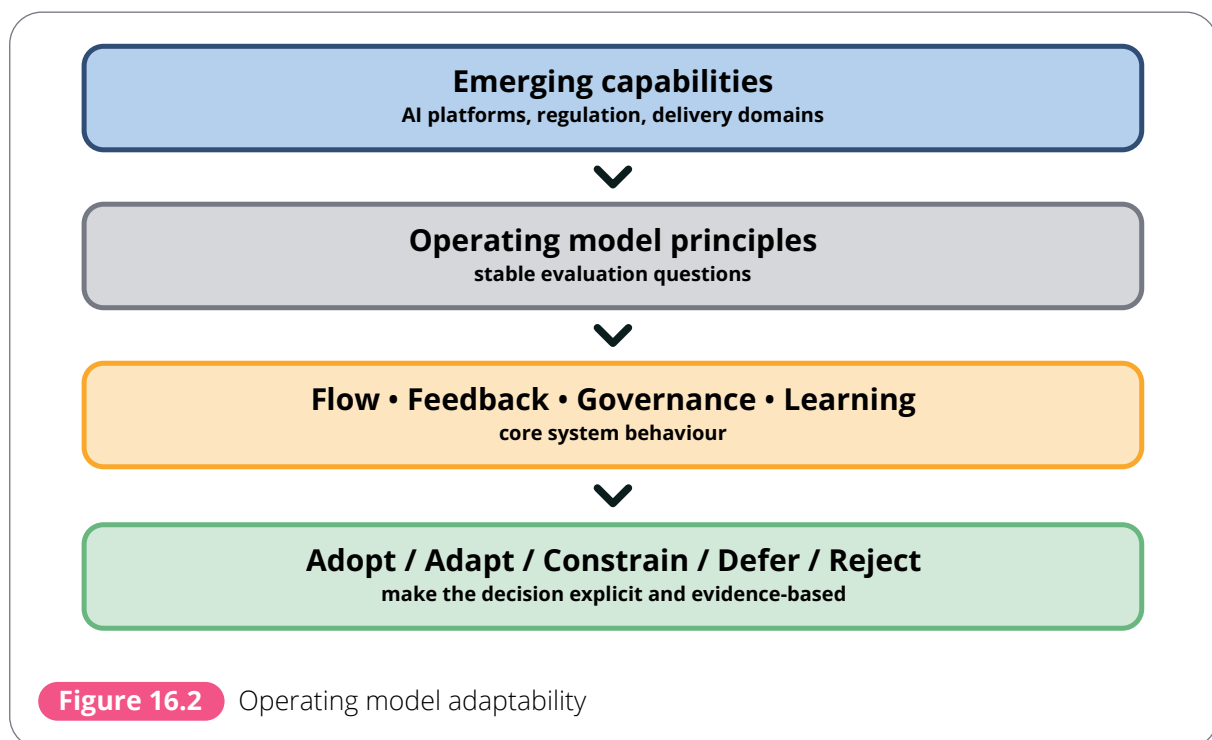
The third pressure is the extension of DevOps into high-stakes digital, AI-enabled, hardware-adjacent, operational technology, and cyber-physical environments. In these contexts, harm may affect rights, safety, financial integrity, essential services, or physical assets. Rollback may be slow, incomplete, or impossible, and accountability may cross suppliers, operators, and regulators, as well as product owners and technology teams.

These pressures interact. A context-aware platform may enable an agentic workflow in a regulated service, creating consequences that no single team or tool can govern alone. The practical implication is that novelty does not justify bypassing the operating model. Emerging capabilities should enter through the existing design, integration, testing, security, deployment, observability, measurement, and governance paths.

16.1.2 What remains stable as technology changes

Future-ready DevOps is principle-based. Tools, architectures, delivery domains, and regulations will continue to change, but the organization still needs a stable basis for deciding whether a change improves the system. The durable principles are:

- **Value before novelty:** begin with a customer, citizen, business, operational, or risk outcome rather than a technology claim.
- **Small and manageable change where possible:** limit exposure, learn from real use, and expand when the evidence supports it.
- **Fast and trustworthy feedback:** combine validation before release with telemetry, user evidence, incident learning, and operational review after release.
- **Controls proportionate to consequence:** embed security, governance, provenance, and assurance in the workflow without treating every change as equally risky.



- **Clear accountability and recovery:** name the accountable decision owner, define the permitted action space, and design containment, rollback, or other recovery before expanding authority.
- **Evidence-based learning:** record assumptions, results, exceptions, and unintended effects so that the organization can improve or stop the practice.

Durable principles do not imply identical implementation. A low-risk internal workflow and a rights-affecting public service may use the same operating questions while requiring very different controls, approval thresholds, and assurance. The strength of the Operating Model is that it preserves the questions while allowing the implementation to fit the context (ISO/IEC, 2023; NIST, 2023a).

Figure 16.2 shows emerging capabilities entering the durable DevOps principles and being evaluated through flow, feedback, governance, and learning. The outcome is a decision to adopt, adapt, constrain, defer, or reject the capabilities depending on whether the change fits existing principles or changes accountability, safety, or control assumptions.

A mature organization does not treat adoption as inevitable. It uses these stable principles to inspect an emerging capability, test it in context, govern it safely, and decide what to use, what to change, and what not to deploy. Technical feasibility is only one input to that decision.

16.2 Applying DevOps principles to emerging capability

The durable principles in Section 16.1.2 provide the basis for evaluating emerging capabilities. The evaluation questions that follow translate these principles into practical adoption decisions, with stronger assurance where potential consequences affect rights, safety, essential services, or physical systems.

16.2.1 Evaluating an emerging capability

The following questions apply the durable principles as a compact decision tool for any new AI capability, platform feature, delivery practice, assurance technique, or operating domain. The answers should be specific enough to shape a pilot, a control design, and a decision record.

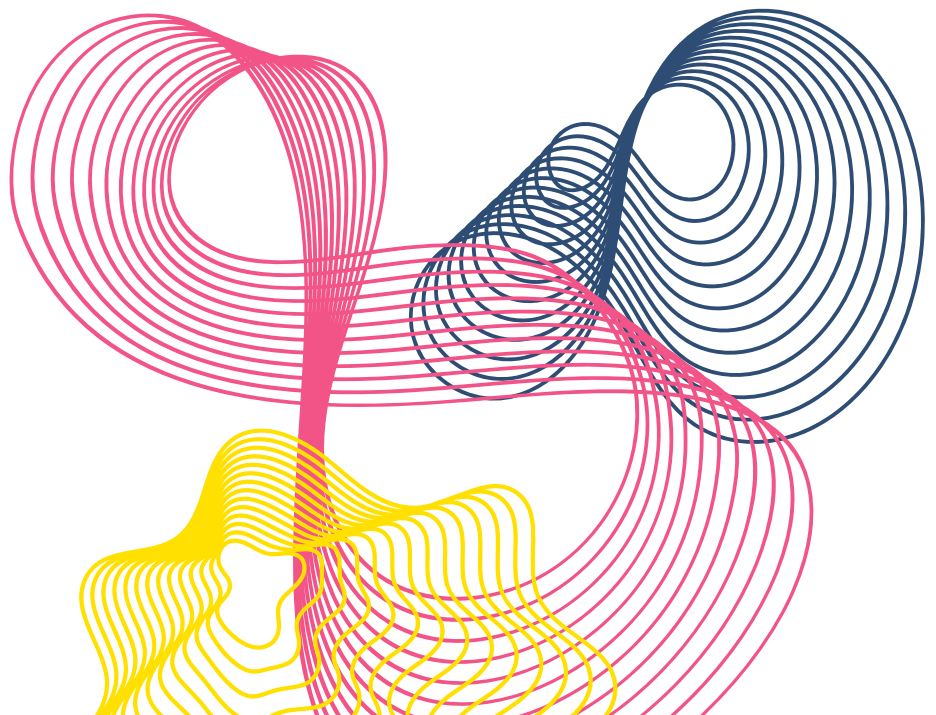


Table 16.1 Evaluating an emerging capability

Question	Why it matters
What outcome should the capability improve?	Prevents novelty, market pressure, or vendor enthusiasm from becoming the justification.
Where does it fit in the DevOps Nine Pillars and DevOps Blueprint?	Connects the capability to the existing operating model instead of creating a separate initiative.
What decisions or actions can it take?	Distinguishes information, recommendation, assistance, and direct production action.
What data, tools, services, and environments can it access?	Makes authority and exposure explicit and supports least-privilege design.
What evidence establishes confidence?	Defines provenance, validation, testing, assurance, and review expectations before wider use.
What is the potential blast radius?	Determines rollout boundaries, segregation, monitoring, and the strength of oversight.
Can the effect be reversed or contained?	Identifies cases where normal rollback assumptions are insufficient or where harm can outlast the technical change.
Who remains accountable?	Prevents automation, suppliers, or shared workflows from creating an accountability gap.
How will behaviour be observed after release?	Makes adoption dependent on production evidence rather than pre-release confidence alone.
What result would cause the organization to stop or restrict use?	Establishes explicit failure, escalation, and review conditions before pressure builds.

The evaluation should lead to one of five explicit decisions. These decisions are not a maturity ladder. Constraining, deferring, or

rejecting a capability can be the most mature response when evidence, accountability, or recovery is weak.

Table 16.2 Evaluation decisions and their meaning

Decision	Meaning
Adopt	Use the capability through normal governed DevOps practices because the evidence and controls are sufficient for the intended scope.
Adapt	Change workflows, platforms, controls, measures, or responsibilities before broader use.
Constrain	Permit only narrowly bounded use because consequence, uncertainty, or reversibility requires stronger limits.
Defer	Continue experimentation or evidence collection before operational adoption.
Reject	Do not use the capability where expected value does not justify the risk, lack of evidence, or lack of control.

The decision should be revisited when material evidence changes. A successful pilot may justify a wider boundary. An incident, new regulatory obligation, control failure, unexpected cost, or change in the operating context may justify restriction. The purpose is disciplined adaptation, not permanent approval after a single review.

16.2.2 Applying DevOps in high-stakes and broader delivery domains

Some environments place additional demands on DevOps because the consequence of change exceeds the assumptions of ordinary software delivery. The impact may involve legal rights, safety, or public trust. It may also affect financial integrity, essential services, or physical systems. Recovery may be delayed or impossible, and independent assurance may be required (ISO/IEC, 2023; NIST, 2023a; European Parliament and Council, 2024).

The stable DevOps principles still apply, but their implementation must reflect the domain. The balance between automation, governance, evidence, and human oversight changes according to the operational context. Table 16.3 summarizes some additional constraints and the corresponding DevOps responses across different delivery domains.

These environments do not invalidate Continuous Integration, Continuous Testing, Continuous Security, Continuous Delivery and Deployment, or Continuous Monitoring and Observability. Instead, they change how those capabilities are applied. They affect the acceptable batch size, the evidence required, the release path, the recovery design, and the level of decision authority. A technically reversible deployment may still have consequences for a person or physical system that cannot be reversed.

High-stakes DevOps will depend on traceability, continuous evidence, and operational resilience, as well as data protection, model governance, and risk-based approvals embedded in the delivery system. To achieve this, continuous assurance moves policy, evidence, and risk decisions into the normal delivery workflow. It reduces retrospective evidence gathering and allows human judgement to focus on material exceptions, policy interpretation, unresolved uncertainty, and high-consequence decisions. Faster change is compatible with stronger assurance when evidence is created as the work happens.

Some actions should remain human-authorized. A production system may be capable of deciding without making that decision appropriate. The operating model should preserve meaningful human oversight, especially where consequences affect rights, safety, financial position, or access to essential services.

Table 16.3 DevOps response to change in various domains

Domain	Additional constraint	DevOps response
AI-enabled decisions	Probabilistic behaviour, data dependence, explainability, and human recourse.	Provenance, evaluation, human oversight, monitoring, contestability, and controlled release.
Device and hardware fleets	Long rollout cycles, intermittent connectivity, compatibility variation, and constrained rollback.	Staged exposure, compatibility checks, fleet telemetry, inventory traceability, and recovery planning.
Operational technology and cyber-physical systems	Physical consequences, safety constraints, certification, and restricted change windows.	Stronger verification, segregation, operational approval, fail-safe design, and rehearsed containment.
Regulated digital services	Privacy, rights, financial or public consequences, and formal evidence obligations.	Continuous evidence, named decision owners, exception control, independent assurance, and retained audit trails.

Case C – CivicLink Public Services (composite): deciding whether to expand bounded autonomy

Context: CivicLink considered expanding an agentic workflow for citizen applications. The proposed system could gather supporting records, classify documents, request missing information, draft a recommendation, and update case-management records. The intended outcome was lower administrative effort and faster response without weakening accessibility, privacy, rights, or accountability.

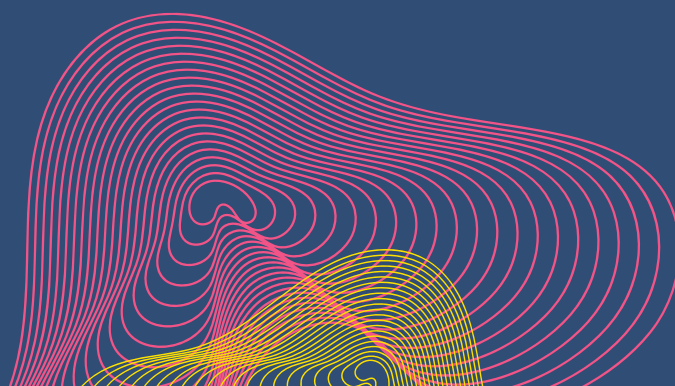
Decision: The organization mapped each action into the DOI® Operating Model. It assessed authority, data access, and user impact, along with blast radius, reversibility, evidence, and human recourse. It allowed advisory summaries and assisted in recommendation drafting, document requests, and low-risk administrative routing within narrow permissions. However, it did not allow the agent to deny a benefit, change entitlement, or communicate a final decision without accountable human review.

Evidence: The pilot required approved data and tools, least-privileged access, provenance, and complete action logs. It also demanded post-action verification, monitoring for drift and unusual tool use, a kill switch, a clear escalation path, a human fallback, and a route for citizens to challenge a decision. Accessibility and language testing identified edge cases in which apparently complete applications were misclassified.

Outcome and trade-off: CivicLink retained the parts that reduced administrative work and improved evidence. Yet it classified broader decision authority as constrained rather than approved. The evidence did not justify expansion, and the potential effect on citizens was not adequately reversible.

Transferable lesson: The appropriate level of agentic autonomy is not determined by what the model can technically execute. Instead, it is determined by outcome, risk, evidence, reversibility, rights, and accountability.

