1

Architecture & Systems Design

TPBOK Domain Guide 1 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

Architecture & Systems Design

TPBOK Domain Guide 1. 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

Architecture & Systems Design

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 1 **The task statements** 7

CHAPTER 1 **Architecture & Systems Design** 8

1.1 Scope 8
1.2 Levels 8
1.3 Principles at work 9
1.4 Knowledge by task 9
1.5 Anti-patterns 18
1.6 Worked scenario 18
1.7 Illustrative questions 21
1.8 Evidence of good practice 22
1.9 References and further reading 23

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 1 of *The Standard for Technical Practitioners* (TPBOK™), published on its own for people who work mainly in Architecture & Systems Design. It holds the domain's 11 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 **24%** of the questions. In the CTP® examination it carries **30%**.

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 1

The task statements

Blueprint weight: ATP 24%, CTP 30%. How software systems are divided, connected and made to behave correctly under load and failure.

ID	Level	Task	Knowledge	Principles
D1-T01	A/C	Choose an interaction style for communication between components.	Client-server; synchronous request/response; asynchronous messaging; publish/subscribe; event streams; coupling in time and in space.	P7, P9
D1-T02	A	Design and consume HTTP APIs correctly.	Resources and representations; HTTP methods and their safety and idempotency; status codes; versioning; pagination; error formats; API contracts.	P4, P7
D1-T03	A	Decompose code into modules that hide their design decisions.	Information hiding; cohesion and coupling; encapsulation; composition over inheritance; pure functions and immutability; interface stability.	P7
D1-T04	A	Apply common design patterns where they fit, and avoid them where they do not.	Creational, structural and behavioural patterns (e.g. factory, adapter, strategy, observer); dependency inversion; speculative generality as an anti-pattern.	P7
D1-T05	C	Define service and module boundaries for a system and the teams that own it.	Bounded contexts; monolith, modular monolith and microservices trade-offs; Conway's Law; data ownership per service.	P3, P6
D1-T06	C	Design reliable event-driven flows.	Delivery guarantees (at-most-once, at-least-once, effectively-once); idempotent consumers; transactional outbox; dead-letter queues; ordering and partitioning; event schemas and evolution.	P4, P9
D1-T07	A/C	Design and operate caching tiers.	Cache-aside, read-through, write-through, write-behind; TTL and invalidation; cache stampede; consistency between cache and source; client, CDN, application and database caches.	P9
D1-T08	C	Reason about consistency and availability trade-offs in distributed data.	CAP and PACELC; consistency models (strong, eventual, read-your-writes); replication; logical clocks and causal order; consensus at concept level.	P5, P9
D1-T09	C	Isolate failure domains so a fault in one part does not take down the rest.	Timeouts; retries with exponential backoff and jitter; circuit breakers; bulkheads; graceful degradation; supervision and restart; cascading failure.	P9
D1-T10	A/C	Scale a system to meet demand.	Vertical and horizontal scaling; stateless services; partitioning and sharding; load balancing; back-pressure; queueing and Little's Law; tail latency; capacity estimation.	P2, P9
D1-T11	C	Evolve a legacy system incrementally.	Strangler fig; anti-corruption layer; branch by abstraction; parallel run; data migration strategies.	P2, P5, P7

CHAPTER 1

Architecture & Systems Design

Template note. Every Part B chapter uses these sections in this order: 1.1 Scope, 1.2 Levels, 1.3 Principles at work, 1.4 Knowledge by task, 1.5 Anti-patterns, 1.6 Worked scenario, 1.7 Illustrative questions, 1.8 Evidence of good practice, 1.9 References.

1.1 Scope

This domain is about how a software system is divided into parts, how those parts talk to each other, and how the whole keeps working when load rises or a part fails.

In scope: interaction styles, API design, modular decomposition, design patterns, service boundaries, event-driven design, caching, distributed data trade-offs, failure isolation, scaling, and evolving legacy systems.

Covered elsewhere:

- *Recording* architecture decisions (ADRs): Domain 2, `D2-T08`.
- Running the system on infrastructure (containers, Kubernetes, observability): Domain 4.
- Architecture of AI features (RAG, vector stores): Domain 3. This chapter's ideas still apply to them.
- Security design (threat modelling, zero trust): Domain 4. This chapter covers the structural side: boundaries and failure isolation.

