DevSecOps A Complete Visual Guide

Sample Book
Chapters 1 to 2

From Secure Foundations to Practical Delivery and Resilient Operations

Visual Guide to Computer Systems

Introduction

This book is designed for readers who are new to DevSecOps and systems engineers who want to organize and deepen their knowledge of the subject, from foundations through practical intermediate and advanced applications.

Rather than asking you to memorize more than you need, the book builds from essential concepts to the intermediate and advanced applications commonly used in day-to-day work, explaining them in enough depth for you to apply them in practice.

Processes, relationships, and structures that can be difficult to picture from text alone are supported by diagrams. By moving between explanations, visuals, and examples, you can build a clearer and more practical understanding of the subject.

Product names, service names, logos, icons, and other identifying marks mentioned in this book are trademarks or registered trademarks of their respective owners. This book is not an official publication, certified training resource, or endorsed learning material of any company or organization.

How to Use This Book

Read the chapters in order when you are new to DevSecOps. The sequence builds the necessary background step by step, beginning with risk, trust, governance, threat modeling, and architecture before moving into implementation, delivery, operations, recovery, and organization-wide scale.

At the beginning of each chapter, review **What You'll Be Able to Do**. It identifies the practical capabilities the chapter is designed to develop and gives you a clear target for the explanations and exercises that follow.

Use the diagrams to understand relationships, processing paths, trust boundaries, state transitions, and responsibility divisions that are difficult to hold in mind from prose alone. The visuals support the text; they do not replace the mechanisms, precautions, or verification steps described in the chapter.

Run the code, commands, policy examples, and hands-on exercises in a safe test environment whenever possible. Follow the stated prerequisites, compare the result with the expected outcome, and perform the verification step rather than treating successful command execution as proof that the control works.

At the end of each chapter, use the **Chapter Checkpoint** as a summary and answer the **Review Questions** without referring immediately to the text. This confirms both conceptual understanding and the ability to apply the material to engineering decisions.

The intermediate and advanced applications in this book build directly on the foundations taught earlier. Work through the practical examples, including negative tests and failure cases, and verify the observed result. That process is the most reliable way to turn familiarity with a technique into operational competence.

Table of Contents

Introduction

How to Use This Book

Volume I — Foundations and Secure Engineering Flow

- The Big Picture
- Reference System Used Throughout

Chapter 1 — DevSecOps as a Continuous Risk-Reduction System

- Start with the Outcome
- The Lifecycle Is a Set of Feedback Loops
- Controls: Prevent, Detect, and Correct
- Choosing How a Check Behaves
- The Delivery System Is Part of the Attack Surface
- A Practical Mapping Exercise
- A First Maturity Baseline
- Diagnosing Weak Controls
- Common Misconceptions
- Chapter Checkpoint

Chapter 2 — Security, Trust, and Risk Foundations for Systems Engineers

- Security Decisions Begin with Consequences
- What Security Must Preserve
- Attack Surface, Trust Boundaries, Privilege, and Blast Radius
- Design Principles That Reduce Dependence on Perfect Detection
- Zero Trust Without the Marketing Layer
- Classification Drives Proportionate Controls
- Threat Actors Relevant to Delivery Systems
- Building the Reference-System Asset Inventory
- Creating a Risk Register That Supports Decisions
- Severity Is One Input, Not the Decision
- Testing Trust Assumptions Under Failure
- Troubleshooting Risk Work
- Misconceptions That Distort Decisions
- Chapter Checkpoint

Volume I — Foundations and Secure Engineering Flow

The Big Picture

DevSecOps makes security a continuous, measurable property of the engineering system rather than a review performed after engineering is complete. The objective is not to run the greatest possible number of scanners. It is to create short, trustworthy feedback loops that prevent avoidable defects, detect meaningful risk early, preserve the integrity of delivery, and help teams recover safely when prevention fails.

A strong DevSecOps system connects five ideas:

1. **Risk provides direction.** Teams identify the assets, trust boundaries, failure modes, and business consequences that matter.
2. **Design removes entire classes of problems.** Secure architecture reduces exposure, privilege, and hidden trust before implementation begins.
3. **Automation makes repeatable decisions.** Source control, CI/CD, identity, static analysis, dependency controls, and policy checks produce consistent evidence.
4. **People retain accountable judgment.** Automation can enforce known rules, but it cannot fully understand business authorization, operational trade-offs, or every novel attack path.
5. **Evidence closes the loop.** A control is credible only when its inputs, decision, result, owner, and failure behavior can be observed.

This volume develops the engineering foundation through **Chapter 11**. It establishes the risk model, governance, threat modeling, secure architecture, source and pipeline trust, workload identity, secure coding, dependency governance, and infrastructure policy needed before trustworthy release and runtime practices can be built on top.

Reference System Used Throughout

The worked examples use a small internet-facing order-processing service. It exposes an API, stores orders in a relational database, and calls an external payment-status service. The application is maintained in source control, tested in CI, configured through infrastructure as code, and intended for deployment to Kubernetes.

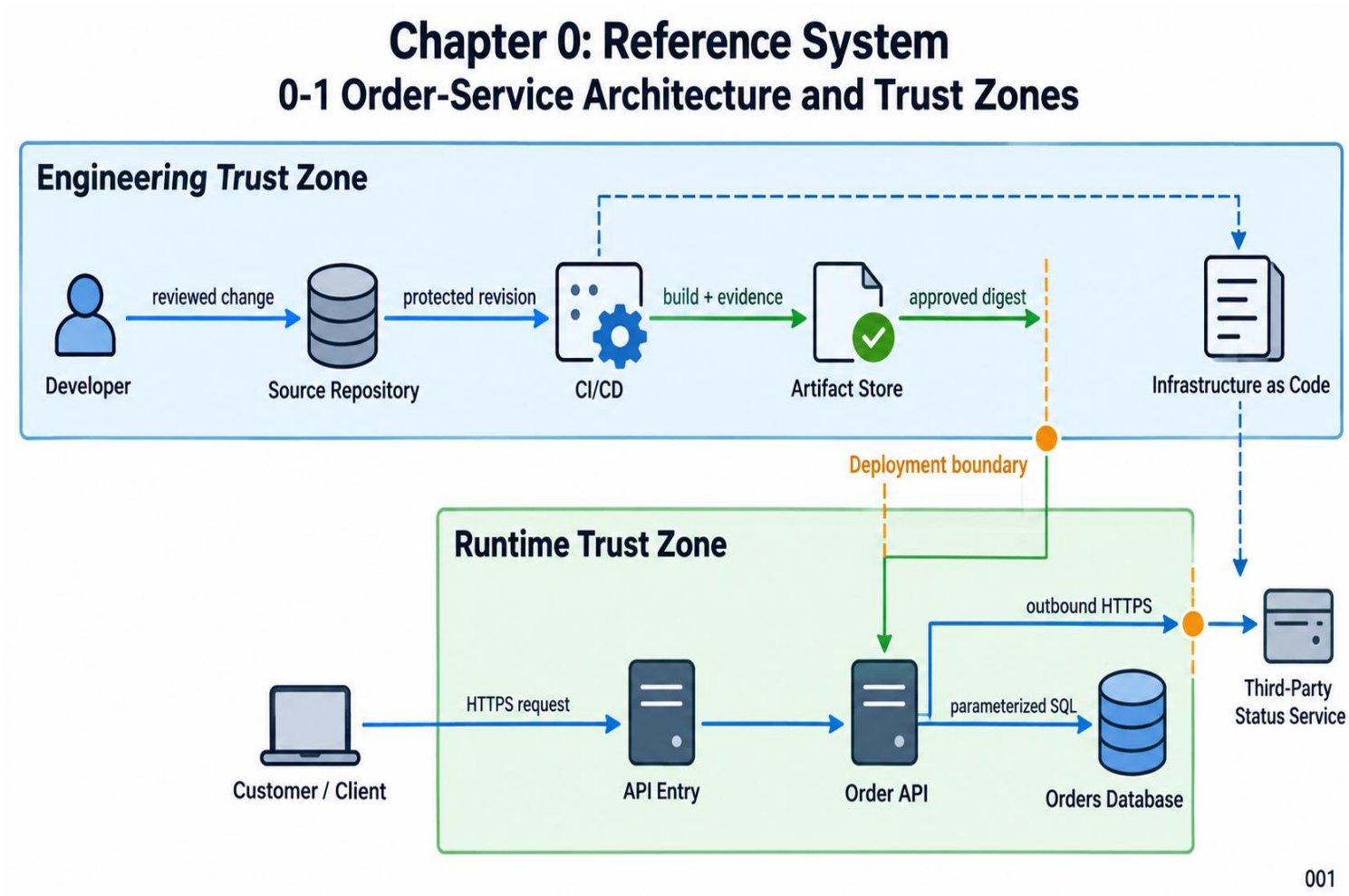

Figure FM-001: Reference order-service architecture and engineering trust zones

This architecture is deliberately small. The goal is to make trust, control placement, and evidence visible without hiding the mechanism behind a large product stack. Product-specific examples are identified as examples; the engineering principle remains applicable to other source platforms, CI systems, cloud providers, operating systems, and runtime models.

Part I — Foundations and Secure Design

Chapter 1 — DevSecOps as a Continuous Risk-Reduction System

What You'll Be Able to Do

By the end of this chapter, you will be able to:

- describe DevSecOps as an engineering feedback system rather than a product category;
- classify controls as preventive, detective, or corrective;
- choose advisory, warning, blocking, or compensating behavior deliberately;
- map a real delivery path and identify trust-bearing components;
- verify whether a claimed control produces observable evidence.

Start with the Outcome

A DevSecOps implementation is successful when it reduces meaningful risk while preserving a delivery flow that engineers can understand, operate, and improve. A pipeline filled with tools can still be insecure if findings arrive too late, privileged automation is poorly protected, exceptions are invisible, or no one can prove what happened during a release.

DevSecOps combines development, security, and operations responsibilities around one delivery system. It does not mean that every engineer becomes a specialist in every discipline. It means that teams share enough context, controls, and evidence to make secure delivery part of ordinary engineering work.

The Lifecycle Is a Set of Feedback Loops

A delivery lifecycle is the path from an idea to a running service and then back to engineering through operational learning. The stages are commonly described as plan, design, code, build, test, release, deploy, operate, respond, and improve.

