

BE SOFTWARE ARCHITECTURE WISE · CHAPTER 5

Architectural Styles I: Monoliths and Modular Designs

Most software, including some of the largest and most successful products, runs as a single deployable application. This chapter explains how such applications should be structured inside. It introduces the vocabulary of architectural styles, patterns, tactics and the architecture quantum, then covers layered architecture, the modular monolith, pipes and filters, microkernel and plug-in architecture, hexagonal, onion and clean architecture, and strategic domain-driven design, and ends with when a monolith is the right answer. It is for developers and architects who design or maintain business applications and want to choose and enforce a structure deliberately.

bewiseapps.com/software-architecture-wise/monolithic-styles/ · Version 2026-11-03. Educational information only, not professional advice.

AT A GLANCE

You will learn to

- Distinguish architectural styles, patterns, tactics and reference architectures, and explain why a style's constraints produce its qualities.
- Compare technical and domain partitioning and predict how each affects the cost of a typical change and the organisation of teams.
- Identify the architecture quanta in a design and explain why shared databases bind services into one quantum.
- Describe layered, modular monolith, pipes-and-filters and microkernel styles, rate their characteristics and recognise their anti-patterns.
- Apply hexagonal, onion or clean architecture to keep business logic independent of frameworks and infrastructure.
- Use strategic domain-driven design to find bounded contexts and describe their relationships in a context map.
- Enforce structural rules in code with architecture tests and module tooling such as import-linter, ArchUnit and Spring Modulith.
- Decide when a monolith is the right choice, how to scale it and which signs indicate it is time to extract services.

Before you start

Chapter 3, especially modularity, coupling and cohesion, and dependencies and layering; Chapter 4, especially the C4 model. Chapter 2's quality attributes are used throughout.

Time

About 3 hours to read; 45 minutes for the test.

Key terms

architectural style · architectural pattern · tactic · technical partitioning · domain partitioning · architecture quantum · closed and open layers · architecture sinkhole · modular monolith · pipes and filters · back pressure · microkernel · ports and adapters · bounded context

Why styles matter

Chapters 3 and 4 gave you the principles of good structure (modularity, coupling and cohesion, dependency direction) and the tools for describing it. This chapter and the next turn those principles into the recurring shapes that real systems take. An architectural style is a named set of constraints on the kinds of parts a system has and how they may connect. The constraints are the point: forbid the parts of a pipeline from sharing state, or forbid one business module from touching another's tables, and you get predictable benefits, at a predictable price. Knowing the main styles lets an architect start from a shape that is known to work for a given set of qualities, explain it in a sentence, and recognise early when a system is drifting away from it.

This chapter covers styles in which the whole application is a single deployment unit: one build, one release, usually one database, with parts calling each other in memory. Such a monolithic architecture avoids the network failures, partial outages and consistency problems of distributed systems, which Chapter 6 takes on. That does not make monoliths primitive. Whether a monolith is a pleasure or a misery to work on depends almost entirely on how it is divided inside, and that is what this chapter is about.

How the chapter fits together

Section 5.1 sets up the vocabulary used for every style in Chapters 5 and 6: styles, patterns, tactics and reference architectures; the difference between technical partitioning (by layer) and domain partitioning (by business capability); monolithic versus distributed deployment; and the architecture quantum, the independently deployable unit to which one set of qualities applies. It ends with a summary table of all the styles in the two chapters.

Sections 5.2 to 5.5 then describe four styles in a consistent format: definition and origin, a diagram, how it works, an example, a table of characteristics, when to use it and when not, anti-patterns, and how to enforce it in code. Layered architecture (5.2) is the familiar presentation, business and persistence stack, with closed and open layers and the sinkhole anti-pattern. The modular monolith (5.3) divides one deployable application by business domain, with explicit module APIs and data ownership, illustrated by Shopify's experience and a module map of a food-delivery platform. Pipes and filters (5.4) arranges processing as a chain of independent steps, from Unix pipes to data pipelines. The microkernel (5.5) builds a minimal core that plug-ins extend, as in IDEs, browsers and rules engines.

Sections 5.6 and 5.7 are about structure inside and between modules rather than about deployment. Hexagonal, onion and clean architecture (5.6) put the business logic at the centre and make databases, user interfaces and external services depend on it, never the other way round, which makes the core testable and its infrastructure replaceable. Strategic domain-driven design (5.7) supplies the method for finding good boundaries: subdomains, bounded contexts and the relationships between them in a context map. Finally, Section 5.8 asks when a monolith is the right answer, how far it can scale, which signs say it is time to split, and offers a decision guide and comparison table for the chapter's styles.

A running example

As in earlier chapters, a food-delivery platform appears throughout: customers order from restaurants, couriers deliver, payments go through an external provider. In Section 5.1 it is classified under four candidate designs; in 5.2 it is a classic layered web application; in 5.3 it becomes eight modules with a dependency rule checked in the build; in 5.4 its order events

flow through a pipeline into a warehouse, where finding the bottleneck is a worked calculation. Later sections return to the same platform, most fully in the context map of Section 5.7, so that the styles can be compared on one familiar system rather than on unrelated toy examples.

The idea to take away

If you remember one thing from this chapter, make it this: **the boundaries inside a system matter more than the boundaries between deployment units**. A monolith divided by domain, with explicit module APIs, one owner for every table and dependency rules checked automatically, can be changed safely by several teams for many years, and any module can later become a service if it truly needs to. A set of services that share a database and call each other's internals is still one tangled system, now with network failures added. Styles are tools for creating and keeping good boundaries; none of them is a goal in itself.

Two habits follow. First, choose the simplest style whose characteristics meet the system's architecturally significant requirements, and record the choice and its trade-offs in a decision record. Second, enforce the style's rules with architecture tests in the build, because every style erodes when its constraints live only in people's heads.

SECTION 5.1 OF 5.8 · INTERMEDIATE · 24 MIN READ · LAST REVIEWED NOVEMBER 2026

Architectural styles, patterns and tactics

BEFORE YOU READ THIS

This section builds on:

- 2.1 Functional and quality requirements
- 3.1 Modularity
- 3.7 Conway's law

Terms it uses from earlier: Quality attribute, Architecturally significant, Connector, Cohesion, Conway's law

THE IDEA IN BRIEF

Think of house types: a terrace, a bungalow, a tower block. Each name tells a builder a lot before any drawing exists: how many floors, how the rooms connect, what it costs, where it works well. A tower block suits a crowded city and is wasteful in a village. Software has its own named shapes, called architectural styles: a layered application, a set of modules in one program, a chain of processing steps, a group of separate services. Knowing the styles, and what each is good and bad at, lets a team pick a sensible shape quickly and talk about it in a few words.

WHAT YOU NEED TO KNOW

- An architectural style is a named set of element types, connector types and constraints on how they combine (Garlan and Shaw); the constraints are what give a style its predictable qualities.
- An architectural pattern, in the tradition of *Pattern-Oriented Software Architecture* (Buschmann et al., 1996), is a proven solution to a recurring structural problem in a stated context; a tactic is a single design decision aimed at one quality attribute; a reference architecture is a domain template.
- Technical partitioning divides a system by technical role (presentation, logic, data); domain partitioning divides it by business capability (orders, payments, delivery). The choice decides how far a typical feature change spreads and how teams can be organised.
- A monolithic style is deployed as one unit; a distributed style is deployed as several units that cooperate over a network and inherits all the costs of the network.
- An architecture quantum is an independently deployable part with high functional cohesion, data included; one set of architecture characteristics applies per quantum, so parts with very different needs must be separate quanta.

- Each style is a trade-off with characteristic strengths and weaknesses; real systems combine styles, and the ratings in this course are relative judgements for typical implementations, not laws.

Why architects name shapes

Most systems are not designed from nothing. Experienced architects recognise that a new problem resembles one they have seen before and reach for a shape that is known to work: a few layers for a straightforward business application, a chain of processing steps for a data feed, a core with plug-ins for a product that customers extend. Naming those shapes gives a team a shared vocabulary. Saying “it is a modular monolith with event-based communication between modules” conveys in ten words what would otherwise take a page of diagrams, and it brings with it a body of knowledge about what tends to go right and wrong.

David Garlan and Mary Shaw put this on a firm footing in *An Introduction to Software Architecture* (Carnegie Mellon technical report, 1994) and in their book *Software Architecture: Perspectives on an Emerging Discipline* (1996). They described an architectural style as a vocabulary of component and connector types together with a set of constraints on how they can be combined. Pipes and filters, for example, has one kind of component (a filter that transforms data) and one kind of connector (a pipe that carries it one way), and it constrains filters not to share state or know who their neighbours are. Roy Fielding, whose dissertation (2000) defined REST as a style, made the same point more sharply: a style is a coordinated set of constraints, and it is the constraints that produce the properties. Forbid filters from sharing state and you get filters that can be tested, reused and rearranged independently; that is not luck, it is a consequence of the rule.

This is the most useful way to read every style in this chapter and the next. A style is not a picture to copy. It is a set of rules you agree to follow, chosen because following them buys specific architecture characteristics, the quality attributes of Chapter 2, at a price. A team that adopts the vocabulary of a style but not its rules gets the price without the benefits.

Styles, patterns, tactics and reference architectures

Four related terms are used, often loosely, for reusable design knowledge. They differ mainly in scope and in how much they prescribe.

An architectural pattern describes a proven solution to a recurring structural problem in a stated context. The idea of patterns came from the building architect Christopher Alexander and entered software through the design patterns of Gamma, Helm, Johnson and Vlissides (1994), which work at the scale of a few classes. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns* by Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad

and Michael Stal (1996), usually called POSA, catalogued patterns at the scale of whole systems: Layers, Pipes and Filters and Blackboard (grouped as getting from mud to structure), Broker for distributed systems, Model-View-Controller and Presentation-Abstraction-Control for interactive systems, and Microkernel and Reflection for adaptable systems. Each POSA pattern is written as context, problem, forces and solution, with consequences, which makes the trade-offs explicit.

A tactic, in the sense used by Len Bass, Paul Clements and Rick Kazman in *Software Architecture in Practice*, is smaller: one design decision that influences one quality attribute response, such as adding a heartbeat to detect a failed server or a cache to cut latency. Section 2.6 used modifiability tactics such as encapsulation and using an intermediary. Patterns package several tactics: the Layers pattern combines encapsulation, restricting dependencies and using an intermediary.

A reference architecture is a template for a whole domain or platform: it names the typical building blocks and how they connect, so that projects adapt it rather than start from a blank page. AUTOSAR, the layered software architecture used for electronic control units in cars, is a well-known example; the architecture centres published by cloud providers offer many smaller ones. A reference architecture is usually built from several styles and patterns at once.

Practitioners do not agree on where the line between style and pattern falls. POSA calls Layers a pattern; Garlan and Shaw call layered systems a style; Mark Richards and Neal Ford, in *Fundamentals of Software Architecture*, reserve “style” for the overall topology of a system and “pattern” for lower-level, reusable structures that can appear inside many styles. This course follows the common usage: **style** for the overall shape of a system or of a large part of it, **pattern** for a reusable solution that can sit at any level. Do not spend energy on the boundary; spend it on the constraints and their consequences.

Four kinds of reusable design knowledge

Term	Scope	Example	Question it answers
Architectural style	Whole system or a large part	Layered, pipes and filters, microservices	What kinds of parts and connections are allowed, and under what rules?
Architectural pattern	Any structural level	Broker, Model-View-Controller, Microkernel	What is a proven solution to this recurring structural problem, and what does it cost?
Tactic	One design decision	Heartbeat, cache, encapsulation, retry with back-off	Which single decision moves this quality attribute in the direction we need?
Reference architecture	A domain or platform	AUTOSAR for car electronics; cloud providers' published architectures	What do systems of this kind usually contain, and how is it usually arranged?

Two ways to cut a system: technical or domain partitioning

The first structural decision in any style is how to divide the system at the top level. There are two basic answers. Technical partitioning groups code by technical role: everything to do with the user interface in one place, all business rules in another, all database access in a third. Domain partitioning groups code by business capability: everything to do with orders in one place, payments in another, delivery in a third, each containing whatever user interface, logic and data access it needs.

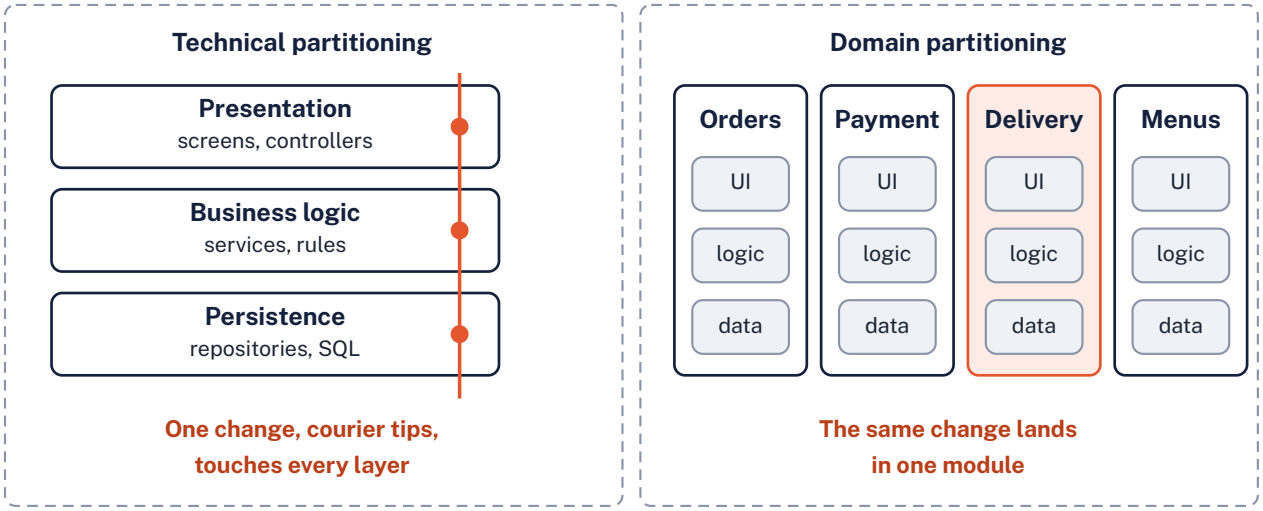
The difference shows up the moment a feature changes. Suppose a food-delivery company lets customers add a tip for the courier. In a technically partitioned system that change needs a new field on a screen (presentation), a new rule for tip limits and payouts (business logic) and a new column and query (persistence): three parts, often owned by three different people or teams. In a domain-partitioned system the same change lands almost entirely in the delivery module. Most business change requests are phrased in domain terms, so domain partitioning tends to keep a change in one place. Technical partitioning keeps a *technical* change in one place, such as replacing the database access library, but those changes are rarer.

The choice also shapes the organisation. By Conway's law, a technically partitioned system tends to be built by a front-end team, a back-end team and a database team, and every feature needs a hand-off between them. A domain-partitioned system can be built by teams that each own a business capability end to end. Martin Fowler made the same recommendation in 2015:

presentation, domain and data layers are a good way to organise a small piece of software, but a larger system's top level should be split into domain-oriented modules, each layered inside if that helps.

Neither is wrong. Technical partitioning is simple, familiar and supported by almost every web framework's default folder structure, and it suits small systems where one team does everything. Domain partitioning costs more thought up front, because someone has to find good domain boundaries (the subject of Section 5.7), but it scales better with the size of the system and the number of teams.

The same feature under two partitionings



With technical partitioning a typical feature crosses every layer; with domain partitioning it lands in one module, which may still be layered inside.

Monolithic and distributed styles

The second top-level distinction is about deployment. A deployment unit is the thing that is built, released and started as one piece: a single executable, a container image, a web archive. In a monolithic architecture the whole application is one deployment unit, usually with one database. It can run as many identical copies behind a load balancer, but every copy contains all the code, and releasing any change means releasing all of it. In a distributed architecture the system is several deployment units that cooperate by sending messages over a network, and each can be released and scaled separately.

Two clarifications avoid much confusion. First, “monolith” describes deployment, not quality. A monolith can be beautifully modular or a big ball of mud (Brian Foote and Joseph Yoder's name for a system with no discernible structure), and so can a distributed system: many teams have learned that splitting a ball of mud into services gives a distributed ball of mud, which is worse. Second, being distributed is not a free upgrade. Every call that crosses a network can be slow,

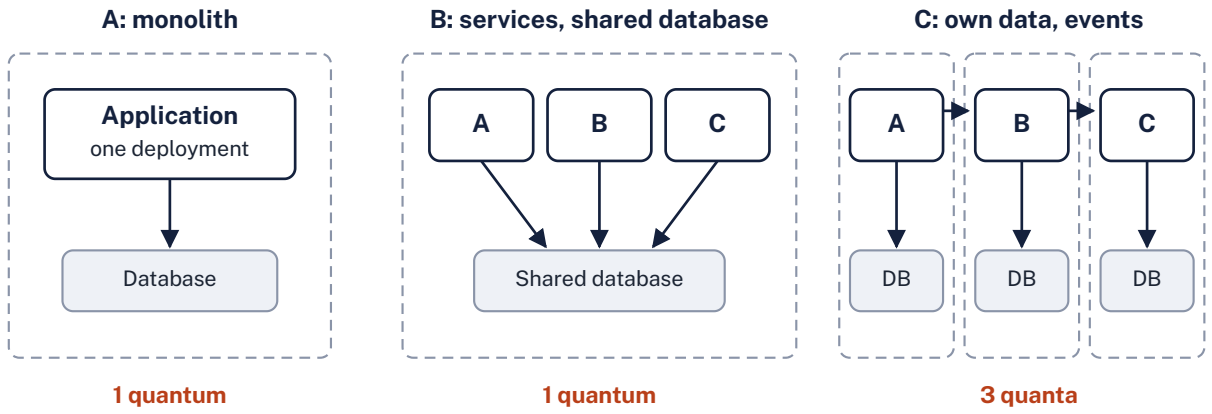
fail halfway or arrive twice, data can no longer be changed in one local transaction, and the system needs monitoring, deployment automation and on-call skills that a monolith can manage without. Chapter 6 covers those costs in detail. This chapter covers the styles that avoid them: everything runs in one deployment unit, and the architecture lives in how the code inside it is organised.

The architecture quantum

Monolithic and distributed is a useful first cut but too coarse for real decisions. Neal Ford, Rebecca Parsons and Patrick Kua introduced a sharper idea in *Building Evolutionary Architectures* (2017), refined by Ford, Richards, Sadalage and Dehghani in *Software Architecture: The Hard Parts* (2021): the architecture quantum. A quantum is a part of the system that can be deployed independently, has high functional cohesion (it does one coherent job for the business) and includes everything it needs to work, its database in particular. The later definition adds that the parts of a quantum are statically coupled (they share code, schemas or infrastructure that must be deployed together) and that synchronous calls bind quanta together operationally for as long as the call lasts.

Why does this matter? Because architecture characteristics apply per quantum. A quantum is scaled, deployed, secured and recovered as a whole, so it has one level of scalability, one level of availability, one release cadence. If two parts of a system need very different characteristics, for example a courier-location feed that must absorb thousands of updates per second and a monthly restaurant-invoicing job, they can only get them by being in different quanta. Conversely, splitting code into several services does not create several quanta if they still share one database: none of them can change the schema or be restored on its own, so they rise and fall together. A monolith is, by definition, one quantum.

Counting quanta



Dashed boundaries mark each quantum: what can be deployed and run on its own, data included

A shared database binds services into one quantum; separate quanta need separate deployment and separate data.

WORKED EXAMPLE

Worked example: classifying four designs

A food-delivery company is reviewing four candidate designs for its platform. For each, the architect asks three questions: how is the top level partitioned, how many deployment units are there, and how many quanta?

Design 1: one Django application with folders `views`, `services` and `models`, one PostgreSQL database	Technical partitioning; one deployment unit; one quantum. Simple, but every feature crosses all three folders.
Design 2: one application with packages `orders`, `payments`, `delivery` and `menus`, each with its own tables, one database server	Domain partitioning; one deployment unit; one quantum. Changes stay local, and the modules could later be separated.
Design 3: four services (orders, payments, delivery, menus) that all read and write the same database tables	Domain partitioning in name; four deployment units but one quantum, because a schema change or a database outage affects all four. All the costs of distribution, few of its benefits.
Design 4: the same four services, each with its own database, exchanging events through a message broker	Domain partitioning; four deployment units; four quanta. Each can be scaled and released independently, at the price of network failures and eventual consistency between them.

Designs 1 and 2 cost the same to run, but design 2 makes typical changes cheaper and keeps future options open.

Design 3 is the warning case: the number of deployment units says little; the number of quanta, which depends on data and coupling, says what can really evolve independently.

IF YOU ARE NEW TO THIS

Step by step: one quantum, one set of qualities

The step most people trip over here is that architecture characteristics belong to a whole architecture quantum, not to each feature inside it. A quantum is deployed, scaled and recovered as one, so everything in it shares one level of scalability, one level of availability and one release cadence. Follow the section's own two contrasting features through one monolith.

- The food-delivery platform runs as one deployment unit with one database. Two of its features need very different things: a courier-location feed that must absorb thousands of updates per second, and a monthly restaurant-invoicing job.
- At dinner time the location feed is overloaded. The only way to add capacity is to run more copies of the whole application, and every copy also carries the invoicing code, which needs no extra capacity at all.
- If the invoicing job has a bug that uses up the process's memory, it can stop the location feed as well, because both run in the same process.
- When the invoicing code needs a small fix, it cannot be released on its own: the whole application is built, tested and redeployed, location feed included.
- To give the feed its own scaling, its own failures and its own releases, it must become a separate quantum, deployed on its own with its own data. Moving its code into a separate service is not enough if it still shares the database, because then neither part can change the schema or be restored on its own.
- **The lesson:** do not ask how many services a design has; ask which parts need very different characteristics. Those parts need separate quanta, and everything else can stay together in one.

How to read the style sections that follow

Sections 5.2 to 5.5 and most of Chapter 6 describe one style each, always in the same order, so that styles can be compared: a definition and where the style came from; a diagram; how it works; a concrete example system; a table rating its main characteristics; when to use it and when not to; common anti-patterns; and how to enforce it in code, because a style that is not checked erodes, as Section 1.7 described.

