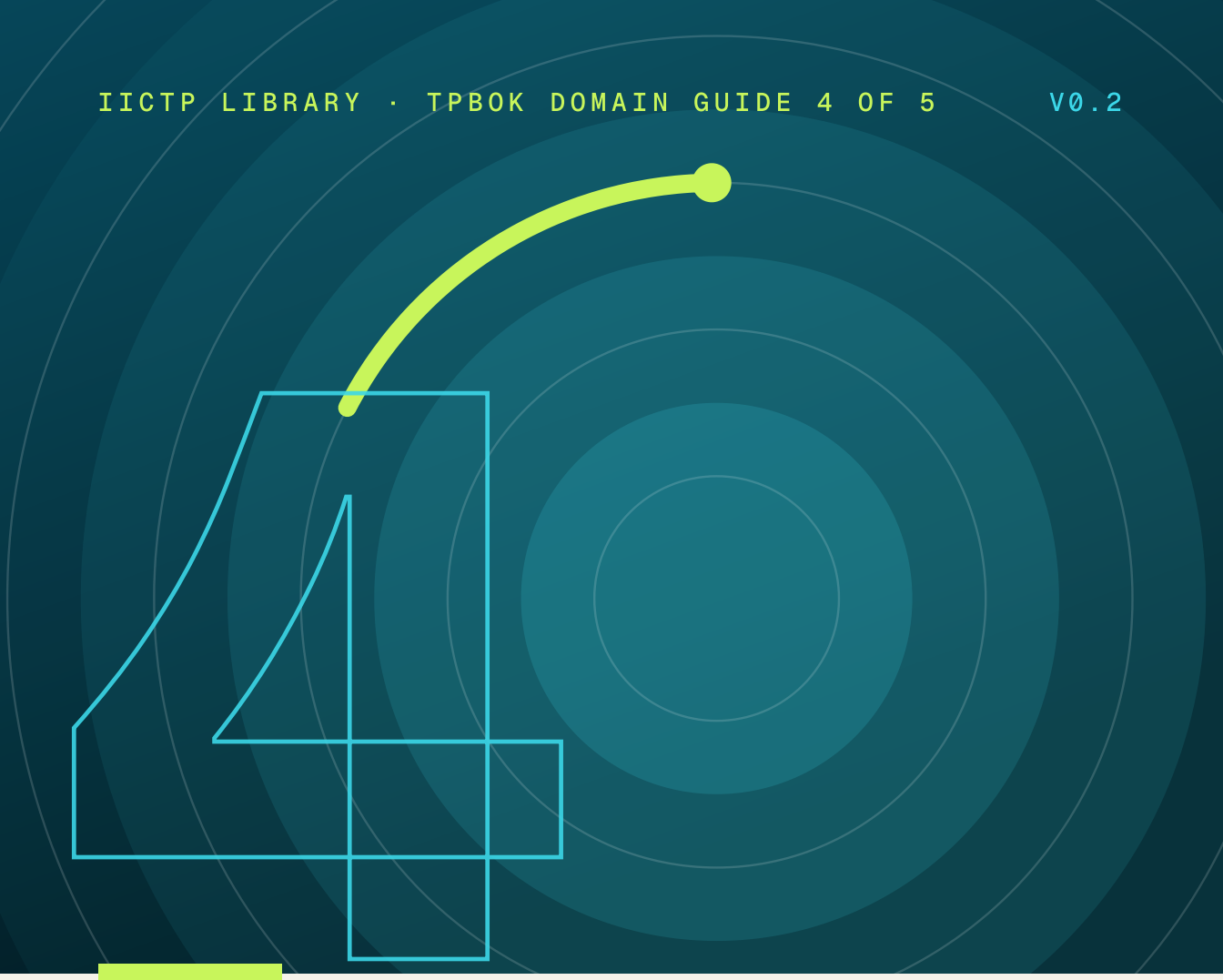

Infrastructure & Security

TPBOK Domain Guide 4 of 5. Task statements, guide chapter, worked scenario, questions and glossary.

International Insignia of Certified Technical Practitioners

22 October 2026

IICTP LIBRARY · VERSION 0.2 · PUBLIC DRAFT FOR COMMENT

Infrastructure & Security

TPBOK Domain Guide 4. The task statements, the guide chapter, worked scenario, illustrative questions and glossary for one domain of the Standard.

International Insignia of Certified Technical Practitioners (IICTP)
22 October 2026

Infrastructure & Security

IICTP Library · Version 0.2. Public draft for comment. 22 October 2026.

© 2026 International Insignia of Certified Technical Practitioners (IICTP).

Licensed under Creative Commons Attribution-NoDerivatives 4.0 International (CC BY-ND 4.0), https://creativecommons.org/licenses/by-nd/4.0/. You may copy and share this book unchanged, for any purpose, with credit to IICTP. Modified versions, including translations, need IICTP's written permission.

This book is drawn from *The Standard for Technical Practitioners* (TPBOK™), version 0.2.

TPBOK™, IICTP, ATP® and CTP® are marks of IICTP and are not licensed by the above. IICTP examination items are confidential and are not part of this book.

Status. This is a draft, published to be criticised. Nothing in it is final. IICTP is a new independent credentialing body and is not ISO/IEC 17024 accredited. Products and organisations named are cited as examples or sources; no endorsement is implied.

Contents

ABOUT THIS GUIDE **How to use this guide** 5

PART A · DOMAIN 4 **The task statements** 7

CHAPTER 4 **Infrastructure & Security** 8

- 4.1 Scope 8
- 4.2 Levels 8
- 4.3 Principles at work 9
- 4.4 Knowledge by task 9
- 4.5 Anti-patterns 19
- 4.6 Worked scenario 20
- 4.7 Illustrative questions 23
- 4.8 Evidence of good practice 25
- 4.9 References and further reading 26

PART A · THE PRINCIPLES **The ten principles** 28

GLOSSARY **Terms used in this domain** 31

INDEX **All 56 task statements** 32

ABOUT THIS GUIDE

How to use this guide

This guide is Chapter 4 of *The Standard for Technical Practitioners* (TPBOK™), published on its own for people who work mainly in Infrastructure & Security. It holds the domain's 10 task statements, the full guide chapter with its worked scenario and illustrative questions, the ten principles, and a glossary of the terms used.

In the ATP® examination this domain carries **20%** of the questions. In the CTP® examination it carries **18%**.

How to read it

- Read the task statements first. They say what a practitioner does.
- Use the chapter for the knowledge behind each task, then try the worked scenario and the questions.
- Task IDs such as `D2-T08` point to other domains. The task index at the back lists all 56.

The two credential levels

Every task statement carries a level tag. The tag says which exam may test it and how deeply.