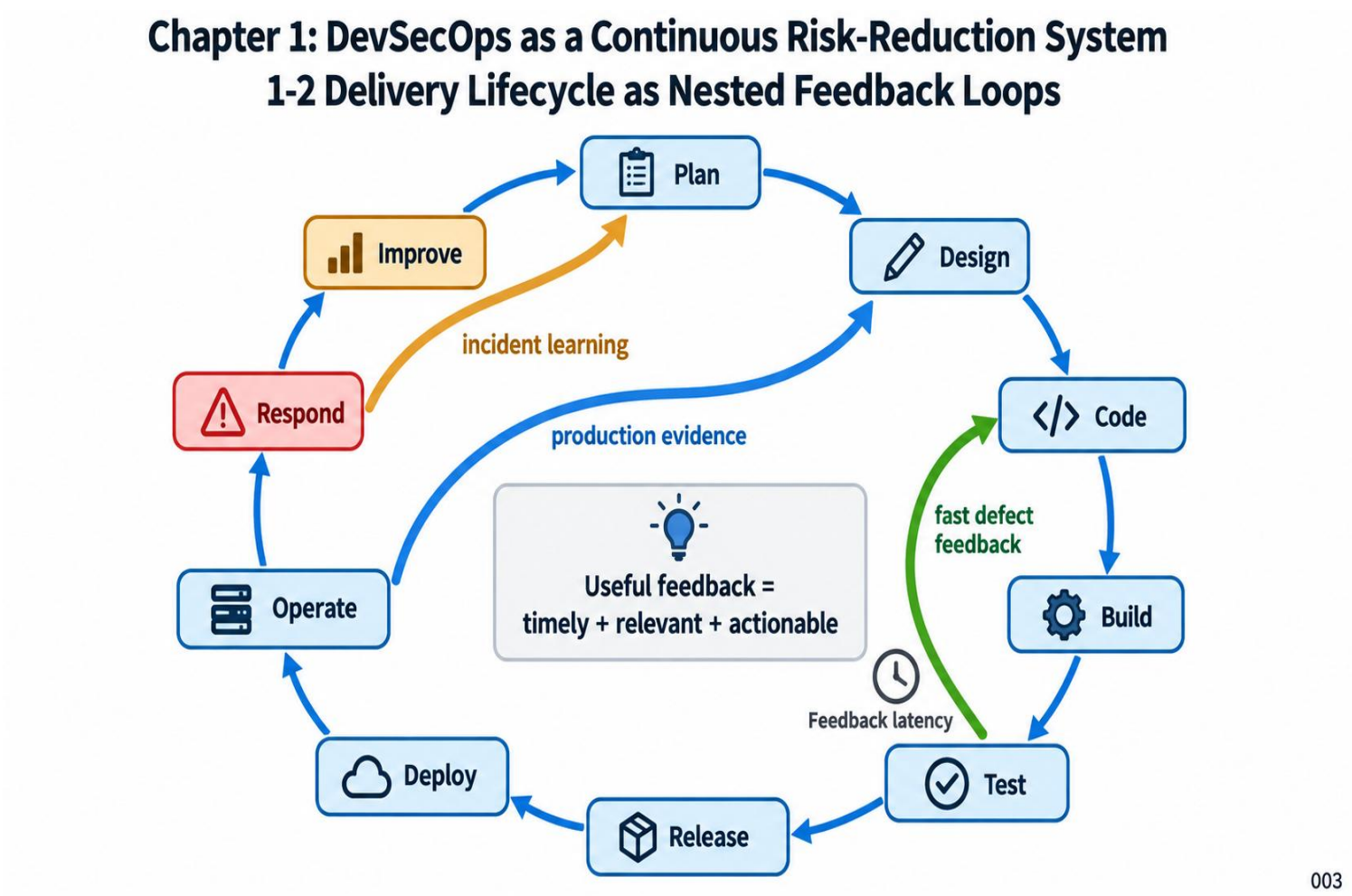

Figure 1-003: Delivery lifecycle as nested feedback loops

The arrows returning to earlier stages are the important part. A security check that reports a defect three weeks after merge creates a long feedback loop. The same defect reported in the editor or pull request is cheaper to understand because the change and its intent are still fresh.

Feedback latency is the elapsed time between introducing a condition and receiving useful information about it. Low latency matters, but speed alone is insufficient. A fast stream of low-confidence warnings creates alert fatigue and teaches engineers to ignore the system. Useful feedback must be timely, relevant, explainable, and connected to an action.

Shift Left and Shift Right

Shift left means moving suitable security feedback earlier in the lifecycle. Examples include checking a policy before infrastructure is provisioned or finding a hard-coded token before a commit leaves a workstation.

Shift right means validating security assumptions in a running environment. Examples include confirming that a workload received only its intended permissions or detecting unexpected outbound traffic.

Neither slogan is a complete strategy:

- Some properties are best prevented early, such as a secret committed to source.
- Some properties can be verified only after deployment, such as effective permissions granted by a cloud control plane.

- Some controls belong in several places. An unsafe infrastructure setting can be checked in source, in a resolved plan, at deployment, and through drift detection.

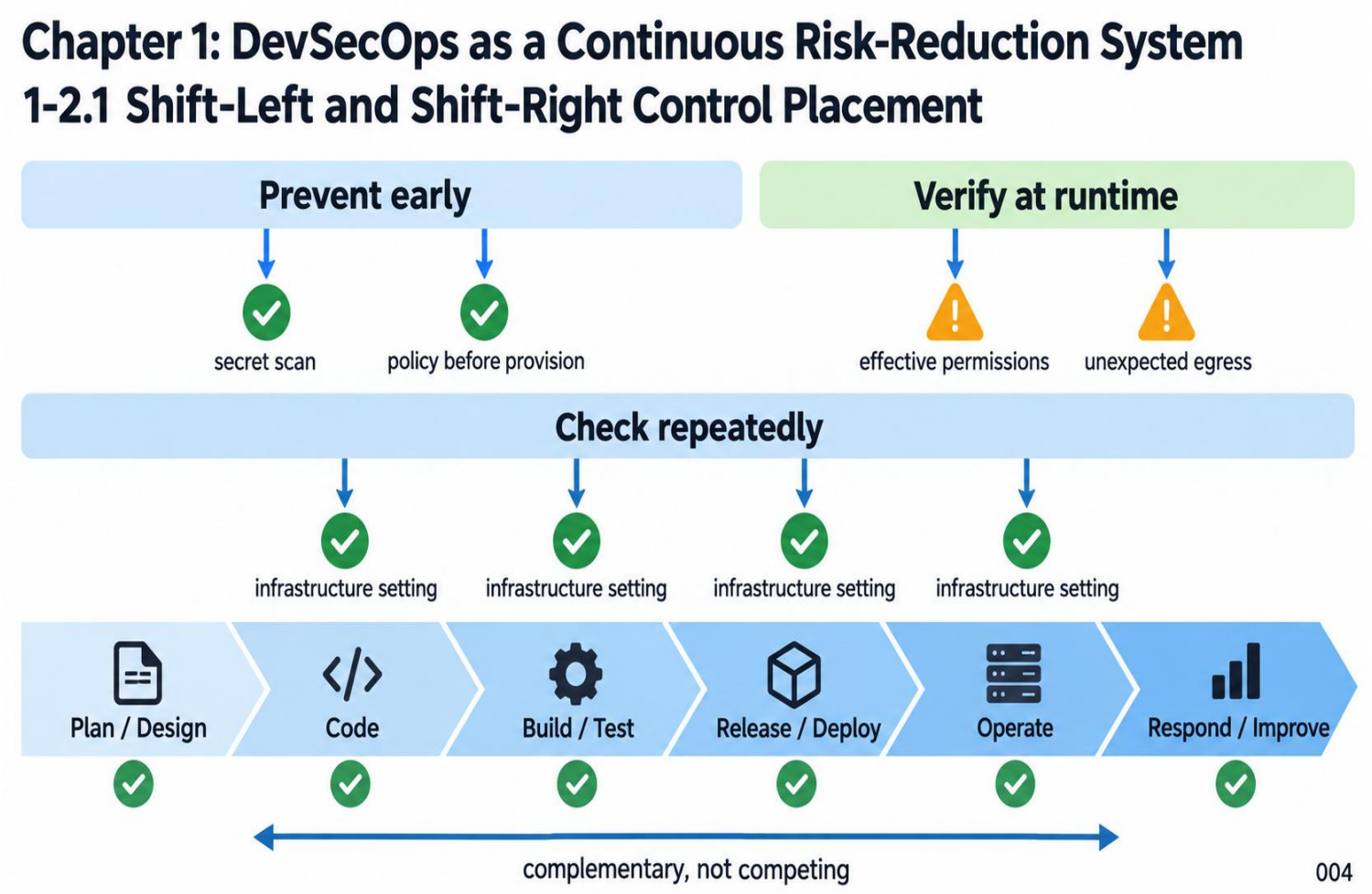

Figure 1-004: Shift-left and shift-right control placement

Think of the lifecycle as a hospital care pathway. Screening before surgery is valuable, monitoring during surgery is still necessary, and post-operative observation catches conditions that neither earlier step could prove absent.

Controls: Prevent, Detect, and Correct

A **security control** is a safeguard that changes the likelihood or impact of an unwanted event. Controls are often grouped by purpose.

Control type	Primary purpose	Example in the reference system	Important limitation
Preventive	Stop an unsafe action or state	Protected branch rejects an unreviewed merge	Can obstruct delivery if the rule is inaccurate or unavailable
Detective	Reveal an event or unsafe state	Secret scanner reports a committed token	Detection does not revoke or contain the token
Corrective	Restore a safer condition	Credential rotation and repository history cleanup	Recovery may be incomplete without verification

Strong systems use several independent barriers. A preventive control may fail because of a defect or emergency bypass. A detective control should expose that failure, and a corrective process should limit the resulting harm.

Automation and Human Judgment

Automation is well suited to decisions with clear, repeatable inputs:

- whether a file matches a prohibited secret pattern;

- whether a pipeline token requests write permission;
- whether an infrastructure plan creates a public storage resource;
- whether a dependency version appears in an approved advisory feed.

Human review remains essential where context determines correctness:

- whether an administrator may approve their own high-risk change;
- whether two application roles are allowed to view the same order;
- whether temporary public access is justified during an incident;
- whether a high-severity vulnerability is reachable in this service.

The goal is not to remove human decisions. It is to reserve human attention for decisions that actually require judgment.

Choosing How a Check Behaves

Not every check should block delivery immediately. A check can operate in several modes.

Mode	Effect	Best use	Failure risk
Advisory	Produces information without changing flow	New or exploratory analysis	Findings may never be acted on
Warning	Makes risk visible and records acknowledgement	Known issue with transitional tolerance	Repeated warnings can become background noise
Blocking gate	Stops the action until a condition is resolved or explicitly excepted	High-confidence, high-impact, cheaply fixable conditions	False positives or tool outages can halt delivery
Compensating control	Uses another safeguard when the preferred control is unavailable	Temporary migration, emergency repair, legacy constraint	Temporary measures can become permanent and weaker