SCENARIO



DECISION AND ACTION GUIDE

Executive decision: The accountable technology and risk authority establishes how emerging capabilities will be evaluated, defines the permitted scope and autonomy boundaries, and names an accountable owner for each bounded experiment and adoption decision.

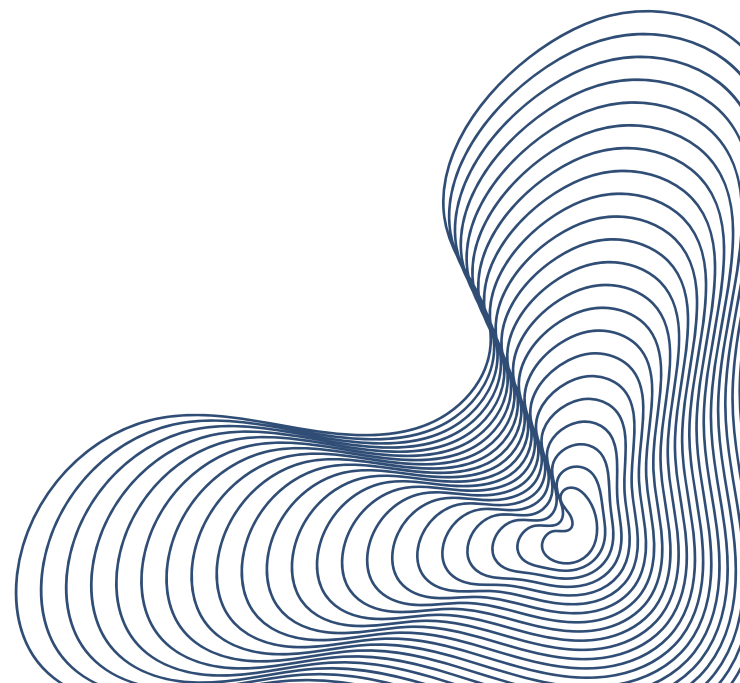
Role-specific actions: Product, architecture, engineering, platform, security, risk, operations, and governance roles map the capability to the Operating Model. Once this mapping is established, these roles define the necessary controls and evidence while running bounded experiments. Based on these outcomes, they recommend whether to adopt, adapt, constrain, defer, or reject the capability.

Evidence to review: An outcome hypothesis, Operating Model mapping, action and access boundary, risk and blast-radius assessment, validation evidence, monitoring plan, containment or recovery test, human-oversight model, and recorded adoption decision.

16.3 Chapter summary

Future-ready DevOps applies durable principles to changing technology rather than rebuilding the operating model for every trend. Increasing autonomy, context-aware and federated platforms, as well as high-stakes delivery domains all increase the need for disciplined validation, security, governance, observability, recovery, and accountability.

Emerging capabilities should be evaluated through explicit outcomes and their fit with the DOI® Operating Model. Furthermore, this evaluation must establish clear authority and require sufficient evidence. It is also critical to assess the potential blast radius and ensure process reversibility. Finally, it should incorporate human recourse and production monitoring. Adopt, adapt, constrain, defer, and reject are all responsible decisions when they are based on evidence. The organization becomes future-ready by experimenting safely, learning from real outcomes, and changing direction when assumptions prove wrong.



This final chapter turns the DevOps Standard into practical action. The earlier chapters described the operating model, the DevOps Nine Pillars, the delivery capabilities, the assessment logic, and the transformation playbook. Those ideas matter only when someone uses them to change how work is funded, designed, delivered, secured, operated, measured, and learned from.

The right next step depends on the role you play in the system. A senior executive has different work ahead than a platform engineer. A consultant or trainer has a different responsibility again, because their work multiplies through other organizations. Each path in this chapter is written for a specific reader, with concrete actions, evidence to collect, and resources to consider.

The guidance is intentionally practical. **Start with evidence. Choose a meaningful scope. Use the DevOps Nine Pillars to see the system clearly. Apply learning to real work. Verify progress through outcomes rather than ceremony.** The same discipline used throughout the book applies here: improve the system a little at a time and make the improvement visible enough that people can trust it.

The Decision and Action Guides in Chapters 1-16 identify decisions, role-specific actions, and evidence within each capability area; this chapter combines them into role-based pathways for continued improvement.

17.1 For leaders and decision-makers

Leaders and decision-makers should begin with the operating system of delivery before making another procurement decision. Many organizations already have a collection of tools, cloud services, pipelines, dashboards, and consultants. The missing element is often a clear view of where the organization is constrained and what investment will change the constraint (Humble et al., 2014; DORA, 2024).

Your first responsibility is to commission an honest DevOps maturity assessment. Use the DEVOPS INSTITUTE® Maturity self-assessment tool, supported by the Nine Pillars (guidance in Appendix A) to structure the assessment and identify constraints. The complete DOI® Maturity self-assessment tool is available through PeopleCert Plus.

Assess value streams and products in addition to departments. A department view often hides the places where work waits, changes hands, loses context, or is delayed by governance. A value-stream view shows how strategy becomes work, how work becomes software, and how software becomes user, customer, citizen, or operational value.

A good assessment separates aspirations from operational reality. Leaders may believe the organization has mature DevOps because

Your next steps

CI/CD tooling exists. Teams may experience a different system: fragile environments, manual approvals, weak incident learning, unclear product feedback, and security review that arrives late. The assessment should surface this gap without blaming teams. The purpose is to find the operating constraint.

The leader's next step is to commission an honest assessment, align sponsors, build the case from evidence, fund a focused pilot, review outcomes, and scale only what has been proven.

Commission the maturity assessment

Treat assessment as a leadership act. It requires sponsorship, time from the right participants, access to evidence, and permission to discuss uncomfortable findings. Include engineering leaders, product leaders, operations or SRE, security, architecture, finance, risk, and business representatives where the value stream crosses business boundaries. If the scope includes AI-enabled services, it should also encompass the owners of data, model behaviour, operational monitoring, and AI governance.

The assessment should produce a current maturity profile, evidence notes, confidence rating, key constraints, and an initial improvement backlog. Ask for evidence rather than opinion. Look for practices that are repeatable, used in production, visible across the chosen scope, and connected to outcomes. Be careful with averages. A single organization score can make weak reliability, brittle security, or a local platform gap disappear into a respectable number.

The assessment should also show whether the organization can manage AI-native work responsibly. Leaders need to know whether AI-generated code flows through the same review, testing, security, and provenance controls as human-authored code. They also need to know whether AI-enabled services are monitored after deployment for quality, drift, cost, fairness where relevant, prompt changes, retrieval changes, and human escalation.

Ask before approving

Ask three questions before approving any major improvement investment:
What constraint are we addressing?
What evidence shows it matters? How will we know if the system improved?

TIP



Build the transformation business case

A credible business case translates delivery and operational problems into business outcomes that leaders can understand.

A strong business case should connect maturity evidence, flow, reliability, security, and risk data. It should define the business problem, the affected value stream, the target outcomes, the required investment, the expected benefits, and the measurement approach. Avoid broad claims about becoming more modern. Leaders who have lived through several change programmes learn to distrust broad claims. Evidence earns attention.

Define a 90-day proof point before scaling. The proof point should be small enough to learn from and important enough to matter. It might reduce lead time for a real value stream, lower deployment risk for a critical service, improve incident response for a customer-facing product, automate compliance evidence for a regulated release path, or establish production monitoring for an AI-enabled capability. Overall, the proof should leave behind a stronger operating pattern that other teams can understand and adapt.

Fund the work that changes the system

Transformation work competes with feature delivery, platform work, reliability work, security automation, technical debt reduction, training, and data or AI governance. Leaders need to fund the whole system. Project-only funding is a common trap.

It creates temporary activity, then disbands the people who learned how the system works.

Priority investment areas include:

- long-lived product teams
- platform capabilities
- test automation
- observability
- security automation
- infrastructure modernization
- coaching and capability development.

Make the invisible work visible. Platform reliability, test data, deployment automation, security guardrails, model monitoring, and engineering enablement rarely fit into a feature roadmap cleanly. They still determine whether the roadmap can be delivered safely.

Decision rights matter as much as budget. If every meaningful decision is escalated to a committee, teams cannot improve flow. Leaders should define guardrails, clarify risk boundaries, and move routine decisions closer to the teams that understand the work. High-risk exceptions, regulatory interpretation, and material policy decisions still require formal oversight. Routine engineering change belongs outside avoidable ceremony.

Use DEVOPS INSTITUTE® learning resources deliberately

Learning resources, training pathways, and assessment mechanisms should be used as part of a wider capability plan. They are valuable when they create a shared language, improve judgement, strengthen practical skills, and change how work is performed. The goal is capability, not credential collection.

Table 17.1 Translating delivery issues into business impact

Delivery issue	Business impact
Slow lead time	Delayed revenue, policy delivery, experimentation, or market timing
High change failure rate	Customer trust erosion, operational disruption, and costly rework
Manual compliance	Higher audit cost, release delays, and fragile evidence
Poor environment quality	Unpredictable delivery



The leader's first 30 days

A leader can make meaningful progress quickly without launching a large programme.

The first 30 days should focus on understanding the system, identifying constraints, and establishing sufficient evidence to support a targeted improvement effort:

- Choose one value stream.
- Name the sponsor.
- Select the assessment participants.
- Gather the pipeline, incident, change, security, and customer-impact evidence.
- Run the assessment.
- Identify top constraints.
- Fund a 90-day proof point.
- Establish a cadence for reviewing outcomes.

Hold the line with intent. Prevent the effort from becoming a tool-selection exercise, dashboard collection, or organizational theatre. Ask what has changed for users, teams, risk, reliability, security, and flow.

Ask whether evidence is improving. Ask which policy, funding model, team boundary, architectural dependency, or platform gap the leadership team must remove next.

17.2 For senior practitioners

This section is intended for experienced engineers, DevOps specialists, SREs, platform engineers, architects, test leaders, security engineers, release engineers, service owners, and technical leads. Senior practitioners should use the DevOps Nine Pillars as both a team-improvement lens and a professional-development framework. As AI and platforms absorb more routine work, the centre of gravity moves from operating individual tools towards systems judgement.

Systems judgement means understanding flow, architecture, reliability, security, evidence,

and organizational constraints. Traditional skills remain essential. Automation, scripting, infrastructure, CI/CD, testing, and reliability engineering become inputs to a wider practice that designs safer delivery paths, platform capabilities, controls, and feedback loops.

Begin by assessing your own capability across the DevOps Nine Pillars. Include systems thinking, platform collaboration, risk integration, and the ability to evaluate AI-generated output and bounded agentic workflows. The goal is not to become expert in every pillar; it is to understand enough of the system to identify the constraint, make evidence-based trade-offs, and work effectively across roles.

The senior practitioner path is simple and demanding: self-assess against the DevOps Nine Pillars, choose one real skill gap, learn deliberately, apply it to a real system, measure improvement, and then broaden capability.

Choose one primary gap and one adjacent skill

The fastest professional growth often comes from pairing a primary skill gap with an adjacent skill. A test leader may choose deployment automation as the adjacent skill because test strategy only matters when it improves release confidence. A platform engineer may choose developer experience because a technically elegant platform that teams avoid is just another internal product failure. An SRE may choose product telemetry because reliability goals should be tied to user paths, beyond internal system health.

Tie learning to a real system. Choose a pipeline, an incident pattern, a testing gap, a deployment bottleneck, a security control, a platform issue, an observability weakness, or an AI monitoring gap. Learning becomes durable when it changes practice. Reading about SLOs is useful. Helping a team define an SLO for a customer-facing path, wire the SLI into monitoring, and use the error budget in release decisions is where capability forms.

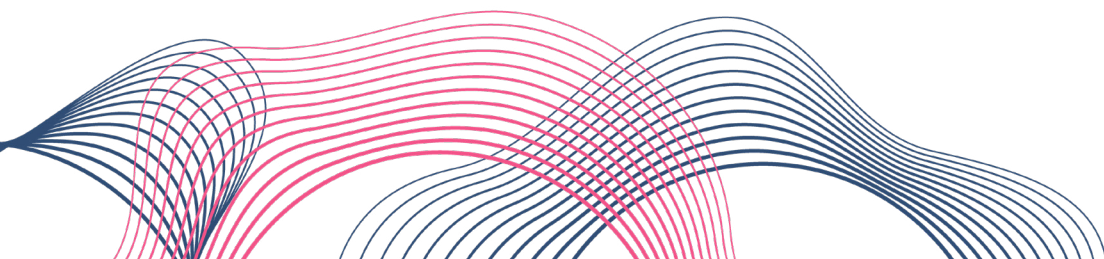


Table 17.2 Senior practitioners' guidance

Development goal	Relevant DEVOPS INSTITUTE®/ PeopleCert path	Apply it at work by
Lead DevOps transformations and operations	DevOps Leader Foundation	Establish leadership practices for transformation.
Common DevOps foundation	DevOps Foundation	Establish shared language for flow, feedback, collaboration, automation, and continuous improvement.
Reliability and SRE practice	Site Reliability Engineering (SRE) Foundation and Practitioner	Define SLOs, reduce toil, improve observability, and treat resilience as product work.
Production understanding	Observability Foundation	Improve metrics, logs, traces, OpenTelemetry awareness, and full-stack diagnostic capability.
Security in delivery	DevSecOps Foundation and Practitioner	Integrate security into design, pipelines, deployment, operations, and team ownership
Quality and feedback	Continuous Testing Foundation	Improve test culture, test strategy, faster feedback, and quality confidence.
Flow and value	Value Stream Management (VSM) Foundation	Map value streams, expose queues, improve flow, and connect work to value realization.
AI-enabled operations	AIOps Foundation	Build language around AI, machine learning, GenAI, big data, and operational intelligence foundations.

AI literacy is now a working skill, not a specialist badge. Senior practitioners need to know where AI can assist, how generated artifacts and recommendations fail, how to validate outputs, how to define permissions and guardrails, and how to monitor behaviour and cost. Judgement, context, ethics, collaboration, and the willingness to challenge an apparently efficient path remain human responsibilities.

Pick one real workflow where improvement will be visible within the next 60 days. The work should produce evidence: shorter lead time, fewer escaped defects, or faster recovery.

Build a professional development path

Start with shared language, then deepen into a specialty. A senior practitioner who lacks the common vocabulary of DevOps will struggle to influence teams. A senior practitioner who only knows the language and lacks deep engineering practice will struggle when the work gets hard. You need both. First, build the foundation. Then choose depth in reliability, security, testing, observability, value streams, platform engineering, or AI-enabled operations.