1.2 Levels

	ATP® (24% of 100 items)	CTP® (30% of 150 items)
Typical candidate	Early-career developer building features inside a system someone else designed	Tech lead or senior engineer responsible for how a system is shaped
What items ask	"What happens if…", "Which is the correct use of…", "What is wrong with this API/design…"	"Which design would you choose and why…", "What is the first thing to change…", "Which trade-off is acceptable here…"

	ATP® (24% of 100 items)	CTP® (30% of 150 items)
Tasks tested	D1-T01, T02, T03, T04, T07, T10	D1-T01, T05, T06, T07, T08, T09, T10, T11

1.3 Principles at work

- **P7 Make change cheap** is the main reason architecture exists. A good structure lets the most likely changes happen in one place.
- **P9 Contain failure** shapes every decision about distribution. Once a system has more than one process, partial failure is normal.
- **P6 Design for human limits**: boundaries between parts of a system are also boundaries between people. A design that ignores who owns what will be bent back into the shape of the organisation (Conway, 1968).
- **P5 Retire risk early**: the expensive architecture mistakes are the ones found late. Prototype the uncertain part first.
- **P2 Keep batches small** applies to migrations: move one capability at a time.

1.4 Knowledge by task

D1-T01 Choose an interaction style A/C

What the practitioner does. Decides how two components communicate, knowing what each choice couples together.

Core knowledge.

- **Synchronous request/response** (e.g. HTTP, gRPC). The caller waits. Simple to reason about and debug. Couples the two parties *in time*: if the callee is slow or down, so is the caller.
- **Asynchronous messaging** (a queue). The sender hands work to a broker and carries on. Removes time coupling, and the broker absorbs bursts. Costs: eventual completion, harder debugging, the need to handle duplicates.
- **Publish/subscribe and event streams.** A producer announces that something happened; any number of consumers react. The producer does not need to know who listens, which lowers coupling *in space*. A log-based stream also keeps history, so new consumers can replay it.
- **Commands vs events.** A command asks for something to happen (“ReserveStock”) and has one intended handler. An event says something has happened (“OrderPlaced”) and can have many. Mixing the two up is a common source of hidden coupling.

ATP level. Recognise each style, and say which one a scenario is using and what the caller experiences when the other side is slow. **CTP level.** Choose a style for a given flow and defend it: what failure looks like, what consistency the user sees, how the choice affects team independence.

Common mistakes. Using a queue to hide a request that really needs an answer now. Chaining many synchronous calls where each one adds latency and a failure point. Publishing "events" that are really commands aimed at one service.

D1-T02 Design and consume HTTP APIs A

What the practitioner does. Builds and calls HTTP APIs that behave predictably for clients, including when requests are retried or fail.

Core knowledge.

- **Resources and methods.** Under HTTP semantics (RFC 9110), GET, HEAD and OPTIONS are *safe* (they do not change state). PUT and DELETE are *idempotent* (repeating them has the same effect as doing them once). POST is neither by default.
- **Idempotency in practice.** Networks time out, and clients retry. For operations that are not naturally idempotent, such as creating a payment, the client sends a unique idempotency key, often in a request header. The server stores the key with the result and returns the same result if the key arrives again.
- **Status codes that matter.** 200 OK, 201 Created, 204 No Content; 400 Bad Request; 401 (not authenticated) vs 403 (authenticated but not allowed); 404; 409 Conflict; 422 Unprocessable Content; 429 Too Many Requests; 500 vs 503 (503 tells the client the problem is temporary).
- **Error bodies.** Return a consistent, machine-readable error format. RFC 9457 ("Problem Details for HTTP APIs") is a widely used standard.
- **Versioning and compatibility.** Adding optional fields is usually safe. Removing or renaming fields, or changing their meaning, breaks clients. Version deliberately and keep old versions alive for a stated period.
- **Pagination.** Offset pagination is simple but skips or repeats items when data changes and gets slow on deep pages. Cursor pagination is stable and fast for large or changing collections.
- **Contracts.** Describe the API in a machine-readable specification (for example OpenAPI) and test against it.

ATP level. Pick the correct method and status code; spot a non-idempotent operation being retried unsafely; read and follow an API contract.

Common mistakes. State-changing GET requests. Returning 200 with an error message in the body. Breaking changes without a new version. Retrying POST without an idempotency key.