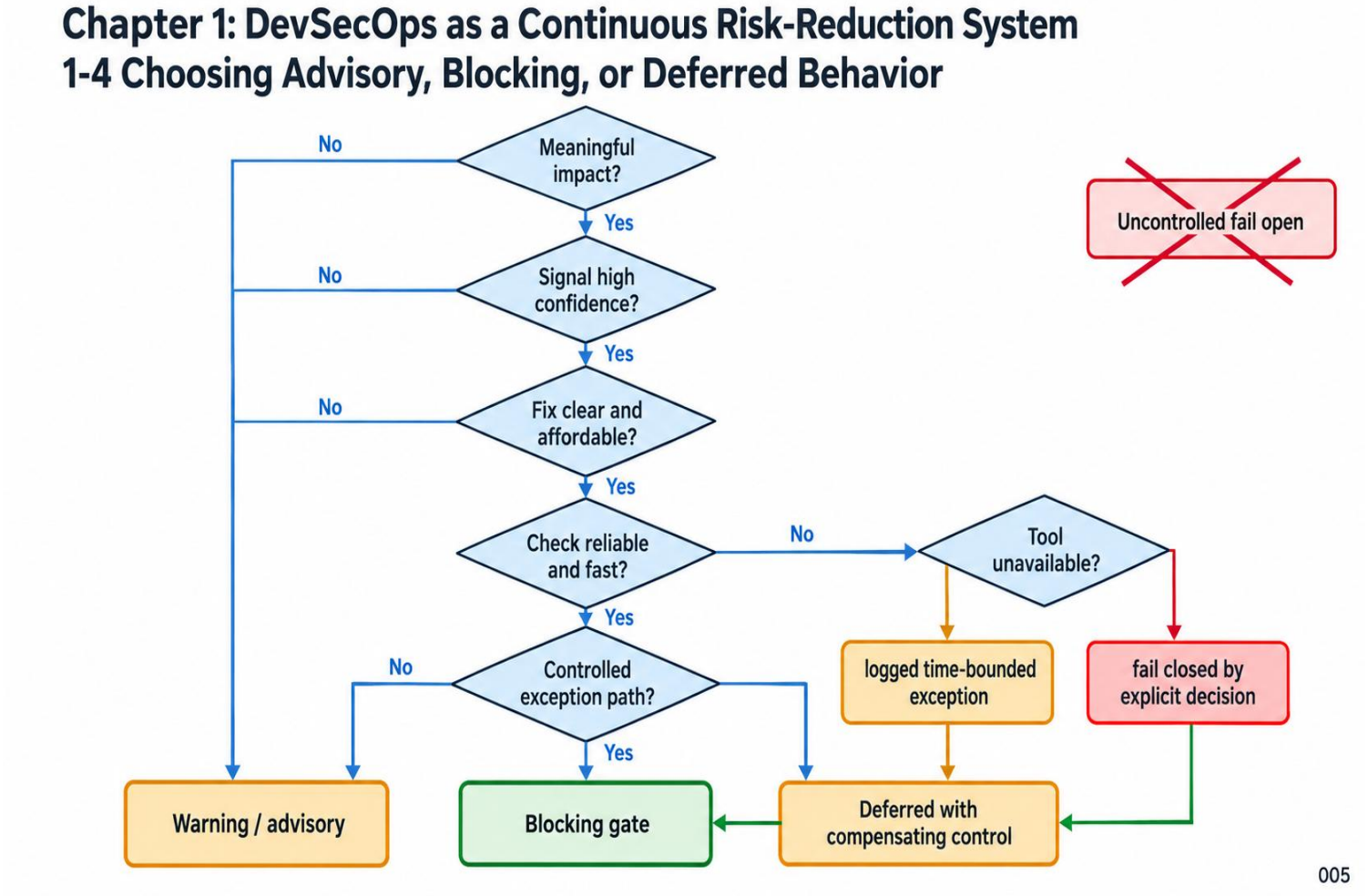

Figure 1-005: Choosing advisory, blocking, or deferred check behavior

A blocking gate is justified when all of the following are substantially true:

- The condition has meaningful security or reliability impact.
- The signal is accurate enough to avoid routine false blocking.
- The team can understand and fix the condition.
- The check completes within an acceptable feedback time.
- A controlled exception path exists for exceptional circumstances.
- Tool failure behavior has been chosen deliberately rather than inherited accidentally.

For example, a confirmed private key in a new pull request is usually appropriate to block. A speculative data-flow warning in generated code may begin as advisory until the rule is tuned.

The Delivery System Is Part of the Attack Surface

An **attack surface** is the set of reachable paths through which an attacker can influence a system or obtain a protected capability. The delivery system often has more privilege than the application itself because it can create artifacts, change infrastructure, retrieve secrets, and deploy to production.

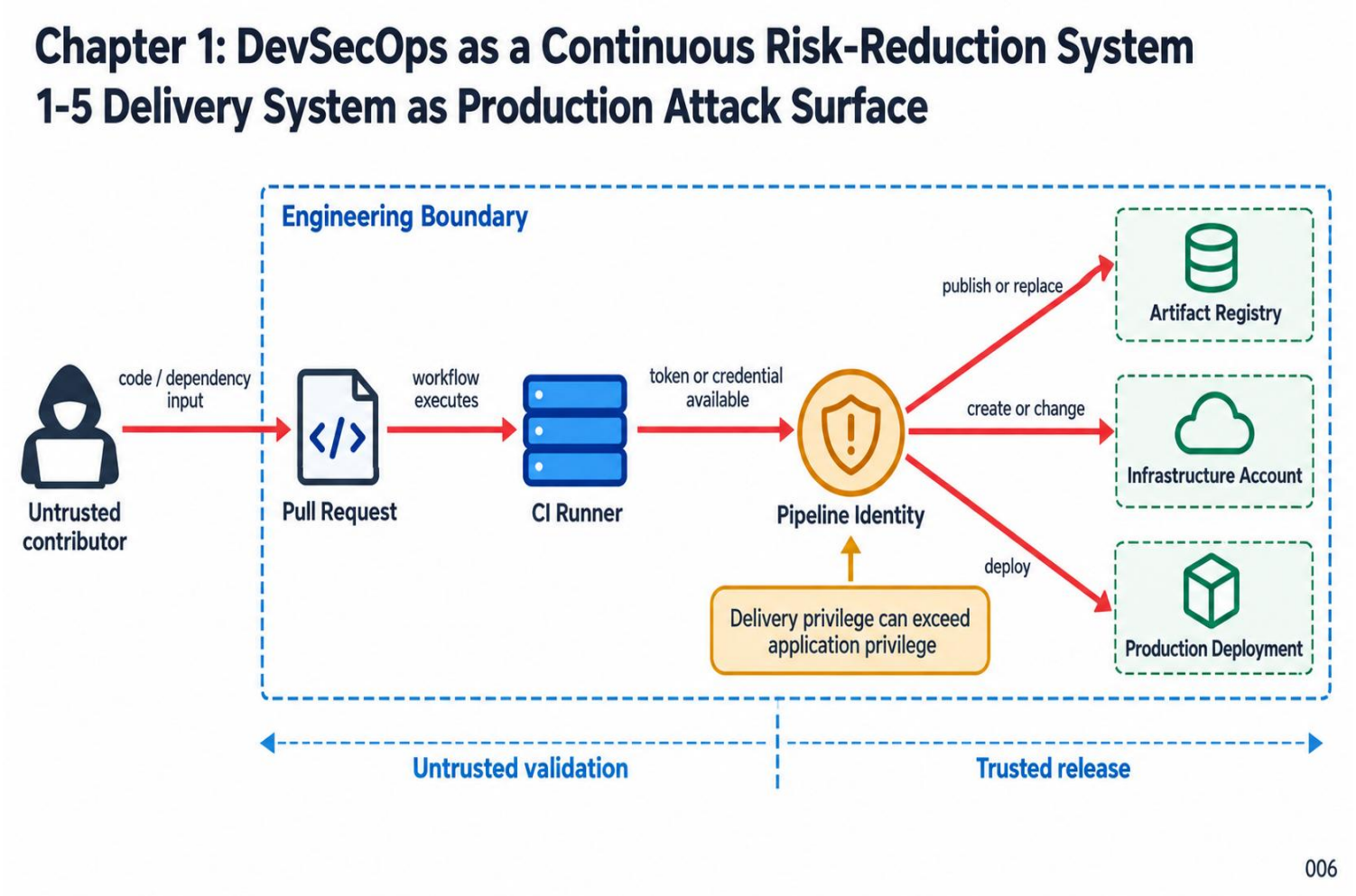

Figure 1-006: Delivery system as part of the production attack surface

If untrusted pull-request code executes on a persistent runner that also performs production deployments, an application-level code contribution can become an infrastructure compromise. The delivery architecture must therefore separate trust levels, minimize credentials, and preserve evidence in the same way as other privileged systems.

A Practical Mapping Exercise

The first useful DevSecOps activity is usually not installing a tool. It is drawing the actual delivery path and identifying where decisions occur.

Prerequisites

You need representatives who understand source control, CI/CD, infrastructure, application delivery, and operations. Use a real service and a recent release rather than an idealized process document.

Draw the Real Path

Record the sequence from change creation to production. Include manual actions, scripts run from laptops, shared accounts, external package stores, deployment approvals, and emergency paths.

A minimal path might be:

```
Developer workstation
    -> source repository
    -> pull-request checks
    -> merge
    -> release pipeline
    -> package or image registry
    -> deployment mechanism
    -> runtime environment
```

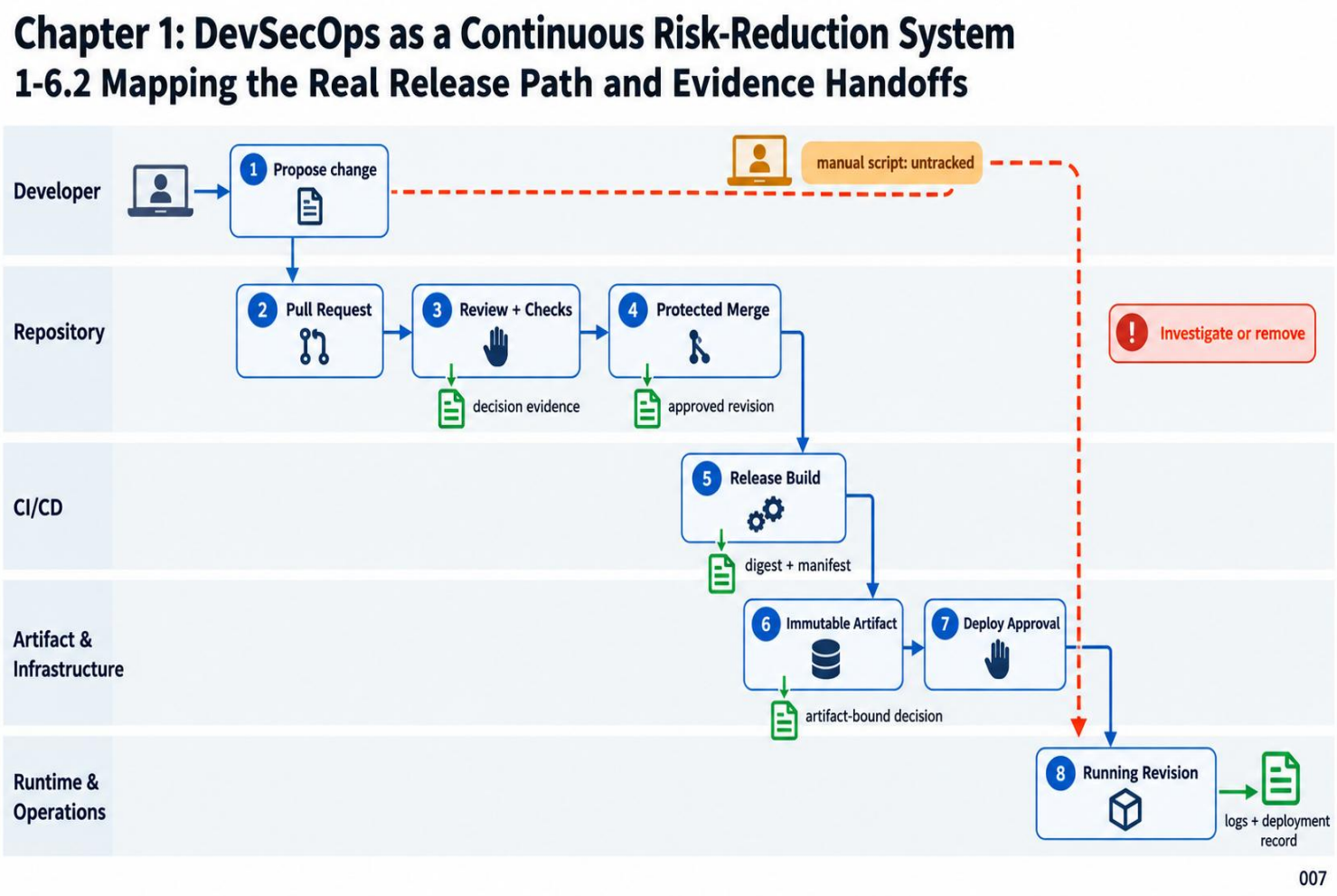

Figure 1-007: Mapping the real release path and evidence handoffs

Record Each Existing Control

For every claimed control, capture the following fields:

Field	Question to answer
Input	What evidence or state is evaluated?
Decision	What rule is applied?
Owner	Who maintains the rule and handles failure?
Output	What result or evidence is produced?
Enforcement	Does it advise, warn, block, or trigger correction?
Failure behavior	What happens when the tool, dependency, or identity is unavailable?