Tag	Exam	What the candidate must show
A	ATP® (Associate)	Explain the concept, recognise it in a scenario, apply it correctly when the situation is ordinary.
C	CTP® (Certified)	Choose between options, justify the choice against trade-offs, and lead others through it when the situation is ambiguous.
A/C	Both	ATP tests the "explain and apply" part, CTP tests the "choose and justify" part.

A task tagged **C** only is not tested on the ATP exam.

Rules every exam item must follow

1. Every item traces to exactly one task statement ID (for example `D1-T06`). Items that cannot be traced are not used.
2. Items test judgement in a scenario wherever possible, not recall of a vendor's product names.
3. No item depends on a single vendor's tool. Tools may appear as examples in this text; the exam asks about the concept.
4. The share of items per domain matches the blueprint for that credential.

PART A · DOMAIN 4

The task statements

Blueprint weight: ATP 20%, CTP 18%. How software is built, deployed, run, observed and protected.

ID	Level	Task	Knowledge	Principles
D4-T01	A	Explain and maintain a CI/CD pipeline.	Build, test, package, deploy stages; artifacts; environments; pipeline as code; fast feedback on failure.	P1, P4
D4-T02	A	Handle secrets safely.	Never in source control; secret managers; rotation; short-lived credentials; secret scanning; what to do after a leak.	P9
D4-T03	A	Prevent common application vulnerabilities.	The OWASP Top 10 (e.g. broken access control, injection, security misconfiguration); input validation at trust boundaries; secure defaults.	P9
D4-T04	A/C	Manage dependencies and the software supply chain.	Dependency scanning; pinning and update policy; software bills of materials; signed artifacts; build provenance.	P8
D4-T05	C	Apply least privilege and zero-trust networking.	Identity-based access; service-to-service authentication (e.g. mutual TLS); network segmentation; short-lived credentials; access reviews.	P9
D4-T06	C	Threat-model a system.	Data flow diagrams; trust boundaries; STRIDE; prioritising threats; turning threats into backlog items.	P5, P9
D4-T07	A/C	Make systems observable.	Metrics, logs and traces; structured logging; correlation IDs; SLIs, SLOs and error budgets; alerting on symptoms not causes. ATP: read and use; CTP: design.	P1, P9
D4-T08	C	Run and scale containerised workloads on an orchestrator.	Containers and images; Kubernetes concepts (pods, deployments, services); resource requests and limits; health probes; horizontal autoscaling; rollout strategies.	P2, P9
D4-T09	C	Manage infrastructure as code.	Declarative definitions; review and plan before apply; state; drift; policy checks in the pipeline; immutable infrastructure.	P4, P7
D4-T10	C	Respond to incidents and learn from them.	Severity levels; incident roles; communication during an incident; recovery objectives (RPO, RTO) and tested backups; blameless retrospectives with tracked actions.	P1, P9, P10

CHAPTER 4

Infrastructure & Security

4.1 Scope

This domain is about how software gets from a commit to production, how it runs there, how the team knows what it is doing, and how it is protected and recovered when something goes wrong.

In scope: CI/CD pipelines, secrets, common application vulnerabilities, software supply chain, least privilege and zero trust, threat modelling, observability, containers and orchestration, infrastructure as code, and incident response and recovery.

Covered elsewhere:

- Structural failure isolation (timeouts, circuit breakers, bulkheads): Domain 1, `D1-T09`. This chapter covers the platform and operational side.
- How teams use integration and release (trunk-based development, flags, release trains): Domain 2. This chapter covers the pipeline itself.
- AI-specific threats (prompt injection, excessive agency): Domain 3, `D3-T08`. The general methods here (threat modelling, least privilege) apply to them.
- Leading people through incidents and building a blameless culture: Domain 5, `D5-T08`. This chapter covers the incident process.

Tools. Products and projects named here (Kubernetes, OpenTelemetry, Terraform and others) are examples of widely used, mostly open-source tools. The exam asks about the concepts. Where Kubernetes is named in the CTP blueprint, items test orchestration concepts it shares with other orchestrators, and use its vocabulary because it is the common one.

4.2 Levels

	ATP® (20% of 100 items)	CTP® (18% of 150 items)
Typical candidate	Developer who uses the pipeline, handles secrets, writes secure code and reads dashboards	Tech lead who designs the pipeline, the observability, the runtime platform choices and the security posture, and leads incident response

	ATP® (20% of 100 items)	CTP® (18% of 150 items)
What items ask	"What is wrong with this code or config...", "What should the developer do now...", "What does this pipeline stage do..."	"What would you change first...", "Which control best reduces this risk...", "What does this SLO mean for..."
Tasks tested	D4-T01, T02, T03, T04, T07	D4-T04, T05, T06, T07, T08, T09, T10

4.3 Principles at work

- **P9 Contain failure** is this domain's core principle. Assume breaches and outages will happen, and limit how far they reach: least privilege, isolated environments, small blast radius, tested recovery.
- **P1 Close the loop**: observability exists so the team learns what the system is doing from evidence, quickly. Incidents end with learning that changes the system.
- **P4 Make "done" explicit and checkable**: pipelines enforce the quality and security bar automatically. Infrastructure is defined in reviewable code, not undocumented console clicks.
- **P8 Verify what you did not write**: most of the code in production is dependencies, base images and infrastructure modules written by others. Check what you pull in.
- **P5 Retire risk early**: threat-model during design, scan in the pipeline, and practise recovery before you need it.

The security principles of Saltzer and Schroeder (1975) still hold and appear throughout this chapter:

- economy of mechanism (keep it simple);
- fail-safe defaults (deny unless allowed);
- complete mediation (check every access);
- open design (security must not depend on a secret design);
- separation of privilege;
- least privilege;
- least common mechanism;
- psychological acceptability (security people will actually use).

4.4 Knowledge by task

D4-T01 Explain and maintain a CI/CD pipeline A

What the practitioner does. Understands what each stage of the pipeline does, keeps it fast and reliable, and fixes it when it breaks.

Core knowledge.

- **Continuous integration.** Every change merged to main is automatically built and tested, so problems surface within minutes of the change that caused them.
- **Continuous delivery.** Every change that passes the pipeline *could* be released at any time. **Continuous deployment** goes one step further: it *is* released automatically (Humble and Farley, 2010).
- **Typical stages:**
  1. Build.
  2. Fast tests: unit tests, linting, static analysis, secret and dependency scanning.
  3. Package into an artifact (for example a container image), versioned and stored.
  4. Deploy to a test environment and run slower tests (integration, end-to-end).
  5. Deploy to production, often progressively (`D2-T07`).
- **Build once, promote the same artifact.** The exact artifact that was tested is the one deployed to each environment. Configuration differs per environment; the code does not. Rebuilding per environment means production runs something that was never tested.
- **Pipeline as code.** The pipeline definition lives in the repository, is versioned and reviewed, and changes go through the same process as code.
- **Speed and trust.** A slow pipeline encourages batching, and a flaky one teaches people to ignore red builds (`D2-T09`). Fast feedback first, slow checks later; fix or quarantine flaky tests promptly.
- **A broken main is the team's top priority.** Fix it or revert, before anything else is merged.

