Glossary

This glossary gathers the technical terms, techniques, and concepts from the world of software development that appear throughout the book. It is meant as a quick reference: if a term is unfamiliar to you, you will find a brief definition here. The entries are ordered alphabetically.

Andon. Visual signaling system from the Toyota Production System that lets anyone stop the line when they spot a problem, leaving the fault visible so it can be resolved right away. In software it inspires the idea of "stopping the line" when a defect appears instead of letting it run.

Automatic rollback. Mechanism that reverts a deployment to the previous version automatically when a problem is detected after release.

Basal Cost. The permanent drain on a team's capacity that exists simply because a system exists and has to be maintained, regardless of whether any changes are added to it. A central concept of the book.

Blameless incident management. A way of analyzing production failures that focuses on which part of the system or process allowed the error, not on pointing at people. It fosters learning and psychological safety.

Blast radius. The reach of the damage caused by a failure or a change: how many users, services, or pieces of data are affected. Reducing it (for example with canary releases) limits the consequences of an error.

Bus factor. The number of people who would have to be absent for a project to grind to a halt for lack of knowledge. A bus factor of one (a single person knows something critical) is a serious risk.

Canary release. Strategy of activating a change first for a small group of users (the "canaries") and, if all goes well, gradually extending it to everyone else. It lets you detect problems with limited impact.

Continuous Delivery. Keeping the code always in a deployable state, so that any validated change can be taken to production at any moment (the decision to do so is still manual).

Continuous Deployment. Automatically taking to production every change that passes validation, with no manual intervention. It goes one step beyond Continuous Delivery.

Continuous Integration (CI). A practice in which the whole team integrates its work into the shared main branch very frequently (at least once a day). Because it requires integrating everyone's code daily, it is incompatible with working on branches that live longer than a day. To make it viable, each integration is validated automatically (build and tests), so problems are caught as early as possible.

Conway's law. The observation that a system's architecture ends up reflecting the communication structure of the organization that builds it. Hence designing how teams are organized is, in practice, also designing the software.

Cycle time. The time from when work on an increment begins until it is deployed and in use. A key metric of how fast the flow is.

Dark launch. Deploying a feature to production while keeping it invisible to the user, so that it can be tested with real traffic before being activated.

DORA metrics (Accelerate). Four metrics, popularized by the DORA reports and the book *Accelerate*, that correlate a team's delivery performance with its business results: deployment frequency, lead time for changes, time to recover from failures (MTTR), and change failure rate.

Ensemble programming. A practice (formerly called mob programming) in which the whole team works at the same time on the same problem and the same code. It is an extension of pair programming to more people.

Extreme Programming (XP). An Agile development method created by Kent Beck, built on a set of values (communication, simplicity, feedback, courage, and respect) that take shape in principles and, finally, in practices. It combines technical practices (TDD, Continuous Integration, pair programming, simple

design) with team and planning practices: that is where its strength lies, compared with approaches focused on management alone.

Feature flag. A switch in the code (also called a feature toggle) that lets you turn a feature on or off without deploying again. It separates the moment of deploying from the moment of making a change visible.

Feedback loop. A loop that returns information about the result of an action. The shorter it is, the sooner you learn and correct course.

Firefighting. A reactive mode in which the team keeps resolving urgent issues and incidents with no time to prevent their cause, which perpetuates the problem.

Flaky test. A test that sometimes passes and sometimes fails without the code having changed. It erodes trust in the test suite and is best fixed or removed.

Flow efficiency and resource efficiency. Two opposite ways of measuring. Resource efficiency looks at whether people are always busy; flow efficiency looks at how much of the time work is actually moving forward versus how much it spends waiting. Lean prioritizes the latter.

Hamburger method. Gojko Adzic's technique for breaking a feature down into its "layers" and generating several options per layer, choosing the simplest combination that delivers value. It helps split work into small increments.

Handoff. The moment when work passes from one person or team to another. Every handoff adds waiting and loss of context, which is why Lean seeks to reduce them.

Jidoka. A principle of the Toyota Production System: giving machines and processes "a human touch" so that they detect a defect and stop the flow, preventing the error from spreading.

Just-in-Time (JIT). A Lean principle of producing only what is needed, in the quantity needed, at the moment it is needed, instead of building up work or inventory in advance.

Kaikaku. Radical, disruptive improvement, usually driven from the top, as opposed to the gradual change of Kaizen.

Kaizen. A philosophy of continuous improvement through many small changes sustained over time. It means "change for the better".

Kanban. A method for managing and improving the flow of work in an evolutionary way: making what is in progress visible, limiting the amount of simultaneous work (WIP), and managing the flow to detect and resolve bottlenecks. It takes its name from the "cards" or visual signals of the Toyota Production System.

Kanban board. A visual tool that represents work in columns corresponding to the stages of the process (for example "to do", "in progress", "done"). Tasks move from column to column, which lets you see the state of the flow and where work piles up at a glance.

KISS (Keep It Simple, Stupid). A principle that reminds us to keep solutions as simple as possible and to avoid unnecessary complexity.

Last responsible moment. A Lean principle of postponing a decision until the point where delaying it any further would eliminate some important option. Deciding then, with the most information available, avoids committing too early without falling into paralysis.

Lead time. The total time from when a need is identified until the solution reaches the user. It measures end-to-end speed.

Lean Manufacturing and the Toyota Production System (TPS). A production system born at Toyota that maximizes customer value by eliminating waste, with principles such as Jidoka, Just-in-Time, and Kaizen. It is the origin of Lean thinking.

