2

Engineering Governance & Agile Delivery

TPBOK Domain Guide 2 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

Engineering Governance & Agile Delivery

TPBOK Domain Guide 2. 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

Engineering Governance & Agile Delivery

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 2 **The task statements** 7

CHAPTER 2 **Engineering Governance & Agile Delivery** 8

- 2.1 Scope 8
- 2.2 Levels 8
- 2.3 Principles at work 8
- 2.4 Knowledge by task 9
- 2.5 Anti-patterns 17
- 2.6 Worked scenario 18
- 2.7 Illustrative questions 20
- 2.8 Evidence of good practice 22
- 2.9 References and further reading 22

PART A · THE PRINCIPLES **The ten principles** 25

GLOSSARY **Terms used in this domain** 28

INDEX **All 56 task statements** 29

ABOUT THIS GUIDE

How to use this guide

This guide is Chapter 2 of *The Standard for Technical Practitioners* (TPBOK™), published on its own for people who work mainly in Engineering Governance & Agile Delivery. 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 **28%** of the questions. In the CTP® examination it carries **24%**.

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 2

The task statements

Blueprint weight: ATP 28%, CTP 24%. How work is organised, integrated, released, measured and decided on.

ID	Level	Task	Knowledge	Principles
D2-T01	A	Use version control to collaborate safely.	Git model (commits, branches, merges, rebase); branching strategies including trunk-based development; commit hygiene; resolving conflicts.	P2
D2-T02	A	Raise and review pull requests.	Small, focused changes; PR descriptions; review checklists; required checks; review turnaround as a flow problem.	P2, P4
D2-T03	A	Work within an iterative delivery cycle.	Scrum accountabilities, events and artifacts; Kanban practices; Definition of Done; increments.	P1, P3, P4
D2-T04	A	Write and maintain work items others can act on.	User stories; acceptance criteria; splitting work vertically; backlog ordering; ticket hygiene (status, owner, links).	P2, P4
D2-T05	A/C	Manage the flow of work.	Work in progress limits; cycle time; throughput; work item age; Little's Law; bottlenecks.	P2, P6
D2-T06	C	Measure software delivery performance without turning measures into targets.	DORA's delivery metrics (change lead time, deployment frequency, change fail rate, failed deployment recovery time, deployment rework rate); Goodhart's Law; why story-point velocity does not compare across teams.	P1, P10
D2-T07	C	Plan and coordinate releases across several teams.	Release trains and cadence; decoupling deploy from release; feature flags and their lifecycle; progressive delivery; dependency management between teams.	P2, P5, P7
D2-T08	C	Record significant technical decisions.	Architecture decision records (context, decision, consequences, status); when a decision deserves a record; superseding decisions.	P7, P10
D2-T09	C	Identify, quantify and pay down technical debt.	Types of debt (deliberate, accidental, bit rot); measuring cost of delay and change effort; making debt visible in the backlog; negotiating capacity for it.	P1, P7
D2-T10	C	Tailor the delivery approach to the situation.	Predictive, iterative, incremental and adaptive life cycles; Cynefin domains; compliance and regulated contexts; automated controls in place of manual approval boards.	P1, P5

CHAPTER 2

Engineering Governance & Agile Delivery

2.1 Scope

This domain is about how engineering work moves from an idea to running software: how it is described, split, integrated, reviewed, released and measured, and how decisions about it are made and recorded.

In scope: version control, pull requests, iterative delivery cycles (Scrum and Kanban), work items, flow management, delivery metrics, release coordination, architecture decision records, technical debt, and choosing a delivery approach for the situation.

Covered elsewhere:

- The CI/CD pipeline as a technical system (stages, artifacts, environments): Domain 4, `D4-T01`. This chapter covers how teams *use* integration and release.
- Content of architecture decisions: Domain 1. This chapter covers how they are recorded.
- Leading people and stakeholders: Domain 5. This chapter covers the delivery system they work in.