ATP level. Say what a given stage does and why it is in that order; read a failed pipeline and identify the cause; explain why the same artifact is promoted.

Common mistakes. Rebuilding for production. Re-running a failing pipeline until it passes. Pipelines that take an hour before the first test result.

D4-T02 Handle secrets safely A

What the practitioner does. Keeps passwords, keys, tokens and certificates out of places they can leak, and responds correctly when one does.

Core knowledge.

- **Never in source control.** A secret committed to Git stays in the history even after a later commit deletes it, and in every clone and fork. Treat a committed secret as leaked.
- **Where secrets belong.**
  - A secret manager or vault, or the platform's secret store, injected at runtime.

  - Local development uses separate, low-privilege credentials kept out of the repository (for example, ignored environment files).
- **Short-lived over long-lived.** Prefer credentials that expire on their own: workload identity, or federated identity from the CI system to the cloud provider (for example via OpenID Connect). A leaked short-lived token is worth far less than a permanent access key.
- **Rotation.** Secrets are rotated regularly and immediately after any suspected exposure. Systems should be able to rotate without an outage.
- **Scanning.** Secret scanning in pre-commit hooks, the pipeline and the repository host catches most mistakes before or soon after they happen.
- **Not in logs, URLs, error messages or prompts.** These are copied widely and kept for a long time (see also `D3-T02`).
- **When a secret leaks:**
  1. Revoke and rotate it first.
  2. Check logs for use of it.
  3. Report it according to the incident process.
  4. Only then clean up the history.

ATP level. Spot secrets in code and config; choose the right place to store one; give the correct order of actions after a leak.

Common mistakes. Deleting the secret in a new commit and assuming the problem is solved. One shared admin key used by every service and developer. Secrets in CI logs because a script echoed its environment.

D4-T03 Prevent common application vulnerabilities A

What the practitioner does. Writes and reviews code that avoids the vulnerability classes responsible for most real-world breaches.

Core knowledge.

- **The OWASP Top 10.** A widely used awareness list of the most critical web application security risks. The TPBOK follows the 2025 edition (OWASP Top 10:2025). The categories practitioners meet most:
  - **Broken access control (A01).** Users act outside their permissions, for example changing an ID in a URL to see another user's record. Check authorisation on the server for every request and every object, not just whether the user is logged in. In the 2025 edition this category also includes server-side request forgery (SSRF), where the server is tricked into fetching a URL the attacker chooses, often reaching internal services.
  - **Security misconfiguration (A02).** Default credentials, verbose errors in production, open storage buckets, unnecessary features enabled.

  - **Software supply chain failures (A03).** Vulnerable, outdated or compromised dependencies, build systems and distribution. See `D4-T04`.
  - **Cryptographic failures (A04).** Sensitive data unencrypted in transit or at rest, weak algorithms, home-made crypto. Store passwords only as salted hashes from slow algorithms designed for passwords (such as Argon2, scrypt or bcrypt).
  - **Injection (A05).** Untrusted input is interpreted as code or a query (SQL, OS commands, LDAP, templates). Use parameterised queries and safe APIs, never string concatenation.
  - **Authentication failures (A07).** Weak session handling, no protection against credential stuffing, missing multi-factor options.
  - **Mishandling of exceptional conditions (A10).** Errors and unusual states handled in ways that fail open, leak information or leave the system inconsistent. Fail safely and closed.
  - Also: insecure design (A06), software or data integrity failures (A08), and security logging and alerting failures (A09).
- **Validate at trust boundaries.** Treat all input from outside the component as untrusted: user input, other services, files, model output (`D3-T08`). Validate type, length, format and range on the server. Client-side checks are for usability only.
- **Encode output for its context.** HTML, URL, JavaScript and SQL each need their own encoding. Frameworks do much of this by default; know when you are bypassing it.
- **Secure defaults.** Deny by default; least privilege for the application's own database user; generic error messages to users, detailed ones to logs.

ATP level. Identify the vulnerability in a code sample and choose the correct fix; explain why a client-side check is not a control.

Common mistakes. Authorisation checked in the UI only. Escaping input by hand instead of parameterising. Stack traces shown to users.

D4-T04 Manage dependencies and the software supply chain A/C

What the practitioner does. Keeps track of the code the team did not write but ships anyway, and protects the path from source to production against tampering.

Core knowledge.

- **Most shipped code is someone else's.** Libraries, transitive dependencies, base images, CI actions, infrastructure modules. Each is a trust decision (P8). Incidents such as Log4Shell (CVE-2021-44228, 2021) and the backdoor found in XZ Utils (CVE-2024-3094, 2024) showed how deep and how quiet this risk can be.
- **Basic hygiene (ATP):**

  - Use lockfiles so builds use exactly the versions that were tested.
  - Run software composition analysis (dependency scanning) in the pipeline.
  - Update regularly in small steps rather than rarely in large jumps.
  - Remove dependencies you do not need.
  - Check that a new package is the real, maintained one, not a look-alike or invented name (see `D3-T03`).
  - Pull internal packages only from internal registries, to avoid *dependency confusion*.
- **Triage, not panic.** Not every reported vulnerability is reachable in your use of the library. Prioritise by severity, exploitability, and whether the vulnerable code is actually used, but do not let "probably not reachable" become a permanent excuse.
- **Software bill of materials (SBOM).** A machine-readable list of components in a release, in formats such as SPDX or CycloneDX. It answers "are we affected?" in minutes when the next big vulnerability is announced.
- **Build integrity (CTP):**
  - Protect the pipeline itself: reviewed changes, limited permissions, pinned versions of CI actions and base images.
  - Sign artifacts and verify signatures before deployment.
  - Produce build provenance (a verifiable record of what was built, from what, by which pipeline). The SLSA framework describes increasing levels of these assurances.
- **Secure development frameworks.** The NIST Secure Software Development Framework (SP 800-218) gathers these practices into a checklist many customers and regulators now reference.

ATP level. Apply dependency hygiene; read a dependency scan result and decide what to do; recognise a suspicious package. **CTP level.** Design the supply-chain controls for a pipeline; plan the response when a widely used dependency has a critical vulnerability; decide which assurances (SBOM, signing, provenance) a context needs.

Common mistakes. No lockfile. Hundreds of open, ignored scanner alerts. CI steps pulling third-party actions at a moving "latest" tag with access to production secrets.

`D4-T05` Apply least privilege and zero-trust networking C

What the practitioner does. Designs access so that every person and workload has only the access it needs, and no request is trusted just because of where it comes from on the network.