The characteristic ratings (Low, Medium, High, each with a reason) follow the approach of Richards and Ford's style comparisons, but they are this course's own judgements. Read them as relative and typical: they compare styles as they are usually built, by a competent team, for

a typical business system. A disciplined team can make a layered monolith that deploys many times a day, and a careless one can build microservices that cannot be deployed at all. The ratings tell you where the style helps and where you will have to work against it.

Two sections in this chapter are not deployment styles. Hexagonal, onion and clean architecture (Section 5.6) describe how to organise the inside of an application or service so that business logic does not depend on technology. Domain-driven design (Section 5.7) is a way of finding the domain boundaries that modular monoliths and microservices rely on. Both apply equally inside a monolith or a distributed system.

The styles in Chapters 5 and 6 at a glance

Style	Main partitioning	Deployment	Typical fit
Layered (5.2)	Technical	Monolithic	Small to medium business applications built by one team
Modular monolith (5.3)	Domain	Monolithic	Growing products with several teams that want one deployment
Pipes and filters (5.4)	Technical (processing steps)	Monolithic or distributed	Data processing: ETL, media, logs, compilers
Microkernel (5.5)	Technical core, plug-ins often by domain	Usually monolithic	Products extended by customers or third parties; rules engines
Hexagonal, onion, clean (5.6)	Inner structure: domain at the centre	Either	Any application whose logic must outlive its frameworks
Service-based (6.2)	Domain	Distributed, few coarse services	A pragmatic step beyond a monolith, often sharing a database
Service-oriented architecture (6.2)	Technical (orchestration, enterprise services)	Distributed	Enterprise integration, now largely historical
Microservices (6.3)	Domain	Distributed, many fine-grained services	Many teams that must release independently at scale
Event-driven (6.4)	Technical (event processing)	Distributed	Highly responsive, decoupled reactions to business events
Serverless and space-based (6.8)	Varies	Distributed	Spiky or extreme load; pay-per-use workloads

Styles combine, and they are not goals

Real systems are hybrids. A modular monolith may be layered inside each module; one of its modules may run a nightly pipes-and-filters job; a microservice may be built in hexagonal style; an event-driven system may use a microkernel to let customers add their own rules. Naming the style of each part is still useful, because each part then inherits a known set of rules and trade-offs.

The common mistake is to treat a style as an aim in itself: “we are moving to microservices” as a strategy, rather than as a means to a stated quality such as independent deployment for twelve teams. Start from the architecturally significant requirements of Chapter 2, pick the simplest style whose characteristics meet them, write the choice down as a decision record and enforce its rules. Section 5.8 gives a decision guide for the styles in this chapter.

WATCH OUT

Common misunderstandings

- **Monolith means legacy.** It means one deployment unit. Many successful products run as well-structured monoliths.
- **More services means more modular.** Modularity is about boundaries and dependencies; services sharing a database are one quantum however many there are.
- **A style is chosen once.** Styles evolve: a layered application may become a modular monolith and then lose a module to a separate service when its needs diverge.
- **Using the framework's folders is choosing a style.** A default folder layout is a technical partitioning chosen by the framework's authors for small applications, not a decision about your system.

CHECK YOUR UNDERSTANDING

▼ **According to Garlan and Shaw, what two things define an architectural style?**

A vocabulary of component and connector types, and a set of constraints on how they may be combined. The constraints are what produce the style's characteristic qualities.

▼ **What is the difference between a tactic and a pattern?**

A tactic is one design decision aimed at one quality attribute response, such as a heartbeat or a cache. A pattern is a larger, documented solution that typically combines several tactics.

▼ **Why might four services that share one database count as a single architecture quantum?**

Because none of them can change the shared schema, be restored or be scaled at the data level without affecting the others; they are deployed and fail together in practice, so they share one set of operational characteristics.

▼ **Which kind of change does technical partitioning keep in one place?**

A technical change, such as replacing the database access library or the user-interface framework. Typical business changes, by contrast, cut across all the technical layers.

▼ **Try it: A ticketing platform has three separately deployed services, Booking, Payments and Reports, all using one shared database. How is it partitioned, how many deployment units and how many quanta? Why might Reports' heavy load matter?**

It is partitioned by domain, with three deployment units but only one quantum, because the shared database ties them together: a quantum includes its data. So the system is not as independent as it looks. Reports' heavy queries can slow Booking on sale day, and one set of quality characteristics applies to all three. To give Reports different needs, make it a separate quantum, for example with its own reporting data store.

SOURCES

- Garlan and Shaw: An Introduction to Software Architecture (SEI, 1994): https://www.sei.cmu.edu/library/an-introduction-to-software-architecture/
- Roy Fielding: Architectural Styles and the Design of Network-based Software Architectures, chapter 1: https://ics.uci.edu/~fielding/pubs/dissertation/software_arch.htm
- Martin Fowler: Presentation Domain Data Layering: https://martinfowler.com/bliki/PresentationDomainDataLayering.html
- Microsoft Azure Architecture Center: Architecture styles: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/

Layered architecture

BEFORE YOU READ THIS

This section builds on:

- 3.4 Dependencies
- 5.1 Styles and patterns

Terms it uses from earlier: Technical partitioning, Deployment unit, Architecture quantum, Conway's law

THE IDEA IN BRIEF

Think of a restaurant. Customers talk only to waiters. Waiters pass orders to the kitchen and never go into the storeroom themselves. Cooks fetch ingredients from the storeroom staff, and only the storeroom staff deal with the suppliers. Each group has one job and talks only to the group next to it, so you can retrain the waiters without touching the kitchen. A layered application works the same way: the screens talk to the business rules, the business rules talk to the data access code, and only that code talks to the database.

WHAT YOU NEED TO KNOW

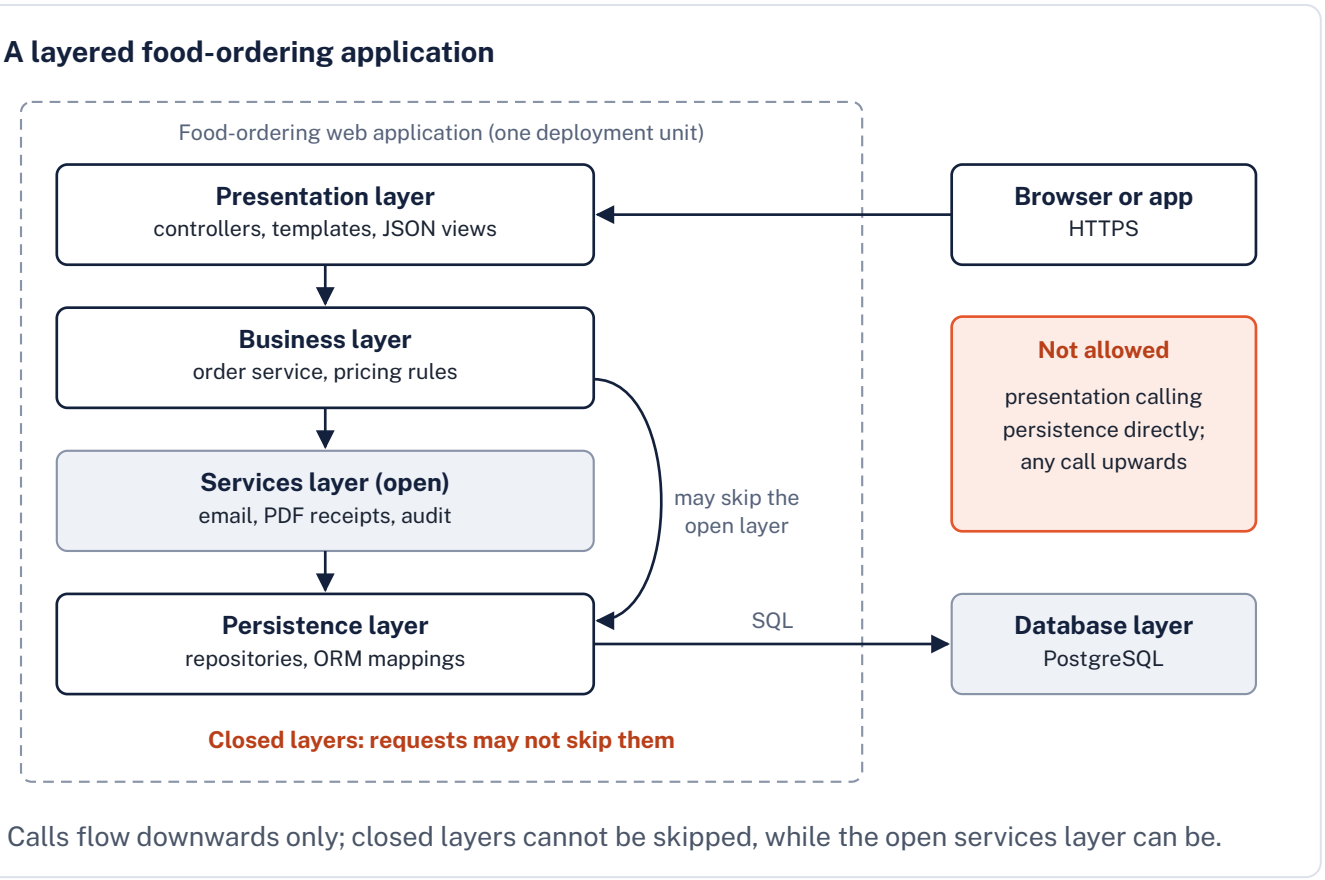
- A layered architecture arranges code in horizontal layers by technical role, typically presentation, business, persistence and database, and lets each layer depend only on layers below it.
- Closed layers must not be skipped, which gives layers of isolation: a change inside one layer does not ripple into others while its interface holds. An open layer, such as a shared-services layer, may be bypassed.
- Layering is technical partitioning, so a typical business feature touches every layer; this slows change and invites team structures split by layer.
- The architecture sinkhole anti-pattern is requests passing through layers that only forward them; as a rule of thumb a minority of such requests is normal, a large majority suggests the wrong style.
- The style rates high on simplicity and low cost, and low on deployability, scalability, elasticity and fault tolerance, because the whole system is one deployment unit.
- Layer rules erode unless enforced automatically, for example with import-linter in Python or ArchUnit in Java, as tests that fail the build.

Definition and origin

A layered architecture organises an application into horizontal layers, each of which has one technical responsibility, offers services to the layer above and uses the services of the layer below. It is the most widely used style for business applications, often by default rather than by decision, because most web frameworks generate a project with folders for views, business logic and models.

The idea is old and comes from systems software. Edsger Dijkstra described the THE multiprogramming operating system (Communications of the ACM, 1968) as a hierarchy of levels, each built only on the ones beneath it, so that each level could be reasoned about and tested on its own. Network protocol stacks such as the OSI reference model use the same principle. In business software, layering arrived with client-server and three-tier systems in the 1990s, and POSA (1996) catalogued it as the Layers pattern. Garlan and Shaw listed layered systems among their basic styles, with the constraint that interactions are limited to adjacent layers.

The typical business application has four layers. The **presentation layer** handles the user interface and the protocol: HTTP requests, screens, JSON. The **business layer** holds the rules and use cases: how an order is priced, when it may be cancelled. The **persistence layer** turns business objects into stored data and back, usually through repositories and an object-relational mapper. The **database layer** is the database itself. Small applications sometimes merge business and persistence; larger ones add layers, such as an application layer that coordinates use cases above a domain layer that holds the core model. The style does not fix the number; it fixes the direction of dependency.



Calls flow downwards only; closed layers cannot be skipped, while the open services layer can be.

How it works: closed layers, open layers and isolation

Each layer can be closed or open. A request must pass through a closed layer: the presentation layer may not reach past the business layer into the persistence layer, even when that would be quicker to write. An open layer may be bypassed. The usual example is a layer of shared technical services, such as sending emails, generating PDF receipts or writing audit records. Only some business operations need them, so forcing every request through that layer would add pointless forwarding code. In the diagram, the business layer may call the services layer or go straight past it to persistence.

Closed layers give layers of isolation. Because the presentation layer knows only the business layer's interface, the team can replace the object-relational mapper, restructure the tables or move from one database product to another without touching a single controller or template, as long as the business layer's interface stays the same. That is information hiding (David Parnas, 1972) applied to a whole tier of the code, and it is the main benefit the style offers. Every open layer, and every exception that lets one layer skip another, weakens it: if controllers query the database directly, a schema change now ripples into the presentation layer too.

The direction of dependency is the other half of the rule. Lower layers must not know about higher ones: the persistence layer must never import a controller, and the business layer must never build HTML. When a lower layer needs to notify a higher one, it does so through an

abstraction the lower layer owns, such as a callback interface or an event, so that the dependency still points downwards in the code. Section 3.4 covers dependency direction, cycles and strict versus relaxed layering in depth.

It helps to keep layers and tiers apart. A layer is a logical grouping in the code. A tier is a physical boundary at run time, such as a separate server or process. A classic three-tier system runs the browser, an application server and a database server as three tiers, while the application server alone may contain three or four layers. Turning a layer into a tier puts a network between two layers that used to call each other in memory, which multiplies the cost of every call; this is rarely worth doing for its own sake.

Example: a classic web application

Consider the ordering website of a regional food-delivery company: customers browse menus, place orders and pay; restaurant staff see incoming orders. One team of five builds it with a mainstream web framework and a PostgreSQL database. A layered structure is the obvious starting point and the code layout makes the layers visible.

```
foodshop/
  presentation/          # HTTP only: parse requests, render responses
    order_views.py
    menu_views.py
    templates/
  business/              # rules and use cases; no HTTP, no SQL
    order_service.py
    pricing.py
  services/              # open layer: shared technical services
    email.py
    receipts.py
  persistence/           # all database access
    order_repository.py
    menu_repository.py
    models.py
  settings.py
```

Package structure of the layered food-ordering application: one top-level package per layer.

WORKED EXAMPLE

Worked example: following one request down the layers

A customer presses **Place order**. The request arrives as `POST /orders` with the basket and delivery address.

Presentation	`order_views.place_order` parses the JSON, checks it is well formed and that the user is signed in, and calls `order_service.place(customer_id, items, address)`. It knows nothing about tables or prices.
Business	`order_service.place` checks the restaurant is open, prices the basket with `pricing.py` (discounts, delivery fee, tax), and asks `order_repository` to save the order. It calls `receipts.send` in the open services layer to email a receipt.
Persistence	`order_repository.save` maps the order to rows in `orders` and `order_lines` and writes them in one database transaction.
Database	PostgreSQL commits the transaction and returns. The result flows back up: the service returns the new order, the view renders `201 Created` with the order number.
Change request	Product asks for a courier tip. The developer adds a tip field to the request and response (presentation), a maximum-tip rule and the tip in the total (business), and a column and mapping (persistence), plus a migration (database). Four layers for one small feature.

Each layer did one kind of job, and each could be tested alone with the layer below replaced by a test double.

The tip shows the cost of technical partitioning: even a small business change cuts across every layer, so it touches more files, more reviewers and, if teams are split by layer, more teams.

The consequences of technical partitioning

Because layering is technical partitioning, the business domain is smeared across the layers. There is no single place called “delivery”: there are delivery controllers, delivery services and delivery repositories, interleaved with everything else in each layer. A developer who wants to understand how tipping works must read across all of them. As the system grows, each layer becomes a large, low-cohesion collection of unrelated code: the business layer contains ordering, payment, menu and courier rules side by side, and nothing stops the payment code from calling deep into the menu code. Over time the layers stay clean but the inside of each layer becomes a big ball of mud.

There is an organisational cost too. Organising teams by layer (a front-end team, a back-end team, a database team) means nearly every feature needs a hand-off and a queue between teams, which Conway's law then cements into the architecture. Martin Fowler calls teams organised by layer an anti-pattern for this reason: developers need not be full-stack, but teams should be. A layered architecture does not force teams to be split by layer, but it makes it the path of least resistance.

Finally, a layered monolith is one architecture quantum: one deployment unit and usually one database. Every change, however small, means building, testing and redeploying the whole application, and the whole application scales and fails as one. That is often perfectly acceptable, but it sets a ceiling.

An alternative cut: vertical slice architecture

Jimmy Bogard's **vertical slice architecture** (2018) organises code around individual requests instead of layers. Each feature, such as *place order* or *cancel order*, holds its own handling, validation and data access from the front of the request to the database. His aim is to "minimize coupling between slices, and maximize coupling in a slice", so that a change touches one feature rather than every layer. The price is the shared structure that layers provided: duplication can build up and handlers can swell, and Bogard says the approach assumes a team that recognises code smells and refactors. Slices also combine well with modules: modules divide the domains, and slices divide the features inside a module.

The architecture sinkhole and other anti-patterns

The architecture sinkhole anti-pattern, a name popularised by Mark Richards, describes requests that flow through layers which do nothing but pass them on. Viewing a menu is the typical case: the controller calls a service method that calls a repository method that runs a query, and the controller and service add nothing. Every layer of pass-through adds code to write, test and read, and a little latency. Some pass-through is the price of isolation and is fine. Richards offers a rule of thumb: if roughly a fifth of requests are simple pass-through, that is normal; if most of them are, the application is probably mostly data access with little business logic, and layering is the wrong style or the layers should be opened. Opening layers, however, gives up isolation, so decide deliberately and record why.

Other common anti-patterns are worth recognising early. **Leaky layers**: SQL in controllers, HTTP request objects passed into services, or ORM entities rendered straight to JSON, so a change in one layer ripples into others. **Upward dependencies**: a repository that calls a service, or a domain object that formats HTML. **The anaemic business layer**: services that only shuffle data while the real rules live in controllers or stored procedures; Fowler named the related Anemic Domain Model anti-pattern in 2003. **The god service**: one `OrderService` of several

thousand lines that every other service calls. **The utility layer that everything depends on**: a `common` package that gradually accumulates business logic and couples everything to everything.

Sinkhole versus useful layering



Pass-through requests cost code and latency without adding behaviour; a few are normal, a majority is a warning sign.

Characteristics of the layered style

Characteristic	Rating	Reason
Simplicity	High	Familiar to almost every developer and the default structure of most frameworks; easy to explain and start.
Overall cost	High (cheap)	One code base, one deployment, one database; minimal infrastructure and operations.
Testability	Medium	Each layer can be tested with the layer below replaced by a test double, but realistic tests often need the whole stack and a database.
Modifiability for business change	Low to medium	Technical changes stay inside a layer, but most feature changes cross every layer.
Deployability	Low	Any change means redeploying the whole application; release risk grows with size.
Scalability and elasticity	Low	The whole application scales as one unit; you cannot scale only the busy part, and start-up of a large application slows elastic scaling.
Fault tolerance	Low	One memory leak or runaway query can take down the whole process; there is no isolation between features.
Performance	Medium	In-process calls are fast, but extra layers and object mapping add overhead, and sinkholes add more.
Evolvability at scale	Low	Low cohesion inside each layer and coupling across features make large layered systems hard to restructure.

IF YOU ARE NEW TO THIS

Step by step: a layer that seems to do nothing

This section's hardest idea is the trade-off behind the architecture sinkhole. A layer that only passes a request on looks like wasted code, yet the rule that forces requests through it is what keeps the layers isolated. Follow two requests through the food-ordering application.

- A customer views a menu. The controller calls a business-layer method, which simply calls the menu repository, which runs the query. The controller and the business method only pass the request on.
- A developer suggests letting the controller call the repository directly. It would save code, but it would turn the business layer into an open layer: one that requests may skip.
- Months later the team restructures the menu tables. Through the business layer, the controller would not need to change, as long as that layer's interface stayed the same. With the shortcut, the change reaches the controller too.
- Placing an order is different: the controller checks the input, the business layer prices the basket and checks the rules, and the repository saves the order in one transaction. Every layer does real work.
- So the useful question is how many requests are like the menu. Richards's rule of thumb: about a fifth being simple pass-through is normal. If most are, the application is probably mostly data access with few rules. Then another style may suit it better, or the team may open some layers on purpose and record why, accepting the loss of isolation.
- **The lesson:** some pass-through is the price of layers of isolation. Count it rather than remove it on sight, and treat a large majority of pass-through requests as a sign that the style does not fit.

When it is a good fit, and when it is not

Layered architecture is a good choice for small and medium applications built by one team, for products whose domain is not yet well understood (domain partitioning needs boundaries you do not yet know), for tight budgets and deadlines, and for applications that are mostly forms over data, with modest rules. It is also a reasonable inner structure for a single module or service that is itself part of a domain-partitioned system: a layered `delivery` module inside a modular monolith gets the best of both.

It is a poor choice as the top-level structure of a large system with several teams, because every team works in every layer and features collide. It is poor when parts of the system need very different scalability or availability, because there is only one quantum. And it is poor when

the business logic is rich and central, because the rules are scattered among controllers, services and stored procedures; there, a domain-centred inner structure such as hexagonal or clean architecture keeps the core independent of the database and the web framework.

The most common evolution path is from a layered monolith to a modular monolith: keep one deployment, but reorganise the top level by domain and keep layers inside each module.

Enforcing the layer rules in code

Layer rules decay quickly. Under deadline pressure, a developer imports the repository straight into a controller because it saves twenty minutes, nobody notices in review, and the next developer copies the pattern. The defence is an architecture test: an automated check of dependencies that runs in the build and fails it when a rule is broken, which makes the rule a fitness function.

In Python, import-linter reads contracts from a configuration file. Its `layers` contract lists layers from highest to lowest and forbids any lower layer from importing a higher one. It does not on its own make layers closed (a higher layer may import any lower layer), so a `forbidden` contract adds the rule that presentation must not import persistence directly; it needs `allow_indirect_imports` switched on, because otherwise the normal chain from presentation through business to persistence would count as a violation. In Java, ArchUnit expresses the same rules as a unit test: `layeredArchitecture()` declares the layers by package and states, for example, that the persistence layer may only be accessed by the service layer. Similar tools exist for other ecosystems, such as dependency-cruiser for JavaScript and TypeScript.