2.2 Levels

	ATP® (28% of 100 items)	CTP® (24% of 150 items)
Typical candidate	Developer working in a team's process: branches, PRs, tickets, sprint events	Tech lead shaping how one or several teams deliver, measure and decide
What items ask	"What is the correct way to...", "What should the developer do next...", "Which ticket is ready..."	"What is the first thing to change in this delivery system...", "How should this be measured...", "Which approach fits this context..."
Tasks tested	D2-T01, T02, T03, T04, T05	D2-T05, T06, T07, T08, T09, T10

2.3 Principles at work

- **P2 Keep batches small** is the central idea of this domain. Small commits, small pull requests, small work items and limited work in progress all shorten the time from idea to feedback.
- **P1 Close the loop**: every practice here exists to get evidence sooner, whether from tests, reviewers, users or production.
- **P4 Make "done" explicit and checkable**: Definitions of Done, acceptance criteria and required checks turn quality from an opinion into an agreement.
- **P3 Separate the "what" from the "how"**: the backlog's order is a product decision; how an item is built is an engineering decision.
- **P10 Act with integrity**: honest status, metrics that are not gamed, and decisions written down with their real reasons.

2.4 Knowledge by task

D2-T01 Use version control to collaborate safely A

What the practitioner does. Uses Git so that the team can integrate often, understand history, and recover from mistakes.

Core knowledge.

- **The model.**
  - A commit is a snapshot with a parent.
  - A branch is a movable pointer to a commit.
  - A merge joins two lines of history. A rebase replays commits onto a new base, rewriting them.
- **Rebase rule of thumb.** Rebasing your own unshared branch is fine and keeps history tidy. Rebasing a branch others have pulled, then force-pushing, rewrites history under them. If you must force-push your own branch, `--force-with-lease` refuses to overwrite work you have not seen.
- **Branching strategies.**
  - *Trunk-based development*: everyone integrates into one main branch at least daily, through short-lived branches or directly. Unfinished work is hidden behind feature flags (`D2-T07`). The DORA research associates it with higher delivery performance.
  - *Long-lived feature branches* delay integration, so conflicts and surprises pile up at the end.
  - *Release branches* are sometimes needed for supporting several versions in parallel. Keep them short-lived and fix forward on main.

- **Commit hygiene.** One logical change per commit; a message that says *why*, not only *what*; no generated files, secrets or large binaries in history.
- **Conflicts.** Integrating often keeps conflicts small. When resolving one, understand both changes; do not just keep "mine".

ATP level. Choose the right Git operation for a situation; explain why a long-lived branch is risky; recover from a common mistake (wrong branch, bad commit) without losing work.

Common mistakes. Force-pushing a shared branch. Branches that live for weeks. Commit messages like "fix" or "wip" on main.

D2-T02 Raise and review pull requests A

What the practitioner does. Proposes changes in a form others can review quickly and well, and reviews others' changes promptly.

Core knowledge.

- **Size.** Reviewers find more problems in small changes and approve them faster. Split large work into a sequence of small PRs, each safe to merge on its own (behind a flag if needed).
- **Description.** What changed, why, how it was tested, and anything the reviewer should look at closely. Link the work item.
- **Required checks.** Automated tests, linting and security scans run on every PR and block merging when they fail (P4). Humans then review what machines cannot: design, clarity, and whether the change does the right thing.
- **Review as flow.** A PR waiting for review is work in progress that delivers nothing. The DORA 2023 report found that teams with faster code reviews had markedly higher software delivery performance. Teams often agree a review turnaround, for example within one working day.
- **What to look for.** Correctness, tests that would fail if the logic broke, security at trust boundaries, readability, and fit with the existing design.

ATP level. Identify what makes a PR hard to review; choose the right response when a reviewer requests changes; know what required checks are for.

Common mistakes. 2,000-line PRs. Approving without reading ("LGTM" culture). Reviews that sit for days. Debating style that a formatter should settle.

D2-T03 Work within an iterative delivery cycle A

What the practitioner does. Takes part in the team's delivery cycle so that each cycle produces something usable and teaches the team something.

Core knowledge.