Example control record:

```
control: required pull-request review
input: proposed change to protected branch
rule: at least one reviewer other than the author; code owner required for /infra
owner: platform engineering
output: repository review and merge audit events
enforcement: block merge
failure_behavior: repository administrators may use emergency bypass; bypass is logged and reviewed next business day
```

Find One Risk Gap and One Friction Point

A **risk gap** is an important failure path with no adequate control. A **friction point** is a control whose cost causes delay, bypass, or avoidance.

For the reference service:

- Risk gap: the release job uses a broad, long-lived cloud key stored in CI.
- Friction point: the full dependency scan takes 25 minutes on every documentation-only change.

A better feedback design would replace the key with a short-lived workload identity and use change-aware scanning for pull requests while retaining a scheduled full scan.

Verify the Map with a Real Release

Select one deployed revision and follow its evidence backward:

- Which commit produced it?
- Which workflow and runner executed?
- Which checks passed, failed, or were bypassed?
- Which artifact was created?
- Which identity deployed it?
- Can an operator distinguish the expected artifact from a substituted one?

If the team cannot answer these questions, the control system is not yet observable enough to be trusted.

A First Maturity Baseline

Tool ownership is a poor maturity measure. Observable practices are more useful.

Level	Observable condition
0 — Implicit	Security depends on individual memory; releases cannot be reconstructed reliably
1 — Visible	Delivery paths, owners, critical assets, and major risks are documented
2 — Repeatable	Core controls run consistently and produce reviewable evidence
3 — Enforced	High-confidence critical rules block unsafe actions with controlled exceptions
4 — Adaptive	Runtime and incident evidence changes design, tests, and policy; control performance is measured

Maturity is not uniform across an organization. A team may have advanced dependency automation but weak identity governance. Assess each capability separately and improve the riskiest gaps first.

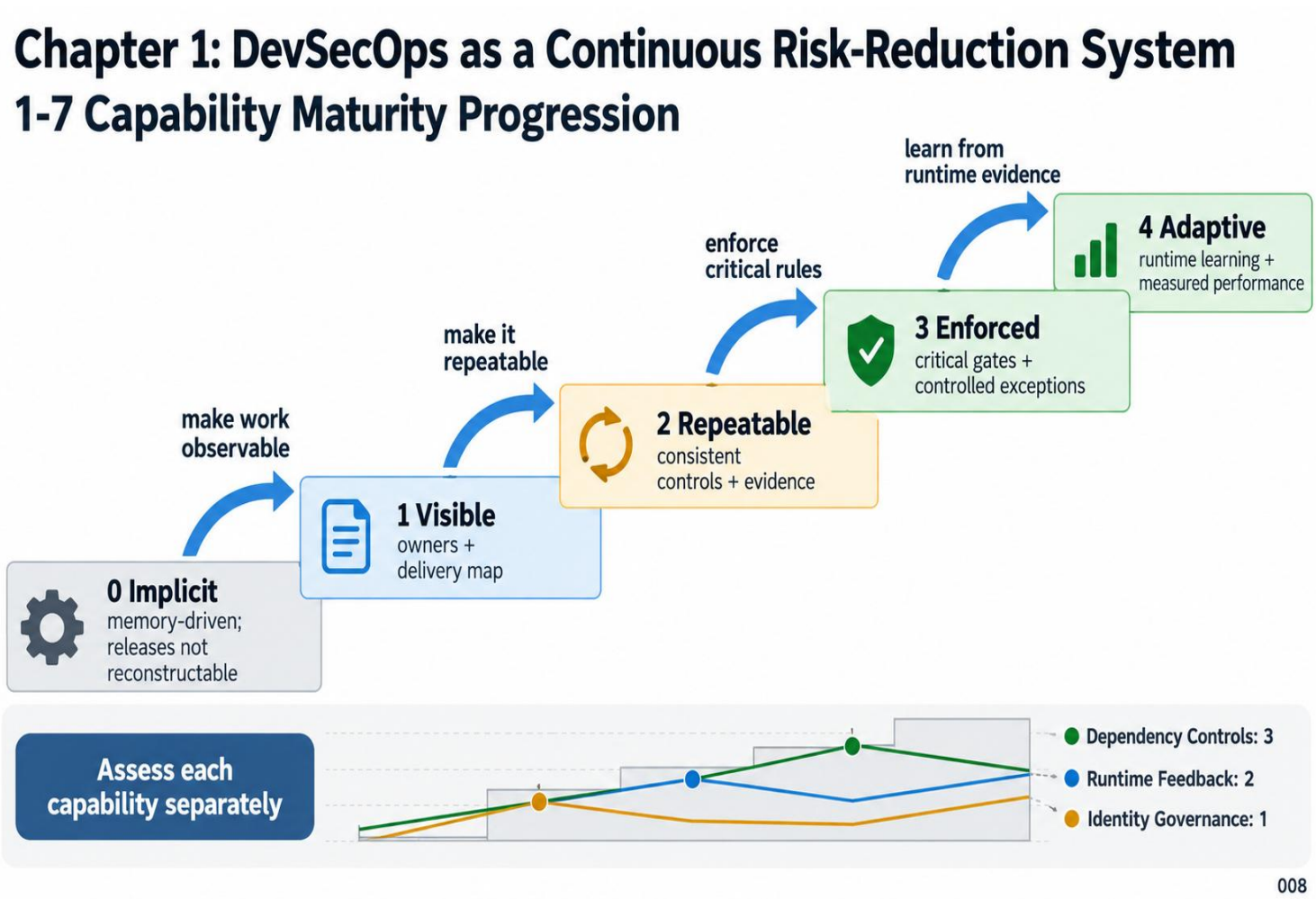

Figure 1-008: First maturity baseline as capability progression

Diagnosing Weak Controls

A control exists in policy but not in releases

Check whether the control has an actual enforcement point. A document saying “all changes require review” is not evidence that branch protection is configured or that administrators cannot bypass it silently.

A tool outage stops all delivery

Determine whether blocking was an explicit risk decision. For some controls, fail-closed behavior is appropriate. For others, a time-bounded, logged exception with compensating review is safer than an uncontrolled operational outage.

Findings are numerous but remediation is low

Inspect signal quality, ownership, and fix guidance. A finding without a responsible owner and service-level expectation is inventory, not risk reduction.

Engineers routinely bypass a gate

Treat repeated bypass as evidence that either the policy, implementation, or operating model is defective. Do not assume the users are the only problem. Measure false positives, elapsed time, exception causes, and whether the check evaluates the correct artifact or context.

Common Misconceptions

“DevSecOps means adding scanners to CI.” Scanners are sensors. They do not define risk, protect pipeline identities, enforce architecture, or ensure remediation.

“Every security finding should block.” Broad blocking creates denial-of-service risk against delivery and encourages suppression. Block high-confidence conditions whose expected loss exceeds the cost of interruption.

“The security team owns security.” Security specialists provide expertise and shared capabilities, but the team that designs, changes, and operates a service retains responsibility for its risks.

“More scans mean more security.” Useful measures include exposure reduced, time to remediate, prevented recurrence, exception age, credential lifetime, and the percentage of releases with traceable evidence.

Chapter Checkpoint

At this point, a systems engineer should be able to:

- describe DevSecOps as an engineering feedback system rather than a product category;
- classify controls as preventive, detective, or corrective;
- choose advisory, warning, blocking, or compensating behavior deliberately;
- map a real delivery path and identify trust-bearing components;
- verify whether a claimed control produces observable evidence.

Review Questions

1. Why can a delivery pipeline contain many security tools and still fail to reduce meaningful risk?
2. Which factors should determine whether a security check is advisory, blocking, deferred, or paired with a compensating control?
3. What evidence would prove that a claimed control actually evaluated a specific release?

Chapter 2 — Security, Trust, and Risk Foundations for Systems Engineers

What You'll Be Able to Do

By the end of this chapter, you will be able to identify assets and trust boundaries, distinguish vulnerability severity from system risk, record accountable treatment decisions, and test whether trust assumptions remain valid during failure and emergency operation.

Security Decisions Begin with Consequences

A control is useful only when it protects something that matters against a credible failure path. Before selecting tools or writing policies, identify the assets, trust assumptions, exposures, and business consequences that make a condition risky.

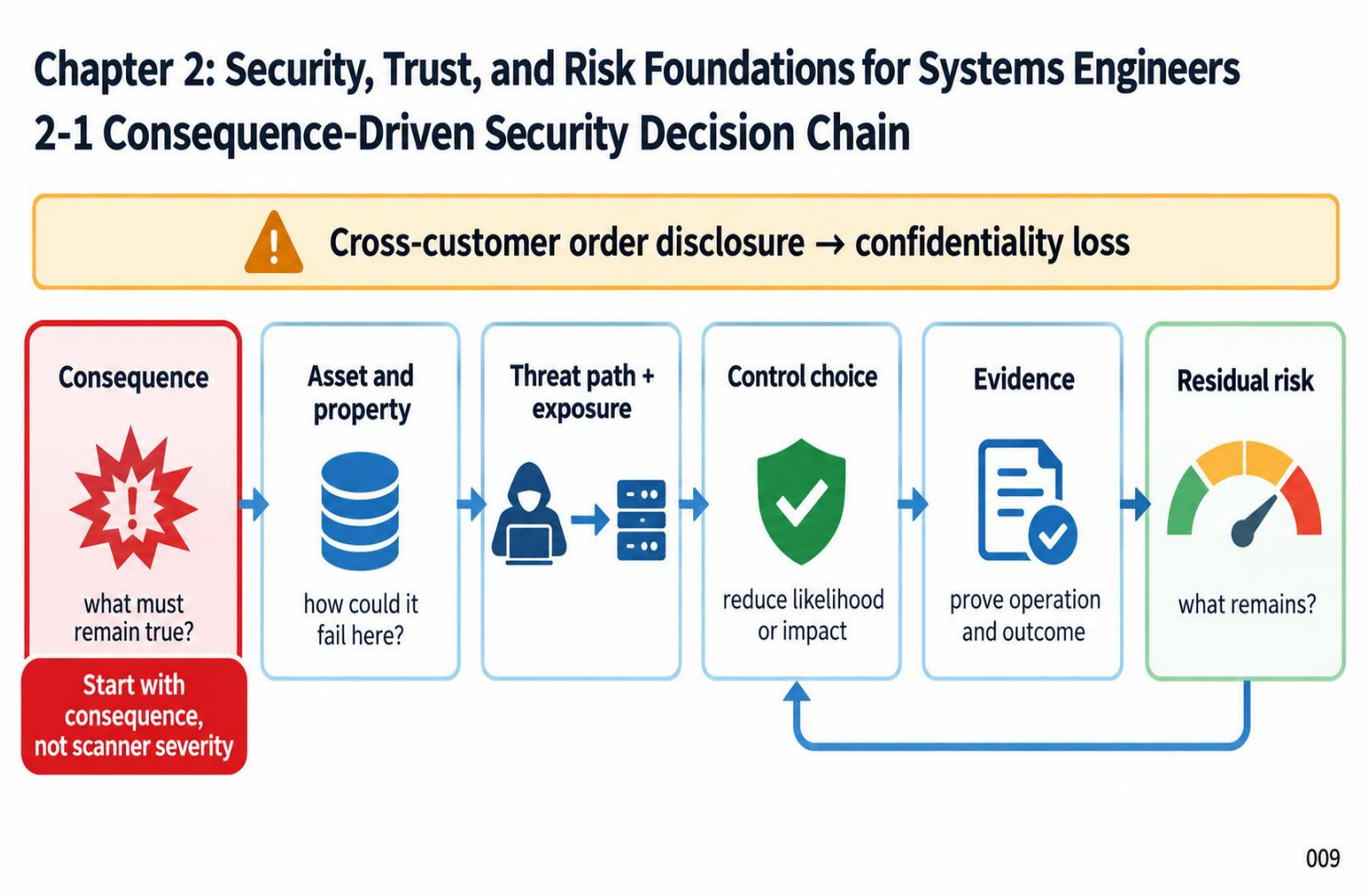

Figure 2-009: Consequence-driven security decision chain

The central vocabulary is straightforward when each term is tied to a concrete example:

- An **asset** is something with value that must be protected, such as order records, deployment credentials, source code, or service availability.
- A **threat** is a circumstance or actor capable of causing harm, such as credential theft or a malicious dependency.
- A **vulnerability** is a weakness that can be exploited or triggered, such as an overly broad role or an injectable query.
- **Exposure** describes how reachable or applicable the weakness is in the real system.
- **Likelihood** is a reasoned judgment about how probable the harmful event is within a stated period and context.
- **Impact** is the consequence if the event occurs, including technical, operational, financial, legal, and customer effects.
- A **control** is a safeguard that changes likelihood, impact, or both.
- **Residual risk** is the risk that remains after existing and planned controls are considered.

These terms prevent a common category error: a severe software defect is not automatically the organization's highest risk. A critical vulnerability in an isolated test tool may be less urgent than a moderate authorization defect in an internet-facing order API with sensitive customer data.

What Security Must Preserve

The familiar confidentiality, integrity, and availability model remains useful, but practical DevSecOps work needs several related properties.

Property	Meaning	Reference-system example
Confidentiality	Information is disclosed only to authorized parties	One customer cannot read another customer's order
Integrity	Information and behavior cannot be changed without authorization	A pipeline cannot silently substitute a different artifact
Availability	Required capabilities remain usable within agreed conditions	The order API remains responsive during dependency slowdown
Authenticity	An identity, message, or artifact is what it claims to be	A workload token belongs to the expected repository and branch
Accountability	Actions can be attributed and reviewed	Administrative changes are tied to individual identities and audit events
Resilience	The system can withstand, recover from, and learn from disruption	Credential compromise is contained, rotated, and followed by control improvement

Security design often involves tension among these properties. A restrictive control that prevents emergency recovery can reduce availability and resilience. An emergency bypass that restores service but is shared and unaudited undermines accountability. The objective is not to maximize one property in isolation; it is to make deliberate trade-offs based on service needs.

Attack Surface, Trust Boundaries, Privilege, and Blast Radius

An **attack surface** is the collection of interfaces and paths through which a system can be influenced. It includes public APIs, administrator consoles, package sources, CI event handlers, identity providers, maintenance channels, and human processes.

A **trust boundary** is a point where data or execution crosses between zones with different identity, ownership, privilege, or assurance. Crossing a trust boundary should trigger explicit validation and authorization.

Privilege is the capability to perform an action on a protected resource. A privilege should be granted to a specific identity, for a defined purpose, within an appropriate scope and duration.

Blast radius is the maximum practical harm that can result from one failure or compromise. Reducing blast radius is often more achievable than assuming every prevention mechanism will work perfectly.

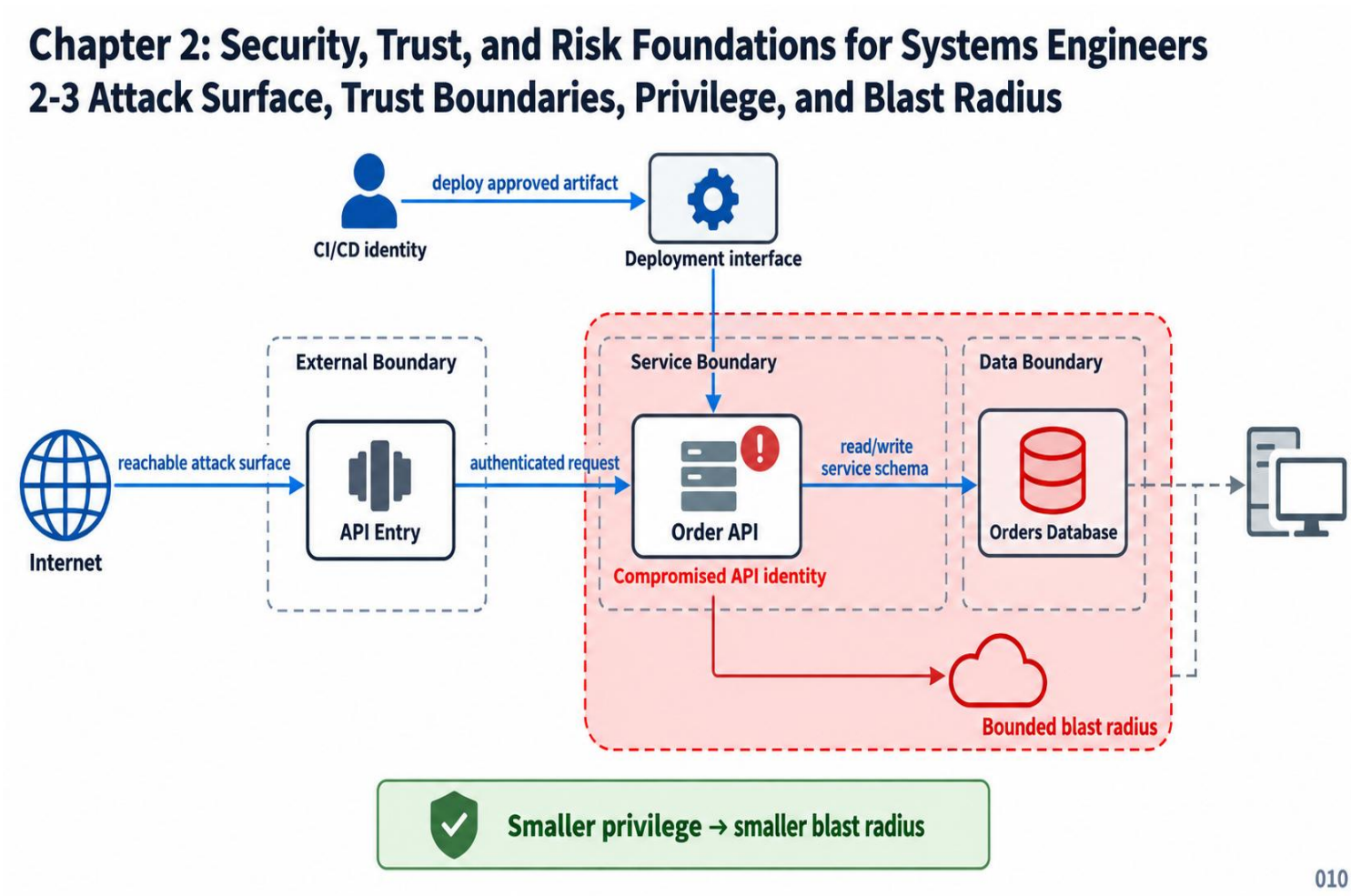

Figure 2-010: Attack surface, trust boundaries, privilege, and blast radius

The diagram is less important than the questions it forces:

- What identity crosses each boundary?
- How is that identity authenticated?
- Which action is authorized after authentication?
- Which data is accepted, transformed, logged, or returned?
- What happens if the upstream party is unavailable or compromised?
- How far can a compromised identity move?

Design Principles That Reduce Dependence on Perfect Detection

Least Privilege

Least privilege means granting only the minimum permissions needed for a task, for no longer than needed. It applies to users, service accounts, pipeline jobs, containers, databases, network paths, and administrative tools.

A release job that only uploads an artifact should not also be able to delete the registry, change identity policies, or read production database credentials. Splitting those capabilities reduces the blast radius of a compromised job.

Defense in Depth

Defense in depth uses multiple controls whose failure modes are not identical. A protected branch, code review, restricted pipeline token, immutable artifact identity, and deployment policy each address a different point in the change path.

The everyday analogy is a building with a perimeter fence, locked entrance, room-level access control, alarms, and incident procedures. Adding five identical padlocks to one door is not equivalent because a single copied key defeats all five.

Secure Defaults

A **secure default** is the safest practical behavior when a user or system has not made an explicit choice. New repositories should not permit arbitrary force-pushes. New storage resources should not be public. New pipeline jobs should start with no write permissions.

Defaults matter because exceptional configuration receives attention, while inherited configuration often does not.

Fail-Safe Behavior

Fail-safe behavior means failure moves the system toward a controlled condition rather than an unexpectedly permissive one. It does not mean that every dependency outage must stop every service.

For an authorization decision, failure to validate a token should normally deny the request. For a noncritical advisory scan, an outage may permit delivery while generating an alert and a required follow-up. The correct behavior depends on the control objective and operational consequence.

Separation of Duties

Separation of duties divides critical actions so that one compromised or malicious actor cannot complete an entire harmful sequence alone. Examples include requiring another reviewer for protected changes or separating routine deployment rights from identity-administration rights.

Separation should be risk-based. Requiring three approvals for every low-risk documentation change wastes attention and encourages rubber-stamping.