Two practices make the checks stick. Introduce them with the existing violations recorded as known exceptions, so the build goes green immediately and only new violations fail it; then reduce the exceptions over time. And keep the configuration next to the code and review changes to it like code, because relaxing a rule is an architectural decision that deserves a decision record.

```
[importlinter]
root_package = foodshop

[importlinter:contract:layers]
name = Layers only depend downwards
type = layers
layers =
    foodshop.presentation
    foodshop.business
    foodshop.services
    foodshop.persistence

[importlinter:contract:closed-business]
name = Presentation must go through business
type = forbidden
source_modules = foodshop.presentation
forbidden_modules = foodshop.persistence
# presentation → business → persistence is the normal path
allow_indirect_imports = True
```

An import-linter configuration (.importlinter) for the food-ordering application: layers may only depend downwards, and presentation may not skip the business layer.

GOING DEEPER

Layers inside other styles

Layering survives inside most other styles. Each module of a modular monolith, each microservice and each plug-in may be layered internally. Hexagonal and clean architecture can be read as layering with one crucial change: the dependency between business logic and persistence is inverted, so that the database code depends on interfaces defined by the business layer rather than the other way round. That inversion is what lets the business core be tested without a database and outlive its frameworks.

CHECK YOUR UNDERSTANDING

▼ **What is the difference between a closed and an open layer?**

A request must pass through a closed layer and cannot skip it; an open layer may be bypassed by the layer above, which is typical for a shared-services layer that only some requests need.

▼ **Why does a layered application tend to need changes in every layer for a new feature?**

Because it is technically partitioned: the code for each business capability is split across presentation, business and persistence layers, and most features are defined in business terms.

▼ **What is the difference between a layer and a tier?**

A layer is a logical grouping of code by responsibility; a tier is a physical run-time boundary such as a separate server. Several layers usually run inside one tier.

▼ **Why does an import-linter layers contract on its own not make the layers closed?**

It only forbids lower layers from importing higher ones; a higher layer may still import any lower layer, so a separate forbidden contract is needed to stop presentation importing persistence.

▼ **Try it: You trace 500 requests through a layered ticketing app, and 400 pass straight through the business and persistence layers with no logic added. What is this called, what does it suggest, and what would you check before acting?**

It is the architecture sinkhole anti-pattern. Some pass-through is the normal price of isolation, about a fifth by the usual rule of thumb, but here it is 80%, which suggests layering may be the wrong style. First check whether those requests are mostly simple reads and whether the rest of the system really gains from isolation. Options include opening a layer or moving to domain partitioning. Do not rewrite on one sample.

SOURCES

- Martin Fowler: Presentation Domain Data Layering: https://martinfowler.com/bliki/PresentationDomainDataLayering.html
- Microsoft Azure Architecture Center: N-tier architecture style: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/n-tier
- import-linter documentation: Layers contract: https://import-linter.readthedocs.io/en/stable/contract_types/layers/
- ArchUnit user guide (layer checks): https://www.archunit.org/userguide/html/000_Index.html
- Jimmy Bogard: Vertical Slice Architecture (2018): https://www.jimmybogard.com/vertical-slice-architecture/

SECTION 5.3 OF 5.8 · INTERMEDIATE · 26 MIN READ · LAST REVIEWED NOVEMBER 2026

The modular monolith

BEFORE YOU READ THIS

This section builds on:

- 3.3 Information hiding
- 5.1 Styles and patterns
- 5.2 Layered

Terms it uses from earlier: Deployment unit, Domain partitioning, Big ball of mud, Architecture test, Transaction

THE IDEA IN BRIEF

Think of a large department store. It is one building with one front door, one set of opening hours and one security team, but inside it is divided into departments: food, clothing, electronics. Each department has its own staff, stockroom and till, and if clothing needs something from food, it asks at the food counter rather than wandering into the food stockroom. A modular monolith is software built like that store: one program, deployed as one piece, but divided inside into business departments that keep their own data and deal with each other only through agreed counters.

WHAT YOU NEED TO KNOW

- A modular monolith is a single deployment unit whose top level is partitioned by business domain into modules, each with a small explicit public API, private internals and its own data.
- It keeps the operational simplicity of a monolith (one build, one deployment, in-process calls, local transactions) while giving many of the modularity benefits people seek from microservices.
- Boundaries erode unless enforced: language visibility, architecture tests (ArchUnit, import-linter), Spring Modulith's verification for Spring applications, Packwerk for Ruby, and separate database schemas per module.
- Each table belongs to exactly one module; other modules use its API or react to its events, never its tables. Joins across modules are replaced by API calls or read models.
- Modules can communicate by in-process calls (simple, immediate, transactional) or in-process events (looser coupling, but the listener's work happens separately and can fail separately).

- Shopify reorganised its large Ruby on Rails monolith into domain components from 2017 and described it in 2019; its later posts show both the value of the approach and how hard full isolation is.
- A well-bounded module can later be extracted into a service with modest effort, because its API, data and dependencies are already explicit.

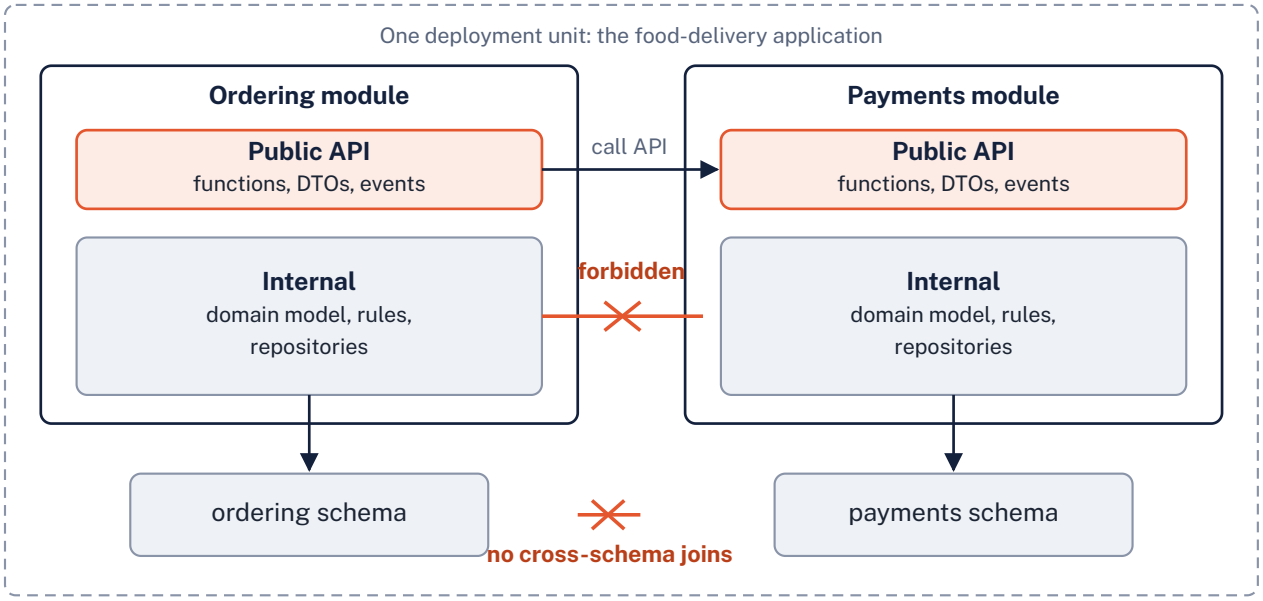
Definition: one deployment, many domains

A modular monolith is an application that is built and deployed as one unit but whose code is divided, at the top level, into modules that follow business boundaries: ordering, payments, delivery, menus. Each module has a small, deliberate public API that other modules may use, a larger internal part that nobody outside may touch, and its own data. It combines the domain partitioning of microservices with the single deployment unit of a classic monolith.

The idea is not new. Modules with hidden internals go back to David Parnas's information hiding (1972), and large well-structured monoliths have always existed. What is newer is the name and the attention. After a decade in which many organisations split systems into microservices and discovered the cost of distribution, practitioners such as Simon Brown and Sam Newman argued that a modular monolith is an underrated option, and that many teams would be better served by one than by a set of services. Richards and Ford give it a chapter of its own in the second edition of *Fundamentals of Software Architecture* (2025).

The difference from the layered style is the top-level cut. A layered application is divided into presentation, business and persistence, with every domain spread across all three. A modular monolith is divided into domains, and each module may be layered inside. The difference from microservices is deployment: the modules share one process, one release and usually one database server, and they call each other in memory.

Anatomy of a modular monolith



Each module exposes a small public API and owns its own schema; reaching into another module's internals or tables is forbidden.

Module APIs and internal code

The single most important rule is that a module is used only through its API. The API is the set of functions, data transfer objects and events that the module's owners promise to keep stable; everything else, including the domain model, the repositories and the database tables, is internal and may change at any time. In practice the API is small: a payments module might expose `authorise_payment`, `refund`, a `PaymentResult` type and a `PaymentCaptured` event, while hiding dozens of classes for providers, retries, ledgers and reconciliation.

Good module APIs share several traits. They speak the language of the domain (`authorise_payment`, not `insert_payment_row`). They pass simple data, such as identifiers and small immutable records, rather than internal objects, so that callers cannot reach through a returned object into the module's internals. They are coarse enough that a typical use case needs one or two calls, not twenty. And they are designed as if a network might one day sit between caller and callee: no shared mutable objects, no reliance on the caller's database transaction, explicit errors. That last trait is what keeps the option of extracting the module later.

Inside the boundary, anything goes that the module's team finds useful: a layered structure, a hexagonal one, a rich domain model or simple transaction scripts. Different modules can make different choices, because nothing outside depends on them.

```
# delivery_platform/payments/__init__.py  (the module's public API)
from .api import authorise_payment, refund, PaymentResult
from .events import PaymentCaptured

__all__ = ["authorise_payment", "refund", "PaymentResult", "PaymentCaptured"]

# delivery_platform/ordering/internal/checkout.py
from delivery_platform import payments             # allowed: public API
# from delivery_platform.payments.internal import ledger   # forbidden

def place_order(order, card_token):
    result = payments.authorise_payment(order.id, order.total, card_token)
    if not result.ok:
        raise PaymentDeclined(result.reason)
    order.confirm(result.payment_id)
    events.publish(OrderPlaced(order.id, order.restaurant_id))
```

A Python package as a module: the payments package exports a small API, and ordering imports only that API.

Enforcing the boundaries

In a monolith every class can technically call every other class, so boundaries hold only if something enforces them. Without enforcement, a deadline-driven shortcut into another module's internals is invisible in review, and a modular monolith decays into a big ball of mud within a year or two. There are several layers of defence, best used together.

Language visibility. Use the strongest hiding the language offers: package-private classes in Java, `internal` in Kotlin and C# (which enforces boundaries only between build modules or assemblies, not between packages in one project), the Java Platform Module System's `exports`, separate build modules (Maven or Gradle subprojects, .NET projects) whose dependencies are declared explicitly. Python and Ruby offer only conventions (a leading underscore, an `internal` sub-package), so tools must do the work.

Architecture tests. Architecture tests check dependencies in the build. ArchUnit (Java) can assert that no class outside a module touches its internal packages and that module dependencies are free of cycles. import-linter (Python) offers a `forbidden` contract, which can stop other modules importing a module's internal package while still letting them use its public API, a `layers` contract whose layers can hold several mutually independent siblings separated by `|`, which makes a one-way dependency order between modules checkable, and an `independence` contract for packages that must not depend on each other at all, even indirectly.

Framework support. Spring Modulith, for Spring Boot applications, treats each direct sub-package of the application's main package as a module, treats the module's base package as its API and its sub-packages as internal, and offers a one-line verification (`ApplicationModules.of(Application.class).verify()`) that fails when a module references another module's internals, when module dependencies form a cycle, or, where a module lists its allowed dependencies, when it uses one that is not listed. It can also generate C4-style component diagrams of the modules. In the Ruby world, Shopify's open-source Packwerk statically analyses references between packages, each declared by a `package.yml` file, and reports undeclared dependencies.

Data boundaries. Give each module its own database schema (or at least its own clearly prefixed tables) and, ideally, its own database user that can access only that schema. Code review alone will not stop a convenient `JOIN` across modules; database permissions will.

```
[importlinter:contract:payments-internals]
name = Only payments may use payments internals
type = forbidden
source_modules =
    delivery_platform.ordering
    delivery_platform.delivery
    delivery_platform.reporting
forbidden_modules =
    delivery_platform.payments.internal
# going through the public API (which itself uses internals) is fine
allow_indirect_imports = True

[importlinter:contract:module-order]
name = Module dependencies point one way
type = layers
layers =
    delivery_platform.reporting | delivery_platform.notifications
    delivery_platform.delivery
    delivery_platform.ordering | delivery_platform.restaurants
    delivery_platform.payments | delivery_platform.menus | delivery_platform.customers
```

import-linter contracts (one per module in practice): other modules may not import payments internals, and module dependencies must follow one direction, with modules on the same line independent of each other.

Data ownership inside a monolith

Data ownership is where modular monoliths most often fail. Sharing one database server is fine; sharing tables is not. If the delivery module reads the `orders` table directly, the ordering team can no longer change that table without coordinating with delivery, and the two modules

are coupled as tightly as if they were one. The rule is simple to state: every table has exactly one owning module, only the owner writes it, and other modules obtain the data through the owner's API or from events the owner publishes.

The rule has consequences that teams must accept deliberately. A report that used to be one SQL query joining orders, payments and couriers now needs either calls to three module APIs or a read model: a reporting module that subscribes to events from the others and keeps its own denormalised copy, optimised for queries. Foreign keys across module schemas are usually dropped, because they would make one module's schema depend on another's; integrity across modules is maintained by the application instead. These costs are real, but they are the same costs a later move to services would impose, paid early and in a cheaper setting.

One advantage remains that distributed systems lose: modules in one process can, when they truly must, share a single local database transaction. Use it sparingly. Each use couples the modules' data operationally and must be undone if a module is ever extracted.

IF YOU ARE NEW TO THIS

Step by step: one owner for every table

The idea that needs the most care here is data ownership inside one program. The modules share one database server, yet every table has a single owner: only that module may read or write it directly. Follow one shortcut and one report through the food-delivery platform.

- The Delivery module needs each new order's details. The orders table sits in the same database, so the quickest code is a query that reads it directly.
- Months later, Ordering reorganises that table, replacing its status column with separate placed and paid times. Delivery's query now fails, and from then on the two teams must plan every change to the table together, as tightly coupled as if they were one module.
- The rule removes the shortcut. Only Ordering touches its tables; other modules use Ordering's public API or the in-process events it publishes, messages inside the program announcing that something has happened. In the design later in this section, Delivery listens for OrderPlaced instead of querying.
- Reports are harder. A cancellations report that used to be one SQL query joining orders, refunds and customers would reach into three modules' tables.
- There are two ways out: call the three modules' APIs and combine the answers, or let Reporting keep a read model: it listens to the modules' events, such as OrderCancelled, and stores its own copy of just the data the report needs.
- Foreign keys between modules' tables are usually dropped too; the application, not the database, keeps those links correct.
- **The lesson:** sharing a database server is fine; sharing tables is not. Events and read models are extra work, but it is the same work a later move to services would need, done early and more cheaply.

Communication between modules: calls or events

Modules talk in two ways, both in memory. An **in-process call** to another module's API is simple, fast and immediate: ordering calls `payments.authorise_payment` and gets the result at once, and an exception propagates normally. It creates a direct dependency, from caller to callee, and the caller cannot finish until the callee does. Use calls when the caller needs an answer to continue, such as whether the payment was authorised.

An in-process event inverts the dependency. Ordering publishes `OrderPlaced` and does not know or care who listens; delivery, notifications and reporting subscribe. Adding a new reaction means adding a listener, without touching ordering. Use events when the publisher does not

need an answer, and when several modules react to the same fact.

Events introduce subtleties that teams new to them often miss. If a listener runs inside the publisher's transaction, a failure in the listener can roll back the order; if it runs after the commit, the order is saved but the listener's work can be lost if the process crashes in between. Spring Modulith addresses this with an event publication registry that records each event-listener pair in the same transaction as the business change and marks it complete only when the listener succeeds, so that incomplete ones can be retried; a hand-built equivalent is the transactional outbox described in Section 6.7. Listeners should also be idempotent, so that a retried event does no harm. Neither style is better; most modular monoliths use calls for queries and commands that need an answer and events for notifications of things that have happened.

Per-module migrations and tests

Ownership lasts only if the tooling respects it. Give each module its own database schema and its own folder of migrations, so that Ordering's changes live beside its code and cannot quietly alter Delivery's tables; Flyway, for example, can run once per schema with separate script locations and history tables. A database login per module, granted only its own schema, turns a stray query into an error. Tests follow the same lines: test each module through its public API, with its neighbours replaced by stand-ins. Spring Modulith's `@ApplicationModuleTest`, for instance, starts one module alone by default. Keep a few whole-application tests for the wiring. Because everything still deploys together, expand and contract still applies during a rolling release.

Shopify's modular monolith

Shopify's core application is one of the largest Ruby on Rails code bases in existence, worked on by more than a thousand developers over more than a decade. In “Deconstructing the Monolith” (Shopify Engineering, February 2019), Kirsten Westeinde described why the company did not split it into microservices and instead, from 2017, turned it into a modular monolith. The pain was familiar: code for unrelated business areas was entangled, changes had unexpected effects elsewhere, and onboarding was slow. Microservices were considered and judged to bring their own costs (separate pipelines and infrastructure, network latency and failures, harder refactoring across services), so the team chose to get modularity without multiple deployment units.

The first step was to reorganise the code from the standard Rails layout (models, views and controllers) into components named after business domains, such as orders, shipping, inventory and billing, in one large automated change. The next was isolation: each component was to have a public API and exclusive ownership of its data, with an internal tool, Wedge,

tracking each component's progress by recording calls across boundaries during the test suite. A September 2020 update reported 37 components in a core monolith of about 2.8 million lines of Ruby, and that same month Shopify open-sourced Packwerk, a static analysis tool for declaring and checking dependencies between packages.

The story is instructive because it is not a simple success. The 2020 update said the original ideas still stood but almost every detail had changed, and that almost no component yet had a complete, strong boundary. A 2024 retrospective on Packwerk reported that its privacy checks had been removed, that some packages organised by name rather than by runtime behaviour misled developers, and that fully isolating even a base package took many months; Packwerk had become a guard on key boundaries rather than the centre of the strategy. The lesson for architects is that a modular monolith is a direction of travel that needs sustained investment, and that retrofitting boundaries into a large existing code base is much harder than keeping them from the start.

WORKED EXAMPLE

Worked example: the food-delivery platform as a modular monolith

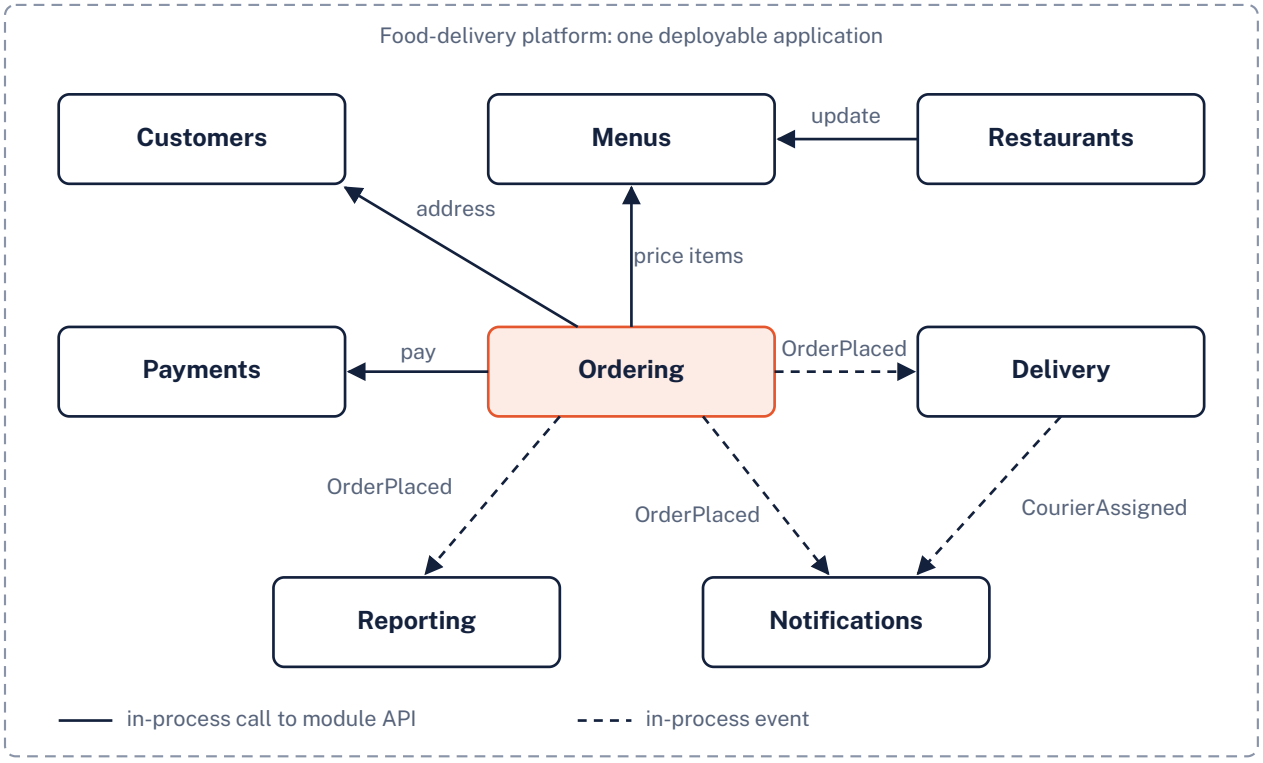
A food-delivery start-up with four teams (about 25 engineers) wants independent ownership without the cost of operating microservices. The architect proposes eight modules in one deployable Python application, with one PostgreSQL server and one schema per module.