Core knowledge.

- **Least privilege.** Every user, service, pipeline and AI agent gets the minimum permissions for its task, for the minimum time. Separate roles for reading, deploying

and administering. Break-glass access for emergencies is logged and reviewed.

- **Zero trust (NIST SP 800-207, 2020).** The traditional model trusted anything inside the network perimeter. Zero trust assumes the network is hostile. Every request is authenticated and authorised based on identity, device and context, whatever its network location. Google's BeyondCorp (Ward and Beyer, 2014) is a well-known early implementation.
- **Workload identity.** Services authenticate to each other with their own identities, for example mutual TLS with short-lived certificates, often issued and rotated automatically by a service mesh or platform. Shared static keys and IP allow-lists are not identities.
- **Segmentation.** Network policies limit which workloads may talk to which, so a compromised service cannot reach everything. Production is separated from test; sensitive data stores are reachable only by their owning service.
- **Access reviews.** Permissions accumulate. Review them regularly, remove access when people change roles or leave, and alert on unusual use.

CTP level. Find excessive privilege in a described design; propose identity-based controls to replace network-based trust; plan segmentation that limits blast radius.

Common mistakes. Every service using one database superuser. "It's on the internal network, so it doesn't need authentication." Developers with permanent production admin rights "just in case".

D4-T06 Threat-model a system C

What the practitioner does. Systematically works out what could go wrong with a system's security, decides what to do about it, and turns that into work.

Core knowledge.

- **Four questions.** The Threat Modeling Manifesto (2020) reduces threat modelling to four questions:
  1. What are we working on?
  2. What can go wrong?
  3. What are we going to do about it?
  4. Did we do a good enough job?
- **Model the system.** A data flow diagram showing processes, data stores, external entities and data flows. Mark *trust boundaries*: the places where data crosses from less trusted to more trusted (internet to API, service to database, user content to an AI model).
- **STRIDE.** A checklist of threat types to ask about at each element and boundary:

Threat	Violates	Example
Spoofing	Authentication	Calling an internal API while pretending to be another service
Tampering	Integrity	Changing the price field in a checkout request
Repudiation	Non-repudiation	A user denies making a transfer and there is no reliable record
Information disclosure	Confidentiality	Another tenant's data returned by a search
Denial of service	Availability	An expensive endpoint called in a loop
Elevation of privilege	Authorisation	A normal user reaching an admin function

- **Respond to each threat.** Mitigate it (add a control), eliminate it (remove the feature or data), transfer it (for example to a provider, contractually), or accept it, explicitly, with an owner. Mitigations become backlog items with acceptance criteria (`D2-T04`).
- **When.** During design (P5), revisited when the design changes significantly, and after incidents. A short session with the people who build the system is worth more than a long document nobody reads (Shostack, 2014).

CTP level. Draw the trust boundaries for a described system, identify threats with STRIDE, and prioritise mitigations.

Common mistakes. Threat modelling only before an audit. Producing a list of threats with no owners or actions. Ignoring internal threats and trusted-but-compromised components.

D4-T07 Make systems observable A/C

What the practitioner does. Instruments systems so the team can tell whether users are being served well, and can find out why when they are not.

Core knowledge.

- **Three kinds of signal:**
  - *metrics*: numbers over time, cheap to store and good for alerting and trends;
  - *logs*: records of events, best when structured (key-value or JSON) so they can be queried;
  - *traces*: the path of one request across services, with timing for each step, essential for finding where time goes in distributed systems.

  OpenTelemetry is a vendor-neutral standard for producing all three.
- **Correlation.** A request or trace ID carried through every service and written in every log line lets you join the three signals for one user's request.
- **What to measure.**

  - The “four golden signals” for user-facing services: latency, traffic, errors and saturation (Beyer et al., 2016).
  - The RED method (rate, errors, duration) for request-driven services.
  - The USE method (utilisation, saturation, errors) for resources such as CPU, disks and queues.
- **SLIs, SLOs and error budgets.**
  - A *service level indicator* is a measure of user experience, for example the proportion of booking requests that succeed within 500 ms.
  - A *service level objective* is a target for it over a window, for example 99.9% over 30 days.
  - The *error budget* is what the SLO allows to fail: 0.1% of 30 days is 43.2 minutes. While budget remains, the team can take risks; when it is spent, reliability work comes first. This turns “how reliable is reliable enough?” into an agreed number.
- **Alert on symptoms, not causes.** Page a human when users are affected or the error budget is burning fast, not every time CPU is high. Every alert should be actionable. Alerts that are routinely ignored train people to ignore the important ones.
- **Privacy.** Logs and traces are data stores. Mask personal data and secrets, and set retention periods.

ATP level. Read a dashboard, a log and a trace to locate a problem; add structured log lines with a correlation ID; explain what an SLO is. **CTP level.** Choose SLIs and SLOs for a service; design alerting around them; calculate and apply an error budget.

Common mistakes. Logs as unstructured prose. Dashboards of every metric with no indication of user impact. 99.999% targets nobody can meet or afford. Alerts that page at 3 a.m. for things that can wait.

D4-T08 Run and scale containerised workloads on an orchestrator C

What the practitioner does. Runs services on a container orchestrator so that they start reliably, scale with load, survive node failures and update without downtime.

Core knowledge.

- **Containers and images.** An image packages an application and its dependencies. Keep images small, built from a trusted, pinned base, scanned, and run as a non-root user. A running container should be replaceable at any time; state lives outside it.
- **Core Kubernetes concepts:**
  - a *pod* runs one or more containers together;
  - a *deployment* keeps a desired number of identical pods running and manages updates;

  - a *service* gives a stable address that load-balances across the healthy pods;
  - configuration and secrets are injected rather than built into images.
- **Resource requests and limits.**
  - *Requests* are what the scheduler reserves for a container, and are what CPU utilisation targets are measured against.
  - *Limits* are the ceiling. A container over its CPU limit is throttled; one over its memory limit is killed.
  - Missing requests make scheduling unpredictable and break CPU-based autoscaling.
- **Health probes:**
  - a *liveness* probe tells the orchestrator to restart a container that is stuck;
  - a *readiness* probe tells it to stop sending traffic to a pod that cannot serve yet;
  - a *startup* probe gives slow-starting applications time before liveness checks begin.

  A liveness probe should check only the process itself. If it checks a shared dependency such as a database, a slow database makes every pod fail liveness at once, and the orchestrator restarts the whole service, turning a slowdown into an outage.
- **Autoscaling.**
  - The *horizontal pod autoscaler* changes the number of pods from metrics such as CPU utilisation relative to requests, or custom metrics like queue length.
  - A cluster autoscaler adds or removes nodes.
  - Scaling takes time (new nodes, image pulls, warm-up), so scale on leading indicators and keep headroom for sudden peaks.
- **Safe updates and disruption.** Rolling updates replace pods gradually, and readiness probes stop traffic reaching new pods until they can serve. Pod disruption budgets keep a minimum number running during maintenance. Spread replicas across nodes and zones.