D1-T03 Decompose code into modules that hide their design decisions A

What the practitioner does. Structures code so that each likely change stays inside one module.

Core knowledge.

- **Information hiding (Parnas, 1972).** Do not divide a system by the steps of its flowchart. Divide it by the decisions likely to change: a data format, an external provider, a business rule. Put each decision in one module and expose an interface that does not reveal it.
- **Cohesion and coupling.** High cohesion means a module's parts belong together. Low coupling means modules depend on each other through small, stable interfaces.
- **Object-oriented fundamentals.** Encapsulation (state is changed only through the object's own operations); polymorphism (callers depend on a behaviour, not a concrete type); composition generally preferred over deep inheritance trees, which couple subclasses to their parents' internals.
- **Functional fundamentals.** Pure functions (same input, same output, no side effects) are easy to test and reason about. Immutable data cannot be changed underneath you by another part of the program. A common structure is "functional core, imperative shell": pure logic inside, input/output at the edges.
- **Simplicity.** Hoare (1981) argued that a design should be simple enough to have obviously no deficiencies, rather than complicated enough to have no obvious ones.

ATP level. Given two designs, say which one localises a stated change. Identify encapsulation leaks (for example, exposing an internal list that callers then modify).

Common mistakes. Modules named after technical layers only ("utils", "helpers", "manager") with no hidden decision. Getters and setters for every field, which hides nothing. Inheritance used only to reuse code.

D1-T04 Apply design patterns where they fit A

What the practitioner does. Recognises recurring design problems and uses known solutions, but only when the problem is actually present.

Core knowledge.

- **Patterns as vocabulary.** The catalogue by Gamma, Helm, Johnson and Vlissides (1994) groups patterns as creational (e.g. factory, builder), structural (e.g. adapter, facade, decorator) and behavioural (e.g. strategy, observer, command).
- **Patterns practitioners meet most:**
  - *adapter*, to wrap a third-party API behind your own interface (information hiding again);

  - *strategy*, to swap an algorithm or rule without changing its callers;
  - *observer* / publish-subscribe, to notify interested parties without knowing who they are;
  - *facade*, to give a simple entry point to a complicated subsystem.
- **Dependency inversion.** High-level policy should not depend on low-level details. Both should depend on an abstraction owned by the high-level side. This is what makes a payment provider or database replaceable and testable.
- **When not to.** A pattern adds indirection. Without a second real use case or a real expected change, it is speculative generality: cost now for a benefit that may never come.

ATP level. Name the pattern in a code sample; choose the pattern that fits a described problem; recognise when a pattern is unnecessary.

Common mistakes. Factories for objects that have one implementation. Interfaces with exactly one implementer and no test or replacement need. Calling any class with "Manager" in its name a pattern.

D1-T05 Define service and module boundaries C

What the practitioner does. Decides where the main seams of a system go, and whether those seams are module boundaries inside one deployable or network boundaries between services.

Core knowledge.

- **Bounded contexts (Evans, 2003).** In a large domain, the same word means different things to different parts of the business ("customer" in billing vs in support). Each bounded context has its own model and language, and contexts integrate through explicit translation.
- **Three structural options:**

Option	Strengths	Costs	Fits when
Monolith	One deployable; simple transactions, debugging and refactoring	Everything deploys together; one team's change can break another's; scales as one unit	Small teams; a domain still being learned
Modular monolith	One deployable with enforced internal boundaries; boundaries are cheap to move	Needs discipline (or tooling) to stop modules reaching into each other	Most growing products; a step before extracting services
Microservices	Independent deployment and scaling; team autonomy; fault isolation	Network failures, eventual consistency, distributed debugging, operational overhead	Several teams that need to release independently, parts with very different scaling or compliance needs, and a mature delivery platform

- **Data ownership.** Each service owns its data, and other services do not read its tables directly. Services that share one database are coupled at the schema, whatever the deployment diagram says.
- **Conway's Law (1968).** A system's structure tends to mirror the communication structure of the organisation that designs it. Use it deliberately: shape teams around the boundaries you want (sometimes called the "inverse Conway manoeuvre"; see also `D5-T09`).

CTP level. Given a domain description and a team structure, propose boundaries and justify them; say when *not* to split; identify a boundary that will cause chatty cross-service calls or shared data.