Customers	Accounts, addresses, preferences. API: `get_customer`, `get_delivery_address`. Team A.
Restaurants; Menus (two modules)	Restaurants holds restaurant accounts and opening hours and updates menus through the Menus API; Menus serves prices and availability to everyone else (`price_items`, `is_open`). Team B.
Ordering	Basket, checkout, order lifecycle. Calls Customers, Menus and Payments; publishes `OrderPlaced` and `OrderCancelled`. Team C.
Payments	Authorisation, capture and refunds through an external provider. API: `authorise_payment`, `refund`; publishes `PaymentCaptured`. Team C.
Delivery	Courier matching and tracking. Listens for `OrderPlaced`; publishes `CourierAssigned` and `Delivered`. Team D.
Notifications and Reporting	Pure listeners: send messages and maintain read models. They call no other module. Team A.
Dependency rule	Payments, Menus and Customers depend on nobody; Ordering and Restaurants call them; Delivery reacts to Ordering's events; Notifications and Reporting only listen. Checked by import-linter in the build.
Deployment	One container image, run as six replicas behind a load balancer, plus a worker process (same code) that handles event listeners and background jobs.

Each team owns whole modules, so most changes land in one module and one team.

The dependency rule makes cycles impossible: if Payments ever needs to know about orders, it receives an order identifier, not an import of Ordering.

Module map of the food-delivery platform



Solid arrows are calls to another module's API; dashed arrows are in-process events. Ordering calls three modules, and its listeners depend only on the events it publishes.

Characteristics of the modular monolith

Characteristic	Rating	Reason
Simplicity	Medium to high	One deployment and in-process calls, but finding good module boundaries takes thought.
Overall cost	High (cheap)	Infrastructure and operations of a monolith; some extra effort for boundary tooling.
Modifiability for business change	High	Domain partitioning keeps most changes inside one module owned by one team.
Testability	Medium to high	Modules can be tested through their APIs with neighbours replaced; whole-system tests run in one process.
Deployability	Low to medium	Still one deployment unit: every release ships all modules, though smaller blast radius per change helps.
Scalability and elasticity	Low to medium	Replicas scale the whole application; one module cannot be scaled alone, though worker processes can separate background load.
Fault tolerance	Low	One process: a crash or memory leak in any module affects all of them.
Performance	High	Calls between modules are in memory, with no serialisation or network latency.
Evolvability	High	Explicit APIs and data ownership make modules easy to restructure or extract later.

When to use it, when not to, and anti-patterns

A modular monolith is a strong default for new products whose domain is reasonably clear, for organisations with up to a handful of teams, for systems that need strong consistency across business areas, and for teams with limited operations capacity. It is also the natural target when modernising a tangled monolith: introducing boundaries inside it is usually cheaper and less risky than breaking it into services in one go. It is less suitable when parts of the system need genuinely different quality attributes (a feed of courier positions that must absorb huge write volumes and a back-office module with modest load), when dozens of teams must release independently many times a day, or when parts must be written in different languages or run in different places for data-residency reasons. Those needs call for separate quanta.

Common anti-patterns: **modules in name only**, where folders follow domains but anything imports anything; **the shared database free-for-all**, where modules keep their own code but join each other's tables; **the chatty boundary**, an API so fine-grained that a use case needs dozens of calls, which will be painful if the module is ever extracted; **the shared kernel that**

grows, a `common` module that slowly absorbs business logic and couples every module to it; and **entity modules**, cutting by data entity (a Customer module, an Order module, a Product module that every use case needs at once) instead of by business capability, which produces modules that cannot do anything alone.

Keeping the option to extract services

A modular monolith is sometimes presented as a stepping stone to microservices, sometimes as a destination. It can be either, and the point is that the decision can be deferred to the last responsible moment. If one module later needs to scale separately or be released independently, extraction is far easier when the module already has an explicit API, owns its data and depends on others only through APIs and events. The work becomes largely mechanical: put a network interface in front of the API, move the schema to its own database, replace in-process events with a message broker, and replace any shared transaction with an explicit saga or compensation.

Sam Newman makes the same argument in *Monolith to Microservices* (2019): a well-modularised monolith is the best starting point for decomposition, because the hard part, finding and enforcing boundaries, is already done. Section 5.8 discusses the signs that it is time to extract, and Section 11.6 covers incremental migration techniques such as the strangler fig.

A coarser split: self-contained systems

Self-contained systems (SCS) divide a large system into a few autonomous web applications, each with its own data, logic and user interface and each owned by one team. They are usually larger and fewer than microservices: the SCS site suggests perhaps 5 to 25 for an online shop, where microservices might number hundreds. Each system should do its main job alone, so calls to others stay out of the request and response cycle, and interfaces connect by links rather than sharing code. On the food-delivery platform, ordering, the restaurant portal and the courier tools could each be one. The style suits clear domains with web interfaces, and each system can itself be a modular monolith. Screens that need data from many domains are harder.

CHECK YOUR UNDERSTANDING

▼ **What distinguishes a modular monolith from a layered monolith?**

Its top level is partitioned by business domain rather than by technical layer; each module has an explicit API, private internals and its own data, and may be layered inside.

▼ **Why should module APIs pass identifiers and simple records rather than internal domain objects?**

Returned internal objects let callers reach into the module's internals and couple them to its model; simple data keeps the internals free to change and the API ready to cross a network later.

▼ **A report needs data from the ordering and payments schemas. What are two options that respect data ownership?**

Call both modules' APIs and combine the results, or build a read model in a reporting module that listens to both modules' events and keeps its own denormalised copy.

▼ **What risk does an in-process event listener that runs after the publisher's transaction commits introduce?**

If the process crashes after the commit but before the listener runs, the listener's work is lost unless the event is recorded durably, for example in an event publication registry or outbox, and retried.

▼ **Try it: In a food-delivery modular monolith, the Reports module joins directly onto the Payments module's tables for a nightly summary. It works and is fast. Why object, what would you replace it with, and how would you stop it recurring?**

Each table belongs to exactly one module. The join couples Reports to Payments' private schema, so a Payments change can silently break it, and later extraction becomes harder. Replace it with a call to Payments' public API or a read model fed by Payments' events, accepting some extra work and staleness. Give each module its own schema and add an architecture test, such as import-linter or database permissions, so that the build fails on any repeat.

SOURCES

- Shopify Engineering: Deconstructing the Monolith (2019): https://shopify.engineering/deconstructing-monolith-designing-software-maximizes-developer-productivity
- Shopify Engineering: Under Deconstruction: The State of Shopify's Monolith (2020): https://shopify.engineering/shopify-monolith
- Shopify Engineering: A Packwerk Retrospective (2024): https://shopify.engineering/a-packwerk-retrospective
- Spring Modulith reference documentation: Fundamentals: https://docs.spring.io/spring-modulith/reference/fundamentals.html

- import-linter documentation: Forbidden contract: https://import-linter.readthedocs.io/en/stable/contract_types/forbidden/
- Redgate: Handling multiple schemas in the same database with Flyway: https://www.red-gate.com/blog/handling-multiple-schemas-in-the-same-database-with-flyway/
- Spring Modulith reference: integration testing application modules: https://docs.spring.io/spring-modulith/reference/testing.html
- Self-Contained Systems: https://scs-architecture.org/
- SCS compared with microservices: https://scs-architecture.org/vs-ms.html

SECTION 5.4 OF 5.8 · INTERMEDIATE · 23 MIN READ · LAST REVIEWED NOVEMBER 2026

Pipes and filters and pipeline architectures

BEFORE YOU READ THIS

This section builds on:

- 2.4 Performance and scalability
- 5.1 Styles and patterns

Terms it uses from earlier: Throughput, Idempotent, Architecture test, Schema

THE IDEA IN BRIEF

Think of a car wash. Each car goes through a series of stations: soak, scrub, rinse, dry. Each station does one job, knows nothing about the others, and simply passes the car along a conveyor belt to the next. You can add a wax station without changing the others, and if drying is slow, cars start to queue in front of it. Pipes and filters works the same way for data: each record passes through a chain of simple steps (the filters), carried from one to the next by a conveyor (the pipe).

WHAT YOU NEED TO KNOW

- Pipes and filters arranges processing as a chain of independent filters connected by one-way pipes; filters share no state and do not know their neighbours, only the data format they read and write.
- Filters come in four kinds: producers (sources), transformers, testers (which pass or drop records) and consumers (sinks).
- Doug McIlroy proposed coupling programs like garden hoses in a 1964 memo; Ken Thompson added pipes to Unix in 1973, and the Unix philosophy of small tools joined by pipes made the style famous.
- Typical uses are ETL and data pipelines, compiler phases, media processing and log processing; the same structure runs as batch (bounded input) or streaming (unbounded input).
- A pipeline's throughput is set by its slowest filter, and a fast producer will overwhelm a slow consumer unless buffers are bounded and back pressure slows the producer.
- The style rates high on simplicity, modularity and reuse of filters, and low on interactivity and on handling shared state or complex error recovery across steps.

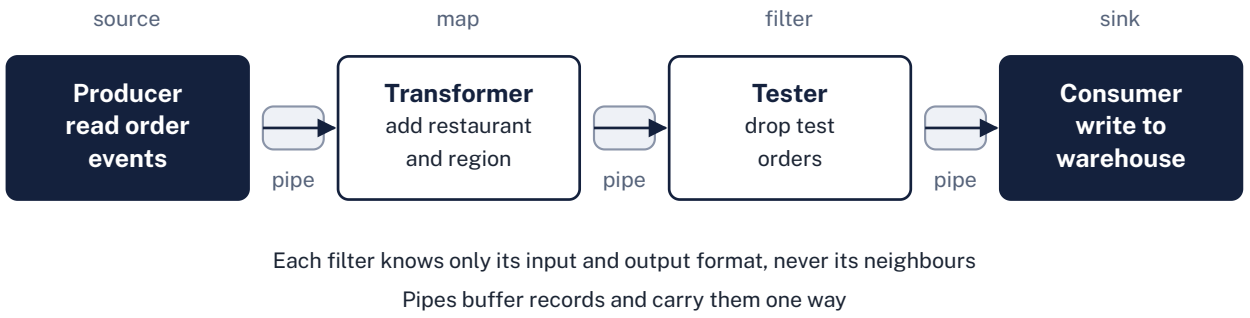
Definition and origin

In the pipes and filters style, processing is a chain of independent steps. Each filter reads data from its input, does one job with it and writes the result to its output. Each pipe carries data one way from one filter to the next, usually with a buffer in between. Garlan and Shaw's description (1994) states the constraints that give the style its properties: filters must be independent, must not share state with other filters, and must not know the identity of the filters at either end of their pipes. A filter knows only the format of what it reads and what it writes.

The style's best-known home is Unix. In a memo dated 11 October 1964, Doug McIlroy wrote that there should be ways of coupling programs like a garden hose, screwing in another segment when the data needs to be massaged in another way. Ken Thompson implemented the `pipe` system call and pipes in the shell in Version 3 Unix in 1973, and the `|` notation followed in Version 4. The Unix philosophy that grew from it, write programs that do one thing well and work together through a universal text interface, is the pipes-and-filters style applied to a whole operating system's toolset. POSA (1996) catalogued Pipes and Filters as an architectural pattern, and Hohpe and Woolf's *Enterprise Integration Patterns* (2003) applied it to messaging between systems.

Richards and Ford call the same style the pipeline architecture, and they name four kinds of filter. A **producer** (or source) starts the flow, reading from a file, a database or a queue. A **transformer** changes each record: parsing, converting, enriching. A **tester** checks a condition and passes the record on or drops it, sometimes routing it to one of several outputs. A **consumer** (or sink) ends the flow, writing to a database, a file or a screen.

Filters and pipes



Four kinds of filter connected by pipes; each filter can be developed, tested and replaced on its own.

How it works

The constraints produce the benefits directly. Because a filter shares no state and does not know its neighbours, it can be written and tested alone, with a list of input records and an expected list of outputs; it can be reused in other pipelines; and filters can be added, removed or reordered by changing only how the pipeline is assembled. Because records flow incrementally, a filter can start work on the first record before the previous filter has finished the last, so stages run concurrently and the whole pipeline needs memory for only a few records at a time, not the whole data set.

The pipe hides how data moves. In a Unix shell it is an operating-system buffer between two processes; on Linux it holds 64 KiB by default, and a writer that fills it simply blocks until the reader catches up. In a Python program it can be a generator, so that each filter pulls the next record from the one before. In a data platform it can be a message queue or a topic in a log such as Apache Kafka, which lets filters run on different machines and survive restarts. The filter code barely changes between these; the operational properties change a great deal.

There are variations on the strict form. A pipeline can fan out (a tester routes records to two different consumers) or fan in (two producers feed one transformer). It can be linear, which is easiest to reason about, or a directed acyclic graph of steps, as in workflow tools such as Apache Airflow. Strictly, a filter with internal state, such as one that counts or sorts, breaks the incremental property, because it must see all input before producing output; `sort` in a Unix pipeline is such a filter. Such filters are allowed but they become the points where memory, latency and restarts need thought.

Examples: where pipelines appear

ETL and data pipelines. Extract, transform, load jobs move data from operational systems into a warehouse: extract yesterday's orders, clean and convert them, join them with restaurant data, load them into reporting tables. Each step is a filter, and modern data platforms (Apache Beam, Spark, dbt models, cloud data-flow services) are largely tools for building and running such pipelines at scale.

Compilers. A classic compiler is a pipeline of phases: lexical analysis turns characters into tokens, parsing turns tokens into a syntax tree, semantic analysis checks types and names, optimisation rewrites an intermediate representation, and code generation emits machine code. Each phase consumes one representation and produces the next, so phases can be developed and replaced separately; the LLVM project, for instance, lets many language front ends share one optimiser and set of back ends. Real compilers relax the strict style, because phases share a symbol table.

Media processing. Video transcoding and audio processing chain decoders, scalers, filters and encoders. GStreamer builds media applications explicitly as pipelines of elements linked through pads, and FFmpeg's filter graphs work the same way.

Log and event processing. Log shippers such as Logstash and Fluent Bit are configured as input, filter and output stages: read log lines, parse them into fields, drop health-check noise, mask personal data, and send them to a search index. The food-delivery platform's observability data flows this way.

Machine-learning feature and inference pipelines. Training data preparation, and the pre-processing and post-processing around a model, are commonly built as pipelines, which Chapter 12 returns to.

```
# Top ten URLs that returned HTTP 500 in today's access log
grep '" 500 ' access.log \
  | awk '{print $7}' \
  | sort \
  | uniq -c \
  | sort -rn \
  | head -10
```

A Unix pipeline: six small tools, each a filter, find the URLs that most often failed with HTTP 500 (in the common access-log format the status code follows the quoted request line).

```
import json

def read_lines(path):                # producer
    with open(path) as f:
        for line in f:
            yield line

def parse(lines):                    # transformer
    for line in lines:
        yield json.loads(line)

def only_failed(events):             # tester
    for e in events:
        if e["status"] == "payment_failed":
            yield e

def count_by_provider(events):       # consumer
    counts = {}
    for e in events:
        counts[e["provider"]] = counts.get(e["provider"], 0) + 1
    return counts

events = only_failed(parse(read_lines("orders.jsonl")))
print(count_by_provider(events))
```

The same idea with Python generators: each function is a filter, and records flow one at a time, so the file is never loaded into memory at once.

Batch and streaming

The same structure runs in two modes. Batch processing handles a bounded input, such as all of yesterday's orders, runs to completion and stops. It is simple to reason about: if a run fails, fix the problem and run it again, provided each run replaces its output rather than appending to it. Nightly payouts to restaurants, monthly invoices and warehouse loads are typical batch pipelines. Their weakness is delay: results are as old as the last run.

Stream processing handles an unbounded input, processing each record shortly after it arrives: courier positions, payment events, clicks. Results are fresh, but new questions appear. A stream never ends, so aggregates must be computed over windows (orders per restaurant per five minutes); records can arrive late or out of order; and the pipeline must run continuously, recover from crashes without losing or double-counting records, and cope when input rates spike. Frameworks such as Kafka Streams, Apache Flink and Apache Beam provide windows, checkpoints and state stores for these problems. Beam's model treats batch as a special case of streaming, with a bounded input, so one pipeline definition can run both ways.

Many organisations run both: a stream for fresh, approximate operational dashboards and a batch job that recomputes exact figures overnight. Section 7.8 discusses the data-architecture side of this choice.

WORKED EXAMPLE

Worked example: finding the bottleneck

The food-delivery platform loads order events into its warehouse through four filters. Measured on one worker, each filter can process the following number of records per second on its own.

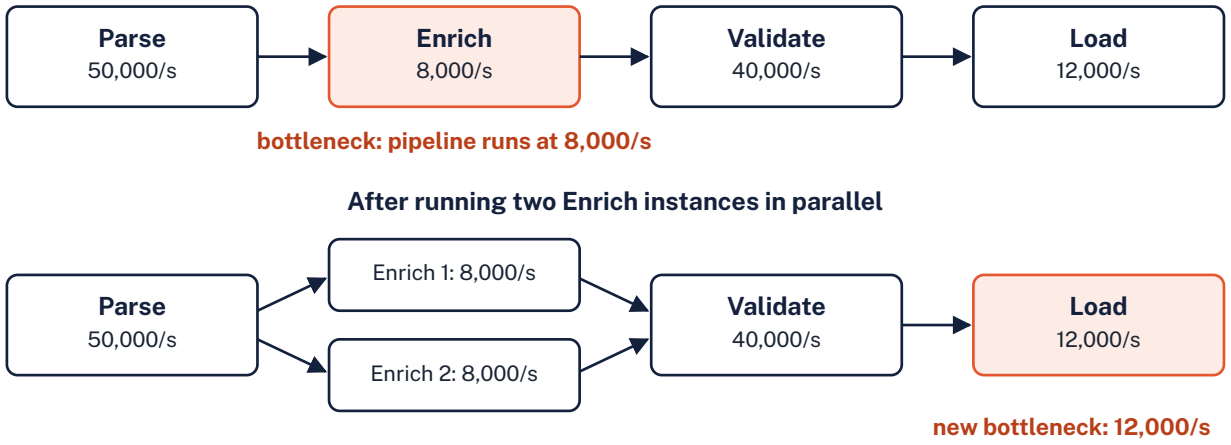
Capacities	Parse 50,000/s; Enrich (looks up restaurant and region) 8,000/s; Validate 40,000/s; Load 12,000/s.
Pipeline throughput	A pipeline can go no faster than its slowest filter: min(50,000, 8,000, 40,000, 12,000) = 8,000 records per second.
Time for one day's data	On a busy day the platform produces 1,000,000 order events. 1,000,000 ÷ 8,000 = 125 seconds.
Scaling the bottleneck	Enrich is stateless, so two instances can run in parallel, each taking half the records: 2 × 8,000 = 16,000/s at that stage.
New throughput	min(50,000, 16,000, 40,000, 12,000) = 12,000/s. Load is now the bottleneck. 1,000,000 ÷ 12,000 ≈ 83 seconds, a 33% saving rather than the 50% a naive reading would suggest.
What the buffers do	Without back pressure, Parse would push records into the pipe before Enrich at 50,000/s while Enrich drains 8,000/s, so the buffer would grow by 42,000 records per second until memory ran out.

Speeding up any filter other than the bottleneck changes nothing; measure first.

Only stateless filters can be parallelised this simply. A filter that keeps state, such as running totals per restaurant, must partition its input by key so that each instance sees all records for its keys.

This is the same reasoning as Amdahl's law and utilisation from Section 2.4: the slowest serial stage sets the limit.

Scaling a pipeline moves the bottleneck



Throughput is set by the slowest filter; parallelising it helps until another filter becomes the slowest.

Back pressure and error handling

When a producer is faster than a consumer, something has to give. The options are to buffer (which only postpones the problem unless the buffer is bounded), to drop or sample records (acceptable for some metrics, never for payments), or to apply back pressure: make the fast stage slow down to the speed of the slow one. Unix pipes do this automatically because a full pipe blocks the writer. Pull-based designs such as Python generators do it by construction, because nothing is produced until the next stage asks for it. Distributed pipelines need it designed in: bounded queues, consumers that pull at their own pace from a log, or protocols such as the JVM's Reactive Streams specification in which a subscriber signals how many records it is ready for. Section 9.3 covers queues and back pressure in depth.

Errors need a policy for each filter, decided up front. A **malformed record** (a poison message) should not stop the pipeline: route it to a dead-letter queue or error file with the reason, count it, and alert if the rate rises. A **transient failure**, such as a lookup service timing out, is retried with back-off. A **crash** of the pipeline needs a restart point: batch jobs rerun from the start or from a checkpoint, and streaming frameworks checkpoint their position and state. Because restarts mean some records are processed twice, consumers that write results should be idempotent, for example by upserting on a natural key rather than inserting blindly. Errors are where the style is weakest: filters are deliberately unaware of each other, so a failure discovered at step four cannot easily undo what step two already did, and a pipeline that needs that kind of coordinated rollback is probably the wrong style.

Characteristics of pipes and filters (single deployment)

Characteristic	Rating	Reason
Simplicity	High	Each filter does one thing; the flow is easy to draw, read and explain.
Modularity and reuse	High	Filters share no state and know only data formats, so they can be reused, reordered and replaced.
Testability	High	A filter is tested with sample input and expected output, without its neighbours.
Overall cost	High (cheap)	One program or job; little infrastructure unless distributed.
Throughput	Medium to high	Stages run concurrently and stateless stages parallelise, but the slowest filter caps everything.
Latency and interactivity	Low	Built for flows of data, not for request-response conversations with users; each stage adds delay.
Deployability	Low to medium	As one program or job, any change redeploys the whole pipeline; filters distributed over queues can be released separately.
Fault tolerance	Low to medium	In one process a crash stops the whole pipeline; checkpoints, retries and dead-letter queues must be added.
Shared state and transactions	Low	Filters are deliberately stateless and independent; cross-step consistency or rollback is awkward.
Scalability	Low to medium in one process	Higher when filters are distributed over queues, at the cost of operations and delivery guarantees.

IF YOU ARE NEW TO THIS

Step by step: when a pipeline fails half-way

The step most people trip over here is what happens when a pipeline fails part of the way through. Filters know nothing about each other, so there is no easy way to undo work already done, and a restart processes some records twice. Follow one day's warehouse load through three kinds of failure.