Avoid learning without changing practice. Professional development activities should lead to measurable improvements in how work is performed. Their value lies not in completion, but in the practical outcomes they enable, such as improved delivery, reliability, quality, governance, and learning.

Use learning as a forcing function for better practice

For every course or book, define a small, applied outcome before you begin.

The applied outcome should include a before-and-after signal. It might be build duration, flaky test rate, defect escape rate, alert noise, deployment failure, on-call burden, lead time for changes, vulnerability age, or customer-impact incidents. Measurement keeps learning honest. It also gives you better stories to tell when you need support from leaders.

Senior practitioners should also build mentoring habits. A senior engineer who quietly fixes every problem is useful for a while and dangerous over time. The better pattern is to create capability around you. Pair on hard incidents.

Explain the reasoning behind architectural decisions. Turn repeated support questions into platform documentation or reusable patterns. Make the work easier for the next team.

Engage with practitioner resources

Use a variety of practitioner resources, such as books, articles, technical papers, professional forums, conferences, community events, and so on, as part of an ongoing development rhythm. Practitioners should draw from multiple sources and perspectives rather than relying on a single framework, vendor, or publication. Regular exposure to new ideas helps maintain relevance as technologies, practices, and delivery models evolve.

Beyond individual learning resources, communities of practice are equally important. A practitioner should have a forum for discussing test strategy, platform patterns, production incidents, security automation, AI governance, and value-stream improvement. The best communities do more than share slides. They review real problems, compare working patterns, and produce usable standards, templates, runbooks, examples, and coaching material.

17.3 For multipliers: consultants, trainers, and coaches

In this section, multipliers are people or groups whose role enables DevOps capability and learning to spread across multiple teams or organizations. Consultants, trainers, coaches, transformation leads, enablement teams, agile coaches, assessment facilitators, and accredited learning partners may act as multipliers because their work influences capability beyond a single team. That reach makes the work valuable. It also raises the standard for evidence, humility, and practical usefulness.

Use the DevOps Standard as an engagement framework. The chapters can become a structured way to assess maturity, facilitate value stream mapping, identify constraints, build a transformation case, define a 90-day pilot, establish measurement, shape governance, and create a transformation backlog. The client should finish the engagement with durable capability and practical artifacts that people can keep using.

The multiplier's job is to help the client see its own system more clearly. That requires facilitation skill, technical literacy, operational realism, and respect for context. A technology company, a regulated organization, a public sector body, and a public-interest AI service will use different sequences. The underlying principles still apply: flow, feedback, reliability, security, measurement, learning, and accountable use of AI.

Multipliers should use this book as an engagement framework: assess, facilitate, coach, enable, build capability, and extend the work until the client can continue without dependence.

Use the DevOps Standard as a client engagement framework

Start with Chapter 12 and Appendix A.

A typical engagement should:

- Assess maturity.
- Map the value stream.
- Build the transformation case.
- Create a practical improvement backlog.

Facilitate a maturity assessment across the DevOps Nine Pillars. Ask participants to score individually first, then discuss differences using evidence. Treat disagreement as a signal. It often reveals where leaders, teams, risk owners, and product stakeholders experience different versions of the same operating system.

Follow the assessment with value stream mapping. Current-state mapping shows how work moves. It exposes queues, handoffs, manual approvals, rework, unclear ownership, environment delays, test bottlenecks, governance costs, and toolchain fragmentation. Future-state mapping should convert those findings into improvement options.

The output should be a backlog of changes that improve flow, reliability, security, and learning.

Then build the transformation case. Use Chapter 13 to define the first 90 days, the pilot scope, the capability sequence, the measurement plan, and the governance cadence. The engagement should help leaders choose a meaningful scope, fund the work, and define what evidence will count as progress. Avoid a generic roadmap. A useful roadmap is rooted in the client's actual constraints.

Design workshops that produce decisions

Workshops should create useful artifacts and decisions. Table 17.3 summarizes the core workshops, their expected outputs, and the evidence that demonstrates they have achieved their intended purpose.

Use DEVOPS INSTITUTE® resources deliberately

Multipliers should focus on building sustainable capability rather than transferring information alone. Learning should be connected to real organizational challenges and reinforced through practical application. Knowledge that is not applied fades quickly, whereas learning embedded in daily work is more likely to produce lasting improvements. Knowledge should be updated as technologies, delivery models, governance practices, and organizational needs evolve.

A client engagement should leave behind capability: people who can assess, prioritize, improve, measure, and continue learning after the engagement ends.

Coach the full system

A common coaching failure is to focus only on team behaviour while ignoring funding, governance, architecture, platform friction, and leadership decisions. Teams can improve practices inside their boundary, but many DevOps constraints sit outside the team. Multipliers should make those constraints visible and help leaders act on them.

Work with evidence. Evidence should guide coaching and consulting decisions. Start by investigating the system rather than accepting symptoms at face value.

- **If deployment is slow**, look at the pipeline, change records, environment process, approval path, test stability, and release calendar.
- **If security is described as a blocker**, map where risk evidence is generated and where decisions wait.
- **If platform adoption is low**, interview developers and observe the onboarding path.
- **If AI use is expanding**, inspect the controls for provenance, review, security, data handling, prompt change, model behaviour, and post-deployment monitoring.

Good coaching is neither abstract nor purely tactical. It connects day-to-day practice with operating model constraints. It helps a test engineer see how test strategy affects release confidence. It helps a security leader to see where automated evidence can replace manual delay. It helps a product leader see why reliability and technical debt must compete for roadmap attention. It helps executives see that invisible work is often the work that makes visible outcomes possible.

Edition maintenance

The field changes quickly. AI-assisted delivery, policy as code, software supply chain risk, platform engineering, observability, and continuous assurance will keep moving forward. A book that teaches principles can age well, but resource references need maintenance. That is ordinary publishing hygiene. The principles remain useful while references require maintenance.

Your next step should improve the system in front of you. Leaders fund and remove constraints. Practitioners build capability through real work. Multipliers help others see, change, and sustain the system.

The DevOps Standard book closes here, but the work continues. DevOps maturity is built through repeated, evidence-based improvement.

Table 17.3 Typical DevOps transformation workshops, practical outputs, and evidence of success

Workshop	Practical output	Evidence of usefulness
Executive alignment	Scope, sponsor, outcomes, investment logic	Leaders can name the first constraint and the decision they must make.
DevOps Nine Pillars assessment	Maturity profile, evidence notes, confidence rating	Participants agree on what is known, uncertain, and constrained.
Current-state value stream mapping	Map of steps, waits, handoffs, rework, controls	The map explains delays better than the org chart.
Future-state design	Improvement options and sequencing logic	Options connect to flow, risk, reliability, security, and value.
First-90-days planning	Pilot plan, owners, measures, review cadence	The plan is fundable, measurable, and small enough to learn from.
Measurement design	Metric definitions, data sources, review forums	Metrics are connected to decisions and countermeasures.
Scaling and governance	Guardrails, platform needs, dependency actions	Governance reduces friction while preserving accountability.

The organization learns by changing small parts of the system, measuring what happens, and carrying that learning into the next decision. A useful next step is specific, funded, accountable, and close enough to real work that people can tell whether it helped.

A shared operating rhythm

The three paths in this chapter should meet in the work. Leaders create conditions, practitioners improve the engineering system, and multipliers help the organization learn faster. When those paths stay separate, improvement becomes fragmented. Leaders fund a programme that teams cannot feel. Practitioners improve local practices without sponsorship. Coaches run workshops without changing decision rights. The operating rhythm below keeps the paths connected.

Start with a common cadence. During active improvement, review assessment evidence monthly without repeating the full assessment each month. Conduct a structured reassessment after each 60-to-90-day improvement cycle. Review value-stream flow and reliability signals with the teams that can act on those signals. Review investment decisions quarterly with leaders who can fund platforms, reliability, security, technical debt, and AI governance work.

Keep the cadence light enough to be useful and steady enough to create memory.

Each review should answer the same practical questions. What changed in the system? What constraint is now visible? What did we learn from production, incidents, security findings, delivery flow, customer behaviour, and AI-enabled workflows? What decision needs to be made? Who owns the next action? What evidence will show whether the action helped?

When it comes to AI, learners, practitioners, and multipliers should insist on review, provenance, telemetry, policy guardrails, model or prompt versioning, human accountability, and post-deployment monitoring. AI adoption should strengthen the DOI® Operating Model rather than create a side channel around it.

AI-RELATED CONTENT



Table 17.4 Guidance for leaders, practitioners, and multipliers

Time frame	Leader focus	Practitioner focus	Multiplier focus
First 30 days	Commission assessment, choose scope, assign sponsor, protect evidence-seeking.	Self-assess against the DevOps Nine Pillars and select one real system problem to improve.	Frame the engagement, gather context, and prepare assessment and mapping workshops.
Days 31-90	Fund the pilot, remove blockers, review outcome evidence, and avoid broad announcements.	Apply learning to a pipeline, service, platform, security control, observability gap, or AI workflow.	Facilitate value stream mapping, convert findings into backlog items, and coach decision-making.
3-6 months	Extend proven patterns, fund platforms and reliability work, and improve governance guardrails.	Mentor others, standardize useful patterns, and measure whether the system is improving.	Help the client scale practices without copying local workarounds or creating a central dependency.
6-12 months	Reassess maturity, adjust funding, and institutionalize learning through normal governance.	Deepen a specialty while broadening adjacent system understanding.	Transfer capability, update learning paths, and leave the organization able to continue without dependence.

This rhythm also protects against a quiet failure mode near the end of a book or training programme: enthusiasm without follow-through. People leave with good intent, then return to queues, approvals, overloaded teams, fragile environments, and competing priorities. Intent disappears quickly when the system remains unchanged. The next step needs sponsorship, ownership, evidence, and calendar time. The simplest useful rule is this: every next step should leave the system easier to understand and safer to change.

If an action produces more dashboards, more meetings, more reports, or more tooling without improving flow, feedback, reliability, security, learning, or value, it deserves a hard look. DevOps maturity improves when evidence changes decisions and decisions change the system.

Start with the system you have. Improve one constraint with evidence. Build capability through real work. Reassess. Then repeat with better judgement than before.

Table 17.5 Your DevOps next step

If you are...	Start here
Executive	Commission an assessment
Engineering leader	Choose one value stream
Platform team	Improve one reusable capability
SRE	Improve one reliability signal
Security	Embed one control into the pipeline
Consultant	Facilitate one assessment workshop
Trainer	Tie one course outcome to workplace change



17.4 Chapter summary

DevOps becomes useful when each role turns the operating model into a specific next decision. Leaders should commission an honest assessment, fund focused improvement, remove systemic blockers, and scale only what evidence supports. Practitioners should strengthen real delivery, reliability, security, platform, measurement, and AI governance capabilities through daily work. Consultants, trainers, coaches, and assessors should help organizations build the judgement and capability to continue without permanent dependence.

These paths must meet in a shared operating rhythm. Every next step should name the constraint, accountable owner, required evidence, and review point. AI-assisted work should remain inside the same governed delivery system, with provenance, validation, telemetry, policy, recovery, and human accountability. DevOps maturity grows through repeated cycles of assessment, focused improvement, verification, and reassessment.

A.

Appendix

This appendix provides guidance and an interpretation framework to support the DEVOPS INSTITUTE® Maturity self-assessment tool for the DevOps Standard. The complete assessment tool is available through **PeopleCert Plus**.

Purpose of this appendix:

Use this appendix to understand how to conduct an initial, evidence-based assessment across the DEVOPS INSTITUTE® Operating Model. It defines the assessment approach, maturity model, scoring guidance, confidence rating, and interpretation principles used by the DEVOPS INSTITUTE® Maturity self-assessment tool. The objective is improvement, not team ranking.

The tool uses thirteen assessment categories. The first nine categories are aligned to the Nine Pillars of DevOps. The remaining four categories assess cross-cutting system capabilities that are essential for AI-native, governable, measurable, and scalable DevOps: value stream management and flow; continuous governance, risk, and evidence; measurement, outcomes, and continuous improvement; and AI-native readiness and controls.

Use this appendix when a team, value stream, platform group, business unit, or organization needs an initial view of maturity before selecting improvement priorities. The assessment should be repeated after improvement work is completed, so progress can be verified with evidence.

Table A.1 DevOps assessment fields and their purpose

Field	Purpose
Category	The capability area being assessed.
Assessment statement	A concise practice statement to score against observable evidence.
Maturity score	A score from Level 1 to Level 5 using the maturity scale in this appendix.
Evidence notes	Specific facts, examples, data, records, and observations that support the score.
Confidence rating	High, Medium, or Low confidence in the score based on the strength of evidence.
Primary gaps	The most important missing, weak, inconsistent, or untrusted capability.
Improvement action	One practical next action that would improve capability or reduce uncertainty.
Scope	Individual, team, value stream, platform group, business unit, or organization-wide.

The DEVOPS INSTITUTE®

Maturity self-assessment tool

Table A.2 Maturity levels and scoring guidance for the DevOps assessment

Level	Name	Descriptor
1	Initial	Ad hoc, hero-driven, inconsistent, and dependent on individual knowledge. Evidence is limited or mostly anecdotal.
2	Managed	Repeatable locally. Some procedures, standards, or automation exist, but practice varies by team or context.
3	Defined	Shared standards exist across teams. Good practice can be explained, but outcomes may still be weakly measured or uneven.
4	Measured	Practices are instrumented and evaluated. Evidence guides decisions, exposes constraints, and supports risk-aware choices.
5	Optimizing	Continuous improvement is embedded. Teams learn from production, improve the platform, adjust governance, and run experiments with evidence.

Scoring rules

- Score observable behaviour, not aspiration or tool ownership.
- Use the lowest consistent level rather than the best isolated example. A strong pilot should not hide weak common practice at organization scope.
- Record evidence for every score. If evidence is weak, keep the confidence rating low even if participants agree.
- Treat disagreement as a signal. Differences between leadership, engineering, security, operations, and product perspectives often reveal the real constraint.
- Calculate a category score as the average of its three statements but interpret the profile as a system. A low category that constrains several others may be the most important priority even if the overall average looks acceptable.

Conducting the assessment

1. Define the scope. Identify whether the assessment covers an individual role, one product team, a value stream, a platform group, a business unit, or the organization.
2. Identify participants. Include people who understand how work actually happens across the assessed scope, including relevant delivery, operational, governance, and business perspectives.
3. Gather evidence. Use interviews and workshops for context, but also review toolchain data, pipeline evidence, incident records, architecture diagrams, change records, security findings, SLO reports, customer-impact data, and AI governance evidence.
4. Score individually first where useful. Individual scoring surfaces differences in perception before group discussion.