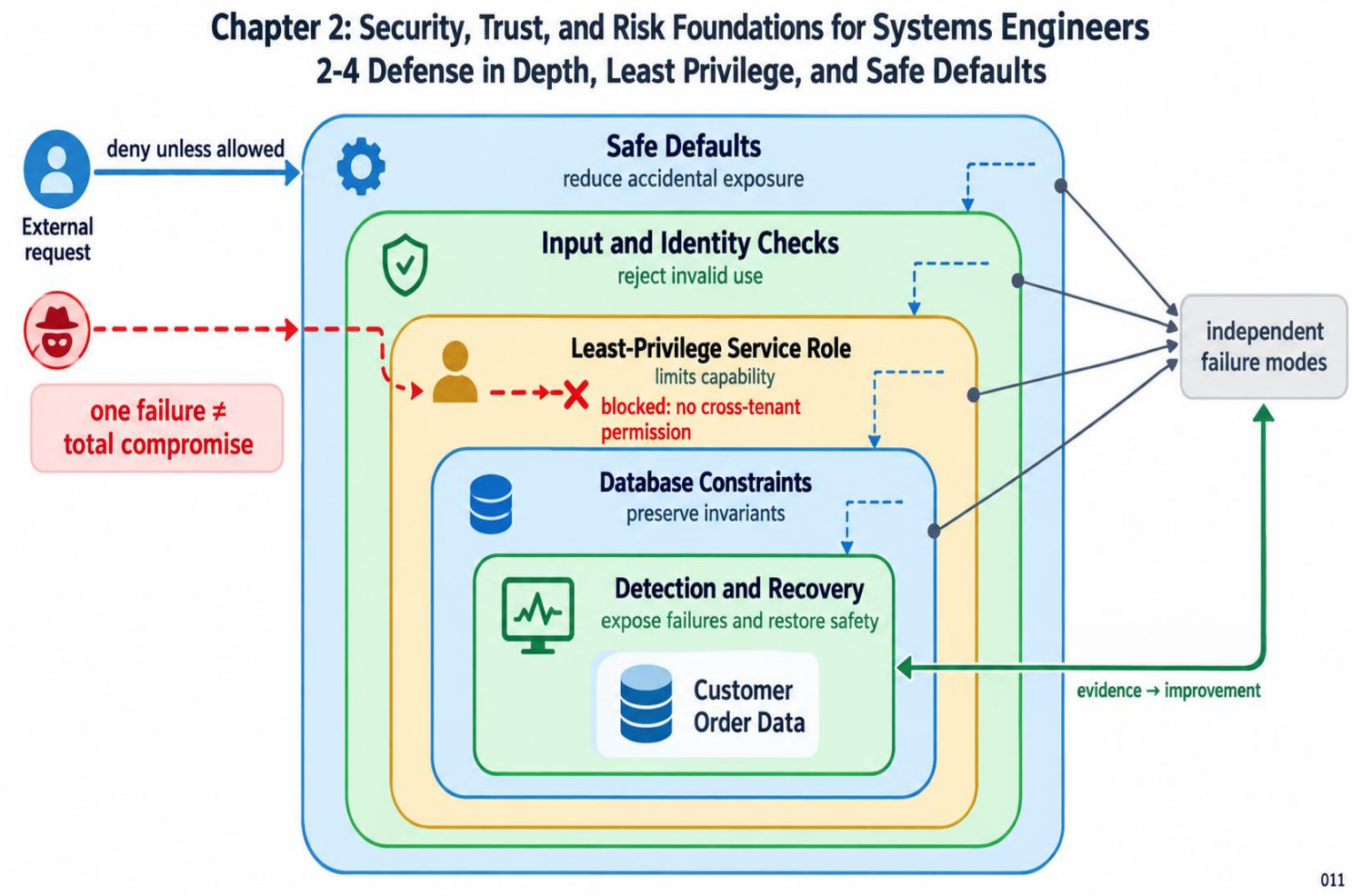

Figure 2-011: Defense in depth built on least privilege and safe defaults

Zero Trust Without the Marketing Layer

Zero trust is an operating model in which access is continuously based on verified identity, context, policy, and resource sensitivity rather than assumed from network location. It is not a single product, and it does not require abandoning all network controls.

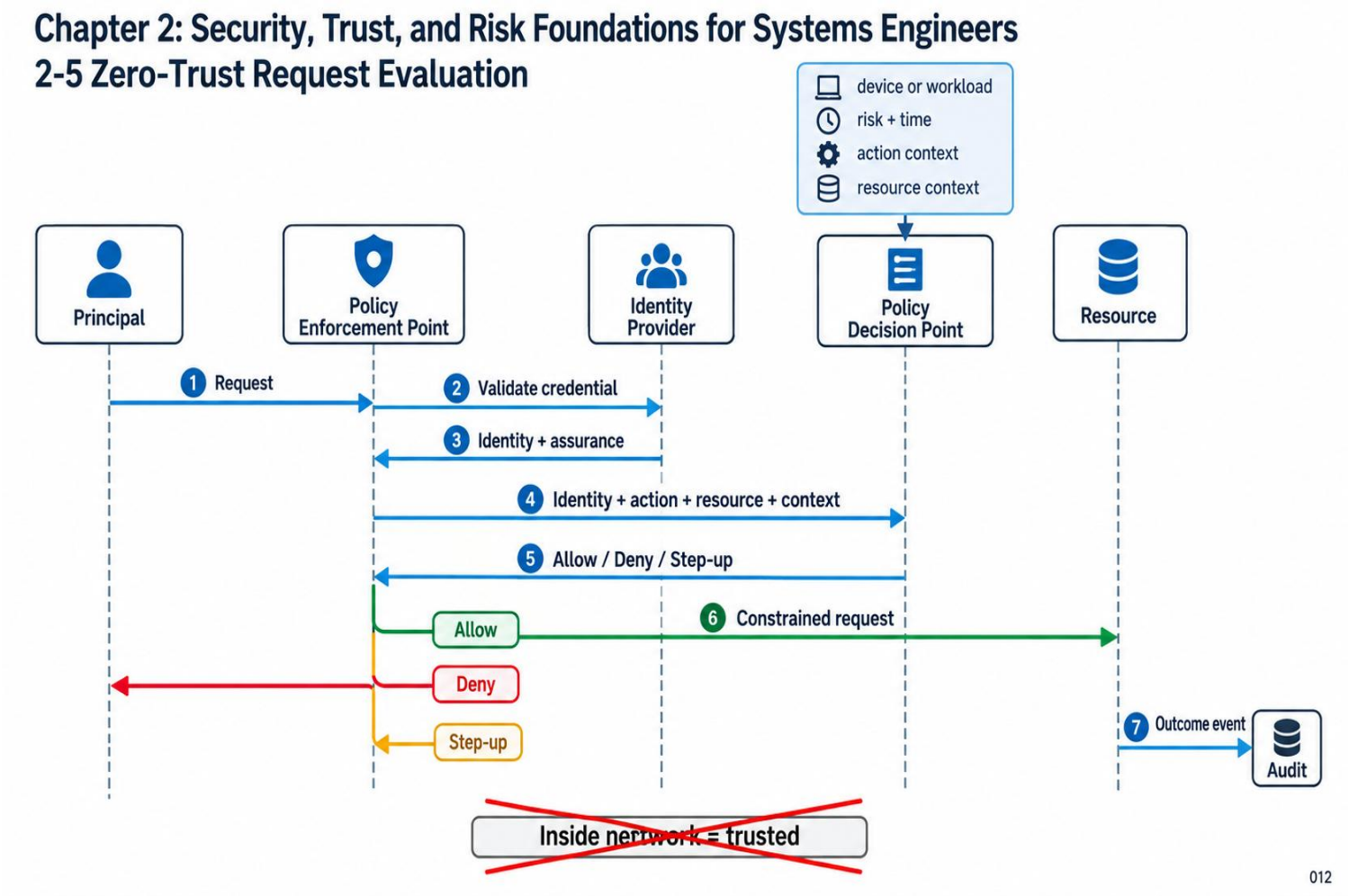

Figure 2-012: Zero-trust request evaluation

For the order service, a zero-trust-oriented design would avoid statements such as “anything inside the cluster is trusted.” Instead:

- each workload has a distinct identity;
- requests are authenticated across service boundaries;
- authorization is specific to the target resource and action;
- credentials are short-lived where practical;
- network location is used as an additional constraint, not proof of identity;
- access and policy decisions are observable.

The practical test is simple: if an attacker reaches an internal network segment, what still prevents them from impersonating a workload or reading unrelated data?

Classification Drives Proportionate Controls

Not every service needs identical controls. Classification makes differences explicit.

Data Classification

A practical scheme might contain four classes:

Class	Typical content	Example handling expectation
Public	Intended for unrestricted distribution	Integrity remains important; confidentiality controls are minimal
Internal	Routine nonpublic operational information	Authenticated access and controlled sharing
Confidential	Customer, commercial, or security-sensitive data	Strong access control, encryption, retention limits, auditable use
Restricted	Data whose compromise creates severe harm	Strict need-to-know access, enhanced monitoring, dedicated approval and recovery requirements

The labels themselves are less important than the handling rules. A classification program that does not change access, retention, logging, or incident response is only a naming exercise.

Service Criticality

Criticality describes the consequence of service failure or compromise. Useful factors include customer dependency, recovery time, safety impact, regulatory obligations, transaction value, and number of downstream systems.

For example:

- A disposable development preview may be low criticality even if it uses production-like technology.
- The public order API may be high criticality because it accepts customer transactions and drives fulfillment.
- The CI identity provider may be critical even though customers never access it directly, because compromise could affect every service.

Environment Classification

Development, test, staging, and production are not inherently risk levels. A development environment containing copied production data can be more sensitive than an empty production shell. Classify the actual data, connectivity, identities, and business effect rather than relying on the environment name.

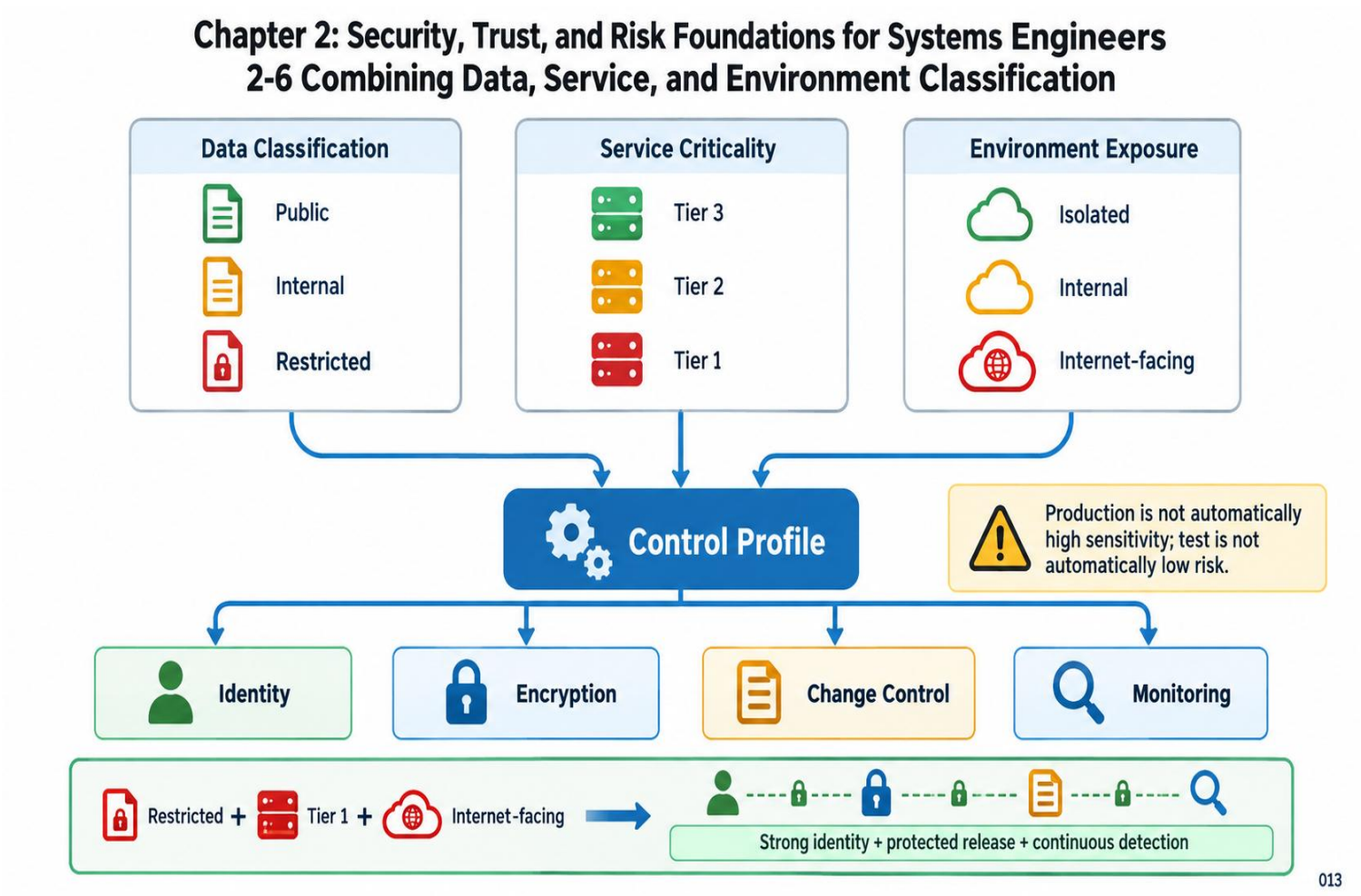

Figure 2-013: How data, service, and environment classifications combine

Threat Actors Relevant to Delivery Systems

A useful threat model includes more than an anonymous internet attacker.