- **Scrum (Scrum Guide, 2020).**
  - *Accountabilities:* the Product Owner (maximising value, ordering the Product Backlog), the Scrum Master (the team's effectiveness), and the Developers (creating the Increment and meeting the Definition of Done). The team is typically 10 or fewer people.
  - *Events:* the Sprint (one month or less) contains Sprint Planning, the Daily Scrum (15 minutes, for the Developers), the Sprint Review, and the Sprint Retrospective.
  - *Artifacts:* each has a commitment. The Product Backlog has the Product Goal, the Sprint Backlog has the Sprint Goal, and the Increment has the Definition of Done.
- **Kanban (Kanban Guide, 2025).** A strategy for optimising the flow of value through three practices:
  - defining and visualising the workflow;
  - actively managing the items in it;
  - improving it.

  The workflow definition includes when items start and finish, how WIP is controlled, explicit policies and a service level expectation. Kanban works alone or alongside Scrum.
- **Definition of Done.** The shared, checkable quality bar an increment must meet. Work that does not meet it is not done and is not shown as done.
- **Purpose of the events.** Each event is a chance to inspect and adapt (P1). A Sprint Review shows working software to stakeholders and changes what comes next. A status meeting does not.

ATP level. Know the accountabilities, events and artifacts and what each is for; act correctly in a scenario (for example, an item that misses the Definition of Done at Sprint end goes back to the Product Backlog and is not presented as finished).

Common mistakes. Daily Scrums run as status reports to a manager. Sprint Reviews that show slides instead of software. "Done" that means "code written, testing next sprint".

D2-T04 Write and maintain work items others can act on A

What the practitioner does. Writes and keeps work items clear enough that anyone on the team can pick one up, build it and know when it is finished.

Core knowledge.

- **User stories.** A short statement of who needs what and why, which invites a conversation. The INVEST checklist (Wake, 2003) says a good item is Independent, Negotiable, Valuable, Estimable, Small and Testable.

- **Acceptance criteria.** Specific, testable conditions, often written as *Given … When … Then …*. "Fast" is not testable. "Search results appear within 1 second for 95% of queries" is.
- **Splitting vertically.** Split work into thin slices that each deliver something usable end to end (a simple version of the whole feature), not horizontal layers (all the database work, then all the API work). Splitting patterns include by workflow step, by business rule, by data variation, and simple-case-first.
- **Ordering.** The backlog is ordered, not just prioritised, and the top items are the most refined.
- **Ticket hygiene.** Status matches reality; one clear owner; links to the PR, design and related items; blocked items marked as blocked with the reason; stale items closed.

ATP level. Recognise a well-formed item and testable acceptance criteria; split a large item vertically; keep a ticket's state accurate.

Common mistakes. Tickets that are only a title. Acceptance criteria that restate the title. Items "in progress" for weeks with no update.

D2-T05 Manage the flow of work A/C

What the practitioner does. Watches how work moves and acts to keep it moving, instead of only starting new work.

Core knowledge.

- **The four flow measures (Kanban Guide, 2025).**
  - *WIP*: items started but not finished.
  - *Throughput*: items finished per unit of time.
  - *Work item age*: time since an unfinished item started.
  - *Cycle time*: time from start to finish.
- **Little's Law applied to flow.** Average cycle time = average WIP ÷ average throughput. For example, 24 items in progress with 8 finished per week means about 3 weeks per item on average. If throughput stays the same, halving WIP halves the average cycle time.
- **WIP limits.** Capping how many items may be in a state forces the team to finish before starting ("stop starting, start finishing") and exposes bottlenecks where work piles up.
- **Work item age.** This is the early warning. An item older than the team's usual cycle time needs attention now, before it becomes a late item.
- **Bottlenecks.** Throughput of the whole system is set by its slowest step (Goldratt's Theory of Constraints). Improving any other step adds to the queue in front of the bottleneck.

- **Forecasting.** A service level expectation like “85% of items finish within 8 days” comes from historical cycle times, not from guesses.

ATP level. Read a board and say where work is stuck; explain why starting new work while items are blocked makes things slower; apply Little’s Law to simple numbers.
CTP level. Choose WIP limits and policies; diagnose a bottleneck from flow data; forecast delivery from historical throughput rather than from estimates.

Common mistakes. Everyone working on a different item, with nothing finishing. Treating “waiting for review” or “waiting for test” as not real work in progress. Forecasting from a single average.

D2-T06 Measure delivery performance without turning measures into targets

C

What the practitioner does. Measures how well the delivery system performs, uses the numbers to find improvements, and protects them from being gamed.