- Once a day the food-delivery platform loads its couriers' completed-delivery records into the warehouse through four filters: Parse, Enrich (which looks up each courier's zone), Validate and Load.
- One record is malformed and Parse cannot read it. Rather than stop the whole job for one bad record, Parse sets it aside in a dead-letter queue, notes why, counts it and carries on.
- Next, the lookup that Enrich depends on stops answering for a few seconds. That is a transient failure, so Enrich retries with back-off, waiting a little longer each time.
- Then the job crashes after Load has written 300,000 of the day's 450,000 records. Those rows stay: the filters do not coordinate, so nothing in the pipeline can easily roll them back.
- The simplest recovery is to run the job again from the start. But if Load simply inserts rows, the first 300,000 deliveries now appear twice, and every report built on them counts them twice.
- So Load is made idempotent: it upserts on each delivery's own identifier, adding a row if the delivery is new and replacing it if it is already there. Rerunning the job then leaves exactly one row per delivery.
- **The lesson:** decide each filter's error policy before the first failure. Set aside bad records, retry brief failures, and make the final step safe to repeat, because restarts will process some records twice.

When to use it, when not to, and anti-patterns

Use pipes and filters when the problem is naturally a one-way flow of data through distinct transformations: ingestion, conversion, enrichment, validation, aggregation, export. It is the default for data engineering, for processing files and media, and for any job where the steps change more often than the overall flow. It also appears inside other styles: a nightly reporting job within a modular monolith, or a stream-processing service within an event-driven system.

Avoid it as the main structure of interactive applications, where users expect immediate, conversational responses and the work is not a flow; for logic that needs many steps to share and update the same state; and for processes that need coordinated rollback across steps, which call for a transaction or a saga.

Common anti-patterns: **the god filter**, one step that parses, enriches, validates and writes, which loses the benefits of the style; **hidden shared state**, filters that communicate through a shared database table or global variable instead of the pipe, which reintroduces the coupling the style removes; **format drift**, a producer that silently changes its output format and breaks every downstream filter, prevented by explicit schemas and schema checks (see Section 7.7); **unbounded buffers**, which turn a slow consumer into an out-of-memory crash; and **non-idempotent sinks**, which double-count after every retry.

Enforcing the style in code

The style's constraints can be made explicit. Give every filter the same simple signature (an iterable of records in, an iterable out), so that composition is mechanical. Define the record format at each pipe as a schema (a dataclass, a JSON Schema, an Avro or Protocol Buffers definition) and validate it at the boundary. Keep filters free of shared mutable state: an architecture test can forbid filter modules from importing the database layer or each other, so that the only way data passes between them is the pipe. Test each filter with small input and output fixtures, and test the assembled pipeline end to end with a representative sample, including malformed records, to prove the error policy works.

In distributed pipelines, monitor per-stage throughput, queue depth and consumer lag: a growing queue in front of one stage is the visible sign of a bottleneck or a failing filter, and it is the first metric to alert on.

CHECK YOUR UNDERSTANDING

▼ **What three constraints did Garlan and Shaw place on filters?**

Filters must be independent entities, must not share state with other filters, and must not know the identity of the filters upstream or downstream of them.

▼ **Why does the Unix command sort break the incremental property of a pipeline?**

It must read all of its input before it can output the first line, so it holds the whole data set and delays everything after it until its input ends.

▼ **A pipeline has stages that can process 20,000, 5,000 and 15,000 records per second. What is its throughput, and what happens if the middle stage is doubled?**

5,000 per second, set by the slowest stage. Doubling it to 10,000 makes the pipeline run at 10,000 per second, still below the other two stages.

▼ **Why should the final consumer of a restartable pipeline be idempotent?**

Because a restart or retry can deliver some records twice; an idempotent write such as an upsert on a natural key gives the same result however many times a record arrives.

▼ **Try it: An order-event pipeline has filters Parse 40,000/s, Enrich 6,000/s, Validate 30,000/s and Load 10,000/s, with 9,000 records/s arriving at peak. What happens? How many Enrich workers fix it, and what then limits throughput?**

Throughput is set by the slowest filter, Enrich at 6,000/s, so a backlog grows at 3,000 records a second at peak. Two Enrich workers give 12,000/s, assuming it scales cleanly. Load then becomes the limit at 10,000/s, which still beats 9,000/s but leaves about 10% headroom. Keep the buffers bounded, with back pressure, so that a slow stage cannot exhaust memory.

SOURCES

- Doug McIlroy's 1964 memo on coupling programs (cat-v.org): https://doc.cat-v.org/unix/pipes/
- Enterprise Integration Patterns: Pipes and Filters: https://www.enterpriseintegrationpatterns.com/patterns/messaging/PipesAndFilters.html
- Microsoft Azure Architecture Center: Pipes and Filters pattern: https://learn.microsoft.com/en-us/azure/architecture/patterns/pipes-and-filters
- Python documentation: Functional Programming HOWTO (generators): https://docs.python.org/3/howto/functional.html

SECTION 5.5 OF 5.8 · INTERMEDIATE · 26 MIN READ · LAST REVIEWED NOVEMBER 2026

Microkernel and plug-in architectures

BEFORE YOU READ THIS

This section builds on:

- 0.6 APIs and integration
- 3.3 Information hiding
- 5.1 Styles and patterns

Terms it uses from earlier: Deployment unit, Breaking change, Interface

THE IDEA IN BRIEF

Think of a games console. The console itself does a few general things: it starts up, shows a menu, reads controllers and draws pictures. Each game is a cartridge that slots in through a standard connector. You can add new games for years without opening the console, and a badly made game cannot rewire it. A microkernel architecture builds software the same way: a small, stable core does the general work, and plug-ins add the parts that vary, through an agreed connector called a contract.

WHAT YOU NEED TO KNOW

- A microkernel (plug-in) architecture has a minimal core system plus independent plug-ins that connect only through published contracts, found through a registry.
- It suits products with a stable workflow and many variants of particular steps, such as tax rules by country, payment methods, file importers or editor languages.
- Plug-ins should depend only on the contract, never on each other or on core internals, and the core should never contain special cases for named plug-ins.
- Isolation is a trade-off: in-process plug-ins are fast but share the host's fate; separate processes, sandboxes or remote calls protect the core at a cost.
- Version the contract separately, check compatibility when plug-ins load, and prefer additive changes; Hyrum's law means anything plug-ins can observe becomes part of the contract.
- Application microkernels aim at extensibility; operating-system microkernels such as L4 or QNX aim at isolation and a small trusted kernel.

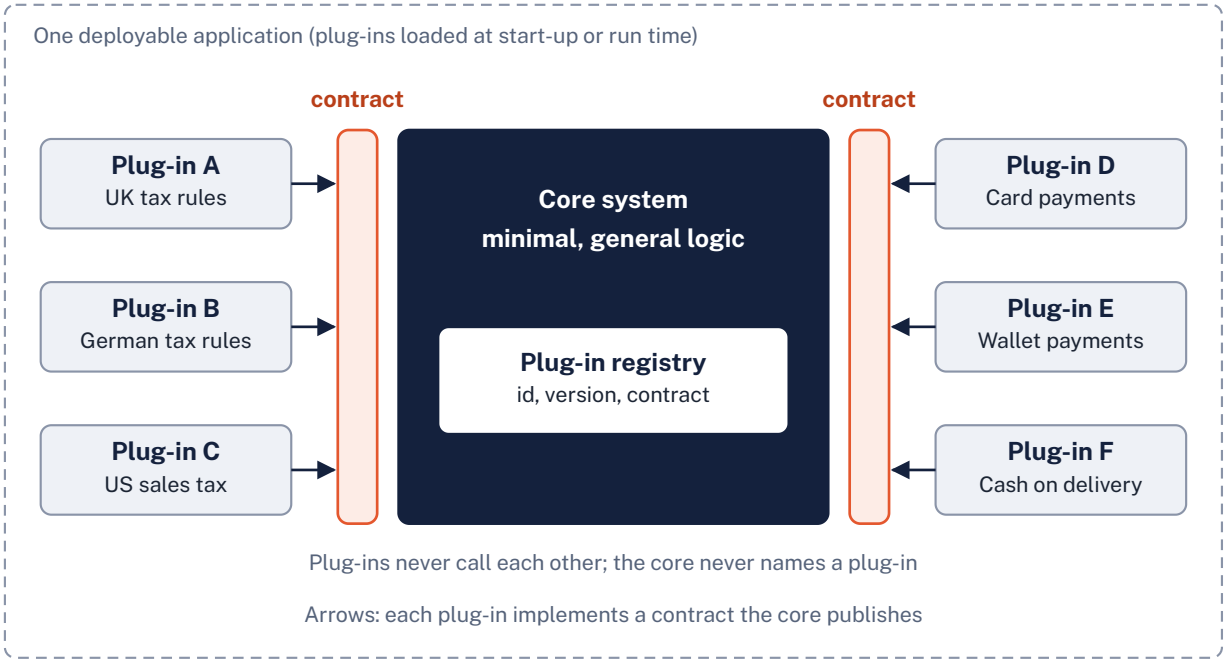
Definition and origin

A microkernel architecture, also called a plug-in architecture, splits an application into two kinds of part: a core system that contains the minimal, general functionality needed to run, and plug-ins that add or change behaviour without the core being modified. Plug-ins connect to the core only through published contracts, and the core finds them through a registry. Mark Richards and Neal Ford catalogue it as one of the classic architecture styles in *Fundamentals of Software Architecture*, and it fits the description of an architectural style given in Section 5.1: a named set of constraints on the elements (core, plug-ins) and connectors (contracts) of a system.

The idea grew up in product software: tools that are installed by many customers, each of whom wants something slightly different. Rather than maintain a separate version for each customer, the vendor ships one core and lets customers, partners or the vendor's own teams add plug-ins. Integrated development environments, web browsers, graphics editors and accounting packages all work this way. The name is borrowed from operating systems, where a microkernel is a very small kernel with everything else running outside it; the two ideas are related but should not be confused, as explained below.

A microkernel application is normally a single deployment unit, so it belongs with the monolithic styles of this chapter: one deployment unit, one process, one set of operational concerns. Plug-ins may be installed separately, but at run time they live inside, or right next to, the core.

The shape of a microkernel system



The core system owns the general workflow and a registry; each plug-in implements a contract the core publishes, and plug-ins never depend on each other.

How it works: core, plug-ins, registry and contracts

The core system contains what every installation needs: the main workflow, shared services such as logging or configuration, and the extension points where plug-ins attach. Richards and Ford describe it as the happy path through the application with little or no custom processing; in a food-delivery checkout, that is pricing a basket, taking payment and confirming the order. A good core is deliberately boring. Every special case that varies by customer, country or product line is a candidate to move out of it.

Plug-ins are independent, self-contained modules that contain specialised processing. They should depend only on the contract, not on each other and not on the core's internal classes. When one plug-in needs something another provides, the need should be met by the core (as a service it exposes through the contract) or the two plug-ins should be merged; plug-in-to-plug-in dependencies quietly turn a clean star into a tangle.

The registry records which plug-ins are available, what each is called, which version it is, which contract it implements and how to reach it. It can be as simple as a dictionary filled at start-up, a directory the core scans for manifests, entries in a package's metadata (as with Python entry points), a database table, or a remote service that lists plug-ins deployed elsewhere. The registry is what lets the core say "give me the tax rules for Germany" without knowing that a class called `GermanTax` exists.

Contracts define what a plug-in must provide and what it may rely on: an interface with method signatures, the data structures that cross it, the error behaviour and a version. In Eclipse a contract is an extension point declared in a plug-in manifest with a schema; in Visual Studio Code it is the extension API plus the contribution points listed in an extension's `package.json`; in your own code it is usually an interface (or a Python `Protocol`) in a small package of its own. Where the plug-in comes from another organisation and cannot follow your contract, an adapter translates between the two, exactly as in hexagonal architecture.

Plug-ins can be connected in three ways. **Point-to-point, in-process** calls are the norm: the core calls a method on the plug-in object. **Separate-process** plug-ins run in their own process and talk to the core through inter-process messages, which protects the core from crashes. **Remote** plug-ins sit behind an HTTP or messaging interface; this allows plug-ins in other languages or owned by other teams, but it adds network latency and failure modes, and the system starts to look like the distributed styles of Chapter 6.

IF YOU ARE NEW TO THIS

Step by step: a core that never names its plug-ins

This section's hardest idea is how a core can use plug-ins without knowing which ones exist. The core depends only on the contract, and a registry tells it at run time which plug-ins are installed. Follow the food-delivery checkout as it adds a country.

- Today the checkout code works out tax itself, in branches such as: if the country is GB, apply the UK rules.
- Launching in Germany means opening the checkout code, adding a German branch, then retesting and releasing the whole checkout. The core knows every country by name, and it grows with each launch.
- In the microkernel design, the core publishes a contract instead: a small interface saying what every set of tax rules must provide, a way to calculate the tax on a priced basket, and exactly what data crosses it.
- The German country team writes a plug-in that implements the contract and states which country it serves, DE. The UK rules move out of checkout into a GB plug-in in the same way, so checkout no longer names any country.
- At start-up the registry finds the installed plug-ins and records which country each serves. When a German customer checks out, the core asks the registry for the tax rules for DE and calls them through the contract, without knowing which piece of code answers.
- Adding the United States later works the same way: a new plug-in with its own tests, found by the registry. The checkout code itself does not change.
- **The lesson:** the core knows the contract and the registry, never the plug-ins. Any line in the core that names a particular plug-in is a sign that the contract is missing something.

Real examples

Eclipse. The Eclipse IDE is the textbook case. Its core is little more than a workbench and a plug-in runtime (built on Equinox, an implementation of the OSGi framework specification). Almost everything a developer sees, from the Java tooling to version-control integration, is a plug-in. Each plug-in has a manifest that declares what it needs, which extension points it offers to others, and which extensions it plugs into. Plug-ins are activated lazily, only when something actually needs them, so that hundreds of installed plug-ins do not slow start-up.

Visual Studio Code. Extensions declare activation events (open a Markdown file, run a command) so they load only when needed, and contribute commands, languages, themes and views through their manifest. Extensions run in a separate extension host process, so a

misbehaving extension cannot freeze the editor's user interface. This is a good example of choosing process isolation for a plug-in ecosystem that includes thousands of third-party authors.

Web browsers. Browser extensions use a standard set of extension APIs and must declare the permissions they need (which sites they may read, whether they may see tabs or downloads) in a manifest. The browser enforces those permissions, because extension authors cannot be trusted by default.

Business rules engines. An insurer's claims system may keep the claims workflow in the core and put each jurisdiction's rules (what is covered, which documents are needed, statutory deadlines) in plug-ins. Tax engines, payroll systems and e-commerce platforms with payment-method plug-ins follow the same pattern: a stable process, with rules or providers that vary by country, product or customer.

Developer tools. The pytest test framework is built around the `pluggy` plug-in library; many of its own features are implemented as plug-ins using the same hooks that third parties use. Build tools, linters and static-site generators commonly do the same.

Application microkernel versus operating-system microkernel

Aspect	Operating-system microkernel	Application microkernel (plug-in) architecture
What is minimal	The kernel: address spaces, threads, scheduling and inter-process communication	The core system: the general workflow of one application
What is outside the core	Device drivers, file systems and network stacks, run as separate user-space servers	Variable business behaviour or features, such as tax rules, payment methods or editor languages
Main goals	Reliability, security and a small trusted computing base	Extensibility, customisation and a third-party ecosystem
How parts talk	Message passing between isolated processes	Usually in-process calls through an interface; sometimes a separate process or remote call
Examples	Mach, the L4 family (including the formally verified seL4), QNX, MINIX 3	Eclipse, Visual Studio Code, web browsers, pytest, insurance and tax rules engines

> **NOTE**
>
> **Do not confuse the two microkernels**
>
> Operating-system microkernels, such as Mach and the L4 family, move device drivers and file systems out of the kernel into separate user-space processes so that a failing driver cannot bring down the whole machine. Linux, by contrast, is a monolithic kernel with loadable modules. The two designs were famously argued over by Andrew Tanenbaum and Linus Torvalds on Usenet in 1992. An application microkernel borrows the shape (small core, everything else outside) but its usual aim is extensibility, not isolation, and its plug-ins normally run in the same process as the core.

Plug-in isolation, versioning and compatibility

Three questions decide whether a plug-in architecture stays healthy over years: how isolated plug-ins are, how the contract evolves, and how incompatibility is detected.

Isolation. In-process plug-ins share the core's memory, threads and permissions. That is fast and simple, but one plug-in with an infinite loop, a memory leak or a security hole affects everything. The options, in increasing order of isolation and cost, are: in-process with defensive calls (timeouts, catching exceptions per plug-in, resource limits); a separate process or worker per plug-in or group of plug-ins; a sandbox such as a WebAssembly runtime or a restricted interpreter, which limits what plug-in code can touch; and a remote service. Choose by asking who writes the plug-ins. Your own team's tax rules can run in-process; arbitrary code from a public marketplace should not.

Versioning. The contract is an API, and the rules of Section 0.6 about breaking changes apply. Version the contract separately from the core, usually with semantic versioning: a new major version for breaking changes, a minor version for backward-compatible additions. Let each plug-in declare which contract version it was built for, and let the core declare which versions it supports. Additive change (new optional fields, new optional hooks) keeps old plug-ins working; when a breaking change is unavoidable, support the old and new major versions side by side for a published deprecation period.

Compatibility checks. Check compatibility when plug-ins are loaded, not when a customer first triggers them. A registry that refuses an incompatible plug-in at start-up, with a clear log message, turns a run-time bug into a deployment failure. Hyrum Wright's observation, known as Hyrum's law, is worth remembering here: with enough users, every observable behaviour of an interface will be depended on by someone, whatever the documentation says. Keep the contract small, keep core internals out of reach (Eclipse marks non-API packages as `internal` for this reason), and treat anything visible to plug-ins as part of the contract.

Characteristics of the microkernel style

Characteristic	Rating	Reason
Extensibility and modifiability (in the planned directions)	High	New behaviour arrives as a new plug-in; the core and the other plug-ins do not change.
Simplicity	Medium	The idea is simple, but contracts, a registry, loading and version checks are real machinery.
Testability	Medium to high	Each plug-in can be tested alone against the contract; the core can be tested with fake plug-ins. Combinations still need testing.
Deployability	Medium	Plug-ins can often be added without redeploying the core; a contract change forces coordinated releases.
Performance	Medium to high	In-process plug-ins are ordinary function calls; isolation by process or sandbox adds overhead.
Scalability and elasticity	Low	It is usually one deployment unit, so it scales as a whole, like any monolith.
Fault tolerance	Low to medium	An in-process plug-in can crash or slow the host; process isolation improves this at a cost.
Overall cost	High (cheap)	One codebase and one runtime to operate; the main cost is designing and evolving contracts.

When to use it, and when not

Good fits. Product software that many customers install and customise. Systems with a stable workflow but many variants of one step: tax and payroll by country, insurance rules by jurisdiction, payment methods, file-format importers, shipping carriers. Platforms that want a third-party ecosystem. Systems where a regulator or customer changes rules often and you want those changes to be small, separately tested releases. The style also works inside a larger system: one service in a distributed architecture can itself be a microkernel.

Poor fits. Systems where you do not yet know what varies. Extension points designed in advance for imagined needs are speculative generality: they add machinery, slow the team down and are usually in the wrong place when the real need arrives (see the discussion of reversibility in Section 1.6). Systems that need very high scalability or fault isolation, since the whole thing scales and fails as one unit. Systems where plug-ins would need to coordinate closely with each other, which defeats the style's main benefit.

A practical rule: wait until you have built the same kind of variation two or three times, then extract the contract from the real cases. A contract derived from three working tax implementations will be far better than one guessed before the first.

WATCH OUT

Common anti-patterns

- **The knowing core.** The core contains `if plugin_name == "stripe"` or special cases for particular plug-ins. Each one is a sign that the contract is missing something.
- **Fat core.** Variable rules creep back into the core because it is quicker than extending the contract, until the core is the old monolith again.
- **Plug-in tangles.** Plug-ins import each other or share hidden state, so they can no longer be installed, tested or removed independently.
- **Leaky contract.** The contract passes core domain objects or database sessions to plug-ins, so every internal change becomes a breaking change for every plug-in.
- **Unversioned contract.** Nobody can tell which plug-ins will break when the core changes, so upgrades become fearful, all-at-once events.
- **One plug-in per customer.** Plug-ins become a way of maintaining hundreds of customer forks. Configuration, not code, should express small differences between customers.

WORKED EXAMPLE

Worked example: designing a plug-in contract for country-specific tax rules

The food-delivery platform used throughout this course is launching in Germany and the United States after the United Kingdom. Each country taxes food, alcohol and delivery fees differently, and US sales tax also varies by state and city. The checkout code already contains `if country == "GB"` branches. The architect proposes a microkernel: checkout is the core, and each country's rules are a plug-in. These are the decisions, in the order they were made.

1. Name the variation point	Only the calculation of tax on a priced basket varies by country. Pricing, discounts, payment and receipts stay in the core. A narrow variation point keeps the contract small.
2. Decide what crosses the contract	Inputs: basket lines with a neutral category (`hot-food`, `alcohol`, `delivery-fee`), net amounts as `Decimal` and an optional region. Output: total tax and a labelled breakdown for the receipt. No core objects (orders, customers, database sessions) are passed, so plug-ins cannot depend on core internals.
3. Make the contract a separate package	`tax_contract` contains only data classes and the `TaxRules` protocol, versioned with semantic versioning. Core and plug-ins both depend on it; neither depends on the other.
4. Define compatibility	A plug-in declares the contract version it needs, such as (2, 1). The core accepts it if the major version matches and the core's minor version is at least as high. Adding an optional field is a minor change; renaming or removing one is a major change.
5. Choose discovery	Plug-ins are ordinary Python packages that register under the entry-point group `fooddelivery.tax_rules`. Installing a package is enough; no core code lists the countries.
6. Decide failure behaviour	If no plug-in is registered for a country, checkout refuses the order with a clear error rather than charging zero tax. A plug-in that raises an exception fails that checkout only, and the error is reported with the plug-in's name and version.
7. Decide isolation	Plug-ins are written by the company's own country teams and are trusted, so they run in-process for speed. If outside tax vendors were allowed to supply plug-ins, the design would move them behind a remote call with a timeout.
8. Specify the tests	A shared contract test suite (rounding, empty baskets, negative refund lines, unknown categories) runs against every plug-in in its own pipeline. Each country team adds its own cases from published tax guidance.

The core no longer changes when a country is added, so the launch of a fourth country is a new package plus its tests, not a change to checkout.