Threat actor or condition	Typical path	Useful design response
External attacker	Public API, stolen credential, exposed service	Strong authentication, input handling, rate control, monitoring
Compromised dependency	Package update, build plugin, container base	Controlled acquisition, review, inventory, restricted build privileges
Malicious insider	Legitimate access used outside intent	Least privilege, separation of duties, individual accountability
Careless authorized user	Accidental public exposure or secret commit	Secure defaults, validation, guardrails, recovery procedures
Compromised automation	Stolen runner token or modified workflow	Short-lived identity, isolation, protected pipeline code, narrow permissions
Supplier failure	Service outage, breach, or unexpected change	Contractual expectations, fallback design, egress limits, incident coordination

This is not a statement that every insider is suspicious or every supplier is unsafe. It is recognition that systems must remain controlled when trusted components behave unexpectedly.

Building the Reference-System Asset Inventory

An asset inventory should be actionable rather than exhaustive. Record enough information to connect ownership, sensitivity, and controls.

Asset	Value and consequence	Owner	Classification	Primary trust dependency
Order records	Customer commitments and transaction history	Order product owner	Confidential	Database identity and authorization
Order API availability	Revenue and customer experience	Service owner	High criticality	Runtime platform and dependencies
Source repository	Defines intended application and pipeline behavior	Engineering manager	Confidential	Repository identities and branch controls
CI release identity	Can publish artifacts and request deployment	Platform owner	Restricted capability	CI issuer, role policy, workflow protection
Infrastructure state	Reveals topology and may contain sensitive values	Cloud platform owner	Confidential or restricted	Backend access and encryption
Third-party service token	Permits external API use and may incur cost	Integration owner	Confidential credential	Secret store and workload identity

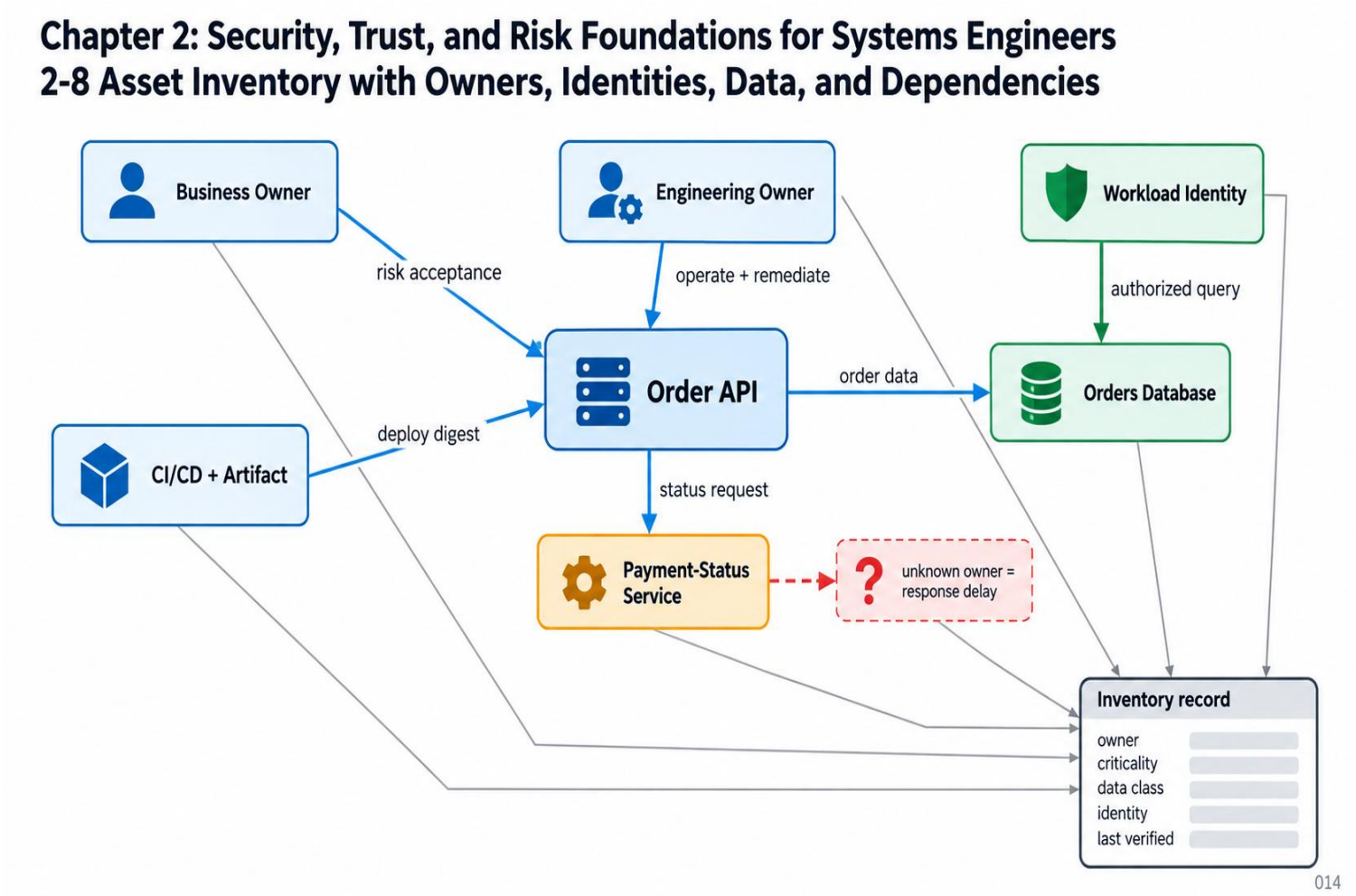

Figure 2-014: Asset inventory overlaid with owners, identities, data, and dependencies

Verification Questions

For every high-value asset, confirm:

- A named role owns risk decisions; “the team” is not sufficiently precise for escalation.
- The classification has actual handling requirements.
- At least one control is observable.
- Recovery is possible and has a responsible owner.
- Dependencies and privileged identities are known.

Creating a Risk Register That Supports Decisions

A **risk register** is a maintained record of identified risks, ownership, treatment, and residual exposure. It should not become a graveyard of vague statements.

A useful risk statement follows this pattern:

> *Because **[condition or weakness]**, **[threat event]** could affect **[asset]**, causing **[business and technical impact]**.*

Weak statement:

> *CI security risk is high.*

Useful statement:

> *Because the release workflow uses a non-expiring cloud key with broad storage and deployment permissions, theft through a compromised workflow could publish a substituted artifact and alter the production service, causing loss of integrity and customer trust.*

A practical register might use the following fields:

Field	Purpose
Risk statement	Describes cause, event, asset, and consequence
Owner	Accepts accountability for the treatment decision
Evidence	Architecture, incident, test, or finding supporting the assessment
Existing controls	Safeguards already operating
Treatment	Avoid, reduce, transfer, or accept
Planned action	Specific change, responsible party, and due date
Residual risk	Remaining exposure after treatment
Review trigger	Date or event that requires reassessment

Four Treatment Choices

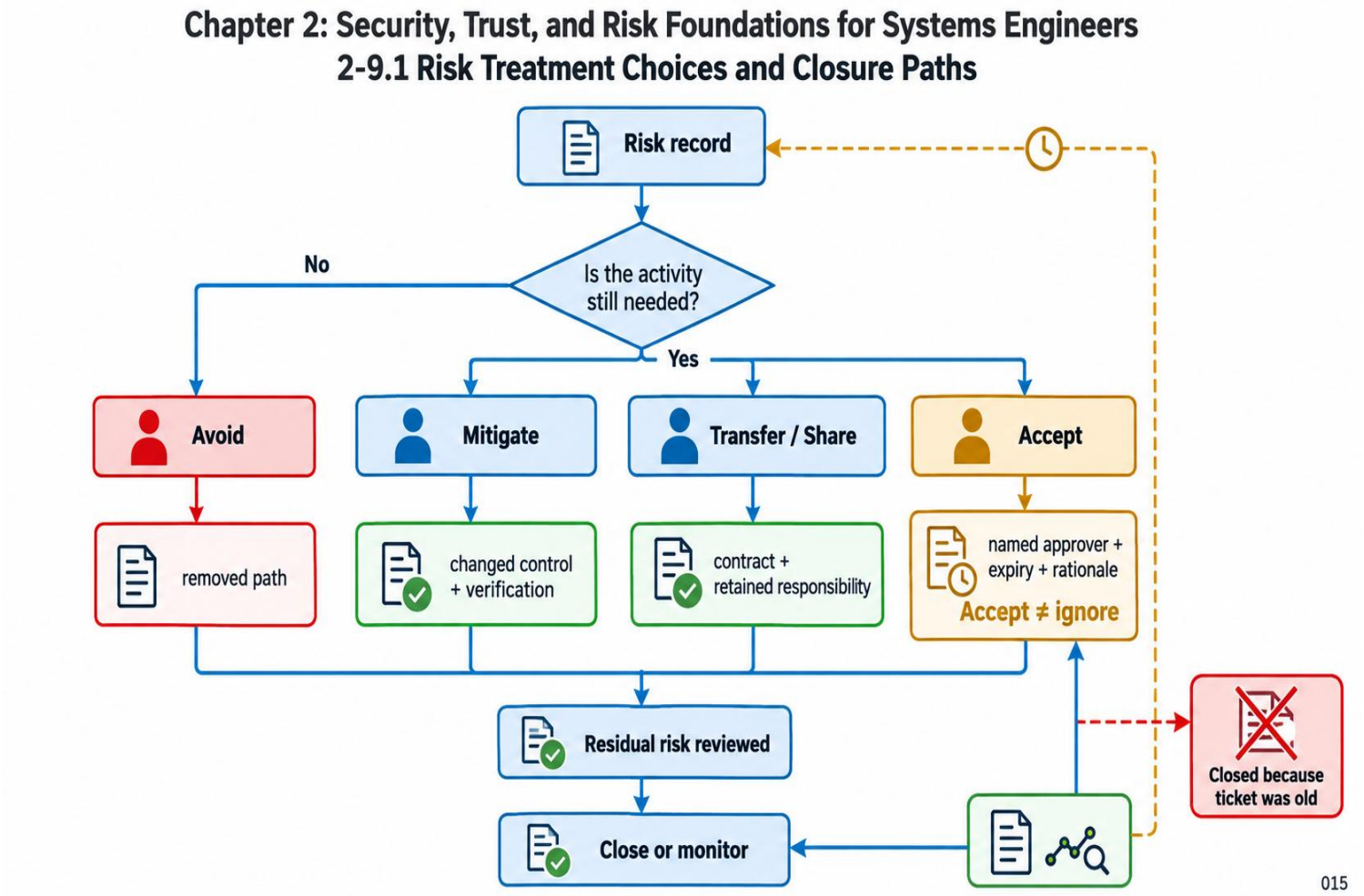

Figure 2-015: Risk treatment choices and closure paths

- **Avoid:** remove the risky capability, such as eliminating public administrative access.
- **Reduce:** add controls, such as short-lived credentials and restricted permissions.
- **Transfer:** shift defined financial or operational consequences through insurance or contracts; accountability for customer impact cannot be completely transferred.
- **Accept:** consciously retain residual risk because further reduction is not proportionate or feasible.