Core knowledge.

- **The DORA software delivery metrics.** DORA’s research (Forsgren, Humble and Kim, 2018, and the annual reports since) measures:
  - *change lead time*: from commit to running in production;
  - *deployment frequency*;
  - *change fail rate*: the share of deployments that need immediate remediation;
  - *failed deployment recovery time*.

  The 2024 report added *deployment rework rate*: the share of deployments that are unplanned and happen because of an incident in production. DORA reports these as measures of throughput and instability, and has repeatedly found that speed and stability go together rather than trading off.
- **Measure applications and services, not people.** DORA itself compares performance within the same application over time, not across teams with different contexts.
- **Goodhart’s Law.** When a measure becomes a target, it stops being a good measure. Deployment frequency can be inflated by splitting deployments, and change fail rate can be lowered by not calling failures failures.
- **Story points and velocity.** These are a team’s private planning aid. Each team calibrates them differently, so they cannot be compared across teams or used as a productivity measure.
- **Use measures in pairs.** Pair a speed measure with a stability measure, and a team measure with a qualitative check such as a retrospective or a developer survey.

CTP level. Design a measurement approach for a team or group of teams, interpret DORA metrics in context, and respond to a request to misuse metrics (for example, ranking teams or tying bonuses to them).

Common mistakes. Leaderboards of teams by velocity or deployment count. Measuring only speed. Reporting metrics nobody acts on.

D2-T07 Plan and coordinate releases across several teams C

What the practitioner does. Gets changes from several teams into users' hands safely and predictably, without making every team wait for the slowest.

Core knowledge.

- **Deploy is not release.** *Deploying* puts code into production. *Releasing* makes a feature available to users. Separating them lets teams deploy continuously and release when the business is ready.
- **Feature flags.** Common kinds include:
  - *release* flags, which hide unfinished work;
  - *experiment* flags, for A/B tests;
  - *operational* flags, kill switches;
  - *permission* flags, which control access for user groups.

  Every flag is a branch in the code. Give release flags an owner and a removal date, or they become permanent debt.
- **Progressive delivery.** Release to a small share of users or a canary group first, watch metrics, then widen. Roll back or switch the flag off if signals worsen (P5, P9).
- **Release trains.** A fixed release cadence (for example, every two weeks) where scope varies but the date does not. Teams catch the train or wait for the next. This makes coordination predictable when continuous release is not yet possible. Some scaling frameworks (for example SAFe's Agile Release Train) build their structure around this idea. The cadence should shorten as automation and confidence improve.
- **Cross-team dependencies.** Make them visible, reduce them through better boundaries (`D1-T05`, `D5-T09`), and use contract tests and backward-compatible changes so teams do not need to release in lockstep.
- **Backward compatibility.** Expand-and-contract (parallel change):
  1. Add the new field, endpoint or column alongside the old one.
  2. Migrate the consumers.
  3. Remove the old one.

CTP level. Plan a release involving several teams; choose between a train, continuous release and progressive rollout for a context; design the flag and rollback strategy.

Common mistakes. “Big-bang” release weekends. Flags never removed. Breaking API changes that force coordinated deployments.

D2-T08 Record significant technical decisions C

What the practitioner does. Writes down important decisions so that later readers know what was decided, why, and when to revisit it.

Core knowledge.

- **Architecture decision records (Nygard, 2011).** A short document per decision, usually one or two pages, with:
  - *Title*;
  - *Status*: proposed, accepted, deprecated or superseded;
  - *Context*: the forces and constraints at the time;
  - *Decision*;
  - *Consequences*: good and bad.
- **Kept with the code.** ADRs are typically stored in the repository, numbered, and reviewed like code.
- **Immutable once accepted.** A changed decision gets a new ADR that supersedes the old one. History shows how thinking evolved.
- **What deserves one.** Decisions that are expensive to reverse, affect several teams, or that someone will later ask “why did we do this?” about. For example: choice of database, service boundaries, an event format, the authentication approach. Not every library upgrade.
- **Honest context.** Record the real reasons, including deadlines and skills on hand, and the options rejected. “We chose X because the team knew it and we had six weeks” is a legitimate, useful record.

CTP level. Decide which decisions need an ADR, write one with real alternatives and consequences, and use ADRs to settle repeated debates.