5. Discuss and converge. Compare scores, ask for evidence, identify gaps, and record confidence. The objective is not consensus at any cost; it is an honest view of system capability.
6. Identify maturity patterns. Look for imbalance across categories, weak cross-cutting governance, fast delivery without observability, security as a late gate, local excellence without organization alignment, and measurement without learning.
7. Prioritize improvement. Select actions that address the strongest constraint, reduce material risk, improve flow or feedback, or increase confidence where evidence is weak.
8. Create an improvement backlog and reassess. Review progress after a fixed period, ideally 60-90 days for an initial cycle.

The assessment can be applied at different scopes, with the evidence and interpretation adjusted to the context being assessed.

Interpreting results and prioritizing improvement

Read results horizontally across categories for balance and vertically within each category for depth. A team may score well in Continuous Integration but poorly in Continuous Monitoring and Observability, meaning it can integrate change faster than it can understand production behaviour. An organization may have strong local teams, but weak common governance, platform, or value stream visibility. The assessment is most useful when it exposes these patterns.

Prioritize improvements by weighing four factors: constraint effect, risk exposure, leverage across multiple categories, and evidence confidence. Where confidence is low, the first action may be investigation rather than implementation. Use results to remove constraints and fund improvement, not to rank teams.

Access the DEVOPS INSTITUTE® Maturity self-assessment tool

The assessment method described in this appendix provides the foundation for using the DEVOPS INSTITUTE®. The complete tool and supporting materials are maintained separately to allow updates independent of future editions of this book.

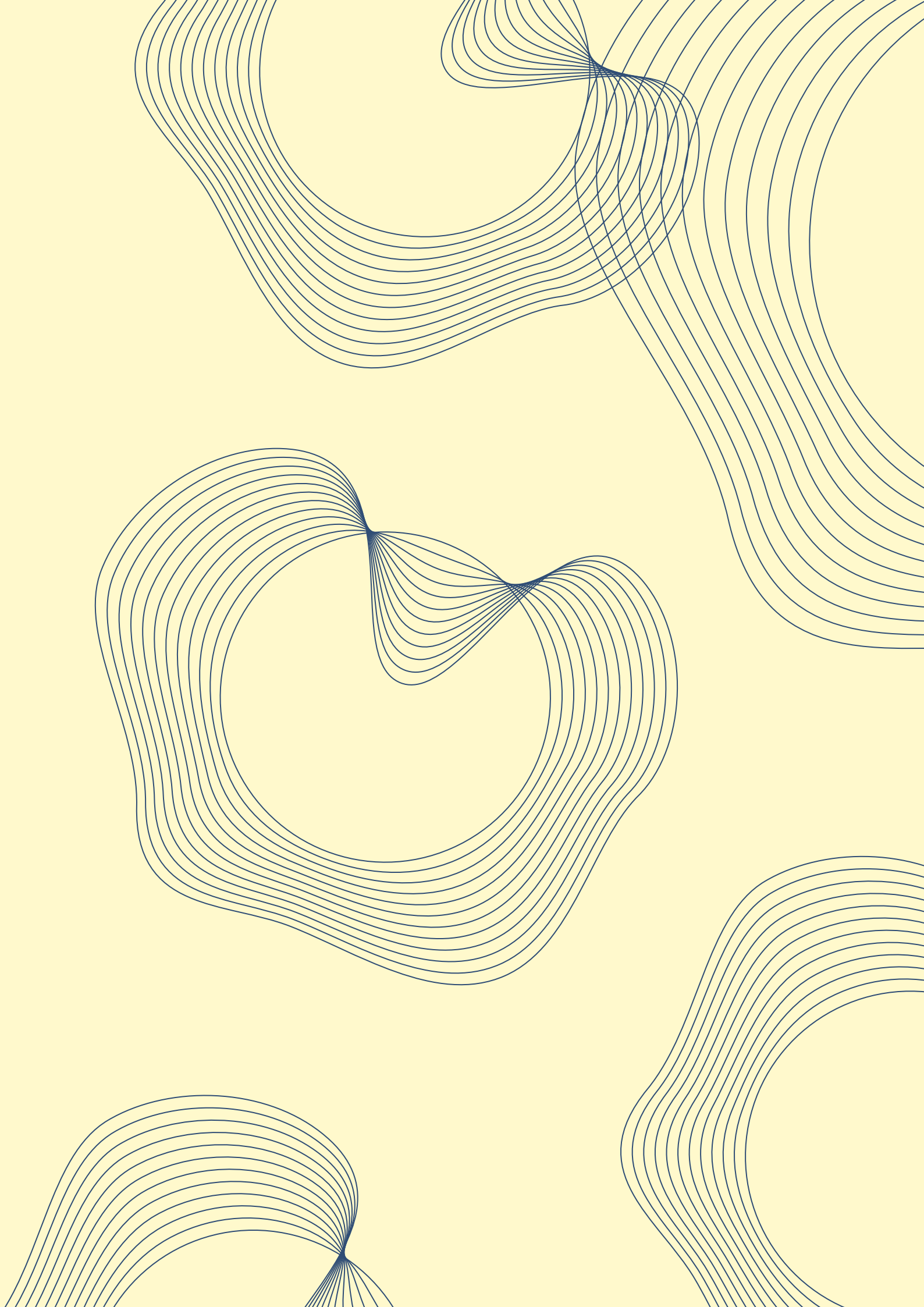
The complete self-assessment tool, including:

- the full assessment instrument
- facilitator guidance
- scoring workbook
- assessment worksheets
- improvement planning templates

is available through **PeopleCert Plus**.

Scan the QR code below or visit PeopleCert Plus to access the latest version of the tool.





B.

Appendix

This appendix is the implementation catalogue for Chapter 14. Chapter 14 explains why metrics are balanced, contextual, governed, and connected to decisions. Appendix B supplies concise definitions and collection guidance without repeating that narrative (Kersten, 2018; DORA, 2024; Pereira & Davis, 2024).

Select only the measures needed for the current decision or constraint. Define scope, ownership, data source, calculation, segmentation, and counter metrics before using a number to compare performance or direct investment.

B.1 Selecting a starter set

Begin with a small set spanning flow, quality, stability, value, and, where applicable, AI behaviour. Add measures only when the existing set is trusted and used.

Table B.1 Stepwise metric selection

Step	Focus question	Recommended action
1. State the decision	What decision will this metric improve?	Use a metric only when it can change a priority, release, reliability, security, product, or AI-governance decision.
2. Name the constraint	What is blocking better outcomes now?	Choose metrics that expose the current constraint: waiting, defects, incidents, rework, low adoption, risk, cost, or AI drift.
3. Pick a starter set	What is the smallest balanced set?	Begin with 5-7 signals: one flow, one quality, one stability, one value, one team-health signal, plus one AI signal when AI affects outcomes.
4. Define before display	Can teams trust the definition?	Document formula, source, denominator, exclusions, owner, segments, and review cadence before adding the metric to a dashboard.
5. Review with context	Who can interpret and act?	Review with the developers, operators, product owners, platform, security, or user-facing teams closest to the work.
6. Pair countermetrics	What harm could optimization create?	Pair speed with quality, quality with flow, stability with value, automation with human trust, and AI use with drift and oversight.
7. Act and verify	What changed because of the signal?	Use the metric to select one improvement, record the decision, and verify whether the change helped in the next review cycle.
8. Expand or retire	Is the metric still useful?	Add metrics only when the starter set is understood. Retire signals that do not change decisions or that confuse teams.

Recommended metrics reference

B.2 Metric definition template

Use the template to make every metric reproducible and actionable before it enters a dashboard or review.

Table B.2 Metric definition template

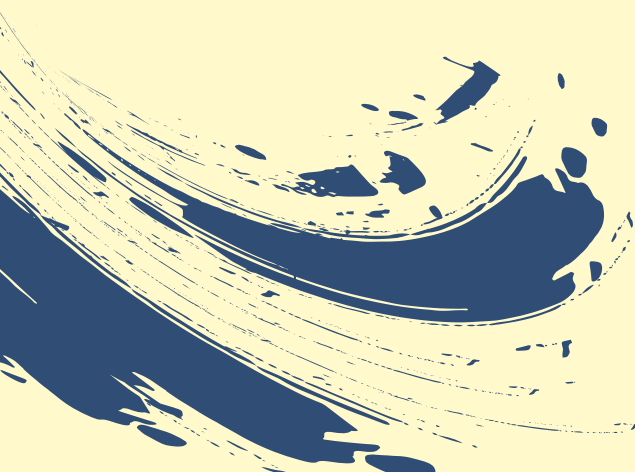
Field	Print-ready guidance
Metric name	Use a plain-language name that teams recognize.
Decision use	State the decision, constraint, or learning question the metric supports.
Formula	Define numerator, denominator, start event, end event, and exclusions.
Unit	Time, count, percentage, ratio, score, cost, percentile, or distribution.
Source	Name the authoritative system: issue tracker, CI/CD, deployment, incident, telemetry, security, product analytics, finance, AI evaluation store.
Segments	Break down by service, value stream, work type, risk class, environment, deployment type, user segment, model or prompt version.
Owner	Name the role or team that can explain the number and act on it.
Cadence	Daily, weekly, monthly, quarterly, or event-triggered.
Countermetric	Pair speed with safety, output with value, automation with quality, and AI use with trust signals.
Action rule	Define the threshold or trend that triggers investigation, action, retirement, or redesign.

B.3 Flow metrics

Use flow metrics to understand movement, waiting, work in progress, and the distribution of capacity across the value stream.

Table B.3 Flow metrics definitions and interpretations

Metric	Definition/collection	Interpretation
Lead time	Request, commitment to value/release; source from issue tracker, VSM, version control, deployment.	Long time usually means queues, handoffs, large batches, unclear ownership, or dependency delay.
Cycle time	Active work start to done/release; source from workflow states, pull requests, CI/CD.	High values point to work size, review delay, unstable validation, or unclear acceptance.
Queue time	Elapsed waiting between workflow states.	Often the most actionable bottleneck signal: approvals, shared services, security review, test environments.
Throughput	Completed flow items per period, segmented by item type.	Use with quality and value; higher counts can reflect better flow or smaller low-value work.
WIP/flow load	Started but unfinished work.	High WIP creates context switching, hides blockers, and stretches cycle time.
Flow efficiency	Active work time divided by total elapsed time.	Low efficiency indicates a system dominated by waiting rather than work.
Flow distribution	Capacity mix: features, defects, risk work, debt.	Shows whether the value stream is creating value, repairing itself, reducing risk, or paying down old decisions.



B.4 Quality metrics

Use quality measures to test whether speed produces useful, secure, maintainable outcomes rather than downstream rework.

Table B.4 Quality metrics definitions and interpretations

Metric	Definition/collection	Interpretation
Defect escape	Production or customer defects compared with all known defects; source from defect, incident, support systems.	Rising escape rate indicates weak requirements, testing, environment fidelity, review, or release validation.
Customer defects	User-reported defects by severity and affected user path.	Severity and impact matter more than raw counts.
Test effectiveness	Ability of tests to find meaningful defects; link test results to defects and incidents.	More useful than coverage alone; ask whether tests fail when behaviour is wrong.
Flaky test rate	Tests with non-deterministic pass/fail results.	High flakiness destroys trust and slows flow.
Security posture	Open vulnerabilities by age/severity, secrets exposure, dependency freshness, policy exceptions.	Security is a quality signal; review with engineering and risk owners.
Rework rate	Capacity spent on preventable fixes, repeat vulnerabilities, failed deployments, support issues.	Rework connects quality directly to flow and value.
Code health	Maintainability, complexity, duplication, dependency risk, debt age.	Use as a leading signal, not proof of production quality.

B.5 Stability metrics

Use stability measures to understand safe change, service behaviour, recovery, SLO performance, and operational load.

Table B.5 Stability metrics definitions and interpretations

Metric	Definition/collection	Interpretation
Change failure rate	Deployments causing degraded service and requiring remediation.	Pair with deployment frequency; frequent small changes can be safer than the infrequent large ones.
Failed deployment recovery time	Time to recover from a failed deployment needing intervention.	Define start and end points; use percentiles and severity segments.
Service restoration time	Time to restore acceptable service after an incident or degradation; this is a broader service metric, not a DORA measure.	Use p50/p85/p95 by severity; averages hide user pain.
SLO attainment	Requests or time meeting agreed service-level objectives.	Connects reliability to user expectations and product risk.
Error budget burn	Rate at which allowed unreliability is consumed.	Guides release risk, reliability investment, and operational focus.
Incident profile	Volume, severity, repeat incidents, customer impact.	Repeat severity often matters more than raw count.
Deployment health	Success, rollback, roll-forward, pause, canary abort, feature disablement.	Shows whether release patterns are safe in practice.
Alert quality	Actionable, urgent, user-impacting, routed correctly, non-duplicate alerts.	Noisy alerts reduce human reliability and trust.

B.6 Value metrics

Use value measures to connect delivery and operational performance to customer, user, mission, risk, cost, and developer outcomes.

Table B.6 Value metrics definitions and interpretations

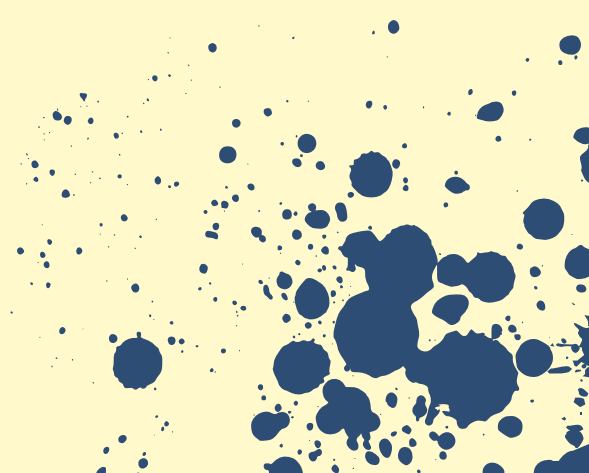
Metric	Definition/collection	Interpretation
Adoption / activation	Target users using a capability; first successful valuable use.	Adoption without success may indicate curiosity rather than value.
Task completion	Completion rate for important journeys: checkout, claim, report, onboarding, support resolution.	Concrete user paths create better engineering questions.
Conversion / mission success	Business, citizen, employee, or mission outcome completed.	Use hypotheses; avoid claiming simple causation.
Retention / repeat use	Continued use over time.	Shows whether value is durable.
Support reduction	Change in tickets, calls, escalations, or contact-centre load.	Pair with release and incident data to avoid false attribution.
Cost per outcome	Cloud, AI, support, or operations cost per transaction or successful journey.	Cost reduction that harms reliability or completion is not value.
Risk reduction	Fewer audit findings, old vulnerabilities, exceptions, manual controls, compliance effort.	Important for regulated, public-sector, and high-stakes contexts.
Developer experience	Onboarding time, platform self-service success, support load, cognitive-load reduction.	Internal platforms should be measured as products.