Common mistakes. Splitting by technical layer (a "database service", a "UI service"). Splitting before the domain is understood. Splitting into more services than there are teams to own them.

D1-T06 Design reliable event-driven flows C

What the practitioner does. Designs flows where services react to events, so that messages are not lost, are not processed twice in a harmful way, and stay in the order that matters.

Core knowledge.

- **Delivery guarantees.** *At-most-once*: may lose messages. *At-least-once*: may deliver duplicates; the usual default. *Exactly-once*: true end-to-end exactly-once is rarely available across systems. In practice teams build "effectively once" from at-least-once delivery plus idempotent processing.
- **Idempotent consumers.** Each message carries a unique ID. The consumer records processed IDs (or makes its update naturally idempotent, for example "set status = shipped" rather than "increment count") so a duplicate has no extra effect.
- **The dual-write problem and the transactional outbox.** A service that updates its database and then publishes to a broker can crash between the two steps, leaving state and events in disagreement. With an outbox, the service writes the event into an outbox table *in the same transaction* as the state change. A separate relay, either polling or change-data-capture, publishes from the outbox. Publishing *before* committing is worse: it can announce something that then rolls back.
- **Dead-letter queues.** After a bounded number of failed attempts, move the message aside so it stops blocking others, then alert and inspect. A DLQ that nobody watches is data loss with extra steps.
- **Ordering.** Log-based brokers typically guarantee order only within a partition. Choose the partition key so that events that must stay in order (all events for one order, for example) share it.

- **Schema evolution.** Events are a public contract. Add fields compatibly, version when meaning changes, and consider a schema registry once many teams depend on them.

CTP level. Identify the failure in a described flow (lost event, duplicate effect, out-of-order processing) and choose the design that fixes it.

Common mistakes. Assuming the broker gives exactly-once end to end. Retrying forever and blocking a partition. Using a random partition key and then relying on order.

D1-T07 Design and operate caching tiers A/C

What the practitioner does. Uses caches to cut latency and load without serving data that is wrong in ways that matter.

Core knowledge.

- **Where caches live.** Client and browser; CDN/edge; application (in-process or a shared cache service); database buffer caches. Each layer has different invalidation powers.
- **Patterns:**
  - *Cache-aside*: the application reads the cache, and on a miss loads from the source and fills the cache. The most common pattern.
  - *Read-through*: the cache itself loads on a miss.
  - *Write-through*: writes go to the cache and the source together. Reads stay fresh, but writes are slower.
  - *Write-behind*: writes go to the cache and reach the source later. Writes are fast, but data can be lost on failure.
- **Invalidation.** With cache-aside, a common safe default is to write the source first, then *delete* the cache entry (not update it), with a TTL as a safety net against missed deletes.
- **Stampede (thundering herd).** When a popular entry expires, many requests miss at once and hit the source together. Mitigations: request coalescing (one loader, others wait), a lock per key, slightly randomised or early expiry.
- **Deciding what to cache.** Ask: how stale may this be, and what happens if it is? A product description can be minutes old. A stock count shown at checkout, an account balance or a permission usually cannot.

ATP level. Explain cache-aside and TTL; spot the stale-read bug in a simple update flow. **CTP level.** Choose cache layers and invalidation for a system, set staleness budgets per data type, and plan for stampedes and cache outages (the source must survive a cold cache).

Common mistakes. Caching without a TTL. Caching personalised or permissioned data in a shared layer. Sizing the database on the assumption that the cache never fails.

D1-T08 Reason about consistency and availability trade-offs C

What the practitioner does. Chooses, per kind of data, what users may see when replicas disagree or the network is split, and explains the cost of that choice.

Core knowledge.

- **CAP, stated correctly.** When a network partition occurs, a distributed data system must give up either consistency (every read sees the latest write) or availability (every request gets a non-error response). It is not "pick any two of three". Partitions are not optional, so the real choice is what to do during one (Brewer, 2000; Gilbert and Lynch, 2002).
- **PACELC (Abadi, 2012).** Even with no partition, there is a trade-off between latency and consistency, because synchronous replication takes time. Many systems choose low latency in normal operation.
- **Consistency models (strongest to weakest, simplified).** Linearizable or strong; sequential; causal; read-your-writes and monotonic reads (session guarantees); eventual. Weaker models are cheaper and more available, and push complexity to the application.
- **Time and order (Lamport, 1978).** Clocks on different machines drift, so wall-clock timestamps cannot reliably order events across machines. The "happened-before" relation and logical clocks give an order consistent with causality. Vector clocks can also detect that two events were concurrent.
- **Consensus.** Agreement protocols (such as Paxos and Raft) let a group of nodes agree on a value despite failures, at the cost of a majority round trip. Practitioners need to know when they are relying on one (leader election, locks, configuration stores), not how to implement one.