Common mistakes. ADRs written after the fact to justify a decision. Long documents nobody reads. Editing accepted ADRs instead of superseding them.

D2-T09 Identify, quantify and pay down technical debt C

What the practitioner does. Makes technical debt visible in terms the business can weigh, and gets a sustainable share of capacity to reduce it.

Core knowledge.

- **The metaphor (Cunningham, 1992).** Shipping code that reflects an incomplete understanding is like borrowing. It is fine if you repay it by refactoring as you learn. The interest is the extra effort every later change costs.
- **Kinds of debt.** Fowler's quadrant (2009) separates deliberate from inadvertent debt and prudent from reckless debt. "We must ship now and will fix X next" is prudent and deliberate. "What's layering?" is reckless and inadvertent. Bit rot, where the world changes around code that does not, is a further source.
- **Quantifying it.** Translate debt into cost the business already understands:
  - time lost per week (flaky tests, slow builds, manual steps);
  - extra lead time on changes in a debt-heavy area;
  - incident frequency;
  - risk exposure (unsupported versions, known vulnerabilities).

  Compare that with the cost of fixing it.
- **Cost of delay (Reinertsen, 2009).** Prioritise by how much value is lost per unit of time while the item waits, divided by its duration. This puts debt work and feature work on the same scale.
- **Paying it down.** Debt items go in the same backlog, sized and ordered with features. Other practices: a steady capacity allocation, the "boy scout rule" for small improvements as you go, and targeted investment where change is frequent. Debt in code nobody touches costs little.

CTP level. Quantify a described debt in time or money, argue for a specific allocation, and choose which debt to pay first.

Common mistakes. "Tech debt sprints" once a year instead of steady work. Calling every dislike "debt". Refactoring stable code nobody changes.

D2-T10 Tailor the delivery approach to the situation C

What the practitioner does. Chooses how much planning, iteration and control a piece of work needs, rather than applying one process everywhere.

Core knowledge.

- **Life cycles.**
  - *Predictive*: plan fully, then execute. Fits stable, well-understood requirements.
  - *Iterative*: refine through repeated versions.
  - *Incremental*: deliver in usable pieces.
  - *Adaptive* (agile): iterative and incremental with frequent re-planning.

  Hybrids are common.
- **Cynefin (Snowden and Boone, 2007).** Helps match the response to the situation:
  - *Clear*: cause and effect are obvious. Apply best practice.

  - *Complicated*: experts can analyse it. Use good practice and analysis.
  - *Complex*: cause and effect are only clear in hindsight. Run safe-to-fail experiments and amplify what works.
  - *Chaotic*: act first to stabilise, then sense.
- **Regulated contexts.** Regulations usually require evidence of review, authorisation, testing and traceability, not a particular meeting. Version control, required PR approvals by someone other than the author, automated test results and deployment logs can provide that evidence continuously. Confirm with your compliance function what is required.
- **Change approval boards.** DORA's research found that approval by an external body was associated with lower delivery performance and was not associated with fewer failures. Peer review plus automated checks performed better. A common pattern: automated approval for standard, low-risk changes; human approval kept for genuinely high-risk ones.

CTP level. Choose and justify a delivery approach for a described context, including regulated environments; propose how to replace manual gates with automated controls without losing required evidence.

Common mistakes. Running Scrum for incident response. Fixed-scope, fixed-date plans for genuinely novel work. Keeping a weekly approval meeting because "the regulator requires it" without checking what is actually required.

2.5 Anti-patterns

Anti-pattern	Symptom	Why it hurts	Remedy	Task
Water-Scrum-Fall	Sprints for building, but separate months of requirements before and testing after	Feedback still arrives at the end; "agile" in name only	Definition of Done that includes testing; deliver thin vertical slices to real users	D2-T03, D2-T10
Merge hell	Long-lived branches merged just before release	Conflicts and integration bugs found at the worst moment	Trunk-based development; feature flags	D2-T01, D2-T07
PR queue	PRs wait days for review	Work in progress grows; authors context-switch; cycle time stretches	Small PRs; review before starting new work; agreed turnaround	D2-T02, D2-T05
Velocity as productivity	Teams ranked or rewarded by story points	Points inflate; estimates stop being useful; trust drops	Keep velocity private to the team; use flow and DORA measures in context	D2-T06
Everything in progress	Each person has several items started; few finish	Long cycle times; late surprises	WIP limits; finish before starting	D2-T05