B.7 DORA metrics

The DevOps Standard uses deployment frequency, lead time for changes, change failure rate, and failed deployment recovery time as the DEVOPS INSTITUTE four-measure subset. DORA's current framework also includes deployment rework rate. Apply the measures at a scope where deployment and failure events can be defined consistently, and use current DORA definitions for external benchmarking (DORA, 2024; DORA, 2026a; DORA, 2026b).

Table B.7 DORA metrics: definitions, collections, and interpretations

Metric	Definition/collection	Interpretation
Lead time for changes	Time from committed change to production deployment. Collect by joining commit, artifact, deployment, and service IDs.	Delivery throughput signal. Use median and p85/p95; separate emergency, normal, risk, and service context.
Deployment frequency	Production deployments per period, or time between deployments. Collect from deployment events; measure user exposure separately when release is decoupled from deployment.	Shows ability to move in small batches. Read with change failure rate, failed deployment recovery time, quality, and value.
Failed deployment recovery time	Time to recover from a failed deployment requiring intervention. Collect from deployment, incident, rollback, and telemetry records.	Recovery capability signal. Define detection, mitigation, restoration, and closure consistently.
Change failure rate	Ratio of deployments requiring immediate intervention such as rollback, hotfix, roll-forward, feature disablement, or emergency patch.	Instability signal. Meaningful only when incidents are honestly recorded and linked to changes.

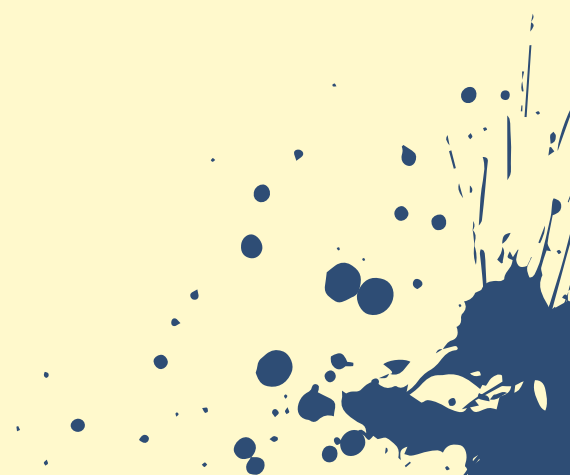


B.8 Pillar-aligned metrics

Use the pillar view to connect capability evidence with outcomes and risk. Start with the pillar that constrains the larger system.

Table B.8 Pillar-aligned metrics

Pillar	Recommended measures	How to read them
Leadership	Decision latency; blocker age; blocker removal; funding for platform, reliability, security, debt; improvement action completion.	Shows whether leaders improve the system around teams rather than request status theatre.
Collaborative Culture	Psychological safety signals; cross-functional participation; incident action completion; handoff count; team health; on-call burden.	Culture is visible in risk raising, incident learning, shared ownership, and treatment of bad news.
Design for DevOps	Deployability score; service coupling; change impact radius; observability-by-design; testability; architecture debt age.	Good design reduces coordination and creates natural places for testing, telemetry, security, and recovery.
Continuous Integration	Commit/merge frequency; branch age; pull-request size; review latency; build duration; mainline green time.	CI is a practice, not only a pipeline. Long branches and slow reviews are integration risk.
Continuous Testing	Defect escape; test effectiveness; flaky tests; execution time; risk coverage; missing-test-linked incidents; AI test quality.	Coverage is a diagnostic, not proof. Measure whether tests protect meaningful behaviour.
Elastic Infrastructure	Provisioning lead time; IaC coverage; environment drift; self-service adoption; platform reliability; policy pass rate; cost variance.	Infrastructure maturity is repeatability, speed, control, recovery, and cost visibility.
Continuous Monitoring and Observability	SLI and SLO coverage; telemetry completeness; trace coverage; alert actionability; MTTD; incident learning; product telemetry; AI monitoring.	Observability is a learning capability, not a tool estate.
Continuous Security	Vulnerability age; remediation time; dependency freshness; secrets exposure; policy as code pass rate; exception age; AI security controls.	Security measures should show whether controls are embedded in delivery, not inspected late.
Continuous Delivery and Deployment	Deployment frequency; change failure rate; failed deployment recovery time; release evidence; rollout success; rollback/roll-forward success; feature-flag debt.	Maturity is delivery confidence: deployable and releasable changes, evidence, small batches, governed exposure, and recovery readiness.



B.9 AI-specific metrics

Where models, prompts, retrieval, AI-generated artifacts, or agents affect delivery or production behaviour, connect their versions and decisions to deployment and telemetry records.

Table B.9 AI-specific metrics

Area	Recommended metrics	Guidance
Predictive model performance	Accuracy; precision; recall; F1; AUC where useful; calibration; false positives; false negatives; decision latency.	Interpret against the cost of being wrong. Segment by risk class, population, language, region, and user path.
Generative AI quality	Groundedness; factuality; hallucination rate; relevance; instruction adherence; refusal quality; safety violations; citation correctness.	Availability and speed are insufficient; a fast assistant can still degrade quality.
Retrieval/RAG quality	Retrieval precision/recall; context relevance; stale-document rate; index freshness; citation validity; no-answer correctness.	Corpus drift can degrade answers even when the model is unchanged.
Drift	Data drift; concept drift; prediction drift; prompt drift; embedding drift; retrieval corpus drift.	Diagnose before acting: fix data, retrain, re-index, re-prompt, update policy, or change product design.
Decision quality	Correct decision rate; override; appeal; reversal; human agreement; fairness/bias indicators; decision latency.	Aggregate accuracy can hide subpopulation harm. Segment every high-impact workflow.
Explainability/audit	Reason-code completeness; model/prompt/version lineage; dataset lineage; decision record completeness; oversight evidence.	Evidence depth should match risk and regulation.
AI operations	AI service latency; p95/p99; token/compute cost; cost per request; fallback; escalation; tool-call success; saturation.	Use with value and quality. A cheaper model can cost more if it increases errors and escalations.
AI-assisted DevOps	AI artifact provenance; review pass rate; defect density in AI-assisted changes; generated-test effectiveness; triage accuracy.	Count accepted, useful outcomes rather than generated output volume.

For metric anti-patterns, review cadence, data governance, interpretation, and the design of a measurement programme, use Chapter 14 as authoritative guidance.

Glossary

accountability

The obligation of a person, team, or organizational role to answer for defined decisions, controls, and outcomes. In this book, shared accountability means that several roles hold explicit, complementary accountabilities for the same system outcome; it does not mean that a task or decision has no named owner.

agentic AI

AI systems that are designed to act on behalf of users by autonomously performing predefined tasks within agreed boundaries, while remaining subject to human oversight and governance.

agile

An umbrella term for a collection of frameworks and techniques that together enable teams and individuals to work in a way that is typified by collaboration, prioritization, iterative and incremental delivery, and timeboxing. There are several specific methods (or frameworks) that are classed as agile, such as Scrum, Lean, and Kanban.

AI governance

The system by which an organization directs, controls, and monitors the use of artificial intelligence to ensure that AI-enabled digital products and services create value while remaining ethical, secure, compliant, and aligned with organizational objectives.

AI-native DevOps

DevOps in which AI participates in the delivery system by creating or modifying artifacts, supporting decisions, monitoring behaviour, or taking bounded actions, while validation, governance, observability, security, and human accountability remain explicit.

AI roles in delivery

The main roles AI may play in the delivery system: generative AI creates or modifies artifacts; machine learning detects patterns, predicts, or recommends; agentic AI acts within defined guardrails and human accountability. A single use case may combine the roles.

AIOps

The use of AI and machine learning to analyse operational telemetry and context, reduce noise, detect and correlate conditions, support diagnosis and prediction, recommend responses, and perform explicitly authorized bounded actions under human accountability.

Application Release Orchestration

The coordination of deployment and release decisions across services, pipelines, environments, dependencies, approvals, risk rules, evidence, exposure controls, and recovery paths.

artifact

A versioned or otherwise managed work product produced or consumed by the delivery system, such as source code, tests, configuration, infrastructure definitions, packages, images, documentation, model assets, prompts, or evidence.

Artificial Intelligence (AI)

The capability of digital systems to perform tasks that normally require human intelligence, such as learning from data, recognizing patterns, making predictions, or supporting decision-making.

automation and orchestration

Automation executes a repeatable task. Orchestration coordinates multiple tasks, tools, policies, dependencies, decisions, evidence, failure handling, and feedback across the delivery system.

blast radius

The scope and potential impact of a failure, change, deployment, security event, or automated action across users, services, systems, or business outcomes.

capability

The ability of an organization, person, process, application, configuration item, or IT service to carry out an activity.

change

The addition, modification, or removal of anything that could have a direct or indirect effect on products and services.

CI/CD pipeline

The automated and governed delivery path that integrates, builds, tests, scans, secures, packages, signs, and produces traceable evidence so a change can be assessed for deployment and release readiness.

Continuous Delivery (CD)

A set of techniques and tools that enables software updates to be deployed to production at any time. Frequent deployments are possible, but deployment decisions are taken case by case, usually because organizations prefer a slower rate of deployment.

continuous deployment

A set of techniques and tools that enables every change that passes automated tests to be automatically deployed to production without additional authorization. Continuous deployment relies on continuous delivery.

continuous governance

Governance embedded in normal delivery through policies, guardrails, automated and human controls, evidence, approvals, exception paths, audit trails, and improvement loops.

Continuous Integration (CI)

A set of techniques and tools that enables developers to frequently merge their code changes into a central repository, followed by automated builds and tests.

continuous monitoring and observability

The combined capability of repeatedly observing known service and delivery conditions and using telemetry to explain internal state, unexpected behaviour, user impact, and the effects of change across development and production.

continuous security

Security built into design, pipelines, platforms, deployment, release, operations, and feedback loops rather than performed as a late-stage gate.

Continuous Testing (CT)

An approach in which automated tests are executed at every stage of the deployment pipeline to ensure that changes meet quality, performance, and business requirements. It reduces risk, improves service reliability, and supports continuous integration and delivery practice.

control

The means of managing a risk, ensuring that a business objective is achieved, or that a process is followed.

cost governance

The decision rights, allocation practices, guardrails, measures, and feedback used to manage technology and AI cost relative to value, quality, reliability, security, performance, sustainability, and risk.

Definition of Done (DoD)

A list of criteria to determine if work has been completed in alignment with user acceptance needs. This ensures that all work, usually in the form of user stories, is held to the same overarching quality standards before being deemed 'done' for quality reviews.

deployment

The movement of any service component into a controlled environment.

DevOps

A socio-technical system that integrates people, process, and technology practices to support the efficient, safe, and reliable delivery of software-enabled products and services.

DevOps Blueprint

A four-layer delivery architecture composed of Elastic Infrastructure, CI/CD Pipeline, Application Release Orchestration, and Value Stream Management. Together, these layers provide the structural foundation for delivering software-enabled value streams efficiently, safely, and reliably.

DevOps culture

The observable pattern of values, behaviours, practices, and leadership responses that enables teams to raise risk early, share ownership, learn without blame, build security into daily work, and improve from evidence.

DEVOPS INSTITUTE® Operating Model

An integrated system that applies the Nine Pillars of DevOps Practices within a four-layer DevOps Blueprint architecture to support the efficient, safe, and reliable delivery of well-governed, software-enabled value streams.

DevOps maturity

The degree to which DevOps capabilities are defined, repeatable, evidenced, measured, governable, scalable, and continuously improving across the operating model.

DevOps Nine Pillars

The interdependent practice domains required for mature DevOps: Leadership, Collaborative Culture, Design for DevOps, Continuous Integration, Continuous Testing, Elastic Infrastructure, Continuous Security, Continuous Delivery and Deployment, and Continuous Monitoring and Observability.

DevSecOps

The shared DevOps capability and mindset in which security decisions, practices, controls, and evidence are integrated throughout design, build, test, delivery, operation, and learning, with security specialists setting guardrails and teams owning secure outcomes.

DORA metrics

The delivery and operational performance measures used in this book: deployment frequency, lead time for changes, change failure rate, and failed deployment recovery time.

drift

A meaningful change in data, behaviour, configuration, model performance, prompts, retrieval content, or operating conditions that can affect system outcomes over time.

elastic infrastructure

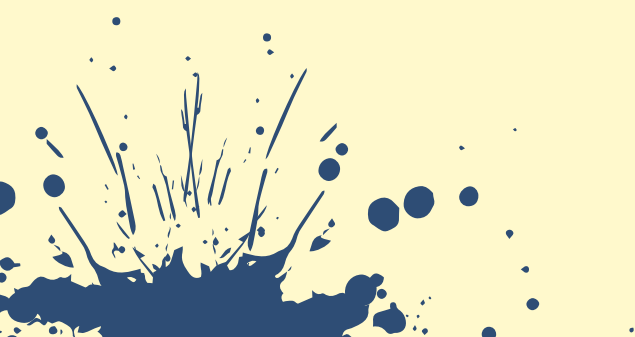
Scalable, consistent, recoverable infrastructure that can be provisioned, changed, governed, secured, and monitored repeatably across environments and workloads.

error budget

The acceptable amount of unreliability over a period, used to guide release and risk decisions.

error budget policy

A policy that defines how a team uses its allowable level of service unreliability to balance delivery speed with reliability, including what actions to take when the budget is at risk or exhausted.



evidence

Recorded, traceable information that supports a decision, demonstrates that a control or requirement was satisfied, or shows an outcome. Examples include build results, test results, security findings, approvals, deployment records, telemetry, and audit trails.

flow metrics

Measures of how work moves through a value stream, including flow time, flow load, flow efficiency, flow distribution, and related bottleneck, wait-time, and throughput signals.

Generative AI (GenAI)

AI systems that can create new content, such as text, images, code, or other outputs, based on patterns learned from existing data.

governance

The system by which an organization is directed and controlled.

groundedness

The degree to which an AI-generated response is supported by approved source information, retrieved context, or verifiable evidence.

guardrail

A predefined boundary or default, implemented in policy, platform, workflow, access control, or automation, which makes safe action easier and prevents or escalates unacceptable action.

incident

An unplanned interruption to a service or reduction in the quality of a service.