CTP level. Diagnose scaling and restart problems from a description; set requests, limits, probes and autoscaling for a service; plan for node and zone failure.

Common mistakes. No resource requests. Liveness probes that call the database. All replicas on one node. Memory limits far below real usage, causing constant restarts.

D4-T09 Manage infrastructure as code C

What the practitioner does. Defines infrastructure in versioned, reviewed code so environments are reproducible and changes are visible before they happen.

Core knowledge.

- **Declarative definitions.** Describe the desired state (networks, databases, clusters, permissions) and let a tool work out the changes. Common tools include Terraform and OpenTofu, cloud-specific template languages, and Kubernetes manifests managed through GitOps.
- **Review the plan, not just the code.** Before applying, generate the plan of what will be created, changed or *destroyed*, and review it in the pull request. A one-line change can replace a database.
- **State.** Many tools keep a state file mapping code to real resources. It must be stored remotely, locked during changes, backed up and protected, since it often contains sensitive values.
- **Drift.** Manual changes in the console make reality differ from code. Detect drift regularly and fix it by changing code, not the console.
- **Policy as code.** Automated checks in the pipeline reject unsafe infrastructure before it exists: public storage, unencrypted databases, open security groups, missing tags.
- **Immutable infrastructure.** Replace servers and images rather than patching them in place, so every running instance matches a known definition.
- **Environments from the same code.** Test and production built from the same modules with different parameters, so a test environment actually tests production's shape.
- **Least privilege for the pipeline.** The identity that applies infrastructure changes is powerful. Restrict it, log its use, and require review for changes to it.

CTP level. Design an IaC workflow with review, policy checks and drift detection; spot a dangerous plan; decide what belongs in code versus runtime configuration.

Common mistakes. Applying from a laptop. State files in the repository. "Temporary" console changes that become permanent.

D4-T10 Respond to incidents and learn from them C

What the practitioner does. Leads or takes part in restoring service quickly and calmly, communicates well during an incident, and makes sure the organisation learns from it.

Core knowledge.

- **Restore first, investigate later.** The first goal is to reduce user impact: roll back, switch off a flag, fail over, shed load. Root-cause analysis comes after service is restored. DORA's failed deployment recovery time measures this ability (D2-T06).
- **Severity levels.** Agreed in advance, based on user impact, so everyone knows who is called and how fast.
- **Roles.** Borrowed from emergency services' incident command practice:

  - an *incident commander* coordinates and decides, and does not debug;
  - an *operations lead* works the problem;
  - a *communications lead* updates stakeholders and users on a regular schedule.

  In small teams one person may hold two roles, but the coordinating role must be explicit.

- **Communication.** Regular, honest updates even when there is no news: what is affected, what is being done, when the next update is. Internal and external audiences need different detail.
- **Security incidents** follow the same structure plus evidence preservation, containment and, where personal data is involved, legal notification duties with fixed deadlines. Involve the data protection and legal contacts early. NIST SP 800-61 Rev. 3 (2025) and the Respond and Recover functions of the NIST Cybersecurity Framework 2.0 (2024) describe the life cycle.
- **Recovery objectives.**
  - The *recovery point objective* (RPO) is how much data loss, measured in time, is acceptable.
  - The *recovery time objective* (RTO) is how long restoration may take.

  Both are business decisions that drive design: backup frequency, replication, standby capacity.

- **Backups are only real if restores are tested.** Test restores regularly, keep at least one copy isolated from the production account (ransomware targets backups), and time the restore against the RTO.
- **Blameless post-incident reviews (Allspaw, 2012).** Ask how the system allowed the failure and why the actions taken made sense at the time, not who is to blame. People who fear blame hide information, and the real causes stay. Actions are specific, owned, tracked to completion, and fed back into SLOs, tests, runbooks and the threat model.
- **Practise.** Game days and chaos experiments (deliberately injecting failure in controlled conditions) test recovery and runbooks before a real incident does (P5).

CTP level. Run an incident: choose the first mitigation, assign roles, and communicate; set RPO and RTO with the business and check that the design meets them; lead a blameless review that produces real changes.

Common mistakes. Debugging for an hour before rolling back. Everyone in the incident channel, nobody in charge. Post-mortems that end with “engineer to be more careful”. Backups that have never been restored.

4.5 Anti-patterns

Anti-pattern	Symptom	Why it hurts	Remedy	Task
Secrets in the repo	Keys in code or config, “removed” in a later commit	Anyone with a clone has them, forever	Secret manager; scanning; rotate on any exposure	D4-T02
Perimeter trust	Internal services skip authentication	One foothold reaches everything	Identity-based access; mutual TLS; segmentation	D4-T05
God credentials	One admin key used by many services, pipelines and people	Huge blast radius; no accountability	Per-workload identities; least privilege; short-lived credentials	D4-T02, D4-T05
Pet servers	Hand-configured servers nobody dares to rebuild	Unknown state; slow recovery; drift	Infrastructure as code; immutable images	D4-T09
Alert fatigue	Hundreds of alerts a week, mostly ignored	Real incidents missed	Alert on SLO burn and user impact; delete unactionable alerts	D4-T07
Probe-induced outage	Liveness probe checks a shared dependency	A slow dependency restarts every pod at once	Liveness checks the process only; handle dependencies with timeouts and degradation	D4-T08
Schrödinger’s backup	Backups run, restores never tested	Recovery fails when it is needed	Scheduled restore tests timed against the RTO	D4-T10
Blame post-mortem	Reviews end with a named culprit	Information is hidden; causes remain	Blameless review; system-level actions with owners	D4-T10
Scanner wallpaper	Thousands of open dependency alerts	Real critical issues lost in noise	Triage by reachability and severity; small regular updates	D4-T04

4.6 Worked scenario

Situation (CTP level). A company runs an online appointment-booking service used by about 3,000 clinics. It handles patient names, phone numbers and appointment reasons. It runs on a managed Kubernetes cluster with a managed relational database.

Last Monday morning, the busiest time of the week, the booking API was down or very slow for 95 minutes:

- On-call engineers saw pods restarting repeatedly and CPU alerts firing, but could not tell where requests were slowing down.
- The autoscaler did not add pods.
- The service was finally restored when someone manually raised the database instance size and restarted everything.

During the review, a second finding emerged: the CI system deploys using a permanent cloud access key with full administrator rights, stored as a CI variable, and that key also appears in an old commit in a public repository fork.

The tech lead is asked to lead the post-incident review and propose changes.

Step 1: The security finding comes first (D4-T02, D4-T10). An admin key in a public place is an active risk, not a review item.