The hardest decision is step 2. A category list that is too coarse forces plug-ins to guess; one that is too detailed couples the contract to one country's law. Expect to revise it, and plan the major-version migration when you do.

The registry rejects incompatible plug-ins at start-up, so a version mismatch shows up in a deployment check, not as wrong tax on a customer's receipt.

```
# package: tax_contract  (version 2.1.0)
from dataclasses import dataclass
from decimal import Decimal
from typing import Protocol

CONTRACT_VERSION = (2, 1)

@dataclass(frozen=True)
class TaxLine:
    sku: str
    category: str      # "hot-food", "cold-food", "alcohol", "delivery-fee"
    net: Decimal       # never float for money

@dataclass(frozen=True)
class TaxResult:
    total_tax: Decimal
    breakdown: dict[str, Decimal]   # label shown on the receipt

class TaxRules(Protocol):
    country: str                    # ISO 3166-1 alpha-2, e.g. "DE"
    requires_contract: tuple[int, int]
    def calculate(self, lines: list[TaxLine], region: str | None) → TaxResult: ...
```

The contract package, versioned and published separately from both the core and the plug-ins.

```
from importlib.metadata import entry_points
from tax_contract import CONTRACT_VERSION

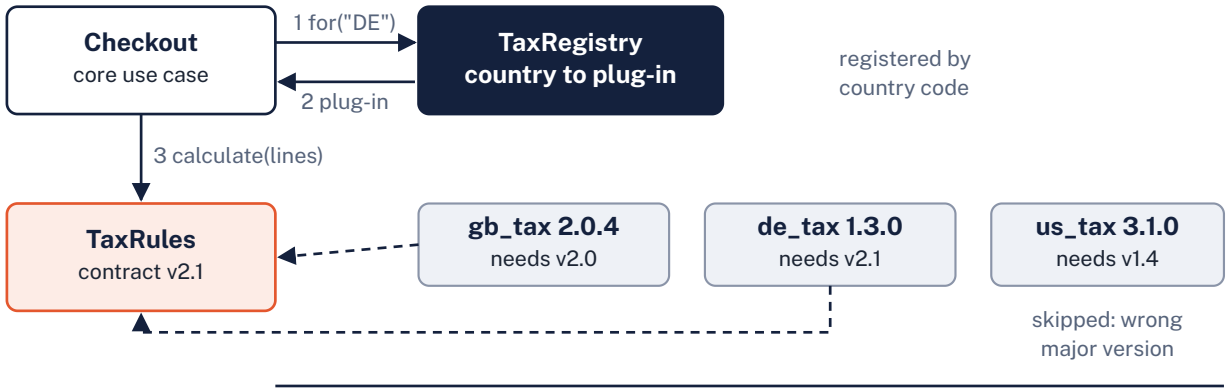
class UnsupportedCountry(Exception): pass

class TaxRegistry:
    def __init__(self, log):
        self._by_country = {}
        for ep in entry_points(group="fooddelivery.tax_rules"):
            plugin = ep.load()()
            major, minor = plugin.requires_contract
            if major ≠ CONTRACT_VERSION[0] or minor > CONTRACT_VERSION[1]:
                log.warning("skipping %s: needs contract %s", ep.name,
plugin.requires_contract)
                continue
            self._by_country[plugin.country] = plugin

    def for_country(self, country):
        if country not in self._by_country:
            raise UnsupportedCountry(country)   # fail loudly, never guess a rate
        return self._by_country[country]
```

The registry in the core: discovers installed plug-ins, checks contract compatibility and looks them up by country.

The tax plug-ins at run time



Loading: at start-up the registry reads installed packages (entry points), checks each plug-in's required contract version and registers it by country; incompatible ones are logged and skipped instead of crashing checkout.

Checkout asks the registry for the plug-in registered for a country and calls it through the contract; a plug-in built for an incompatible contract version is skipped at start-up.

Enforcing the style in code

A plug-in architecture only stays one if its rules are checked automatically. The structure that works in most languages has three kinds of package: a small **contract** package with interfaces and data types only; the **core**, which depends on the contract; and one package per **plug-in**, each depending on the contract and nothing else of yours.

Then enforce the dependency rules with tooling of the kind covered in Section 3.4: in Python, import-linter can forbid the core from importing any plug-in package and forbid plug-ins from importing each other or the core; in Java, ArchUnit tests or module systems (JPMS, OSGi) can do the same. Fail the build when a rule is broken.

Add a **contract test suite**: a set of tests written once against the contract and run against every plug-in in that plug-in's own pipeline. It checks behaviour the contract promises (rounding, empty input, error types), not each country's tax law, which the plug-in's own tests cover. Finally, add a start-up check or a deployment smoke test that loads every installed plug-in and verifies its contract version, so that incompatibilities are caught before users find them.

Questions people ask

Is a microkernel just the strategy pattern?
At code level the strategy pattern is the same mechanism: choose an implementation of an interface at run time. The microkernel is the architectural version: plug-ins are separately built, versioned and often separately installed, a registry discovers them, and the contract is a published API with compatibility rules.

Can plug-ins have their own data?
Yes, and often they should, but treat it like the module-owned data of a modular monolith: a plug-in's tables or files belong to it, and the core reaches them only through the contract.

Should configuration replace plug-ins?
For small, data-like differences (a rate, a threshold, a label), yes: configuration is cheaper and safer. Plug-ins are for differences that need logic. Many systems use both: a plug-in per country plus configuration for each country's current rates.

CHECK YOUR UNDERSTANDING

▼ What three things must a plug-in architecture define besides the core itself?

The plug-ins, the contracts they implement, and a registry that lets the core discover and look up plug-ins without naming them in code.

▼ Why should plug-ins not depend on each other?

Because then they can no longer be installed, removed, tested and versioned independently, which is the main benefit of the style; shared needs belong in the core or the contract.

▼ Why does Visual Studio Code run extensions in a separate process?

So that a slow or crashing extension cannot freeze or crash the editor's user interface: it trades some performance and complexity for isolation from thousands of third-party authors.

▼ When is configuration better than a plug-in?

When the difference between variants is data, such as a rate, a limit or a label, rather than logic. Configuration is simpler and safer to change.

▼ Try it: A ticketing checkout uses payment-method plug-ins. The core contains if plugin == "voucher", and the voucher plug-in reads the core's bookings table and calls the card plug-in directly. Critique this and say what the contract should provide.

This breaks the style's rules. The core must hold no special cases for named plug-ins, and plug-ins must depend only on the contract, never on core internals or each other. Define a versioned contract, such as can_pay and take_payment, passing the booking details the plug-in needs. Move voucher logic behind it, and let the core chain methods for split payments. Anything plug-ins can observe becomes part of the contract, so keep it small.

SOURCES

- Azad Bolour: Notes on the Eclipse Plug-in Architecture (eclipse.org): https://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html
- Visual Studio Code: Extension Host: https://code.visualstudio.com/api/advanced-topics/extension-host
- Python Packaging User Guide: Entry points specification: https://packaging.python.org/en/latest/specifications/entry-points/
- pluggy: the plug-in system behind pytest: https://pluggy.readthedocs.io/en/latest/

- Semantic Versioning 2.0.0: https://semver.org/

SECTION 5.6 OF 5.8 · INTERMEDIATE · 25 MIN READ · LAST REVIEWED NOVEMBER 2026

Hexagonal, onion and clean architecture

BEFORE YOU READ THIS

This section builds on:

- 2.6 Modifiability and deployability
- 3.5 SOLID at scale
- 5.2 Layered

Terms it uses from earlier: Dependency inversion, Test double, Layer, Interface, Framework

THE IDEA IN BRIEF

Think of a travel adapter. Your laptop charger has one plug shape; each country has its own socket. You do not rebuild the laptop for every country: you carry small adapters that fit the socket on one side and your plug on the other. Hexagonal and clean architecture build software the same way. The important part, the business rules, has its own standard sockets (called ports). Databases, websites, payment companies and test tools connect through adapters, so any of them can be swapped without touching the rules.

WHAT YOU NEED TO KNOW

- Hexagonal (ports and adapters, Cockburn, 2005), onion (Palermo, 2008) and clean architecture (Martin, 2012 blog, 2017 book) share one idea: business logic at the centre, independent of frameworks and infrastructure, with dependencies pointing inwards.
- A port is an interface owned by the application; an adapter translates between a port and a technology. Driving adapters call the application; driven adapters implement interfaces the application calls.
- The dependency rule says inner code must never name anything in an outer ring; dependency inversion lets control flow outwards while source code dependencies point inwards.
- The main payoffs are fast, isolated tests of business rules and the ability to replace infrastructure such as payment providers by writing a new adapter.
- The costs are extra interfaces, mapping code and indirection, which are not worth paying for simple CRUD applications or short-lived tools.
- Enforce the rule automatically, for example with import-linter, ArchUnit's onionArchitecture rule or separate build modules.

One idea, three names

Hexagonal architecture, onion architecture and clean architecture are three formulations of one idea: put the business logic at the centre, make it independent of user interfaces, databases, frameworks and external services, and make all source code dependencies point towards it. They describe the internal structure of an application, so they combine with any deployment style: a single monolith, a modular monolith whose modules are each built this way, or a microservice.

They grew out of a frustration with the classic layered architecture. In the layered style of Section 5.2 the business layer depends on the persistence layer, which depends on the database, so business rules end up shaped by tables and object-relational mapping, and testing a rule means starting a database. The family of styles in this section flips that dependency with the dependency inversion principle: the business code declares the interfaces it needs, and the infrastructure code implements them. The principle itself is covered in Section 3.5; this section shows the architectures built on it.

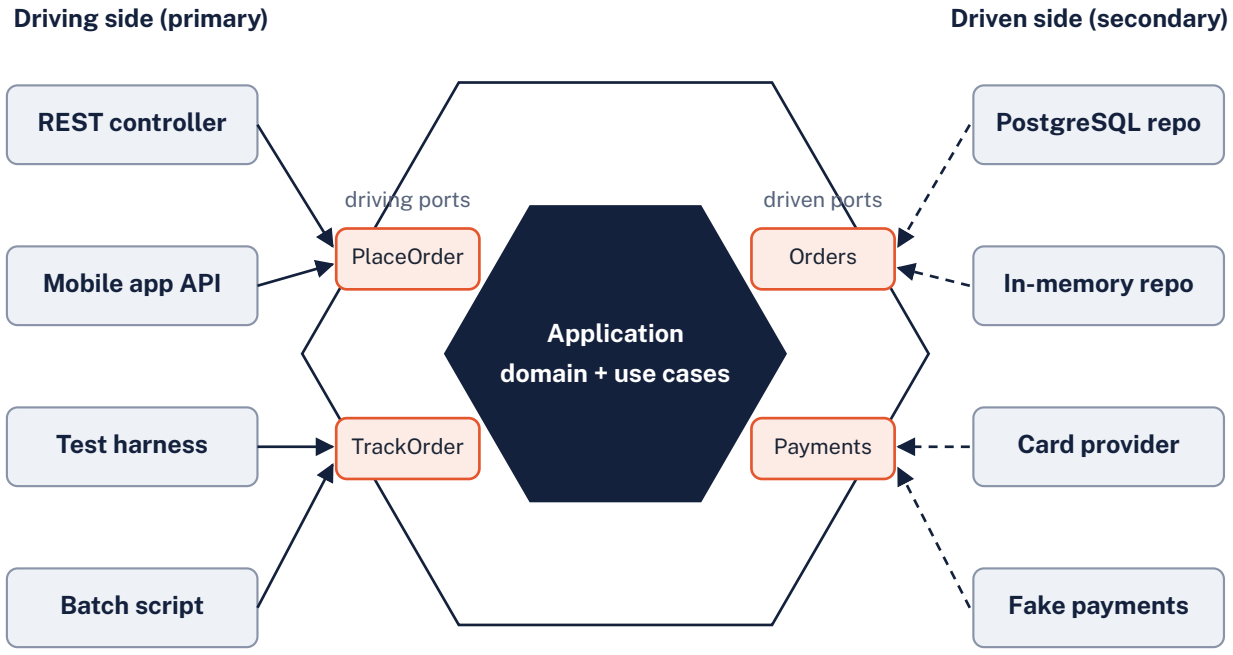
Hexagonal architecture: ports and adapters

Alistair Cockburn first described the idea on the Portland Pattern Repository wiki and wrote it up as an article in 2005 under the more descriptive name ports and adapters. His stated intent was to “allow an application to equally be driven by users, programs, automated test or batch scripts”, and to develop and test it in isolation from its eventual databases and devices. In 2024 he and Juan Manuel Garrido de Paz published a short book, *Hexagonal Architecture Explained*, restating the pattern.

A port is a purposeful conversation between the application and the outside world, expressed as an interface: "place an order", "store orders", "charge a card". An adapter is the code that translates between a port and a particular technology: an HTTP controller, a SQL repository, a payment provider's SDK, a command-line script, an in-memory fake. One port can have many adapters. The application inside the hexagon knows only its ports.

The hexagon has no meaning as a shape; Cockburn chose it to escape the top-to-bottom picture of layers and to leave room to draw several ports on each side. What matters is the inside/outside boundary and the asymmetry between two kinds of port.

Driving and driven ports



Solid arrows: adapter calls a port. Dashed arrows: adapter implements a port. All dependencies point inwards.

Driving adapters call the application through driving ports; the application calls driven ports, which driven adapters implement, so the database and payment provider are replaceable plug-ins.

Driving and driven sides

Driving (primary) ports and adapters sit on the left of the usual picture. A driving adapter receives a request from a primary actor (a user's browser, a mobile app, another system, a test, a scheduled job) and calls a driving port, which is the application's own API: in practice a use case such as `PlaceOrder`. The adapter turns HTTP, a message or a command-line argument into a call with domain values, and turns the result back into a response.

Driven (secondary) ports and adapters sit on the right. The application calls them when it needs something from the outside world: storing an order, charging a card, sending a notification, reading the time. The port is an interface owned by the application and written in its terms (`OrderRepository.save(order)`), and a driven adapter implements it for a technology (PostgreSQL, a card provider's API, an SMS gateway). This is where dependency inversion happens: at run time the application calls the adapter, but in the source code the adapter depends on the application's interface, not the other way round.

Cockburn's original text speaks of primary and secondary actors; the terms driving and driven are now more common, and he uses them himself. The distinction matters for testing: driving ports are exercised by test harnesses acting as users, and driven ports are replaced by test

doubles. It also matters for change: new channels (a voice assistant, a partner API) arrive as new driving adapters, and new infrastructure arrives as new driven adapters.

Someone has to create the adapters and plug them into the application. That happens in a single composition root, typically the program's start-up code, which is the only part of the system that knows about every adapter. A dependency-injection framework can do this work, but plain constructor calls in `main` are often clearer.

IF YOU ARE NEW TO THIS

Step by step: calling in and calling out

The idea that needs the most care here is the difference between the two sides of the hexagon. On the driving side an adapter calls the application; on the driven side the application calls an adapter. Yet in the source code every dependency still points inwards. Section 3.5 introduced dependency inversion; follow one order to see where it happens.

- A customer places an order. The REST controller, a driving adapter, receives the HTTP request, turns it into plain values such as the order ID and total, and calls the PlaceOrder use case. The call goes inwards.
- PlaceOrder saves the order first, then needs to authorise the card. It calls authorise on PaymentGateway, an interface that the application itself declares, in its own words. It has no idea which payment company is behind it.
- The card provider's adapter, outside the hexagon, is a driven adapter: it implements PaymentGateway using the provider's own toolkit. At run time the call goes outwards, from the use case to the adapter, but in the source code the adapter refers to the application's interface, never the other way round.
- Saving works the same way. PlaceOrder calls save on OrderRepository; PostgresOrderRepository implements it with SQL, and an in-memory version implements the same port for tests.
- Something must choose the adapters. That happens once, in the composition root at start-up, which creates the PostgreSQL and card provider adapters and hands them to PlaceOrder. A test does the same with in-memory and fake ones.
- **The lesson:** on the driving side the outside calls in; on the driven side the inside calls out, through an interface it owns. Either way the application never names a technology, which is what lets adapters be swapped and tested.

Onion architecture

Jeffrey Palermo described the onion architecture in a series of blog posts starting in July 2008, aimed at long-lived business applications with substantial behaviour (he advised against it for small websites). He drew concentric rings: the domain model at the centre, domain services around it, application services around those, and an outer ring containing the user interface, infrastructure (including the database) and tests. His key tenets are that the application is built around an independent object model, that inner layers define interfaces which outer layers implement, that all coupling is towards the centre, and that the application core can be compiled and run separately from infrastructure.

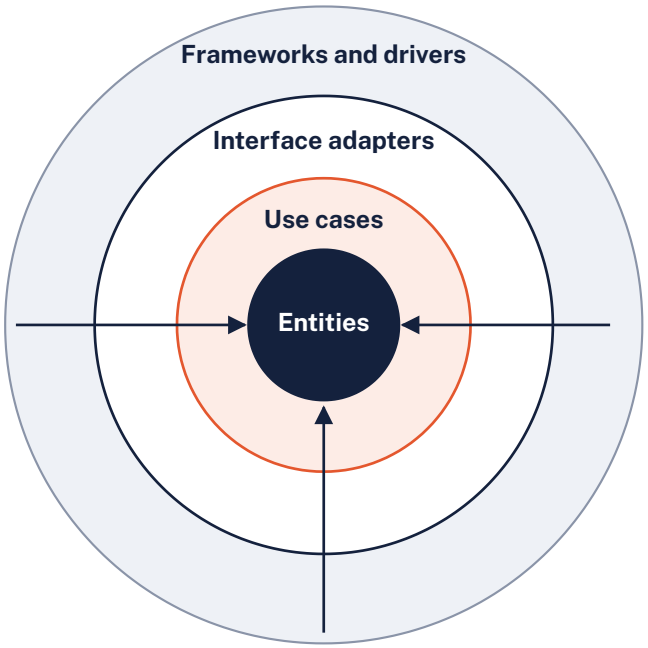
Compared with hexagonal architecture, onion adds named rings inside the application and makes explicit a point Cockburn's picture leaves implicit: the database is not the foundation of the system but an external service at the edge. Repository interfaces live in the core; their implementations live in the infrastructure ring.

Clean architecture and the dependency rule

Robert C. Martin published "The Clean Architecture" on his blog in August 2012 as a synthesis of hexagonal architecture, onion architecture and other similar proposals, and expanded it into the book *Clean Architecture* in 2017. His picture has four rings. **Entities** hold enterprise-wide business rules. **Use cases** hold application-specific rules that orchestrate entities. **Interface adapters** convert data between the form the use cases need and the form external agencies need: controllers, presenters and gateways. **Frameworks and drivers** form the outer ring: the web framework, the database, the UI.

The dependency rule ties it together: source code dependencies point only inwards, and nothing in an inner ring may name anything declared in an outer ring, whether a class, a function or a data format. When control must flow outwards (a use case saving an order), the inner ring declares an interface and the outer ring implements it. Data that crosses a boundary should be simple structures in the inner ring's terms, never a database row or an HTTP request object. Martin also argues that the top-level structure should reveal the system's purpose (orders, payments, deliveries) rather than its framework, an idea he calls screaming architecture.

The clean architecture rings



Outer ring: web framework, ORM, database, message broker, UI

Interface adapters: controllers, presenters, repository implementations

Use cases: application-specific rules, e.g. PlaceOrder

Entities: enterprise-wide rules, e.g. Order, Money

The dependency rule: source code dependencies point only inwards

Each ring may depend only on rings inside it; entities depend on nothing, and frameworks and drivers sit at the edge where they can be replaced.

Hexagonal, onion and clean architecture compared

Aspect	Hexagonal (ports and adapters)	Onion	Clean
Originator and date	Alistair Cockburn; written up as an article in 2005	Jeffrey Palermo; blog series from July 2008	Robert C. Martin; blog post August 2012, book 2017
Picture	A hexagon with an inside (application) and an outside (adapters), left side driving, right side driven	Concentric rings: domain model, domain services, application services, then UI, infrastructure and tests	Concentric rings: entities, use cases, interface adapters, frameworks and drivers
Central rule	The application talks to the world only through ports; adapters translate	All coupling is towards the centre; the core defines interfaces that the outer ring implements	The dependency rule: source code dependencies point only inwards
Internal structure prescribed	None: only inside versus outside	Yes: named inner rings for the domain	Yes: entities and use cases, plus guidance on presenters and data crossing boundaries
Main emphasis	Symmetry: users, tests, scripts and other programs can all drive the application	Putting the domain model, not the database, at the centre of a business application	Independence from frameworks, UI, database and external agencies; testable business rules
Typical vocabulary	Port, adapter, driving (primary), driven (secondary), actor	Domain model, domain services, application services, infrastructure	Entity, use case (interactor), gateway, presenter, controller

A use case with a port and two adapters

The following Python example implements the food-delivery platform's `PlaceOrder` use case in the hexagonal way. The first snippet is the inside: a small domain class, two driven ports written as `typing.Protocol` classes, and the use case, which acts as the driving port. The second snippet is the outside: a PostgreSQL adapter and an in-memory test double that implement the same port. The third shows why it matters: a fast unit test, and the composition root where production adapters are plugged in.

```
from dataclasses import dataclass
from decimal import Decimal
from typing import Protocol

@dataclass
class Order:
    id: str
    total: Decimal
    status: str = "pending-payment"

class OrderRepository(Protocol):        # driven port
    def save(self, order: Order) → None: ...

class PaymentGateway(Protocol):         # driven port
    def authorise(self, order_id: str, amount: Decimal,
                  idempotency_key: str) → bool: ...

class PlaceOrder:                       # driving port + use case
    def __init__(self, orders: OrderRepository, payments: PaymentGateway):
        self.orders, self.payments = orders, payments

    def __call__(self, order_id: str, total: Decimal) → Order:
        if total ⩽ 0:
            raise ValueError("empty basket")
        order = Order(order_id, total)
        self.orders.save(order)                # save first: a crash cannot lose a
charged order
        approved = self.payments.authorise(order_id, total, idempotency_key=order_id)
        order.status = "placed" if approved else "payment-failed"
        self.orders.save(order)
        return order
```

Inside the hexagon: a domain rule, two driven ports and the PlaceOrder use case. Nothing here imports a web framework, a database driver or a payment SDK. The order is saved before the card is authorised, and the order ID doubles as the idempotency key, so a crash or a retry cannot charge a customer without an order or charge them twice.

```
class PostgresOrderRepository:          # real driven adapter
    def __init__(self, conn):
        self.conn = conn
    def save(self, order):
        with self.conn.cursor() as cur:
            cur.execute(
                "INSERT INTO orders (id, total, status) VALUES (%s, %s, %s) "
                "ON CONFLICT (id) DO UPDATE SET status = EXCLUDED.status",
                (order.id, order.total, order.status))
        self.conn.commit()

class InMemoryOrderRepository:          # test double, same port
    def __init__(self):
        self.saved = {}
    def save(self, order):
        self.saved[order.id] = order
```

Outside the hexagon: a real adapter and a test double for the same port. The use case cannot tell them apart.

```
class DecliningPayments:
    def authorise(self, order_id, amount, idempotency_key): return False

def test_failed_payment_is_recorded():
    orders = InMemoryOrderRepository()
    place = PlaceOrder(orders, DecliningPayments())
    place("A-17", Decimal("23.50"))
    assert orders.saved["A-17"].status == "payment-failed"

# composition root (main.py): the only place that knows every adapter
# place_order = PlaceOrder(PostgresOrderRepository(conn), CardProviderGateway(key))
# app.add_route("/orders", make_rest_controller(place_order))   # driving adapter
```

A unit test drives the use case directly, with no web server, database or network; production wiring happens in one place at start-up.