Lean Software Development. The adaptation to software development of the Lean principles: eliminate waste, amplify learning, deliver fast, and optimize the whole. The central framework of the book.

Legacy code. Old code, often without tests and hard to understand, that is costly and risky to modify.

Linting. Automatic analysis of the code that detects errors and style deviations before it is integrated.

Manifesto for Agile Software Development (Agile Manifesto). The document that sets out the four values and twelve principles of Agile development. Its values prioritize individuals and interactions, working software, customer collaboration, and responding to change.

MTTR (mean time to recovery). The average time a system takes to recover after a failure. In resilient environments it matters more to lower the MTTR than to try to avoid every failure at any cost.

Muda, Muri, and Mura. The three types of waste in Lean: Muda (activity that adds no value), Muri (overburden on people or processes), and Mura (unevenness or lack of uniformity in the work).

Mutation testing. A technique that introduces small changes ("mutations") into the code to check whether the tests catch them. It measures the real quality of the tests, not just their quantity.

MVP (minimum viable product). The simplest version of a product that lets you validate a hypothesis with real users and learn with the least effort.

Observability. The ability to understand what is happening inside a system and to diagnose problems you had not anticipated, without having to touch it or deploy new code just to be able to ask it a new question.

Output and outcome. Output is what gets delivered (features, deliverables); outcome is the real impact it produces on users or the business. What matters is the outcome, not the volume of output.

Pair programming. Two people programming together on the same code: one writes while the other reviews and thinks out loud, swapping roles. It improves quality and shares knowledge.

Poka-Yoke. Error-proof design: mechanisms that make it hard or impossible to make a mistake, instead of relying on the person not slipping up.

Pomodoro Technique. A time-management method based on blocks of focused work (usually 25 minutes) separated by short breaks. On a team, synchronizing those blocks protects focus time.

Pre-commit hooks. Automatic checks (fast tests, linting) that run locally before committing a change, to stop errors as early as possible.

Product Engineer. An engineer with a product mindset: they understand the business problem and the user's context, and they take part in deciding what to build, not just in executing tasks.

Product-market fit. The situation in which a product genuinely solves a problem the market has and values, to the point of adopting and paying for it.

Refactoring. Improving the internal structure of the code without changing its external behavior, so that it is clearer and easier to maintain.

Root cause analysis. Looking for the real origin of a problem instead of stopping at the symptom. Its best-known technique is the *5 Whys*: asking "why" repeatedly (often five times) until you reach the systemic cause worth correcting.

Scrum. An Agile framework that organizes development into fixed-length iterations (sprints) with defined roles and ceremonies.

Set-Based Development. A Lean practice of exploring several possible solutions in parallel and discarding the worst ones as you learn, instead of betting early on a single option. It reduces the risk of committing too soon to a design that later turns out to be wrong.

Slack. Capacity deliberately left unassigned to absorb the unexpected, learn, and improve. Without slack, any incident overwhelms the team.

Smoke test. A quick test run after a deployment to check that the system's essential functions are still operational.

Spike. A time-boxed technical investigation whose goal is to learn or reduce uncertainty (trying out an idea, validating a technology), not to produce final code.

Team Topologies. A model by Matthew Skelton and Manuel Pais for organizing teams so that they optimize the flow of value while limiting each team's cognitive load. It defines four team types (stream-aligned, platform,

enabling, and complicated-subsystem) and three interaction modes between them (collaboration, X-as-a-Service, and facilitation).

Technical bankruptcy. The point at which the accumulation of technical debt and complexity makes it unviable to keep evolving a system without rewriting it: every change costs so much that development grinds to a halt.

Technical debt. The future cost we take on when we deliberately take a shortcut in the design or implementation to deliver value sooner. Like a financial debt, it accrues "interest": every later change becomes slower and riskier until it is "repaid" (refactored). It is not the same as careless or low-quality code: debt is a deliberate decision taken in the knowledge that it will have to be paid back.

Technical slicing. Splitting work by technical components or layers (database, backend, frontend) instead of by deliverable value. Also called horizontal slicing. It tends to delay the delivery of anything useful, unlike vertical slicing.

Test coverage. The percentage of the code that the automated tests exercise. It is a useful signal, not a goal in itself: high coverage does not guarantee good tests.

Test-Driven Development (TDD). The discipline of writing first a failing test, then the minimum code that makes it pass, and then refactoring. In its Outside-In TDD variant you start from a test for a complete use case and work inward.

Throughput. The amount of value or increments a team delivers per unit of time. Together with lead time, it describes the performance of the flow.

Toil. Manual, repetitive operational work with no lasting value that grows with the system and consumes the team's capacity. It is best automated or eliminated.

Trunk Based Development. A strategy in which the whole team works on a single main branch (the "trunk"), integrating small changes into it several times a day. It avoids the use of separate branches that diverge from the trunk and make integration harder.

Value stream. The complete sequence of activities from understanding a customer need to delivering the solution to them. Looking at it as a whole helps reveal where waits and waste pile up.

Vertical slicing. Splitting a feature into increments that cut through all the layers and add value on their own, instead of cutting it by technical layers. It lets you deliver and learn sooner.

WIP and WIP limit. WIP (Work In Progress) is work that has been started but not yet finished or delivered. Setting a WIP limit (a maximum number of tasks in progress) improves flow and reduces lead time.

YAGNI (You Aren't Gonna Need It). A principle that advises not building something until it is genuinely needed, avoiding speculative functionality that inflates the Basal Cost.