1. The key is revoked immediately, and a new deployment path is set up (Step 6).
2. The cloud audit log is searched for any use of the key from unknown locations. None is found.
3. Because patient data could have been exposed, the data protection lead is informed and decides on notification duties.
4. The finding is handled as its own security incident, with its own record.

Step 2: Reconstruct the outage without blame (D4-T10, D4-T07). The team builds a timeline from database metrics, orchestrator events and the few logs available:

1. A weekly reporting job started at 09:00 and ran heavy queries against the primary database, slowing all queries.
2. Booking API pods each had a liveness probe that ran a database query. With the database slow, probes timed out, so the orchestrator restarted pods, all of them, repeatedly.
3. Restarting pods opened new database connections on start-up, adding more load to the struggling database.
4. The autoscaler could not scale on CPU because the pods had no resource requests set, so utilisation could not be calculated.
5. Without traces or correlation IDs, responders could see that requests were slow but not where.

No individual made an unreasonable decision. Each setting had seemed sensible when added.

Step 3: Fix the runtime configuration (D4-T08, D1-T09).

- **Liveness probe:** now checks only that the process responds. **Readiness probe:** checks that the pod has finished starting. Neither calls the database.
- **Database dependency:** handled in code with timeouts, a connection pool limit and a circuit breaker. Under database slowness the API returns a clear “try again” for bookings and keeps serving cached clinic information.
- **Resource requests and limits:** set from measured usage, so autoscaling works.
- **Autoscaling:** scales on request rate as well as CPU, with a minimum of 6 pods on weekday mornings.
- **Placement:** replicas spread across three zones, with a pod disruption budget.
- **Reporting job:** moved to a read replica.

Step 4: Define reliability in numbers (D4-T07). With product and clinic-support leads, the team defines one SLI for the booking API: the proportion of booking requests that succeed within 1 second. They set an SLO of 99.9% over 30 days.

	Minutes
Error budget for 30 days (0.1% of 43,200 minutes)	43.2
Monday's outage	95

The budget was overspent more than twice over. Under the agreed policy, the team prioritises reliability work over new features until the rolling 30-day window is back within the SLO.

Step 5: Make the system observable (D4-T07).

- OpenTelemetry tracing is added to the API and database client.
- Logs become structured, with a correlation ID on every line. Patient names and phone numbers are masked.
- The old CPU alerts are replaced with alerts on how fast the error budget is being spent.
- A dashboard shows the SLI, latency by endpoint, database saturation and pod restarts together.

Step 6: Remove the god credential (D4-T05, D4-T09, D4-T04).

- **CI to cloud:** the CI system now authenticates through identity federation, receiving short-lived credentials scoped to deploying this service only.
- **Infrastructure changes:** made through a separate infrastructure pipeline with plan review and policy checks. Nobody applies from a laptop, and break-glass admin access is logged and alerted.
- **Secret scanning:** enabled on all repositories, including forks under the organisation's control.
- **Pipeline:** CI actions are pinned to specific versions, and images are scanned and signed before deployment.

Step 7: Threat-model the booking flow (D4-T06). Prompted by the leaked key, the team holds a two-hour session on the booking flow. Using STRIDE at the trust boundaries, they identify three gaps:

- **Information disclosure:** clinic staff can enumerate other clinics' appointments by changing IDs. A server-side authorisation check per object fixes it, with a test (see `D4-T03`).
- **Repudiation:** there is no reliable record of who cancelled an appointment. An audit log fixes it.

- **Denial of service:** the public booking endpoint has no rate limit. Rate limits per clinic and per IP address fix it.

Each becomes a backlog item with an owner.

Step 8: Recovery and practice (D4-T10).

- **Objectives:** the business sets an RPO of 5 minutes and an RTO of 1 hour for booking data. Point-in-time database recovery meets the RPO.
- **Restore test:** a restore into an isolated account took 50 minutes, within the RTO.
- **Practice:** a quarterly game day will simulate database slowness and a zone failure.
- **Review:** the blameless review is shared across engineering, every action has an owner and a date, and the incident record links to the ADRs for the probe policy and the deployment-identity change (`D2-T08`).

What a weaker answer looks like.

- "Make the database permanently bigger." This hides the probe design, missing requests and absent tracing until the next spike.
- "Remind engineers not to commit keys." This leaves an admin key that can be committed, and no scanning to catch it.
- "The on-call engineer should have restarted the database sooner." This blames a person for a system that gave them no information.

Mini-scenario (ATP level). A developer accidentally commits a `.env` file containing the staging database password, notices, and deletes the file in the next commit. *Is the problem solved? What should they do?*

No. The password is still in the repository history and in every clone. The developer should:

1. Rotate the password immediately and check access logs.
2. Report it under the team's process.
3. Add the file to `.gitignore`.
4. Enable a pre-commit secret scan.
5. Load secrets from the secret manager rather than a committed file (D4-T02).

4.7 Illustrative questions

These show the style of item for this domain. They are written for this guide and are not taken from the exam bank.

Q1. `D4-T03` · **ATP** A developer builds a query like this: `"SELECT * FROM users WHERE email = '" + email + "'"`, where `email` comes from a login form. What is the correct fix?

- A. Check the email format in the browser before submitting
- B. Use a parameterised query (prepared statement) so the input is always treated as data
- C. Remove single quotes from the input
- D. Run the query with a database user that has administrator rights so errors are logged

Answer: B.

- **A** is bypassed by any attacker calling the API directly.
- **C** is incomplete escaping that attackers routinely get around.
- **D** makes the impact of injection worse.

Q2. `D4-T02` · **ATP** A deployment pipeline needs to deploy to a cloud account. Which approach to credentials is best?

- A. Store an administrator access key as a pipeline variable
- B. Commit a deployment key to the repository in an encrypted file, with the decryption password in the README
- C. Use the cloud provider's identity federation so the pipeline receives short-lived credentials limited to deploying this service
- D. Use the lead developer's personal credentials

Answer: C. It removes long-lived secrets and limits what a compromised pipeline could do.

- **A** is a long-lived, over-privileged secret.
- **B** keeps the key and the password together.
- **D** ties deployments to one person and their full access.

Q3. `D4-T07` · **CTP** A service has an availability SLO of 99.9% measured over 30 days. Roughly how much total unavailability does the error budget allow in that window?

- A. 4.3 minutes
- B. 43.2 minutes
- C. 7.2 hours
- D. 3 days

Answer: B. 30 days × 24 hours × 60 minutes = 43,200 minutes, and 0.1% of that is 43.2 minutes.

Q4. `D4-T08` · **CTP** During a database slowdown, every pod of an API service is restarted repeatedly by the orchestrator, turning a slowdown into a full outage. The pods' liveness probe runs a query against the database. What is the most appropriate change?