Anti-pattern	Symptom	Why it hurts	Remedy	Task
Flag graveyard	Hundreds of old flags in the code	Untested combinations; confusing behaviour; risk of wrong switch	Owner and removal date per release flag; regular clean-up	D2-T07
Approval theatre	Weekly approval meeting signs off changes nobody in the room understands	Slows delivery without reducing failures	Peer review plus automated checks; human approval only for high-risk changes	D2-T10
Invisible debt	Debt discussed in corridors, never in the backlog	Cannot be prioritised; grows until it forces a rewrite	Quantify it; put it in the backlog; allocate capacity	D2-T09

2.6 Worked scenario

Situation (CTP level). A payments software company has four product teams, about 30 engineers, working on one platform. They release once a month:

- Each team works on long-lived feature branches for most of the month.
- In the last week, everything is merged into a release branch, tested manually, and taken to a weekly change approval meeting.
- Releases often slip by days. Hotfixes are frequent and stressful.
- Management has started comparing teams by velocity.
- The company is regulated, and the head of compliance says “every production change must be approved”.

The new tech lead is asked how to “release faster without breaking compliance”.

Step 1: Establish a baseline (D2-T06, P1). Before changing anything, the tech lead pulls three months of data from Git, the ticket system and the deployment log:

- *Change lead time*: about 4 to 6 weeks.
- *Deployment frequency*: monthly, plus about 6 hotfixes a month.
- *Change fail rate*: about 1 in 4 releases needs a hotfix.
- *Failed deployment recovery time*: often more than a day, because a fix must go through the same approval route.

These numbers are shared with the teams as a picture of the system, not as a scorecard.

Step 2: Find where the time goes (D2-T05). The combined board shows about 60 items in progress and about 15 finished per week.

	Average WIP	Throughput	Average cycle time (WIP ÷ throughput)
Today	60 items	15 / week	4 weeks
After WIP limits	30 items	15 / week	about 2 weeks, if throughput holds

Item ages show the biggest queues are “waiting for review” and “waiting for release testing”. The tech lead introduces WIP limits per team and a rule that reviewing comes before starting new work.

Step 3: Integrate continuously (D2-T01, D2-T02, D2-T07).

- Teams move to short-lived branches merged to main at least daily.
- Unfinished features are hidden behind release flags, each with an owner and a removal date.
- PRs are kept small.
- Required checks (unit and integration tests, static analysis, dependency scan) run on every PR.

Merge hell disappears because there is no longer a big merge.

Step 4: Meet compliance with evidence, not a meeting (D2-T10). The tech lead and the head of compliance read the actual requirement together. It requires that every change is reviewed and authorised by someone other than its author, tested, and traceable. It does not name a meeting. The new process:

- **Standard changes:** a PR approved by a second engineer, green required checks, and an automated record linking ticket, PR, approver, test results and deployment. This becomes the approval evidence, and it is more complete than the meeting minutes were.
- **High-risk changes** (schema changes to the ledger, changes to authentication): human sign-off from a named owner, kept as before.
- **Audit:** an internal auditor samples the automated records each quarter.

Step 5: Release on a cadence, then faster (D2-T07). The teams first move to a two-week release train, then to deploying daily with release controlled by flags. Payment features roll out to 5% of merchants first, and the team watches error rates before widening.

Step 6: Quantify the worst debt (D2-T09). The integration test suite is flaky. About 10 pipeline runs a day fail for no real reason, and each rerun costs about 20 minutes of an engineer waiting or context-switching.

	Time lost
Per day	10 × 20 min = 200 engineer-minutes, about 3.3 hours
Per week	about 17 hours, roughly 0.4 of a full-time engineer

Flaky tests also teach people to ignore red builds, which is a quality risk. The tech lead asks for two engineers for three weeks to stabilise the suite, and shows that this pays back in under four months on time alone.