Infrastructure as Code (IaC)

The practice of provisioning and managing infrastructure through versioned, machine-readable code or configuration that can be reviewed, tested, automated, and reproduced.

Internal Developer Platform (IDP)

A product-managed internal platform that provides self-service paths for building, testing, deploying, observing, securing, and governing delivery work.

leadership

The ability to inspire, influence, motivate, and support individuals, teams, and organizations in their work towards a defined vision.

machine learning

AI techniques that learn patterns from data to classify, predict, recommend, or support decisions on new inputs.

monitoring

Repeated observation of a system, practice, process, service, or other entity to detect events and to ensure that the current status is known.

observability

The ability to understand the internal state of a complicated system by analysing its external outputs, such as metrics, logs, and traces.

operating model

A conceptual and/or visual representation of how an organization co-creates value with its customers and other stakeholders, as well as how the organization runs itself.

outcome

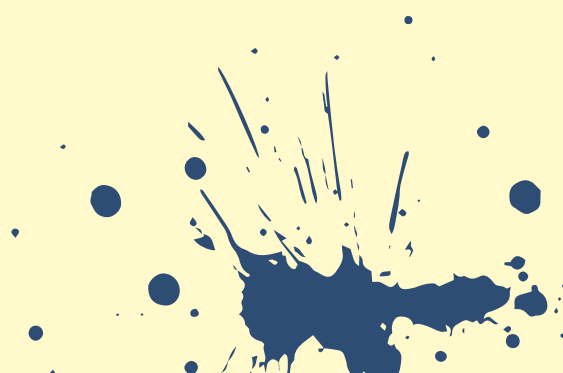
A result for a stakeholder enabled by one or more outputs.

output

A tangible or intangible deliverable of an activity.

people, process, and technology

The three practice dimensions considered within each of the Nine Pillars. People addresses roles, skills, behaviours, and accountability; process addresses ways of working, controls, and workflows; and technology addresses the tools, platforms, automation, and technical capabilities that enable the practices.



platform

A product-managed, shared set of self-service capabilities that reduces cognitive load and helps teams deliver software safely, efficiently, and consistently.

Platform Engineering

The discipline of designing, building, and operating product-managed, shared, self-service platform capabilities that reduce cognitive load and help teams deliver safely, efficiently, and consistently.

policy as code

The practice of expressing executable policy requirements in versioned code or configuration so they can be automatically evaluated, enforced, evidenced, and reported.

practice dimension

One of the three perspectives—People, Process, and Technology—used to consider how practices within each DevOps Pillar are enabled and applied.

practice domain

An area of DevOps practice represented by one of the Nine Pillars. Each practice domain brings together related capabilities and practices that contribute to the performance of the DEVOPS INSTITUTE® Operating Model.

problem

A cause, or potential cause, of one or more incidents.

product

A configuration of an organization's resources designed to offer value for a consumer.

programme

A set of related projects and activities, and an organization structure created to direct and oversee them.

progressive delivery

A release approach that gradually increases user exposure to a change based on observed evidence, production telemetry, and predefined promotion rules.

project

A temporary structure that is created for the purpose of delivering one or more business products according to an agreed business case.

provenance

Traceable information about the origin and history of an artifact, model, dataset, prompt, recommendation, or decision, including relevant source context, tool or model, human review, changes, and validation evidence.

psychological safety

Condition where people can participate without fear.

recovery

The activity of returning a configuration item to normal operation after a failure.

release

A version of a product, service, or other configuration item, or a collection of configuration items, made available for use.

release candidate

A built, tested, secured, and packaged version being evaluated as potentially ready for deployment and release.

release readiness

The evidence-based state in which a release candidate is fit for deployment and the team understands the deployment, monitoring, exposure, recovery, and approval conditions.

reliability

The ability of a product, service, or other configuration item to perform its intended function for a specified period of time or number of cycles.

responsibility

The duty and authority to perform assigned work or make a defined contribution. Responsibility can be delegated where governance permits.

risk

A possible event that could cause harm or loss, or make it more difficult to achieve objectives.

service

A means of enabling value co-creation by facilitating outcomes that consumers want to achieve, without the consumer having to manage specific costs and risks.

Service Level Indicator (SLI)

A quantitative indicator of service behaviour.

Service Level Objective (SLO)

The target value or range for a SLI.

shadow delivery

The introduction of code, configuration, infrastructure, AI-generated artifacts, or operational changes outside the normal governed delivery and evidence process.

Site Reliability Engineering (SRE)

A discipline that incorporates aspects of software engineering and applies them to infrastructure and operations problems. The main goal is to create scalable and highly reliable software systems.

software supply chain security

Practices that protect the artifacts, dependencies, tools, services, registries, build systems, model assets, generated code, and evidence that feed the delivery system.

standard

A document established by consensus and approved by a recognized body that provides, for common and repeated use, mandatory requirements, guidelines, or characteristics for its subject.

technical debt

The implied cost of additional rework caused by choosing an easy solution now instead of a better approach that would take longer.

telemetry

Metrics, events, logs, traces, and other signals emitted by systems, delivery pipelines, and AI-enabled components for monitoring, analysis, correlation, and decision support.

trunk-based development

A Continuous Integration practice in which developers integrate small changes frequently into a shared mainline or trunk, minimizing long-lived branches.

value delivery system

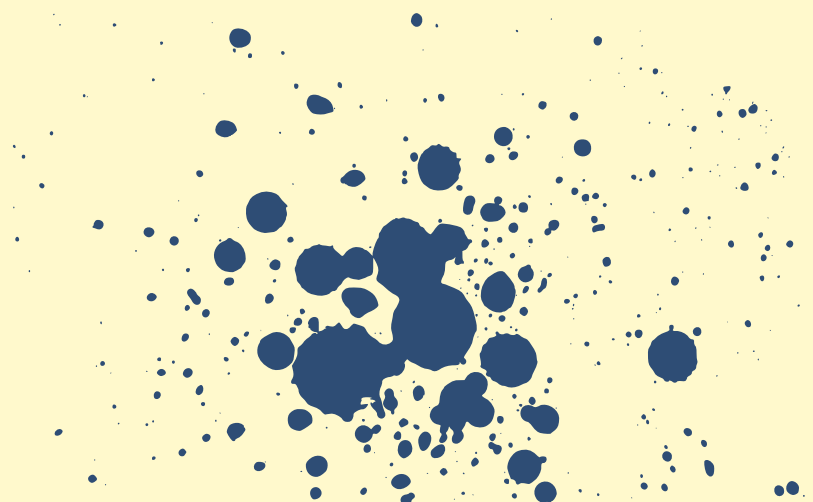
The end-to-end system by which customer or business intent becomes secure, reliable, observable, and valuable software-enabled outcomes.

value stream

A series of steps that an organization uses to create and deliver products and services to a service consumer.

Value Stream Management (VSM)

The discipline of making work, wait time, dependencies, risk, delivery performance, and outcomes visible across a value stream so teams can improve flow and value.

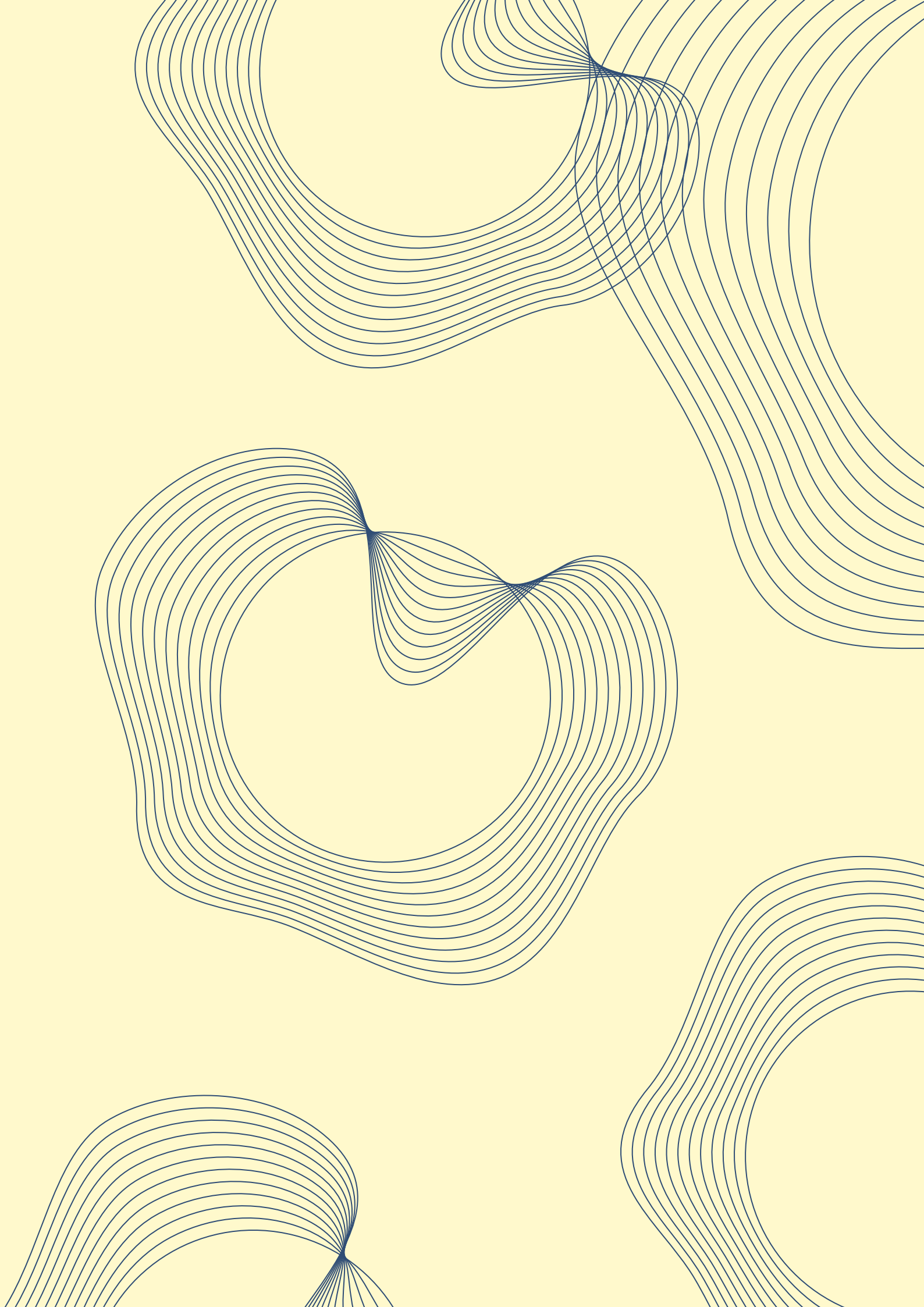


Bibliography

- Adkins, H., Beyer, B., Blankinship, P., Lewandowski, P., Oprea, A., & Stubblefield, A. (2020). *Building secure and reliable systems: Best practices for designing, implementing, and maintaining systems*. O'Reilly Media. <https://sre.google/books/building-secure-reliable-systems/>
- Autio, C., Schwartz, R., Dunietz, J., Jain, S., Stanley, M., Tabassi, E., Hall, P., & Roberts, K. (2024). *Artificial intelligence risk management framework: Generative artificial intelligence profile* (NIST AI 600-1). National Institute of Standards and Technology. <https://doi.org/10.6028/NIST.AI.600-1>
- Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., et al. (2001). *Manifesto for agile software development*. <https://agilemanifesto.org/>
- Beyer, B., Jones, C., Petoff, J., & Murphy, N. R. (Eds.). (2016). *Site reliability engineering: How Google runs production systems*. O'Reilly Media. <https://sre.google/sre-book/table-of-contents/>
- Beyer, B., Murphy, N. R., Rensin, D. K., Kawahara, K., & Thorne, S. (Eds.). (2018). *The site reliability workbook: Practical ways to implement SRE*. O'Reilly Media. <https://sre.google/workbook/table-of-contents/>
- Chen, Y., Shetty, M., Somashekar, G., Ma, M., Simmhan, Y., Mace, J., Bansal, C., Wang, R., & Rajmohan, S. (2025). AIOpsLab: A holistic framework to evaluate AI agents for enabling autonomous clouds. *Proceedings of Machine Learning and Systems*, 7. https://proceedings.mlsys.org/paper_files/paper/2025/hash/d1f9e4a9f109b6e8b75ed362736f22ec-Abstract-Conference.html
- CISA, NSA, FBI, & international cybersecurity partners. (2023). *Shifting the balance of cybersecurity risk: Principles and approaches for secure by design software*. Cybersecurity and Infrastructure Security Agency. <https://www.cisa.gov/securebydesign>
- CNCF TAG App Delivery. (2023a). *Platforms white paper*. Cloud Native Computing Foundation. <https://tag-app-delivery.cncf.io/whitepapers/platforms/>
- CNCF TAG App Delivery. (2023b). *Platform engineering maturity model*. Cloud Native Computing Foundation. <https://tag-app-delivery.cncf.io/whitepapers/platform-eng-maturity-model/>
- Conway, M. E. (1968). How do committees invent? *Datamation*, 14(4), 28–31.
- Cunningham, W. (1992). The WyCash portfolio management system. In *Addendum to the proceedings on object-oriented programming systems, languages, and applications (OOPSLA 1992)* (pp. 29–30). ACM.
- Dang, Y., Lin, Q., & Huang, P. (2019). AIOps: Real-world challenges and research innovations. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)* (pp. 4–5). <https://doi.org/10.1109/ICSE-Companion.2019.00023>
- Deogun, D., Bergh Johnsson, D., & Sawano, D. (2019). *Secure by design*. Manning Publications.
- DORA. (2024). *Accelerate state of DevOps report 2024*. Google Cloud. <https://dora.dev/research/2024/dora-report/>
- DORA. (2025). *State of AI-assisted software development*. Google Cloud. <https://dora.dev/dora-report-2025/>

- 
- DORA. (2026a). *DORA's software delivery performance metrics*. <https://dora.dev/guides/dora-metrics/>
- DORA. (2026b). *A history of DORA's software delivery metrics*. <https://dora.dev/insights/dora-metrics-history/>
- Edmondson, A. (1999). Psychological safety and learning behaviour in work teams. *Administrative Science Quarterly*, 44(2), 350–383. <https://doi.org/10.2307/2666999>
- European Parliament and Council. (2024). *Regulation (EU) 2024/1689 of 13 June 2024 laying down harmonised rules on artificial intelligence (Artificial Intelligence Act)*. *Official Journal of the European Union*, L 2024/1689. <http://data.europa.eu/eli/reg/2024/1689/oj>
- FinOps Foundation. (2025). *FinOps framework 2025*. Linux Foundation. <https://www.finops.org/insights/2025-finops-framework/>
- Forsgren, N., Humble, J., & Kim, G. (2018). *Accelerate: The science of lean software and DevOps*. IT Revolution.
- Fowler, M. (2006). *Continuous integration*. <https://martinfowler.com/articles/continuousIntegration.html>
- Fournier, C., & Nowland, I. (2024). *Platform engineering: A guide for technical, product, and people leaders*. O'Reilly Media.
- Hornbeek, M. (2019). *Engineering DevOps: From chaos to continuous improvement... and beyond*. BookBaby.
- Hornbeek, M. (2024). *Continuous testing, quality, security, and feedback: Essential strategies and secure practices for DevOps, DevSecOps, and SRE transformations*. Packt Publishing.
- Hornbeek, M., et al. (2025). *Intelligent continuous security: AI-enabled transformation for seamless protection*. O'Reilly Media.
- Humble, J., & Farley, D. (2010). *Continuous delivery: Reliable software releases through build, test, and deployment automation*. Addison-Wesley Professional.
- Humble, J., Molesky, J., & O'Reilly, B. (2014). *Lean enterprise: How high performance organizations innovate at scale*. O'Reilly Media.
- ISO/IEC. (2023). *ISO/IEC 42001:2023, information technology - artificial intelligence - management system*. International Organization for Standardization.
- Kersten, M. (2018). *Project to product: How to survive and thrive in the age of digital disruption with the Flow framework*. IT Revolution.
- Kim, G., Humble, J., Debois, P., Willis, J., & Forsgren, N. (2021). *The DevOps handbook: How to create world-class agility, reliability, & security in technology organizations* (2nd ed.). IT Revolution.
- Kruchten, P., Nord, R. L., & Ozkaya, I. (2012). Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6), 18–21. <https://doi.org/10.1109/MS.2012.167>
- Majors, C., Fong-Jones, L., & Miranda, G. (2026). *Observability engineering: Achieving production excellence* (2nd ed.). O'Reilly Media.