CTP level. For a described system, decide which data needs strong consistency and which can be eventual, and predict what the user will see during a partition or replica lag.

Common mistakes. One consistency level for everything. "Last write wins" by wall-clock time, silently losing updates. Reading from a replica straight after writing to the primary and showing users their own change missing.

D1-T09 Isolate failure domains C

What the practitioner does. Makes sure a slow or failed component degrades the system gracefully instead of taking everything down with it.

Core knowledge.

- **Timeouts on every remote call.** Without one, a slow dependency holds threads and connections until the caller runs out. A callee's timeout should be shorter than its caller's, and a deadline can be passed down the call chain.
- **Retries, carefully.**
  - Retry only operations that are idempotent (or made so with a key, `D1-T02`).
  - Use exponential backoff with jitter, so clients do not retry in lockstep.
  - Cap attempts.
  - Retry at one layer, not every layer: if each of three layers makes up to three attempts, one user request can become 27 calls to the bottom service just as it is struggling.
- **Circuit breakers (Nygard).** After repeated failures the breaker *opens*, and calls fail fast without touching the dependency. After a pause it goes *half-open* and lets a few trial calls through. Success *closes* it.
- **Bulkheads.** Separate pools (threads, connections, instances) per dependency or per kind of work, so one exhausted pool cannot drain the rest.
- **Graceful degradation.** Decide in advance what the product does without each dependency: show cached results, hide a recommendation panel, queue the action for later.
- **Supervision and restart (Armstrong, 2003).** Rather than handling every possible error inside a component, isolate it, let it fail, and have a supervisor restart it in a known good state. The same idea shows up in process managers and orchestrators that restart unhealthy instances.
- **Cascading failure.** One overloaded component slows its callers, whose queues grow, which slows *their* callers. The tools above exist to break that chain.

CTP level. Given an outage description, find where isolation was missing and choose the change with the biggest effect.

Common mistakes. Default timeouts of "none" or several minutes. Retries without backoff. Holding a database connection or lock while waiting on a remote call.

D1-T10 Scale a system to meet demand A/C

What the practitioner does. Estimates the capacity a system needs and chooses how it will grow.

Core knowledge.

- **Vertical vs horizontal.** Vertical (a bigger machine) is simple but has a ceiling and a single point of failure. Horizontal (more machines) needs work to be divisible.

- **Statelessness.** Instances that keep no session state locally can be added, removed and load-balanced freely. State goes to a shared store.
- **Partitioning (sharding).** Splitting data by a key spreads load. A poor key creates hot partitions, and cross-partition queries and transactions become expensive.
- **Little's Law (Little, 1961).** Average items in a system = arrival rate × average time in the system ($L = \lambda W$). It holds for any stable system, and it is the quickest capacity check a practitioner has. For example, 200 requests/s at 50 ms each means about 10 requests in flight on average.
- **Back-pressure and load shedding.** When demand exceeds capacity, queues grow without bound unless something pushes back: bounded queues, rate limits, 429/503 responses, dropping low-priority work.
- **Tail latency (Dean and Barroso, 2013).** Averages hide the slowest requests, and fan-out magnifies them. If a request touches 100 servers and each has a 1% chance of being slow, about 63% of requests ($1 - 0.99^{100}$) hit at least one slow server.
- **Measure before scaling.** Find the actual bottleneck (CPU, database, locks, a downstream API) before adding machines that wait on the same bottleneck.

ATP level. Explain why stateless services scale horizontally; apply Little's Law to a simple numeric scenario. **CTP level.** Estimate capacity for a peak, choose the scaling approach for each tier, and plan back-pressure.

Common mistakes. Sizing for averages, not peaks. Adding application servers in front of a saturated database. Session state in local memory behind a load balancer.