Step 7: Stop the velocity comparison (D2-T06, P10). The tech lead explains to management why points cannot be compared across teams, and offers instead the DORA metrics per service, tracked over time, alongside a short quarterly developer survey.

Step 8: Record the decisions (D2-T08). ADRs record:

1. The move to trunk-based development with feature flags.
2. The automated change evidence process, with the compliance requirement quoted in the context.
3. The flag lifecycle policy.

What a weaker answer looks like.

- "Adopt a scaling framework across the company." This adds structure without addressing the queues, the branches or the evidence problem.
- "Remove the approval step." This ignores a real regulatory requirement instead of meeting it better.
- "Ask teams to increase velocity by 20%." Points inflate, and nothing actually speeds up.

Mini-scenario (ATP level). A developer has worked on a feature branch for three weeks. The PR has 1,800 changed lines and conflicts with main, and two reviewers have not started it after four days. *What should the developer have done differently, and what should they do now?*

- **Differently:** integrate with main daily and deliver the feature as a series of small PRs, with unfinished parts hidden behind a flag.
- **Now:** split the PR into smaller pieces that can each be reviewed and merged on their own, starting with the least risky (D2-T01, D2-T02).

2.7 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. `D2-T01` · **ATP** You pushed a feature branch yesterday, and a colleague has since pulled it and added commits. You now need the latest changes from main in your branch. What is the safest approach?

- A. Rebase your branch onto main and force-push it
- B. Merge main into your branch

- C. Delete the branch and create a new one from main
- D. Commit your work directly to main

Answer: B.

- **A** rewrites history your colleague has already pulled.
- **C** loses their work.
- **D** bypasses review.
- **B** brings main's changes into the branch without rewriting anything others depend on.

Q2. `D2-T04` · **ATP** Which acceptance criterion is testable as written?

- A. The search page should be fast
- B. Search should be user-friendly
- C. Given 10,000 products, when a user searches by product name, then matching results appear within 1 second for 95% of searches
- D. Search should work like competitors' search

Answer: C. It names the conditions, the action and a measurable outcome. The others cannot be checked objectively.

Q3. `D2-T05` · **ATP** A team has on average 24 items in progress and finishes on average 8 items a week. Assuming the team's process is stable, what is the average time an item takes from start to finish?

- A. 1 week
- B. 3 weeks
- C. 8 weeks
- D. 32 weeks

Answer: B. Average cycle time = WIP ÷ throughput = 24 ÷ 8 = 3 weeks.

Q4. `D2-T06` · **CTP** A director proposes ranking the six teams by deployment frequency each month and linking part of their bonus to it. What is the most appropriate response from a tech lead?

- A. Agree, since deployment frequency is a validated DORA metric
- B. Agree, but rank by velocity instead
- C. Advise against it: a measure used as a target will be gamed (for example, by splitting deployments), and the teams' contexts differ. Instead, track each service's metrics over time, pair speed with stability measures, and use them to find improvements
- D. Stop measuring delivery performance altogether

Answer: C.

- **A** ignores Goodhart's Law and the differences in context.
- **B** is worse, because velocity is not comparable across teams.
- **D** gives up useful information.
- **C** keeps the measures and removes the incentive to game them.

Q5. `D2-T10` · **CTP** A team is building a product for a market the company has never served. Nobody can say with confidence which features customers will pay for. Which approach best fits?

- A. A detailed 12-month plan with fixed scope, approved up front
- B. Short cycles of small, safe-to-fail experiments with real customers, expanding what works and dropping what does not
- C. Copy the delivery plan of the company's most successful existing product
- D. Wait until market research gives a complete specification

Answer: B. This is a complex situation in Cynefin terms: cause and effect only become clear through trying. Probing with experiments fits it. **A** and **C** assume knowledge the team does not have, and **D** delays the learning that would provide it.

2.8 Evidence of good practice

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

- Main is integrated at least daily. Branches live hours or a couple of days, not weeks.
- PRs are small, have descriptions, pass required checks, and are reviewed within an agreed time.
- The board reflects reality, WIP is limited, and the team can say how old its oldest item is.
- Work items have testable acceptance criteria, and the Definition of Done is written down and applied.
- DORA metrics, or an equivalent, are tracked per service over time and used in retrospectives, not in rankings.
- Release flags have owners and removal dates, and the flag count is not growing without limit.
- Significant decisions have ADRs. Technical debt is in the backlog with an estimated cost.
- In regulated settings, change evidence is produced automatically and an auditor could follow it from ticket to deployment.