Notice what the use case does not contain: no SQL, no HTTP status codes, no provider-specific exception types, no framework decorators. The test runs in milliseconds and covers a case (a declined payment) that would be awkward to arrange against a real provider's sandbox. The PostgreSQL adapter needs its own tests, but those are focused integration tests of one class against a real database, not tests of business rules. This is the testability benefit described in Section 2.6: controllability and observability through injected dependencies.

WORKED EXAMPLE

Worked example: swapping the payment provider

The food-delivery platform must move from one card provider to another within six weeks because of a pricing change. Compare the work in two versions of the system: one where checkout code calls the old provider's SDK directly in 23 places, and one built with a `PaymentGateway` driven port and an adapter.

Direct calls: find the impact	Search finds the SDK in checkout, refunds, subscriptions and three admin screens. The SDK's own exception types are caught in 11 places, and its `Charge` object is stored in a database column.
Direct calls: the change	Every call site changes, plus a data migration for the stored objects. Every test that touched payments was written against the old SDK's sandbox and must be rewritten. Estimate: 5 to 6 weeks, with a risky all-at-once switch.
Ports and adapters: find the impact	One adapter class (about 200 lines) implements `PaymentGateway`. The core only knows `charge`, `refund` and the core's own `PaymentResult` type.
Ports and adapters: the change	Write a second adapter for the new provider; run the existing contract tests for the port against it; choose the adapter per order with a feature flag so that 1%, then 10%, then 100% of payments use the new one. Estimate: 2 weeks, reversible at every step.
What the port cost	Designing the port and adapter at the start took about 3 extra days, and every new payment feature needs a small change to the port as well as the adapter.

The value of a port is proportional to the probability and cost of the change it isolates. Payment providers, message brokers and third-party APIs change often; your own core database engine may never change, and wrapping it in elaborate ports is harder to justify.

The port must speak the domain's language (`charge`, `PaymentResult`), not the old provider's. A port that mirrors one vendor's API protects nothing.

Characteristics of the ports-and-adapters family

Characteristic	Rating	Reason
Testability	High	Business rules can be tested in memory, fast and deterministically, by plugging test doubles into the driven ports.
Modifiability of infrastructure	High	Replacing a database, payment provider or web framework means writing a new adapter; the core does not change.
Modifiability of business rules	Medium to high	Rules live in one place, free of framework code, but a feature still crosses port, use case and adapter.
Simplicity	Low to medium	More types, interfaces and mapping code than a straightforward layered application; harder for newcomers at first.
Development speed (early)	Medium	Extra structure slows the first features of a simple CRUD application; it pays back as the domain grows.
Overall cost	Cheap to medium	No extra infrastructure; the cost is developer time spent writing and maintaining ports, adapters and mapping code.
Performance	Medium to high	Interface calls cost almost nothing; the real overheads are object mapping between layers and losing some ORM conveniences.
Scalability, deployability, fault tolerance	Not decided by this style	These depend on the deployment style (monolith or services) that the hexagon sits in.

Costs, criticism and when to use it

The benefits are real, and so are the costs. Every port is an extra interface; every boundary may need mapping code to convert between, say, an ORM entity, a domain object and an API response. For an application that mostly moves form fields into tables with little logic in between, this structure adds files without adding protection, and a framework's conventions (an active-record model, a generated admin interface) may give a better result faster. Some experienced practitioners argue that strict versions of clean architecture produce too much indirection and that most teams never swap their database; others reply that the main payoff is not swapping technology but fast, reliable tests and business code that can be read without knowing the framework. Both are right in different contexts.

Use it when the business logic is substantial and will outlive the current frameworks; when the system integrates with external services that are likely to change; when fast, isolated tests of business rules are important; and as the internal structure of modules in a modular monolith or of services built with domain-driven design (Section 5.7). **Be cautious** with simple CRUD

applications, prototypes and short-lived tools, where the framework's default structure is usually good enough. A pragmatic middle path is common: ports for external services and anything likely to change, and the framework's own conveniences elsewhere.

WATCH OUT

Common anti-patterns

- **Technology-shaped ports.** A port called `SqlQueryExecutor` or one that mirrors a vendor's SDK keeps the coupling it was meant to remove. Ports should speak the domain's language.
- **Framework leakage.** Use cases that accept HTTP request objects or return ORM entities have outward dependencies in disguise.
- **Logic in adapters.** Business rules in controllers or repository classes are invisible to the use-case tests and duplicated across adapters.
- **An interface for every class.** Ports belong at the boundary with the outside world, not between every pair of internal classes.
- **Anaemic pass-through use cases.** Use cases that only forward calls to a repository are the architecture sinkhole of Section 5.2 in a new form; accept them for simple reads, but notice if they are the majority.
- **A composition root everywhere.** Adapters created deep inside business code (`PostgresOrderRepository()` in a use case) defeat the substitution the style depends on.

Enforcing the dependency rule

As with layers, the rule erodes unless a machine checks it. In Python, import-linter can declare the rings as a *layers* contract (adapters above application above domain) and add *forbidden* contracts that stop the domain and application packages importing web frameworks, database drivers or SDKs. In Java, ArchUnit has a ready-made `onionArchitecture()` rule in which you name the domain model, domain service, application service and adapter packages, and it fails a test if any dependency points outwards. Splitting the code into separate build modules (a domain module with no framework dependencies at all) makes outward dependencies impossible to compile, which is the strongest check of all. These checks are examples of the fitness functions covered in Chapter 11.

A package structure that makes the rule visible helps reviewers: for example `orders/domain`, `orders/application` (use cases and ports) and `orders/adapters/{rest,postgres,payments}`. Organising by business area first and ring second keeps the structure aligned with the domain boundaries of sections 5.3 and 5.7.

CHECK YOUR UNDERSTANDING

▼ Is a REST controller a driving or a driven adapter? And a SQL repository?

A REST controller is a driving adapter: it receives requests and calls the application. A SQL repository is a driven adapter: it implements an interface that the application calls.

▼ In clean architecture, may a use case return an ORM entity to a controller?

No. The ORM belongs to the outer ring, so a use case that names it has an outward dependency. It should return a simple structure or domain object defined in an inner ring.

▼ Where are adapters created and connected to the application?

In a composition root, usually the start-up code, which is the only place that knows about every adapter.

▼ What does onion architecture add to the hexagonal picture?

Named rings inside the application (domain model, domain services, application services) and the explicit statement that the database is an external concern at the edge, not the foundation.

▼ Try it: A ticketing product has complex seat-allocation rules and a simple admin screen for editing venue contact details. Would you use hexagonal architecture for both? Explain what you would gain or lose in each case.

Use it for seat allocation only. Those rules gain fast, isolated tests, and the database or broker can be replaced by swapping an adapter, which suits logic that matters and changes. The admin screen is simple CRUD, so ports and mapping code would add indirection with little to protect. Use a plain approach there, keeping it separate from the core. Apply the pattern where complexity justifies its cost, and enforce the dependency rule automatically.

SOURCES

- Alistair Cockburn: Hexagonal architecture: https://alistair.cockburn.us/hexagonal-architecture/
- Jeffrey Palermo: The Onion Architecture, part 1 (2008): https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/
- Robert C. Martin: The Clean Architecture (2012): https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
- ArchUnit user guide (layered and onion architecture rules): https://www.archunit.org/userguide/html/000_Index.html
- import-linter documentation: https://import-linter.readthedocs.io/en/stable/

SECTION 5.7 OF 5.8 · INTERMEDIATE · 26 MIN READ · LAST REVIEWED NOVEMBER 2026

Strategic domain-driven design: bounded contexts and context maps

BEFORE YOU READ THIS

This section builds on:

- 3.1 Modularity
- 3.7 Conway's law
- 5.6 Hexagonal and clean

Terms it uses from earlier: Adapter, Modular monolith, Big ball of mud, Conway's law

THE IDEA IN BRIEF

In a hospital, the word "patient" means something different to the reception desk (name, address, appointment), the ward (bed, medication, observations) and the billing office (insurer, invoice). If one giant form tried to serve them all, it would be huge and everyone would fight over it. It works better when each department keeps its own simple record and they agree how to pass information between them. Domain-driven design applies that idea to software: it finds the natural departments of a business, gives each its own model and vocabulary, and makes the connections between them explicit.

WHAT YOU NEED TO KNOW

- Domain-driven design (Eric Evans, 2003) builds software around a model of the business domain developed with domain experts and expressed in a ubiquitous language used in speech and code alike.
- Subdomains are core (where the business competes), supporting (specific but not differentiating) or generic (solved problems); the classification decides where to invest design effort and what to buy.
- A bounded context is a boundary within which one model and one language apply consistently; the same word can safely mean different things in different contexts.
- A context map shows the relationships between contexts: partnership, shared kernel, customer-supplier, conformist, anticorruption layer, open host service, published language and separate ways.
- Event storming (Alberto Brandolini) discovers contexts by putting domain events on a timeline with the people who know the business.

> - Aggregates are consistency boundaries changed through a root that enforces invariants; bounded contexts are the best guide to module and service boundaries.

Why domain-driven design belongs in a chapter about structure

Every style in this chapter needs boundaries. A modular monolith needs module boundaries; the microservices of Chapter 6 need service boundaries; even a hexagonal application needs to know what belongs inside the hexagon. Domain-driven design (DDD) is the most widely used method for finding those boundaries in the business domain itself, rather than in technical layers.

Eric Evans introduced it in *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Addison-Wesley, 2003). His argument is that in most business software the hardest problems are not technical but in understanding the domain: what a policy, a claim, a delivery or a refund really is, and which rules govern it. Software should therefore be built around a deliberately designed model of the domain, developed jointly by developers and domain experts, and that model should be protected from confusion with other models. In 2015 Evans published a short *DDD Reference* that summarises the patterns and adds a few, such as partnership and big ball of mud as context-map patterns, and domain events as a building block.

DDD has two halves. **Strategic design** divides a large domain into subdomains and bounded contexts and maps the relationships between them; it is architecture, and it is the main subject here. **Tactical design** provides building blocks (entities, value objects, aggregates and others) for writing the model inside one context; it is design, and it is covered briefly.

Ubiquitous language

A ubiquitous language is a shared vocabulary, built by developers and domain experts together, that is used consistently in conversation, documents, tests and code. If dispatchers say a courier is "stacked" when carrying two orders from the same restaurant, then the code says `stack`, not `multiOrderAssignment`. When a term turns out to be ambiguous or awkward, the team changes the language and the code together.

This sounds like a matter of naming, but it has architectural consequences. Translation between the words of the business and the words of the code is where misunderstandings, and therefore defects, are born. And the moment two groups use the same word with different meanings, you have found evidence of two different models, which is the starting point for bounded contexts.

Subdomains: core, supporting and generic

The business domain as a whole divides into subdomains: areas of activity such as menus, ordering, payments and dispatch. Practitioners classify them into three kinds, building on Evans's distinction between the core domain and generic subdomains; the three-way split with supporting subdomains was popularised by later writers, notably Vaughn Vernon in *Implementing Domain-Driven Design* (2013).

The core domain is what makes the business different from its competitors and is the reason the software is worth building. For a food-delivery company it is probably dispatch: matching orders to couriers and predicting arrival times. A supporting subdomain is specific to the business and necessary, but not a source of advantage: menu management, courier onboarding. A generic subdomain is a problem many businesses share, with good existing solutions: payments, identity, email and SMS.

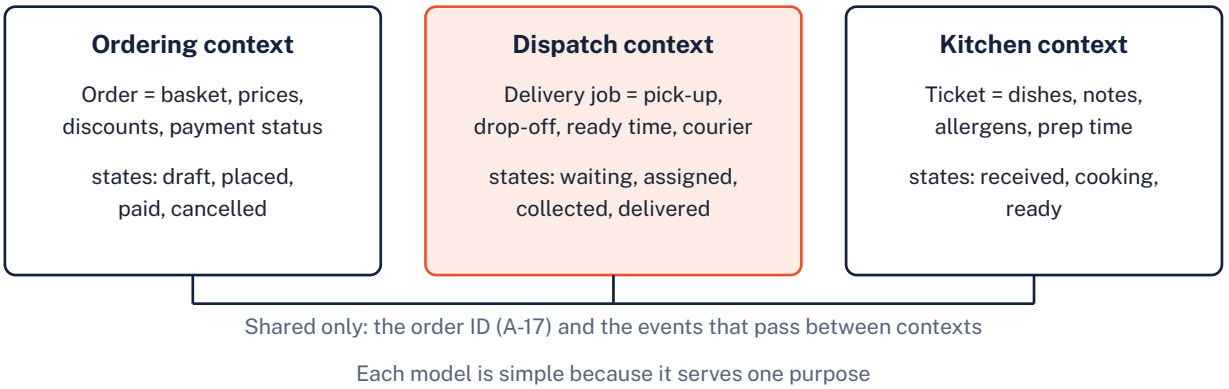
The classification is an investment decision. The core deserves the most experienced people, the richest modelling and code you own completely, because that is where better software turns directly into business advantage. Supporting subdomains deserve sound but simpler designs, often a straightforward layered module. Generic subdomains should usually be bought, used as a service or built as thin adapters over open-source components: building your own identity provider is effort spent where you cannot win. The classification also changes over time; a supporting capability can become core when the business strategy changes.

Bounded contexts

A bounded context is an explicit boundary within which one model, and one ubiquitous language, applies consistently. Inside it, "order" has exactly one meaning. Outside it, the same word may mean something else, and that is acceptable, because each model is designed for one purpose. Evans observed that trying to unify the model of a large system into one is neither feasible nor cost-effective; the attempt produces enormous shared classes that serve every purpose badly and that every team must change.

Subdomains and bounded contexts are different things, though often aligned. A subdomain is part of the problem: it exists in the business whether or not software does. A bounded context is part of the solution: a boundary you choose for a model and, usually, for code and data owned by one team. The ideal is one bounded context per subdomain, but legacy systems often have one context spanning several subdomains, and a large subdomain may need several contexts.

One word, three models



"Order" means a basket with prices to Ordering, a pick-up and drop-off job to Dispatch and a list of dishes to cook in the Kitchen context; each model stays small because it serves one purpose.

IF YOU ARE NEW TO THIS

Step by step: the business's parts and the software's boundaries

The step most people trip over here is the difference between a subdomain and a bounded context. They often line up, so they are easy to confuse, but one is part of the business problem and the other is a boundary the team chooses in the software.

- Dispatch, matching orders to couriers and predicting arrival times, is a subdomain of the food-delivery business. It exists whether or not any software does: a dispatcher with a phone and a map would still be doing it.
- A bounded context is a decision about the software: a boundary inside which one model and one language apply. The team decides that inside the Dispatch context, the thing being handled is a 'delivery job' with a pick-up, a drop-off, a ready time and a courier.
- Ideally each subdomain gets exactly one bounded context, usually owned by one team, and Dispatch and its context line up like that.
- The fit is not always so neat. In an older system, one shared Order class may serve ordering, dispatch and the restaurants' kitchen view at once: one model, and so one context, stretched across several subdomains, which every team must change and nobody can keep simple.
- The opposite also happens: a large subdomain may need several bounded contexts, each with a smaller model.
- The subdomains can stay as they are while the team redraws its contexts, as the worked example later in this section does when it splits that shared Order class into three models.
- **The lesson:** find subdomains by studying the business; choose bounded contexts by deciding where one model and one language should hold. Aim for one context per subdomain, and notice where the two do not match.

Context maps

A context map shows the bounded contexts of a system and the relationships between them. The relationships describe both the technical integration and the power and communication between the teams, which is why context mapping is as much about organisations as about code. It is a direct application of Conway's law: the map shows where the organisation's communication paths and the system's integration paths must line up.

Many relationships are **upstream/downstream**: the upstream context's decisions affect the downstream one, but not the reverse. The downstream team then has a choice about how to cope, and that choice is what most of the patterns name.

Context map relationship patterns

Pattern	What it means	Who adapts to whom	Food-delivery example
Partnership	Two teams succeed or fail together, so they plan jointly and evolve their interface together.	Both, by agreement	Ordering and Dispatch release the order-to-delivery hand-over together.
Shared kernel	Two contexts share a small, explicitly agreed part of the model and its code; changes need both teams' consent.	Both, carefully	Dispatch and Courier onboarding share a tiny courier-profile library (vehicle type, courier ID).
Customer-supplier	The upstream team (supplier) commits to meeting the downstream team's (customer's) needs, which are part of its planning.	Upstream serves downstream	Menu catalogue supplies menus and prices to Ordering, and Ordering's needs go on Menu's backlog.
Conformist	The downstream team adopts the upstream model as it is, because it cannot influence it and translation is not worth the cost.	Downstream conforms	Dispatch uses the maps provider's route and location model directly.
Anticorruption layer	The downstream team builds a translation layer so that the upstream model cannot leak into its own.	Downstream translates	Menu translates dozens of restaurant point-of-sale formats into its own menu model; Payments isolates the provider's API.
Open host service	The upstream context offers a well-defined protocol or API for all consumers, instead of a custom integration for each.	Upstream publishes once	Payments offers one documented payments API to Ordering, refunds and subscriptions.
Published language	A documented, shared exchange format, often used with an open host service.	Both use the language	Order events (OrderPlaced, OrderCancelled) have a versioned, documented schema.
Separate ways	The contexts are not integrated at all, because the benefit is lower than the cost.	Neither	Courier payroll reporting is done in a separate finance tool fed by a monthly export, not live integration.

Two of these patterns deserve special attention. An anticorruption layer (ACL) is a set of adapters and translators that a downstream context builds so that a foreign model, often a legacy system or an external provider, cannot leak into its own model. It is the same idea as the driven adapter of Section 5.6, applied between contexts, and it is the default choice for integrating with anything you do not control. An open host service together with a published language is the upstream's way of serving many consumers well: one documented, versioned protocol instead of a custom integration for each. A shared kernel should stay small; a large one turns two contexts back into one with two teams arguing over it, and Evans's 2015 reference adds big ball of mud as the honest label for an area with no clear boundaries at all, to be contained rather than extended.

Event storming: discovering the contexts

Boundaries are discovered with domain experts, not designed alone. Event storming, created by Alberto Brandolini and first described on his blog in 2013, is the most popular workshop format for this. The people who know the business and the people who build the software stand at a long wall of paper and write domain events (things that happened, in the past tense: *Order placed*, *Courier assigned*, *Food collected*) on orange sticky notes, placing them on a timeline.

The group then adds the commands that cause events, the people and external systems involved, the policies that react to events ("whenever an order is placed, find a courier") and, crucially, the hot spots where people disagree or nobody knows. Clusters of events that share a language and a purpose suggest bounded contexts; places where the same word is used differently suggest boundaries; places where a hand-over happens suggest integration points. A big-picture session for a whole business takes a day or two and needs almost no preparation beyond the right people and a lot of wall space.

Tactical patterns in brief

Inside one bounded context, DDD offers building blocks for the model. An entity has an identity that persists while its attributes change: an order is the same order after its status changes. A value object has no identity and is defined only by its values: two `Money(12.50, "GBP")` objects are interchangeable, and value objects are usually immutable. A domain event records something meaningful that happened in the domain.

An aggregate is a cluster of entities and value objects treated as one unit for changes, with one entity as its root. Outside code may hold references only to the root and must change the aggregate only through it, so the root can enforce the aggregate's invariants (rules that must always hold, such as "no items can be added after an order is placed"). One transaction should change one aggregate; other aggregates are referenced by ID and updated through domain

events, which makes aggregates the natural unit of consistency and later of distribution. A repository loads and saves whole aggregates and looks to the domain like a collection; its interface is a driven port in hexagonal terms.

```
from dataclasses import dataclass, field
from decimal import Decimal

@dataclass(frozen=True)
class Money:                                # value object: equal if values equal
    amount: Decimal
    currency: str
    def __add__(self, other):
        if other.currency ≠ self.currency:
            raise ValueError("currency mismatch")
        return Money(self.amount + other.amount, self.currency)
    def times(self, n: int):
        return Money(self.amount * n, self.currency)

@dataclass
class OrderLine:                            # entity inside the aggregate
    dish_id: str
    qty: int
    price: Money

@dataclass
class Order:                                # aggregate root
    id: str
    restaurant_id: str                      # other aggregates referenced by ID only
    lines: list[OrderLine] = field(default_factory=list)
    status: str = "draft"
    events: list = field(default_factory=list)

    def add(self, dish_id, qty, price):
        if self.status ≠ "draft":
            raise ValueError("order already placed")    # invariant
        self.lines.append(OrderLine(dish_id, qty, price))

    def place(self, minimum: Money):
        total = Money(Decimal(0), minimum.currency)
        for line in self.lines:
            total = total + line.price.times(line.qty)
        if total.amount < minimum.amount:
            raise ValueError("below restaurant minimum")  # invariant
        self.status = "placed"
        self.events.append({"type": "OrderPlaced", "order_id": self.id, "total":
str(total.amount)})
```

An Order aggregate: a value object (Money), an entity (OrderLine), the aggregate root enforcing invariants, and a domain event recorded when the order is placed.