Risk acceptance should identify the owner, rationale, scope, compensating controls, review date, and expiration or trigger. “Accepted by security” without business ownership is not an adequate decision.

Severity Is One Input, Not the Decision

A vulnerability score usually estimates technical severity under general assumptions. Real prioritization also depends on:

- whether the vulnerable component exists in the released artifact;
- whether the vulnerable behavior is reachable;
- whether the service is exposed to a likely attacker;
- whether exploitation is observed in the wild;
- what privileges the process holds;
- what data and downstream systems are accessible;
- whether compensating controls are effective;
- whether a safe fix exists and what operational risk it introduces.

Avoid pseudo-precision. Multiplying arbitrary 1-to-5 likelihood and impact values can help sort a backlog, but the result is not an objective probability. Record the reasoning and evidence so another engineer can challenge it.

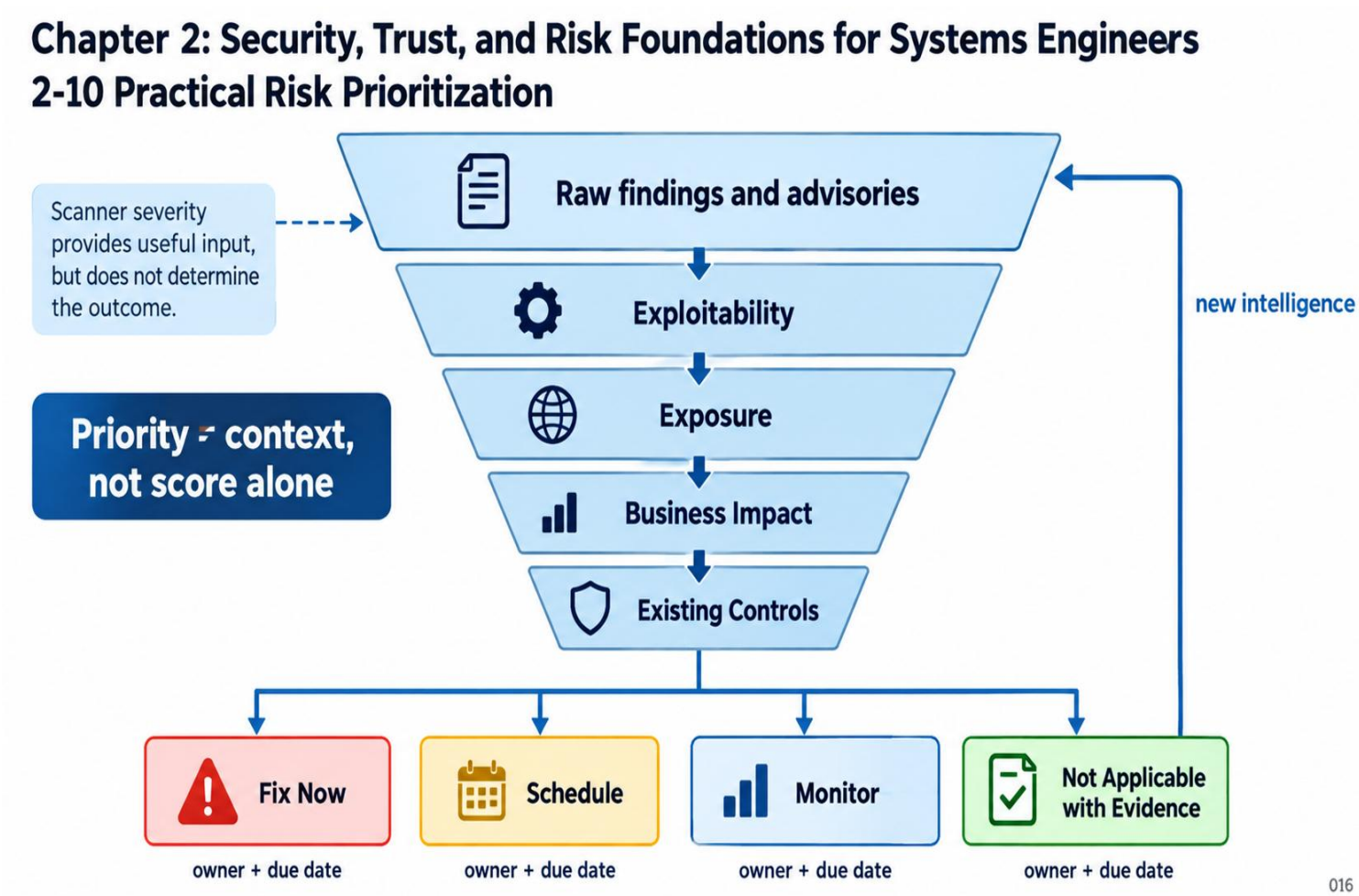

Figure 2-016: Severity combined with exploitability, exposure, business impact, and compensating controls

Testing Trust Assumptions Under Failure

Architecture diagrams often assume that dependencies behave normally. Review them under adverse conditions.

Identity Provider Failure

Ask whether existing sessions remain valid, whether new sessions fail closed, and how administrators recover. An emergency local account may improve resilience but create a high-risk bypass if it is shared or unmonitored.

Secret Store Failure

Determine whether the application uses a short cached credential, stops immediately, or silently falls back to a stale static secret. Each choice affects availability and security.

Network Partition

Check whether retries amplify load, whether duplicate transactions occur, and whether authorization decisions rely on unavailable remote state.

Maintenance Access

Review what changes when an engineer uses an emergency path. Trust assumptions must remain explicit during stressful conditions, not only normal operation.

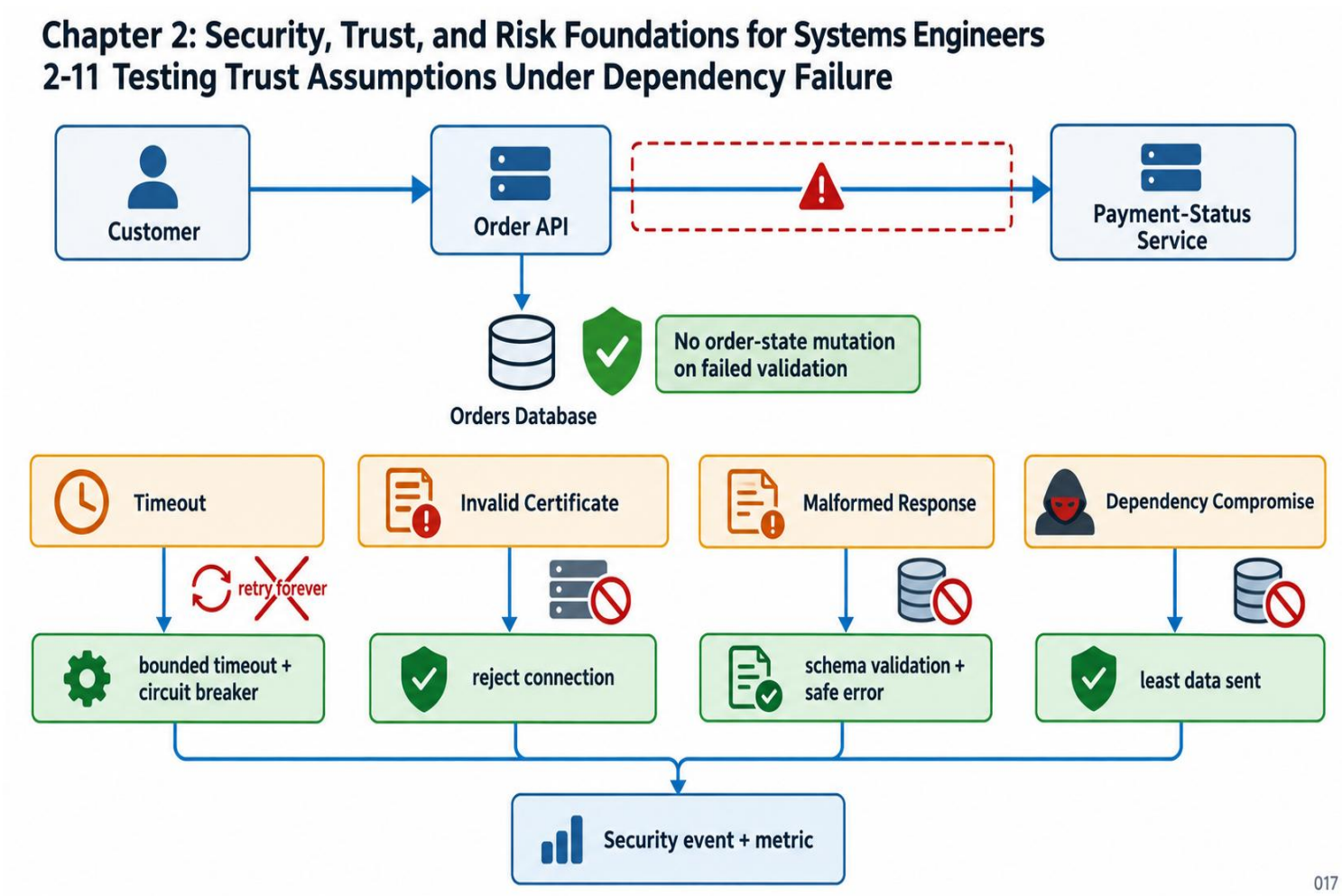

Figure 2-017: Testing trust assumptions under dependency failure

Troubleshooting Risk Work

Entries use vague language

Rewrite each entry with a cause, event, asset, and consequence. If no one can describe how harm occurs, the risk cannot be tested or treated.

Everything is classified as high

Introduce service and data criteria, then compare relative consequences. A classification system that produces one answer cannot prioritize work.

No one owns residual risk

Assign ownership to the role with authority over the affected business objective and resources. Security engineers can advise but should not silently accept another owner's business risk.

A control is listed but cannot be observed

Replace statements such as "access is restricted" with evidence such as an effective permission test, policy decision log, or denied operation.

Misconceptions That Distort Decisions

"Risk equals severity." Severity describes one technical dimension. Risk includes exposure, business consequence, and controls.

“Internal means trusted.” Internal networks, repositories, and managed services can be compromised or misconfigured. Trust must be specific and verified.

“Encryption solves data security.” Encryption protects selected data states. It does not determine who may decrypt, prevent authorized misuse, or guarantee endpoint authenticity.

“Zero trust means no trust.” Systems must trust identities, issuers, code, and policies. Zero trust makes those decisions explicit, narrow, and continuously evaluated.

Chapter Checkpoint

You should now be able to identify assets and trust boundaries, distinguish vulnerability severity from system risk, record accountable treatment decisions, and test whether trust assumptions remain valid during failure and emergency operation.

Review Questions

1. How does vulnerability severity differ from business risk in a specific system context?
2. How do trust boundaries, privilege, and blast radius influence control placement?
3. What information makes a risk-treatment or risk-acceptance decision accountable and reviewable?