- A. Increase the liveness probe timeout to 5 minutes
- B. Make the liveness probe check only that the application process is responsive, and handle database slowness in the application with timeouts and graceful degradation
- C. Remove resource limits so the pods have more CPU
- D. Add more replicas so some survive the restarts

Answer: B. Liveness should reflect whether *this* process needs restarting, not the state of a shared dependency.

- **A** delays the restart of genuinely stuck pods.
- **C** and **D** do not stop the synchronised restarts.

Q5. `D4-T06` · **CTP** In a threat-modelling session, the team notes that a customer could change the `price` field in the checkout request sent from their browser, and the server would accept it. Which STRIDE category does this threat belong to, and what is the right mitigation?

- A. Spoofing; require multi-factor authentication
- B. Tampering; calculate the price on the server from trusted data and ignore any price sent by the client
- C. Repudiation; log every checkout request
- D. Denial of service; rate-limit the checkout endpoint

Answer: B. Changing data in transit or in a request violates integrity, which is tampering. The fix is to never trust a client-supplied value for something the server can determine itself. The other options address different threats.

4.8 Evidence of good practice

What a reviewer or assessor would expect to find in a team that does this domain well:

- The pipeline builds once and promotes the same artifact. It runs tests, static analysis, secret scanning and dependency scanning on every change, and gives first results within minutes.
- No long-lived secrets in code, CI variables or images. The pipeline and workloads use short-lived, narrowly scoped identities, and secrets are rotated.
- An SBOM can be produced for any release, and the team can answer “are we affected?” for a new critical vulnerability on the same day.
- Services authenticate to each other. Production access is limited, logged and reviewed.
- A recent threat model exists for each critical flow, with owned actions.

- Each user-facing service has SLIs, SLOs and an error budget policy. Alerts are tied to user impact. Traces and structured logs with correlation IDs are available.
- Workloads have requests, limits and correct probes, and are spread across failure zones.
- Infrastructure is created only through reviewed code with policy checks, and drift is detected.
- RPO and RTO are agreed and restores are tested against them. Incidents have blameless reviews whose actions are completed.

4.9 References and further reading

Foundational papers and essays

- Allspaw, J. (2012). Blameless postmortems and a just culture. *Code as Craft* (Etsy engineering blog).
- Saltzer, J. H., & Schroeder, M. D. (1975). The protection of information in computer systems. *Proceedings of the IEEE*, 63(9), 1278–1308.
- Ward, R., & Beyer, B. (2014). BeyondCorp: A new approach to enterprise security. *;login:*, 39(6), 6–11.

Standards and frameworks

- NIST (2020). *Zero Trust Architecture*, SP 800-207.
- NIST (2022). *Secure Software Development Framework (SSDF) Version 1.1*, SP 800-218.
- NIST (2024). *The NIST Cybersecurity Framework (CSF) 2.0*, CSWP 29.
- NIST (2025). *Incident Response Recommendations and Considerations for Cybersecurity Risk Management: A CSF 2.0 Community Profile*, SP 800-61 Rev. 3.
- OpenTelemetry project (CNCF). *OpenTelemetry specification.*
- OWASP Foundation (2025). *OWASP Top 10:2025.* https://top10.owasp.org/
- OWASP Foundation. *Application Security Verification Standard (ASVS).*
- SLSA project (OpenSSF). *Supply-chain Levels for Software Artifacts.*
- SPDX (ISO/IEC 5962:2021) and OWASP CycloneDX. Software bill of materials formats.
- Threat Modeling Manifesto working group (2020). *Threat Modeling Manifesto.*

Practitioner books

- Beyer, B., Jones, C., Petoff, J., & Murphy, N. R. (Eds.) (2016). *Site Reliability Engineering: How Google Runs Production Systems.* O'Reilly. (Free online.)
- Beyer, B., Murphy, N. R., Rensin, D. K., Kawahara, K., & Thorne, S. (Eds.) (2018). *The Site Reliability Workbook.* O'Reilly. (Free online.)
- Humble, J., & Farley, D. (2010). *Continuous Delivery.* Addison-Wesley.

- Morris, K. (2020). *Infrastructure as Code* (2nd ed.). O'Reilly.
- Shostack, A. (2014). *Threat Modeling: Designing for Security.* Wiley.

Standards crosswalk

- IEEE Computer Society (2024). *SWEBOK Guide v4.0*: Software Security, Software Engineering Operations and Software Testing knowledge areas.

PART A · THE PRINCIPLES

The ten principles

These ten principles sit underneath every domain. They are drawn from what PMI, Scrum, Kanban, IEEE SWEBOK, TOGAF and the DORA research programme agree on once their vocabulary is removed (see Appendix C2), plus two that the AI era has made urgent (P8, P9). Exam scenarios often test a principle through a task: an item on `D2-T05` (flow) is usually also an item on `P2` (small batches).

P1. Close the loop

Work in short cycles that end with evidence: a test result, a user's reaction, a production metric. Decide the next step from that evidence, not from the plan alone. *In practice:* a design is checked by a prototype before it is committed to; a sprint ends with working software that someone has looked at; an incident ends with a change, not just a report. *Roots:* Deming's PDCA cycle; Scrum's transparency, inspection and adaptation; DORA's emphasis on fast feedback.

P2. Keep batches small

Smaller pieces of work move faster, fail smaller and are easier to review. Time spent waiting grows with the amount of work in progress. *In practice:* small pull requests, frequent integration to the main branch, work items that fit in a few days, limits on work in progress. *Roots:* Little's Law ($L = \lambda W$); Kanban; trunk-based development in the DORA research.

P3. Separate the "what" from the "how"

Someone owns the outcome and its value; the people doing the work own how it is built. Both are accountable, for different things. *In practice:* a product owner orders the backlog; engineers decide the design; neither silently overrides the other. *Roots:* Scrum accountabilities; PMI sponsor and team roles; Dijkstra's separation of concerns.

P4. Make “done” explicit and checkable

Quality is agreed before work starts and checked by something other than the author’s opinion, ideally by automation. *In practice:* acceptance criteria on every work item; a definition of done; tests and checks that block a merge when they fail. *Roots:* Scrum’s Definition of Done; Kanban’s explicit policies; Deming’s “build quality in”.

P5. Retire risk early

Do the riskiest, least understood part first, while it is still cheap to change course. *In practice:* spikes and prototypes; threat modelling during design, not after release; canary releases before full rollout. *Roots:* Royce (1970), “do it twice”; Brooks (1986) on prototyping; progressive delivery.

P6. Design for human limits

People and teams have limited attention. The structure of a system tends to copy the structure of the organisation that builds it. *In practice:* teams sized so that everyone can talk to everyone; module and service boundaries chosen with team ownership in mind; on-call loads that people can sustain. *Roots:* Brooks’s Law and the n(n−1)/2 communication paths; Conway (1968); Team Topologies.