D1-T11 Evolve a legacy system incrementally C

What the practitioner does. Replaces or restructures an existing system while it keeps serving users, without a big-bang rewrite.

Core knowledge.

- **Strangler fig (Fowler, 2004).** Put a routing layer in front of the old system, move one capability at a time to new code, and route that capability's traffic over. The old system shrinks until it can be switched off.
- **Anti-corruption layer (Evans, 2003).** A translation layer that stops the legacy model leaking into the new one.
- **Branch by abstraction.** Inside a codebase, put an abstraction in front of the component to be replaced, build the new implementation behind it, switch over, then remove the old one, all on the main branch.
- **Parallel run.** Send the same input to old and new paths, compare the results, and switch over only when they agree.
- **Data migration.** Often the hardest part. Options include dual writes with reconciliation, change-data-capture from the old store, and one-off backfills. Plan for

rollback.

- **Choosing what to move first.** Something valuable enough to matter and contained enough to finish. A capability with different scaling or compliance needs is often a good first candidate.

CTP level. Plan a migration sequence, choose the first slice, and say how correctness is checked and how to roll back.

Common mistakes. The full rewrite that must reach feature parity before it delivers anything. Moving code without moving data ownership, which produces a distributed monolith.

1.5 Anti-patterns

Anti-pattern	Symptom	Why it hurts	Remedy	Task
Distributed monolith	Services must be deployed together; services share a database	All the costs of distribution and none of the independence	Real data ownership; contracts between services; merge services that always change together	D1-T05
Synchronous call chain	One user request triggers 4+ sequential remote calls	Latency adds up; any link failing fails the request	Asynchronous steps where the user does not need to wait; aggregate data closer to the reader	D1-T01, D1-T09
Speculative generality	Abstractions with one implementation "for later"	Indirection everyone pays for; usually the wrong abstraction	Wait for the second real case, then extract	D1-T04
Dual write	Update DB, then publish event, in two steps	State and events disagree after a crash	Transactional outbox	D1-T06
Retry storm	Every layer retries immediately on failure	Multiplies load on a failing service and stops it recovering	Backoff with jitter; retry at one layer; circuit breakers	D1-T09
Cache as source of truth	Database cannot handle load if the cache is cold	One cache outage becomes a full outage	Size the source for a cold start; warm-up; stampede protection	D1-T07
Big-bang rewrite	New system must match all old features before launch	Years without value; requirements move	Strangler fig; one capability at a time	D1-T11

1.6 Worked scenario

Situation (CTP level). A regional online grocery marketplace has one monolithic application and one relational database. About 40 engineers work in five teams: catalogue, cart and checkout, payments, delivery, and customer support tools. On sale

days, checkout slows down and sometimes the whole site goes down. Last month a sale-day outage started when the external payment provider slowed down. The CTO has asked the tech lead for a plan, and some engineers want to “move to microservices”.

Step 1: Get evidence before redesigning (P1). The tech lead pulls traces from the last outage. Checkout requests hold a database connection while calling the payment provider. Normally that call takes about 300 ms. During the outage it took about 2 seconds.

Step 2: Apply Little’s Law to explain the outage (D1-T10). Peak traffic is about 120 checkout requests per second.

	Arrival rate (λ)	Time holding a connection (W)	Connections in use (L = λW)
Normal	120 /s	~0.3 s	~36
Provider slow	120 /s	~2 s	~240

The database connection pool has 100 connections, so during the outage it was exhausted. Every other page that needed the database (search, cart, the catalogue) then failed too. The problem was not the monolith as such. It was a missing failure boundary.

Step 3: Fix failure isolation first (D1-T09, D1-T02). These changes are small, fast and do not need a new architecture:

- Never hold a database connection or transaction across the payment call. Record “payment pending”, commit, call the provider, then record the result.
- Set a 3-second timeout on the provider call. Retry at most once, with backoff, and only with an idempotency key so the customer is never charged twice.
- Put a circuit breaker around the provider. When it is open, checkout offers “pay on delivery” or a clear “try again in a few minutes” (graceful degradation).
- Give payment calls their own bulkhead (a separate pool), so they cannot starve the catalogue.