2.9 References and further reading

The research archive (`books_and_standards/pdfs`) holds the Scrum Guide 2020, Scrum Primer, Scrum@Scale Guide, Kanban Guide 2025 and DORA 2022 and 2023 reports. The Scrum Guide is published under Creative Commons Attribution-ShareAlike 4.0. The Kanban Guide 2025 names CC BY-SA 4.0 in its opening and CC BY 4.0 in its closing licence section. This chapter paraphrases both and quotes neither.

Primary guides and research

- DORA / Google Cloud (2023). *Accelerate State of DevOps Report 2023.*
- DORA / Google Cloud (2024). *Accelerate State of DevOps Report 2024.* (Introduces deployment rework rate.)
- DORA. *DORA's software delivery performance metrics* and *A history of DORA's software delivery metrics.* https://dora.dev/ (read 21 October 2026).
- Forsgren, N., Humble, J., & Kim, G. (2018). *Accelerate: The Science of Lean Software and DevOps.* IT Revolution.
- Schwaber, K., & Sutherland, J. (2020). *The Scrum Guide.*
- Vacanti, D. S., & Orderly Disruption Limited (2025). *The Kanban Guide.*

Foundational papers and essays

- Cunningham, W. (1992). The WyCash portfolio management system. OOPSLA '92 Experience Report.
- Fowler, M. (2009). TechnicalDebtQuadrant. martinfowler.com.
- Hodgson, P. (2017). Feature toggles (aka feature flags). martinfowler.com.
- Nygard, M. T. (2011). Documenting architecture decisions. cognitect.com blog.
- Royce, W. W. (1970). Managing the development of large software systems. *Proceedings of IEEE WESCON.*
- Snowden, D. J., & Boone, M. E. (2007). A leader's framework for decision making. *Harvard Business Review*, 85(11), 68–76.
- Takeuchi, H., & Nonaka, I. (1986). The new new product development game. *Harvard Business Review*, 64(1), 137–146.
- Wake, B. (2003). INVEST in good stories, and SMART tasks. xp123.com.

Practitioner books

- Goldratt, E. M. (1984). *The Goal.* North River Press.
- Humble, J., & Farley, D. (2010). *Continuous Delivery.* Addison-Wesley.
- Reinertsen, D. G. (2009). *The Principles of Product Development Flow.* Celeritas.
- Vacanti, D. S. (2015). *Actionable Agile Metrics for Predictability.* ActionableAgile Press.

Standards crosswalk

- IEEE Computer Society (2024). *SWEBOK Guide v4.0*: Software Engineering Management, Software Engineering Process, Software Configuration Management and Software Engineering Economics 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

Architecture decision record (ADR)

A short, numbered record of one significant decision: its context, the decision, and its consequences. Superseded, not edited.

Change fail rate

The share of deployments that cause a failure needing immediate remediation (DORA).

Cost of delay

The value lost per unit of time while an item waits to be delivered.

Cycle time

Elapsed time from when a work item starts to when it finishes.

Definition of Done

The shared, checkable quality bar an increment must meet before it counts as done.

Expand and contract

Making a breaking change safely: add the new form beside the old, migrate consumers, then remove the old.

Feature flag

A runtime switch that turns behaviour on or off without deploying new code.

Release train

A fixed release cadence where the date is fixed and the scope varies.

Service level expectation (SLE)

A forecast of how long a work item should take, with a probability (e.g. 85% within 8 days).

Technical debt

The extra future effort caused by code or design that no longer fits what is now known.

Trunk-based development

All developers integrate into one main branch at least daily, using short-lived branches if any.

Work item age

Elapsed time since an unfinished work item started.

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.

Engineering Governance & Agile Delivery

Git and pull requests, Scrum and Kanban, flow, DORA metrics, release trains, decision records, technical debt and tailoring delivery.

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

- Domain Guide 1: Architecture & Systems Design
- Domain Guide 3: Applied AI & Emerging Tech Integration
- Domain Guide 4: Infrastructure & Security
- 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