P7. Make change cheap

Requirements, technology and teams will change. Prefer decisions that are easy to reverse, and invest in the ones that are not. *In practice:* information hiding behind stable interfaces; decisions recorded with their reasons; feature flags that separate deploying from releasing. *Roots:* Parnas (1972); the Agile Manifesto; architecture decision records.

P8. Verify what you did not write

Code, text and data produced by AI tools, libraries, or other teams are inputs to be checked, not facts to be trusted. The person who ships it is responsible for it. *In practice:* AI-generated code goes through the same tests and review as human code; dependencies are scanned and pinned; model outputs are validated against a schema before use. *Roots:* Hoare (1981) on simplicity and verifiability; software supply-chain security practice; the OWASP Top 10 for LLM Applications.

P9. Contain failure

Assume parts of the system will fail and limit how far a failure can spread. Recovering fast matters more than never failing. *In practice:* least privilege; timeouts and circuit breakers; isolated failure domains; practised recovery; error budgets. *Roots:* Saltzer and Schroeder (1975); Armstrong (2003); Google SRE; DORA's recovery-time metric.

P10. Act with integrity and keep it safe to speak up

Be honest about what you know, what you built and what went wrong. Build teams where raising a problem is rewarded, not punished. *In practice:* blameless post-incident reviews; honest estimates and status; protecting user data; declaring conflicts of interest. *Roots:* ACM/IEEE-CS Software Engineering Code of Ethics; Westrum's generative culture as used in DORA; Deming's "drive out fear".

GLOSSARY

Terms used in this domain

Error budget

The amount of failure an SLO allows over its window (e.g. 0.1% of 30 days = 43.2 minutes).

Infrastructure as code (IaC)

Defining infrastructure in versioned, reviewed code that a tool applies, instead of manual changes.

Least privilege

Giving every person and workload only the access its task needs, for only as long as it needs it.

Liveness / readiness probe

Orchestrator health checks: liveness decides whether to restart a container; readiness decides whether to send it traffic.

Recovery point objective (RPO)

The maximum acceptable data loss, measured in time, after a failure.

Recovery time objective (RTO)

The maximum acceptable time to restore service after a failure.

Service level indicator (SLI)

A measure of user experience, such as the share of requests that succeed within a time limit.

Service level objective (SLO)

A target for an SLI over a time window, such as 99.9% over 30 days.

Software bill of materials (SBOM)

A machine-readable list of the components in a piece of software.

STRIDE

Threat categories: Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege.

Trust boundary

A point where data or control passes between parts of a system with different levels of trust.

Zero trust

A security model in which no request is trusted because of its network location; every access is authenticated and authorised.

INDEX

All 56 task statements

The tasks of this guide's domain are shaded. **A** is tested in the ATP®, **C** in the CTP®.

D1-T01	A/C	Choose an interaction style for communication between components.
D1-T02	A	Design and consume HTTP APIs correctly.
D1-T03	A	Decompose code into modules that hide their design decisions.
D1-T04	A	Apply common design patterns where they fit, and avoid them where they do not.
D1-T05	C	Define service and module boundaries for a system and the teams that own it.
D1-T06	C	Design reliable event-driven flows.
D1-T07	A/C	Design and operate caching tiers.
D1-T08	C	Reason about consistency and availability trade-offs in distributed data.
D1-T09	C	Isolate failure domains so a fault in one part does not take down the rest.
D1-T10	A/C	Scale a system to meet demand.
D1-T11	C	Evolve a legacy system incrementally.
D2-T01	A	Use version control to collaborate safely.
D2-T02	A	Raise and review pull requests.
D2-T03	A	Work within an iterative delivery cycle.
D2-T04	A	Write and maintain work items others can act on.
D2-T05	A/C	Manage the flow of work.
D2-T06	C	Measure software delivery performance without turning measures into targets.
D2-T07	C	Plan and coordinate releases across several teams.
D2-T08	C	Record significant technical decisions.
D2-T09	C	Identify, quantify and pay down technical debt.
D2-T10	C	Tailor the delivery approach to the situation.
D3-T01	A	Explain how large language models behave, at the level needed to use them safely.
D3-T02	A	Use AI developer tools within policy.
D3-T03	A	Verify AI-generated code and content before relying on it.
D3-T04	A	Write prompts that are clear, bounded and repeatable.
D3-T05	C	Design retrieval-augmented generation (RAG) systems.
D3-T06	C	Choose and operate a vector store.
D3-T07	C	Evaluate AI features before and after release.

D3-T08	C	Secure AI systems.
D3-T09	C	Integrate AI into production services reliably and affordably.
D3-T10	C	Govern AI use responsibly.
D3-T11	A/C	Explain how machine learning models are trained and evaluated.
D3-T12	C	Choose the right approach for an AI problem.
D3-T13	C	Prepare and govern training data.
D3-T14	C	Fine-tune and adapt pretrained models.
D3-T15	C	Deploy and operate machine learning models.
D4-T01	**A**	**Explain and maintain a CI/CD pipeline.**
D4-T02	**A**	**Handle secrets safely.**
D4-T03	**A**	**Prevent common application vulnerabilities.**
D4-T04	**A/C**	**Manage dependencies and the software supply chain.**
D4-T05	**C**	**Apply least privilege and zero-trust networking.**
D4-T06	**C**	**Threat-model a system.**
D4-T07	**A/C**	**Make systems observable.**
D4-T08	**C**	**Run and scale containerised workloads on an orchestrator.**
D4-T09	**C**	**Manage infrastructure as code.**
D4-T10	**C**	**Respond to incidents and learn from them.**
D5-T01	A	Communicate technical information clearly in writing.
D5-T02	A	Collaborate within a delivery team.
D5-T03	A	Give and receive code review feedback constructively.
D5-T04	A	Act ethically as a practitioner.
D5-T05	C	Lead a squad toward clear outcomes.
D5-T06	C	Mentor and develop other engineers.
D5-T07	C	Align engineering careers with expectations.
D5-T08	C	Build psychological safety and a learning culture.
D5-T09	C	Shape team boundaries to manage cognitive load.
D5-T10	C	Work with stakeholders on scope and trade-offs.

Infrastructure & Security

CI/CD, secrets, vulnerabilities, supply chain, zero trust, threat modelling, observability, orchestration, infrastructure as code and incident response.

Part of the IICTP Library. Every book is free to download at iictp.org.

- Domain Guide 1: Architecture & Systems Design
- Domain Guide 2: Engineering Governance & Agile Delivery
- Domain Guide 3: Applied AI & Emerging Tech Integration
- Domain Guide 5: People Leadership in Engineering
- Illustrative Questions
- Glossary of Technical Practice
- Casebook
- Code of Ethics and Professional Conduct

IICTP · Version 0.2 · 22 October 2026