Step 4: Set boundaries, then extract carefully (D1-T05, D1-T11). The five teams map well to bounded contexts. The tech lead recommends:

- Enforce module boundaries inside the monolith first (a modular monolith), so each team owns its code and tables.
- Extract only **payments** as a separate service for now. It has different compliance requirements, an external dependency that fails independently, and a team that wants to release on its own schedule. Do it with the strangler pattern: route payment endpoints to the new service one at a time, using a parallel run against the old path to compare results.

- Do not extract everything. Splitting catalogue, cart and delivery now would add network failure points without solving a current problem.

Step 5: Decouple downstream work with events (D1-T06). When an order is placed, delivery scheduling and customer notifications do not need to finish before the customer sees "order confirmed". The checkout module writes an `OrderPlaced` event to an outbox table in the same transaction as the order. A relay publishes it. Delivery and notification consumers keep a table of processed event IDs, so redelivered events have no extra effect. Events are partitioned by order ID so that each order's events stay in sequence.

Step 6: Cache by staleness budget (D1-T07, D1-T08).

Data	Staleness allowed	Choice
Product descriptions and images	Minutes	CDN plus cache-aside with a TTL of a few minutes
Prices	Short	Cache-aside with a short TTL; delete the entry when a price is updated; confirm the price at checkout from the source
Stock at checkout	None	Not cached. A reservation is made against the database with a strongly consistent check, so two customers cannot buy the last item. Stock shown on browse pages may be slightly stale and is labelled "limited stock" rather than an exact number.

To protect the database from a stampede when popular product entries expire at the start of a sale, cache loads are coalesced per key.

Step 7: Record the decisions (D2-T08). The tech lead writes three short ADRs:

1. No database connections held across remote calls.
2. Modular monolith, with payments extracted.
3. Outbox for order events.

Each one records the context above and what would make the team revisit it.

What a weaker answer looks like.

- "Rewrite into 15 microservices." This adds distribution without fixing the missing timeout and bulkhead, and the outage would happen again across a network.
- "Add more database connections." This raises the ceiling slightly, and the pool fills again as soon as the provider slows down.
- "Retry the payment call three times at each layer." This multiplies load on a struggling provider and risks double charges.

Mini-scenario (ATP level). A developer adds caching to the product page. Products are read from the cache, and on a miss loaded from the database and cached with no expiry. An admin updates a price, but the old price keeps showing. *What went wrong, and what is the usual fix?*

- **What went wrong:** the update path changes the database but leaves the old cache entry in place, and with no TTL the entry never expires.
- **The fix:** after writing the price to the database, delete that product's cache entry, and set a TTL as a safety net (D1-T07).

1.7 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. `D1-T02` · **ATP** A mobile app sends `POST /payments`. The request times out on the client, the app retries, and the customer is charged twice. Which change best prevents this?

- A. Change the endpoint to `GET /payments` so it is safe to retry
- B. Have the client send a unique idempotency key with each payment attempt, which the server stores and checks before charging
- C. Increase the client timeout to 60 seconds
- D. Return HTTP 500 whenever the payment provider is slow

Answer: B.

- **A** is wrong: it misuses GET for a state change.
- **C** reduces how often the bug happens but does not remove it.
- **D** does not stop the duplicate charge.
- **B** makes the retry harmless.

Q2. `D1-T06` · **CTP** An order service commits a new order to its database and then publishes an `OrderPlaced` event to the message broker. Occasionally, after a crash, orders exist that the delivery service never hears about. What is the most appropriate fix?

- A. Publish the event before committing the database transaction
- B. Keep failed events in memory and retry them every minute
- C. Write the event to an outbox table in the same transaction as the order, and publish from the outbox with a separate relay
- D. Have the delivery service poll the order service's database tables directly

Answer: C.

- **A** creates the opposite bug: events for orders that rolled back.

- **B** loses the events when the process crashes, which is exactly the failing case.
- **D** couples the delivery service to another service's schema.
- **C** makes the state change and the event atomic.

Q3. `D1-T09` · **CTP** A recommendation service becomes slow. Within minutes, the product page, which calls it synchronously, starts timing out for every user, even though recommendations are an optional panel. Which change would have the greatest effect?

- A. Add more instances of the product page service
- B. Add a short timeout and a circuit breaker around the recommendation call, and render the page without the panel when it is open
- C. Retry the recommendation call up to five times
- D. Cache the full product page for 24 hours

Answer: B.

- **A** adds more callers that wait on the same slow dependency.
- **C** adds load to a struggling service.
- **D** serves stale prices and stock.
- **B** turns an optional dependency into an optional failure.