- NIST. (2022). *Secure software development framework (SSDF) version 1.1: Recommendations for mitigating the risk of software vulnerabilities* (NIST SP 800-218). National Institute of Standards and Technology. <https://doi.org/10.6028/NIST.SP.800-218>
- NIST. (2023a). *Artificial intelligence risk management framework (AI RMF 1.0)* (NIST AI 100-1). National Institute of Standards and Technology. <https://doi.org/10.6028/NIST.AI.100-1>
- NIST. (2023b). *Guide to operational technology (OT) security* (NIST SP 800-82 Rev. 3). National Institute of Standards and Technology. <https://doi.org/10.6028/NIST.SP.800-82r3>
- NIST. (2024). *Secure software development practices for generative AI and dual-use foundation models: An SSDF community profile* (NIST SP 800-218A). National Institute of Standards and Technology. <https://doi.org/10.6028/NIST.SP.800-218A>
- Nygard, M. T. (2018). *Release it!: Design and deploy production-ready software* (2nd ed.). Pragmatic Bookshelf.
- OpenSSF. (2026). *OpenSSF scorecard*. <https://openssf.org/projects/scorecard/>
- OpenTelemetry. (2026). *OpenTelemetry specification 1.59.0*. <https://opentelemetry.io/docs/specs/otel/>
- OWASP Foundation. (2025a). *OWASP top 10 for LLM applications 2025*. OWASP GenAI Security Project. <https://genai.owasp.org/resource/owasp-top-10-for-llm-applications-2025/>
- OWASP Foundation. (2025b). *OWASP Top 10:2025*. <https://owasp.org/Top10/2025/>
- PeopleCert. (2024a). *DEVOPS INSTITUTE® The DevOps Engineering Playbook*. Athens.
- PeopleCert. (2024b). *DEVOPS INSTITUTE® The DevOps Journey Playbook*. Athens.
- PeopleCert. (2025a). *PRINCE2 Agile (Version 2)*. Athens.
- PeopleCert. (2025b). *PRINCE2 Project Management (Version 7)*. Athens.
- PeopleCert. (2025c). *PRINCE2 Project Management Glossary (Version 7.2)*. Athens.
- PeopleCert. (2026a). *DEVOPS INSTITUTE® Master Glossary*. Athens.
- PeopleCert. (2026b). *ITIL AI Governance (Version 5)*. Athens.
- PeopleCert. (2026c). *ITIL Foundation (Version 5)*. Athens.
- PeopleCert. (2026d). *ITIL Master Glossary (Version 5.1)*. Athens.
- PeopleCert. (2026e). *Membership: DevOps Resources*. <https://www.peoplecert.org/Membership/>
- Pereira, S., & Davis, A. (2024). *Flow engineering: From value stream mapping to effective action*. IT Revolution.
- Skelton, M., & Pais, M. (2025). *Team topologies: Organizing business and technology teams for fast flow* (2nd ed.). IT Revolution.
- SLSA. (2026). *SLSA specification, version 1.2*. <https://slsa.dev/spec/v1.2/>
- Westrum, R. (2004). A typology of Organisational Cultures. *Quality and Safety in Health Care*, 13(Suppl. 2), ii22–ii27. <https://doi.org/10.1136/qshc.2003.009522>



Index

A

accountability, 29-31, 36-38, 42, 44, 49, 52, 54, 56, 58-59, 62, 70, 73, 80, 82, 86, 88, 92-94, 108, 111, 114, 119, 129-130, 136, 138, 143, 169, 184, 191-193, 196, 199, 201, 204, 206-209, 223, 230, 232, 237-242, 244-245, 253, 255, 268, 271

agentic AI, 62, 268

agile, 251, 268

AI governance, 44, 46, 56, 58-60, 62, 70, 91, 99, 107, 164, 173, 193, 247-248, 251, 253, 255, 257, 268

AI roles in delivery, 268

AI-native DevOps, 28-29, 52-53, 56, 61-62, 268

AIOps, 169-170, 178, 239, 250, 268

Application Release Orchestration, 45, 66, 70, 73, 75, 77-78, 127-128, 268, 270

artifact, 32, 35-36, 42, 46-48, 52, 54-57, 61-62, 70, 72-73, 75, 77, 82, 85, 90, 92, 101, 105, 110-111, 113-115, 119, 123, 127, 133, 135, 140, 142, 149-150, 152-156, 160, 162-163, 171, 175, 181, 186, 194, 196, 200, 203, 225, 250-252, 265, 267-268, 272-273

Artificial Intelligence (AI), 29-30, 32, 34-35, 37-38, 40, 42, 44-45, 47, 52-58, 60-62, 65, 70, 77-78, 84-86, 88-91, 94, 98-99, 104-106, 108-111, 113-114, 116-123, 125, 127-130, 132-133, 136, 140-146, 148-150, 152, 155-157, 160, 164-165, 167, 169, 172-173, 176-178, 181, 184, 186, 193-194, 198, 201, 203-204, 206-208, 210-212, 221-223, 229, 231-232, 237-239, 241, 249-254, 260-261, 264, 266-269, 271

automation and orchestration, 269

B

blast radius, 54-55, 59, 170, 176, 238, 242, 244-245, 269

C

capability, 29-31, 33-50, 52-53, 61-62, 65-67, 69-75, 77-78, 80-81, 83-84, 87-89, 91-92, 94, 96, 100-101, 104, 106-108, 117, 119-120, 124, 126, 131, 134-136, 138-145, 147-150, 152, 158, 162-164, 177, 182-185, 187, 189-190, 192-194, 196, 198, 200-206, 208-209, 215, 217, 220, 228-232, 234-235, 237-238, 240-243, 245-246, 248-252, 254-256, 258, 264-266, 268-272

change, 30, 32, 34-35, 37, 40-43, 45-50, 52-62, 66, 69, 71-73, 76-78, 80, 82-88, 90, 93-94, 96-98, 100-107, 109-117, 119-132, 134-136, 138-142, 144-145, 148-149, 152-154, 157-160, 162, 165-167, 169-175, 177-178, 181, 185, 187-188, 191-194, 196-200, 202-204, 206-216, 219-223, 225, 228, 230-232, 234-244, 246-250, 252, 254, 257-258, 260, 263, 265-267, 269-270, 272-273

CI/CD pipeline, 55, 147, 154, 167, 269

Continuous Delivery (CD), 45, 47, 55, 61, 64-66, 70, 72-74, 77, 121, 124-125, 136, 138, 143-144, 147, 154, 158, 167, 173, 183-184, 186, 200, 213, 243, 247, 249, 261-262, 266, 269-270

continuous deployment, 269

continuous governance, 75, 231, 233, 256, 269

Continuous Integration (CI), 45, 47, 53, 55, 61, 64-66, 70, 72-74, 77, 110-112, 118, 121, 123-124, 138, 143-144, 147, 154, 167, 173, 183-187, 190, 200, 213, 243, 247, 249, 258, 261-262, 266, 269-270, 273

continuous monitoring and observability, 269

continuous security, 161, 269

Continuous Testing (CT), 53, 61, 64-66, 69-70, 74, 77-78, 110-111, 114-116, 123-124, 185, 190, 192, 199-200, 243, 250, 266, 269-270

control, 29-31, 33-34, 36-38, 41-49, 53-59, 61-62, 66-67, 70-73, 75-78, 80-81, 83-85, 87-88, 90, 92, 94, 97, 100-102, 104-107, 109-110, 112, 114-116, 119, 124, 126-131, 136, 138-140, 143-160, 162-165, 167-170, 173-175, 178, 181-182, 184-185, 187, 190-193, 196-197, 199-202, 205, 207-208, 210-211, 213-214, 220, 222, 224-225, 228, 230-235, 237-239, 241-243, 245, 247, 249, 252-254, 256, 262, 264, 266, 268-271

cost governance, 269

D

Definition of Done (DoD), 115, 270

deployment, 30-32, 34-38, 42, 46-48, 52-53, 57, 59-60, 65-66, 69-73, 77, 84-87, 90-91, 93, 97-98, 101-103, 106, 113-115, 117-118, 124-136, 138, 140, 142-147, 152, 154-156, 159-160, 164, 167-170, 173-178, 183, 185-187, 192-193, 197-200, 203-205, 207-208, 213-217, 219-220, 222-225, 235, 240, 243, 247-250, 252, 261-263, 265-272

DevOps, 28-34, 36-50, 52-53, 56, 58-59, 61-62, 64-67, 69-70, 74-78, 80-84, 86-89, 91-92, 94, 96-98, 104, 106-108, 110, 123-124, 138-139, 149-150, 154, 159, 164-165, 167, 170, 173, 178, 180-187, 192-194, 196, 200-201, 204, 206-207, 209-212, 214, 218-220, 222, 224, 226, 228-229, 231, 233-243, 245-246, 249-257, 265-268, 270, 272

DevOps Blueprint, 30, 38, 65-66, 70-71, 74, 76, 78, 242, 270

DevOps culture, 86-88, 91, 270

DEVOPS INSTITUTE® Operating Model, 29-30, 65-68, 256, 270, 272

DevOps maturity, 32-33, 42, 48-49, 69, 88, 182, 184, 187, 194, 237, 246, 252, 254-255, 270

DevOps Nine Pillars, 30, 38, 61, 65-66, 69, 74, 76-78, 80, 181-182, 185-186, 193-194, 200, 209, 237, 242, 246, 249, 251, 253-254, 270

DevSecOps, 61, 88, 92, 150-151, 155, 157, 163, 200, 207-208, 250, 270

DORA metrics, 174, 181, 192, 214-216, 224, 226, 265, 270

drift, 32, 58, 61, 70, 72, 77, 87, 98-99, 103, 106, 116-117, 123, 129, 140, 142, 148-149, 154, 165, 169, 172, 178, 181, 185, 187, 222-223, 226, 234, 244, 247, 260, 266-267, 270

E

elastic infrastructure, 61, 64-66, 70-72, 77, 139, 185, 200, 266, 270

error budget, 43-46, 169, 171-172, 177-178, 211, 249, 270

error budget policy, 172, 270

evidence, 32, 34-35, 37-38, 41-42, 44-50, 52-62, 65-67, 70, 72-78, 82-92, 94, 98-99, 102, 104-106, 110-118, 120-121, 123-130, 133-136, 138-140, 142-144, 146-148, 152-154, 156, 158-164, 166-174, 176-178, 181-185, 187-194, 196, 198-203, 205-210, 212-213, 219, 221-225, 228, 230-235, 237-240, 242-258, 266-273

F

flow metrics, 65, 174, 192, 199, 208, 218, 262, 271

G

Generative AI (GenAI), 271

governance, 29-34, 36-47, 50, 52-53, 56, 58-62, 65-67, 70, 72, 75-76, 78, 80-82, 85, 89-91, 97-99, 102, 104-105, 107-108, 120, 124-129, 135, 138, 141, 143, 145-146, 150, 154, 156-157, 163-164, 172-174, 181, 183-184, 187, 191-194, 196, 199, 201, 203-205, 207-209, 211, 222-223, 229-233, 235, 238, 240-241, 243, 245-248, 250-258, 267-269, 271-272

groundedness, 222, 271

guardrail, 35, 42, 46, 54, 57-59, 65, 72, 75-76, 80, 82-83, 87, 91-92, 94, 108, 139, 141, 146, 148-150, 154, 158-159, 162, 170, 172, 177, 204-205, 209, 230, 232, 235, 238-239, 248, 250, 253-254, 268-271

I

incident, 30, 32, 35, 38, 41-42, 45-46, 48-49, 52, 54-56, 58-61, 65, 71-74, 82-88, 90-92, 98-99, 105, 107-108, 119, 122-123, 129, 132, 134, 140, 142, 145, 148, 150, 157-161, 164, 167-170, 172-176, 178, 183-185, 187-188, 191-193, 196, 198-199, 203, 208, 211, 213-216, 218-221, 223-224, 234, 240, 243, 247-251, 253, 257, 260-261, 263-266, 271-272

Infrastructure as Code (IaC), 31, 71, 140, 149, 152, 185, 200, 266, 271

Internal Developer Platform (IDP), 143-144, 271

L

leadership, 32, 40, 50, 65, 69, 74, 77, 80-84, 86, 90, 92-94, 96, 135, 145, 149, 183, 193-194, 197, 201, 208, 218, 223-224, 247, 249-250, 252, 257, 270-271