Q4. `D1-T10` · **ATP** A service receives 50 requests per second, and each request takes 200 ms on average. On average, how many requests are being processed at the same time?

- A. 4
- B. 10
- C. 25
- D. 250

Answer: B. By Little's Law, $L = \lambda W = 50 \times 0.2 = 10$.

1.8 Evidence of good practice

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

- Most changes touch one module or service, and most deployments involve one team. Track this as a rough measure of coupling.
- Every remote call has a timeout. Retries are bounded and idempotent. There is a written answer to "what does the product do if dependency X is down?"
- Each service or module owns its data, with no cross-service table access.

- Event flows have IDs, idempotent consumers and a monitored dead-letter queue.
- Capacity was estimated for the known peak and checked under load before that peak.
- Significant structural decisions have short ADRs that new team members can read.

1.9 References and further reading

The research archive (`research_papers/pdfs`) holds full texts of several of these. Practitioner books are cited for further reading only; the TPBOK does not reproduce their content.

Foundational papers

- Armstrong, J. (2003). *Making reliable distributed systems in the presence of software errors.* Doctoral thesis, Royal Institute of Technology (KTH), Stockholm.
- Brooks, F. P. (1987). No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4), 10–19. (First presented 1986.)
- Conway, M. E. (1968). How do committees invent? *Datamation*, 14(4), 28–31.
- Dean, J., & Barroso, L. A. (2013). The tail at scale. *Communications of the ACM*, 56(2), 74–80.
- Hoare, C. A. R. (1981). The emperor's old clothes. *Communications of the ACM*, 24(2), 75–83.
- Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 558–565.
- Little, J. D. C. (1961). A proof for the queuing formula: L = λW. *Operations Research*, 9(3), 383–387.
- Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), 1053–1058.

Trade-offs in distributed data

- Abadi, D. (2012). Consistency tradeoffs in modern distributed database system design. *IEEE Computer*, 45(2), 37–42.
- Brewer, E. (2000). Towards robust distributed systems. Keynote, ACM Symposium on Principles of Distributed Computing (PODC).
- Gilbert, S., & Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2), 51–59.

Standards and specifications

- IETF RFC 9110 (2022). *HTTP Semantics.*
- IETF RFC 9457 (2023). *Problem Details for HTTP APIs.*
- ISO/IEC/IEEE 42010:2022. *Software, systems and enterprise: Architecture description.*

- IEEE Computer Society (2024). *Guide to the Software Engineering Body of Knowledge (SWEBOK), Version 4.0*: Software Architecture and Software Design knowledge areas.

Practitioner books

- Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley.
- Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures.* Doctoral dissertation, University of California, Irvine.
- Fowler, M. (2004). StranglerFigApplication. martinfowler.com.
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.
- Kleppmann, M. (2017). *Designing Data-Intensive Applications.* O'Reilly.
- Nygard, M. T. (2018). *Release It! Design and Deploy Production-Ready Software* (2nd ed.). Pragmatic Bookshelf.

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

Bounded context

A part of a system inside which every term has one meaning and one model. Boundaries between contexts are explicit.

Bulkhead

Separating resources (threads, connections, instances) so that exhaustion in one area cannot drain another.

Circuit breaker

A wrapper around calls to a dependency that stops calling it for a while after repeated failures, so callers fail fast instead of waiting.

Dead-letter queue

A place where messages that repeatedly fail processing are moved, so they stop blocking the queue and can be inspected.

Idempotent

An operation that has the same effect whether it is performed once or many times.

Information hiding

Putting each design decision likely to change inside one module, behind an interface that does not reveal it.

Outbox (transactional)

Writing an outgoing event to a table in the same database transaction as the state change, then publishing it separately, so the state and the event cannot disagree.

PACELC

If there is a Partition, choose Availability or Consistency; Else, choose Latency or Consistency.

Strangler fig

Replacing a legacy system by routing one capability at a time to new code until the old system can be removed.

Tail latency

The latency of the slowest requests (e.g. the 99th percentile), which users of large fan-out systems feel most.

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.

Architecture & Systems Design

Service and module design, APIs, events, caching, consistency, failure isolation, scaling and legacy migration.

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

- Domain Guide 2: Engineering Governance & Agile Delivery
- 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