M

machine learning, 54, 169, 250, 268, 271

monitoring, 36, 42, 45, 52-53, 55-56, 58-59, 61-62, 67, 72, 77, 99, 105-106, 108-109, 114, 117-119, 123-124, 126, 129-131, 144-145, 148, 156-157, 160, 163-164, 167, 169-170, 172-173, 176-177, 183-184, 187, 192, 194, 232, 234-235, 242-245, 247-249, 252-253, 266, 268, 271-273

O

observability, 30, 32, 34, 36-38, 42-43, 45, 47-49, 52, 61, 65, 71, 74, 87-88, 96-100, 103, 106-108, 115, 119, 132, 135, 138, 141, 143-144, 148, 164-177, 186, 191, 193, 196-197, 199-202, 204-205, 207-209, 213, 218-219, 234, 240, 245, 248-250, 252, 254, 258, 268-269, 271

operating model, 29-33, 36-37, 39-40, 42-44, 47, 52-53, 56, 61, 66-67, 70, 75-76, 78, 80, 82-84, 146, 165, 181, 184, 196, 203-204, 207-208, 229, 233-234, 238, 240, 242-243, 245-246, 252, 255, 270-271

outcome, 29, 32-34, 38-39, 43, 45-50, 53, 55, 57, 60-62, 67, 69-70, 73-78, 81-84, 87, 92, 96, 99, 103-104, 106, 121, 123, 129, 133-135, 138-139, 143, 146, 149, 161, 165, 167, 170, 172, 177-178, 182-184, 188-193, 196-203, 205, 207-210, 212-215, 218-220, 222, 225, 228, 230, 232-233, 235, 237-238, 240-242, 244-250, 252-254, 256-257, 260, 263-264, 266-271, 273

output, 34, 43-44, 47-48, 52-53, 55-59, 61, 80, 83-85, 89-90, 94, 99, 105, 116, 119-120, 122, 127-129, 133, 148, 152, 156-157, 166, 172, 183, 189, 193, 201, 204, 209, 213-214, 222, 249-250, 252-253, 261, 267, 271

P

people, process, and technology, 40, 69, 74, 185, 270-271

platform, 29-31, 33-34, 36-42, 44, 46-49, 58-59, 61-62, 65-67, 69-70, 75-78, 80-83, 86-89, 91-94, 96, 100, 104, 107-108, 110, 114, 119-120, 123-124, 131, 138-139, 141-150, 158-159, 161-164, 167-168, 170, 178, 181-184, 187-189, 191-194, 197, 199-205, 207-209, 211, 213, 219-220, 225, 228-235, 238-242, 245-258, 260, 264, 266, 269, 271-272

Platform Engineering, 36, 61, 64-65, 104, 138-139, 142-143, 149, 228, 231, 233, 237, 272

policy as code, 59, 92, 144, 193, 197, 199, 252, 266, 272

practice dimension, 271-272

practice domain, 31, 38, 65-67, 69-70, 74, 80, 86, 96, 110, 270, 272

problem, 29-31, 35, 39, 41-42, 45-48, 57, 70, 85-86, 88, 91, 94, 111-112, 114, 116, 119-120, 129, 133, 143-144, 146, 153, 166-168, 174, 182, 190, 193, 196, 198-199, 204, 206, 208, 212, 215, 217-218, 220, 222-224, 228, 248, 250-251, 254, 272-273

product, 31, 35, 37-38, 40, 42-43, 45-47, 50, 56, 58, 61, 65, 69, 72, 74-75, 77, 80-83, 85-89, 96, 98-99, 103-105, 107-108, 113, 119, 124, 127, 135-136, 138-140, 143-147, 149, 155, 158, 160-161, 164, 168-172, 174-178, 183-184, 188-190, 192, 196-197, 200, 202, 205, 208, 211-213, 215-216, 218-222, 228, 230-235, 237-238, 240, 246-252, 257, 260-261, 263-264, 266-270, 272-273

programme, 38, 43, 45, 50, 80, 96, 155-156, 181-182, 190, 202, 204, 206, 210, 214, 223, 226, 229, 248-249, 253-254, 267, 272

progressive delivery, 61, 272

project, 43, 45-46, 50, 107, 234, 272

provenance, 31-32, 35-36, 38, 44, 54-57, 60-62, 76-77, 85, 89-91, 104-105, 107-108, 113, 119, 123, 127, 136, 153, 155-156, 160, 162, 173, 183, 192, 203, 234-235, 240, 242, 244, 247, 252-253, 255, 267, 272

psychological safety, 61, 90, 175, 185, 200, 272

R

recovery, 43, 47-48, 54-55, 62, 65, 69-71, 73, 75, 77, 85, 93, 97, 101-102, 104-108, 121, 124, 126-127, 130, 132, 134-136, 140-141, 157, 159, 170, 174, 185-187, 190, 192, 199-200, 203, 210, 212-216, 219-220, 223, 225, 234, 237-239, 241-243, 245, 250, 255, 263, 265-266, 268, 270, 272

release, 30-32, 34-35, 37-38, 42-50, 52, 55-56, 59, 61, 65, 70-77, 80, 82-83, 86-87, 93, 96-99, 101-102, 106, 108-110, 112-113, 115-117, 119, 122-136, 140, 148, 150-151, 155-156, 158, 162-163, 166-167, 171-172, 174, 176-178, 183, 186-187, 190, 192, 196, 198, 200-204, 207, 211-212, 217-220, 232, 234-235, 238, 240, 242-243, 248-249, 252, 260, 262-266, 268-270, 272

release candidate, 110, 124-127, 156, 272

release readiness, 47, 74, 126-127, 190, 269, 272

reliability, 29-30, 32, 34, 36, 38, 40-42, 45-49, 52, 55-57, 62, 65, 67, 69, 72, 74, 77, 82, 86-87, 96-99, 104, 107, 111, 115-116, 138-139, 141, 147-149, 158-159, 164, 169-172, 175, 177-178, 181, 184, 187, 191, 194, 199-200, 202, 205, 207-208, 211, 214, 218, 220-221, 231, 233-234, 247-255, 260, 263-264, 266, 269-270, 272

responsibility, 40, 43, 55-56, 58, 62, 67, 69, 73, 80, 82, 86, 92, 108, 114, 119-120, 148-149, 161, 172, 199, 210, 223-224, 234-235, 237, 242, 246, 250, 272

risk, 29, 32, 34-39, 41-46, 48, 52-57, 59, 61-62, 65, 71-78, 80-87, 90, 92-93, 97, 101, 104-105, 107-110, 112, 114-115, 117-124, 126-131, 134-136, 139, 141, 148, 150, 152-155, 157-158, 160-163, 168-173, 182, 184, 188-189, 191-193, 196, 198-215, 217-220, 222, 224, 230-235, 237-238, 240, 242-245, 247-249, 251-253, 256, 258, 260-270, 272-273

S

service, 32, 34, 37, 40, 42-43, 45-46, 50, 54-56, 58, 60-61, 65, 69, 72-73, 75, 77, 82, 86-88, 91-93, 97-100, 104-106, 108-109, 115-117, 124, 126-133, 135, 138-141, 143-149, 153, 155-156, 158, 160, 166-172, 174-175, 177-178, 181, 185-188, 190, 192, 194, 199, 202-204, 207-208, 212-216, 218-220, 223, 228, 230-232, 234-235, 239-243, 246-249, 251, 254, 261-263, 265-273

Service Level Indicator (SLI), 46, 77, 127, 171, 176, 178, 203, 249, 266, 273

Service Level Objective (SLO), 45-46, 77, 127, 171-172, 174, 176-178, 193, 200, 203, 212-213, 219, 223, 225, 249, 257, 263, 266, 273

shadow delivery, 61, 114, 273

Site Reliability Engineering (SRE), 29, 36, 42-46, 50, 55, 77, 127, 136, 164, 168-171, 178, 186, 211, 247, 249-250, 254, 273

software supply chain security, 273

standard, 28-31, 35-36, 39, 42, 52, 55, 61-62, 86, 104-105, 107, 110, 114, 117, 119, 127, 143, 145, 147-148, 150, 161, 174, 183-184, 187, 193-194, 229, 232, 237, 251, 257, 270, 273

T

technical debt, 47, 49, 82, 87, 107, 191, 199, 208, 215, 248, 252-253, 273

telemetry, 30, 32-34, 46, 53, 55-56, 59-61, 65, 70, 72-75, 77-78, 98-99, 104, 106, 108, 126, 129-131, 136, 139, 141, 147, 159, 163-171, 173, 176-178, 183, 190, 192, 198, 202-203, 211, 220, 231, 234-235, 239-240, 243, 249, 253, 255, 261, 265-269, 271-273

trunk-based development, 112, 273

V

value delivery system, 40-42, 50, 273

value stream, 30-31, 33, 35-38, 43-45, 50, 52, 56, 58-59, 61, 66-67, 74-75, 87, 91, 135, 139, 146, 174, 181, 188, 190, 196-199, 201-204, 206-209, 212, 215-217, 228-235, 237, 246-251, 253-254, 256-258, 261-262, 270-271, 273

Value Stream Management (VSM), 66, 70, 73-74, 77, 230, 233, 235, 250, 262, 270, 273

Endnotes

1. The composite-case format is a DEVOPS INSTITUTE® synthesis informed by established work on Continuous Delivery, SRE, software-delivery performance, organizational design, service management, project management, and AI risk management (Humble & Farley, 2010; Beyer et al., 2016; Forsgren et al., 2018; Autio et al., 2024; DORA, 2024; PeopleCert, 2025c; Skelton & Pais, 2025; PeopleCert, 2026d).
2. In this book, “no authoritative standard” means that no single, universally governing, cross-vendor publication defines the entire DevOps operating system. It does not diminish the authority of established methods, frameworks, standards, research, or engineering guidance within their own scopes (Beck et al., 2001; Beyer et al., 2016; Humble & Farley, 2010; DORA, 2024; PeopleCert, 2025c; PeopleCert, 2026d).
3. The definition above is original to this DEVOPS INSTITUTE® edition. It intentionally extends narrower movement-, culture-, and tool-centred definitions while preserving formal ITIL and PRINCE2 Project Management meanings when those disciplines are explicitly applied (PeopleCert, 2025c; PeopleCert, 2026d).
4. DEVOPS INSTITUTE® original definition and model for this edition.
5. DEVOPS INSTITUTE® synthesis informed by the Agile Manifesto, ITIL, PRINCE2 Project Management, and Google SRE (Beck et al., 2001; Beyer et al., 2016; PeopleCert, 2025c; PeopleCert, 2026d).
6. DEVOPS INSTITUTE® synthesis informed by the Agile Manifesto, ITIL, PRINCE2 Project Management, and Google SRE (Beck et al., 2001; Beyer et al., 2016; PeopleCert, 2025c; PeopleCert, 2026d).
7. DEVOPS INSTITUTE® synthesis informed by the NIST AI Risk Management Framework and Generative AI Profile (NIST, 2023a; Autio et al., 2024).
8. DEVOPS INSTITUTE® synthesis informed by the NIST AI Risk Management Framework, the NIST Generative AI Profile, and the NIST SSDF AI Profile (NIST, 2023a; Autio et al., 2024; NIST, 2024).
9. DEVOPS INSTITUTE® original Nine Pillars model for this edition (Hornbeek, 2019).
10. DEVOPS INSTITUTE® original DevOps Blueprint for this edition (Hornbeek, 2019).
11. Composite worked example informed by Continuous Delivery, SRE, secure-software guidance, ITIL, PRINCE2 Project Management, and the DEVOPS INSTITUTE® Operating Model (Humble & Farley, 2010; Beyer et al., 2016; Hornbeek, 2019; NIST, 2022; PeopleCert, 2025c; PeopleCert, 2026d).
12. DEVOPS INSTITUTE® adaptation of the team types and interaction modes described in Team Topologies (Skelton & Pais, 2025).



- 
13. DEVOPS INSTITUTE® synthesis informed by Continuous Delivery guidance and the ITIL glossary definitions (Humble & Farley, 2010; Hornbeek, 2019; PeopleCert, 2026d).
 14. Informed by Hornbeek (2019) and PeopleCert (2026d)
 15. DEVOPS INSTITUTE® synthesis informed by the FinOps Framework (FinOps Foundation, 2025).
 16. DEVOPS INSTITUTE® synthesis informed by the NIST SSDF, SLSA provenance and verification guidance, and OpenSSF Scorecard (NIST, 2022; Hornbeek, 2025; OpenSSF, 2026; SLSA, 2026)
 17. DEVOPS INSTITUTE® synthesis informed by the NIST SSDF AI Profile and the OWASP Top 10 for LLM Applications (NIST, 2024; Hornbeek, 2025; OWASP Foundation, 2025a).
 18. DEVOPS INSTITUTE® synthesis informed by the OpenTelemetry specification and signal model (OpenTelemetry, 2026).
 19. DEVOPS INSTITUTE® synthesis informed by AIOps research, Google SRE, OpenTelemetry, and current research on operational AI agents (Beyer et al., 2018; Dang et al., 2019; Chen et al., 2025; OpenTelemetry, 2026).
 20. The five-level DevOps maturity scale and the assessment instrument in Appendix A are original DEVOPS INSTITUTE® constructs for this edition. They are informed by evidence-based improvement and maturity practices, but the categories, statements, and scoring guidance are not a DORA, ITIL, or PRINCE2 assessment model.
 21. DEVOPS INSTITUTE® original maturity model for this edition.
 22. DEVOPS INSTITUTE® synthesis based on DORA research. The figure shows the four-measure DEVOPS INSTITUTE subset; DORA's current software delivery framework uses five metrics. It adds deployment rework rate - the ratio of unplanned deployments made in response to production incidents - and groups the measures into throughput and instability. Readers using DORA for external benchmarking should use DORA's current five-metric definitions (DORA, 2024; DORA, 2026a; DORA, 2026b)
 23. DEVOPS INSTITUTE® synthesis informed by Engineering DevOps (Hornbeek, 2019) and the Flow Framework and Flow Engineering (Kersten, 2018; Pereira & Davis, 2024).
 24. DEVOPS INSTITUTE® original organization-scale synthesis for this edition, informed by established work on team boundaries, platform products, value streams, governance, architecture, and long-lived product funding (Kersten, 2018; CNCF TAG App Delivery, 2023a; Skelton & Pais, 2025).
 25. Composite case informed by work on product and value-stream funding, Team Topologies, platform products, software-delivery performance, and Lean Enterprise (Humble et al., 2014; Kersten, 2018; CNCF TAG App Delivery, 2023a; DORA, 2024; Skelton & Pais, 2025).



THE LANGUAGE OF COLLABORATION



9 789925 397006