The example keeps the aggregate small, references the restaurant by ID rather than holding a restaurant object, and enforces its rules in methods rather than leaving them to callers. A common mistake is the opposite: aggregates so large that every user's change locks the same rows, or domain objects that are only bags of getters and setters while the rules live in service classes (Martin Fowler calls this the anaemic domain model).

WORKED EXAMPLE

Worked example: a context map for the food-delivery platform

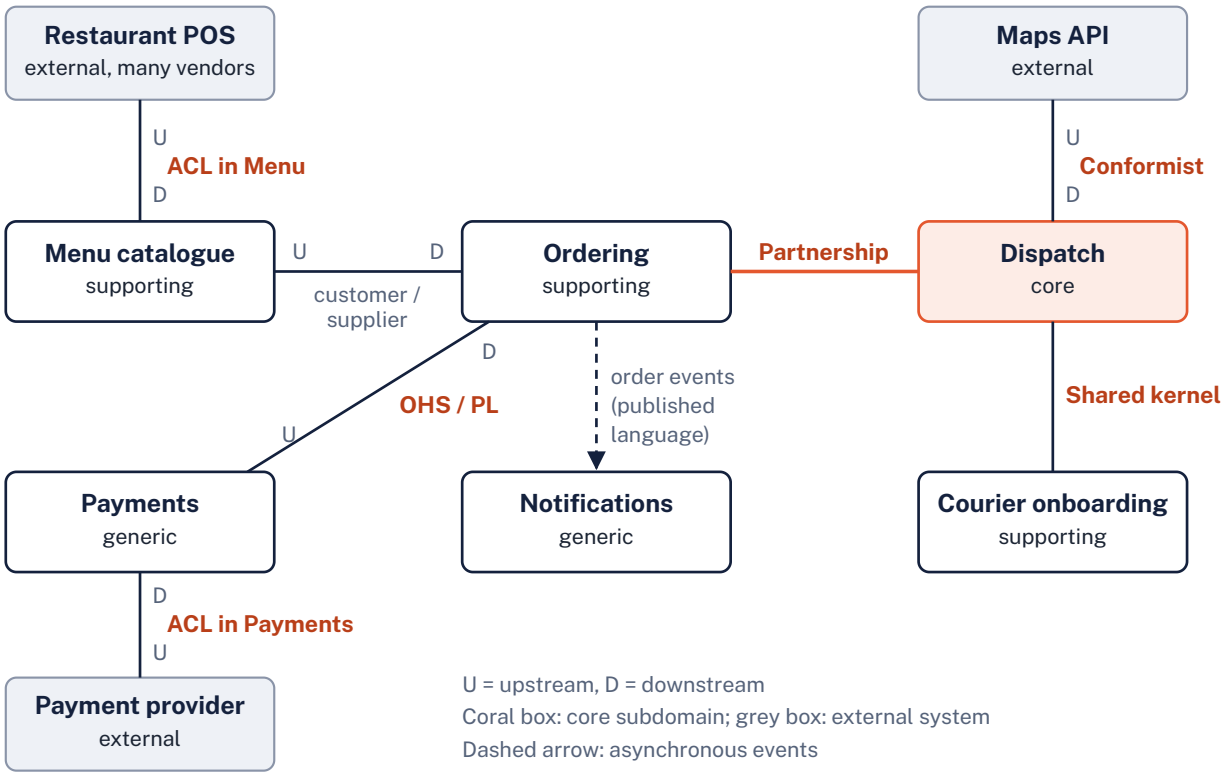
The platform has grown to about 60 engineers in eight teams, and every change seems to touch the shared `Order` class, which now has 94 fields. The architect runs a two-day event-storming workshop with product managers, operations staff, restaurant support and engineers, then draws the context map shown below.

1. Find the subdomains	From the event timeline: menu management, ordering and checkout, payments, dispatch (matching couriers to orders and predicting arrival times), courier onboarding, customer notifications, restaurant integration.
2. Classify them	Dispatch is the **core**: faster, cheaper deliveries are what win customers and restaurants. Menu, ordering and courier onboarding are **supporting**: necessary and specific to this business but not where it competes. Payments and notifications are **generic**: solved problems, best bought or built thinly over a provider.
3. Draw bounded contexts	Each subdomain gets its own model. "Order" in Ordering (basket, prices, payment status) is split from "delivery job" in Dispatch (pick-up, drop-off, ready time) and "ticket" in the restaurant's kitchen view. The 94-field class becomes three focused models linked by the order ID.
4. Name the relationships	Ordering and Dispatch: partnership. Menu to Ordering: customer-supplier. Payments: open host service with a published language, and an anticorruption layer towards the external provider. Menu: anticorruption layer towards restaurant point-of-sale systems. Dispatch: conformist to the maps API, a deliberate choice because translating its rich model would cost more than it saves.
5. Decide investment	The best engineers and the most careful modelling go to Dispatch. Payments uses a provider through a thin adapter. Notifications uses a bought messaging service.
6. Decide the deployment	The contexts first become modules of the existing application, turning it into a modular monolith (Section 5.3), each with its own schema and public interface. Dispatch, which has very different scaling and release needs, is the first candidate to become a separate service later.

Classifying Ordering as supporting surprised some stakeholders. The test is not importance (the company cannot work without ordering) but differentiation: would a better ordering model win the business more customers than a better dispatch model?

The conformist relationship with the maps provider is a conscious trade-off, recorded in a decision record, not an accident. If the provider changes or is replaced, Dispatch will feel it.

The context map



Each line names the relationship pattern; U and D mark upstream and downstream, and the anticorruption layers sit on the downstream side towards systems the platform does not control.

Bounded contexts as module and service boundaries

Bounded contexts are the best available guide to where to draw module boundaries in a monolith and service boundaries in a distributed system. A context has its own model, its own language, its own data and ideally one owning team, which are exactly the properties that make a module cohesive and loosely coupled. In a modular monolith each context becomes a module with its own schema; inside it, the hexagonal structure of Section 5.6 keeps the model independent of infrastructure.

If a context later needs to be deployed separately, its boundary is already the right shape for a service, and its relationships on the context map tell you what the integration must look like. This is why many practitioners recommend finding contexts first and deciding deployment second. The reverse also holds: services cut across bounded contexts (an "order service" that every context writes to) produce the tightly coupled distributed monolith discussed in Section 6.3. Note also that a bounded context is not automatically one microservice; a context can be deployed as several services, and several small contexts can share one deployment.

What strategic DDD buys and costs

Aspect	Effect	Reason
Modifiability within a context	High	One model, one language and one owner: changes stay local and are easy to reason about.
Team autonomy	High	Context boundaries align with team boundaries, so teams can change their models without negotiating every change.
Fit between software and business	High	Ubiquitous language and joint modelling reduce translation errors between experts and code.
Upfront and ongoing effort	Medium to high	Modelling sessions, domain experts' time, translation layers and contract maintenance all cost effort.
Data duplication and integration	Medium cost	Each context keeps its own copy of what it needs, which must be kept in step through events or APIs.
Value for simple domains	Low	For CRUD-heavy systems with little logic, tactical DDD in particular adds ceremony with little return.

WATCH OUT

Common mistakes

- **Tactical DDD without strategy.** Teams adopt entities, repositories and aggregates everywhere but never draw bounded contexts, so they get the ceremony without the boundaries.
- **One context per database table or per noun.** Contexts follow language and purpose, not data entities; "Customer service" and "Order service" for every noun produce chatty, coupled parts.
- **Modelling without domain experts.** A ubiquitous language invented by developers alone is just a naming convention.
- **Treating the map as permanent.** Contexts and relationships change as the business and organisation change; revisit the map at least yearly.
- **Gold-plating generic subdomains.** Engineers often find payments or identity more interesting than the core; the classification exists to resist that.

CHECK YOUR UNDERSTANDING

▼ What is the difference between a subdomain and a bounded context?

A subdomain is part of the business problem and exists whether or not software does; a bounded context is a boundary chosen in the solution within which one model applies. Ideally they align, but they need not.

▼ Your team integrates with a legacy billing system whose model is confusing and that you cannot change. Which context-map pattern protects your model?

An anticorruption layer: translators on your side that convert the legacy model into your own terms, so it cannot leak into your context.

▼ Why should one transaction usually change only one aggregate?

Because the aggregate is the unit whose invariants must hold together; changing several at once creates large locks and coupling, whereas other aggregates can be updated through domain events, which also allows them to be distributed later.

▼ Why is payments usually a generic subdomain for a food-delivery company?

Because taking payments is a solved problem shared by many businesses and is not where the company competes, so it is better bought or built thinly over a provider than modelled in depth.

▼ Try it: In a ticketing firm, "ticket" means a sold seat to Sales and a customer complaint to Support, and one shared Ticket class has grown to 80 fields. Payments comes from an outside provider. What contexts and relationships would you propose?

Split the class into bounded contexts, Sales and Support, each with its own model and language, so "ticket" can safely differ, and Support might say "case". Seat allocation is probably the core subdomain, Support is supporting and email is generic, so invest and buy accordingly. Protect Sales from the outside payment provider with an anticorruption layer. Between Sales and Support, a customer-supplier relationship can work, sharing only a booking reference. Check with domain experts through event storming.

SOURCES

- Eric Evans: Domain-Driven Design Reference (2015, PDF): https://www.domainlanguage.com/wp-content/uploads/2016/05/DDD_Reference_2015-03.pdf
- Martin Fowler: Bounded Context: https://martinfowler.com/bliki/BoundedContext.html
- Martin Fowler: DDD Aggregate: https://martinfowler.com/bliki/DDD_Aggregate.html

- EventStorming (Alberto Brandolini): https://www.eventstorming.com/

When a monolith is the right answer

BEFORE YOU READ THIS

This section builds on:

- 5.1 Styles and patterns
- 5.3 Modular monolith
- 5.7 Domain-driven design

Terms it uses from earlier: Modular monolith, Bounded context, Deployment unit, Big ball of mud, Load balancer

THE IDEA IN BRIEF

A new restaurant usually starts with one kitchen where every cook can see and help every other. Opening five separate kitchens across town on day one, each with its own staff, deliveries and manager, would cost far more and make every change to the menu a negotiation. Only when the one kitchen is truly overwhelmed, or one dish needs special equipment, does it make sense to open another. Software is similar: a single, well-organised program is usually the cheapest and fastest way to start, and splitting it up is worth it only when there is clear evidence that one unit has become the bottleneck.

WHAT YOU NEED TO KNOW

- A monolith is a deployment decision (one unit built and released together), not a verdict on internal quality; a well-modularised monolith is a respectable, often superior, architecture.
- Martin Fowler's microservice premium and MonolithFirst (2015) argue that microservices pay off only above a level of complexity, and that most successful microservice systems began as monoliths.
- David Heinemeier Hansson's majestic monolith (2016) makes the case for small teams; Stefan Tilkov (2015) warns that unmodularised monoliths are hard to split, so enforce module boundaries from the start.
- A monolith is usually right for small teams, uncertain domains, early products, strong consistency needs and limited operations capacity.
- Monoliths scale through stateless replicas behind a load balancer, read replicas, caching, background workers and, if needed, partitioning by tenant.
- Split when there is evidence: team contention, deployment queues, wildly different scaling or compliance needs; split incrementally along bounded contexts.

The question behind the chapter

The styles in this chapter all deploy as one unit. Chapter 6 takes the same questions across a network, where every call can fail, be slow or arrive twice. Before crossing that line it is worth asking directly: when is a monolith the right answer? For much of the 2010s the industry's default answer was "rarely", and many teams split young systems into microservices because large, famous companies had done so. A decade of experience has produced a more balanced view, and this section sets it out.

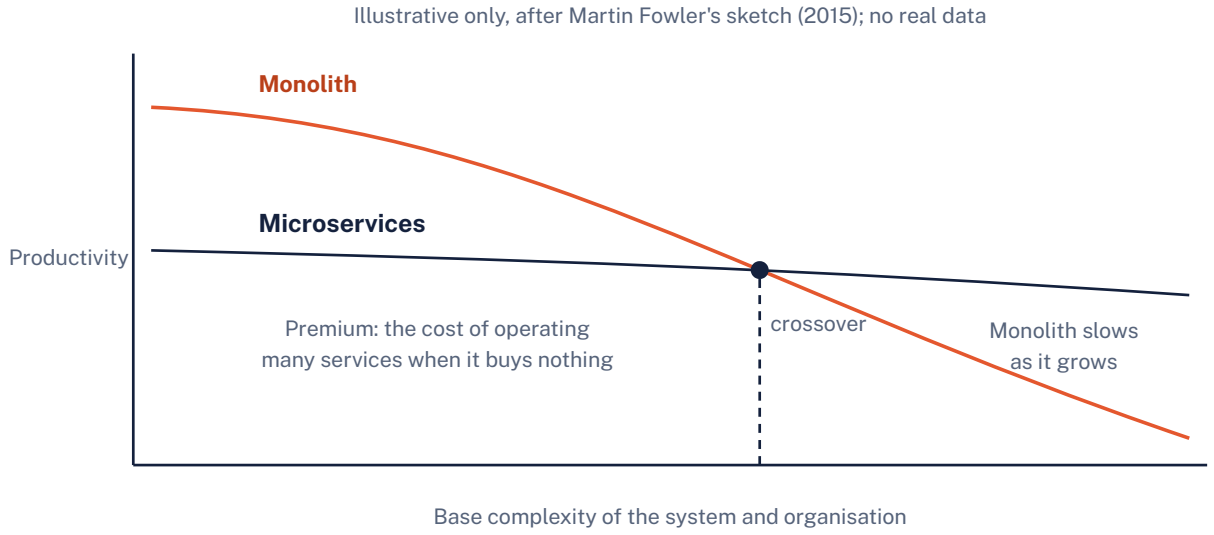
The word monolith needs care. As Section 5.1 explained, it describes the deployment (one unit that is built, tested and released together), not the internal quality. A monolith can be a well-structured modular system or a big ball of mud, and the same is true of a set of services. Many of the problems blamed on monoliths are really problems of poor modularity, and splitting a poorly modularised system into services produces a distributed big ball of mud, which is worse.

MonolithFirst and the microservice premium

In May 2015 Martin Fowler described the microservice premium: microservices bring real costs of their own (automated deployment of many units, monitoring, handling partial failure, eventual consistency, versioned APIs between teams), and those costs slow a team down unless the system is complex enough for microservices to pay them back. His advice was to default to a single, well-modularised monolith and move to microservices only when complexity demands it.

A month later, in an article titled MonolithFirst (June 2015), he added two observations from the projects he had seen: almost all successful microservice systems had started as a monolith that grew too big and was then broken up, and almost all systems built as microservices from scratch had ended up in serious trouble. One reason is that good service boundaries are hard to find before the domain is understood, and a wrong boundary between services is far more expensive to move than a wrong boundary between modules. He described several routes: build a monolith and peel services off its edges later; build a deliberately disposable monolith; or start with a few coarse-grained services and split them further once boundaries settle.

The microservice premium



A conceptual sketch: microservices only pay off beyond some level of complexity; below it their operational cost is a premium that buys nothing.

The majestic monolith, and the opposing view

In February 2016 David Heinemeier Hansson, creator of Ruby on Rails, published "The Majestic Monolith" on the Signal v. Noise blog. He described Basecamp, then built by a team of about a dozen programmers, as one integrated application serving web, mobile, desktop and email clients, and argued that small teams should embrace a well-crafted monolith instead of copying the architecture of organisations with thousands of engineers. His key line was “Don't distribute your computing!” unless you truly have to.

Practitioners do not all agree. Also in June 2015, Stefan Tilkov argued on Fowler's site, in "Don't start with a monolith", that splitting a monolith later is often harder than expected, because its parts grow tightly coupled through shared libraries, domain objects and the database. If the system is clearly large enough to justify separate parts and the domain is well understood, he recommended designing separate subsystems from the start. The two positions are less far apart than they sound: both say the hard part is the boundaries, and a monolith whose modules are enforced from day one (Section 5.3) answers most of Tilkov's concern.

Real cases point both ways but share a lesson. Segment described in 2018 how it had consolidated more than 140 small services back into a single service after operational overhead and diverging shared libraries slowed delivery; Amazon Prime Video moved one monitoring workload from distributed serverless components to a single process in 2023 (see

Section 1.3). Equally, organisations with hundreds of teams, like the large-scale cases of Section 6.3, gain real independence from services. The lesson is fit: the deployment style must match the organisation, the domain's maturity and the qualities required.

When a monolith is the right answer

A small team. With one to a few teams, the communication cost that microservices remove is low, while their operational cost is the same as for a large organisation. A monolith lets everyone understand and change the whole system.

An uncertain domain or an early product. When you are still discovering what the product is, boundaries will move. Moving a boundary between modules is a refactoring; moving it between services is a migration with data, APIs and deployments.

Strong consistency needs. A single database transaction across orders, payments and stock is simple and correct in a monolith. Across services it needs sagas, outboxes and compensation (Section 6.7).

Limited operations capacity. Each service needs its own pipeline, monitoring, alerting, scaling rules and on-call knowledge. A team without a platform for this will spend its time on plumbing.

Performance-sensitive interactions. In-process calls take nanoseconds to microseconds; network calls take milliseconds and can fail. Chatty interactions between parts of the domain are far cheaper inside one process.

In all these cases the recommended form is a modular monolith, not an unstructured one. It keeps the option to extract services open, as Section 5.3 explains.

IF YOU ARE NEW TO THIS

Step by step: moving a boundary that was drawn too early

This section's hardest idea is why a wrong boundary costs so much more between services than between modules, one of the main reasons behind MonolithFirst. Early on, a team is still learning the domain, so some boundaries will be in the wrong place. Follow one piece of data that has to move.

- A young food-delivery product has an Ordering part and a Dispatch part. At first the team keeps each order's ready time, when the kitchen expects the food to be ready, in Ordering.
- Months later they learn that Dispatch uses the ready time far more, to plan couriers. It belongs in Dispatch, as in the model of Section 5.7.
- In a modular monolith, moving it is a refactoring: move the code and the column to the other module, update the callers, let the tests find anything broken, and ship it in one release.
- If Ordering and Dispatch were separate services, the same move would be a migration: change two APIs, copy live data between two databases while orders keep arriving, release both services in a safe order, and keep the old path working until every caller has switched.
- Early in a product, moves like this happen again and again, because the team is still learning how the business works. Each one is cheap inside one deployment and expensive across a network, which is why Fowler advises starting with a monolith.
- **The lesson:** while you are still discovering where the boundaries belong, keep them inside one deployment, where moving one is cheap. Enforce them as modules from the start, so that any later split, made when there is evidence for it, follows boundaries that have already proved themselves.

Scaling a monolith

A common belief is that monoliths cannot scale. They scale further than most systems ever need, using the X axis of the scale cube described by Martin Abbott and Michael Fisher (running many identical copies) together with a few standard tactics. Chapter 9 treats each in depth (Section 9.1).

Horizontal replicas behind a load balancer. Run many identical copies of the application and spread requests across them. This only works if the application is stateless: sessions, uploaded files and caches must live in shared stores, so any replica can serve any request (Section 9.2). It also gives redundancy: one replica failing no longer stops the service.

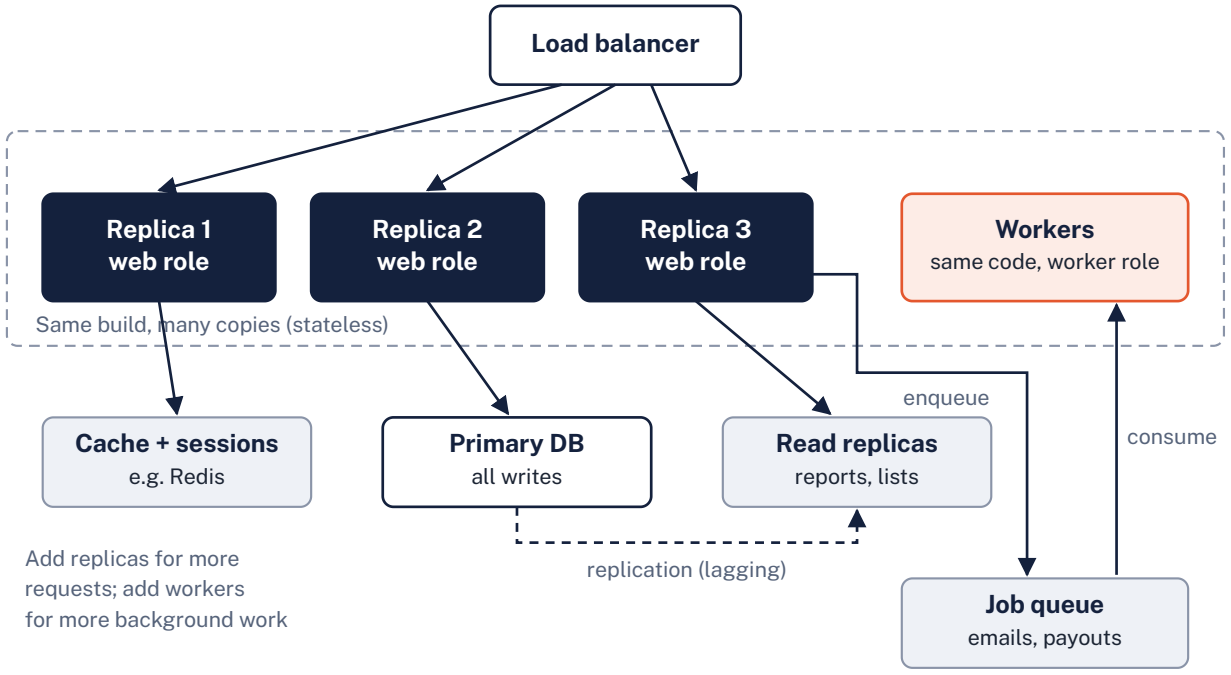
Read replicas. Most systems read far more than they write. Database read replicas copy the primary's changes and serve reads such as order history and reports, at the cost of replication lag: a replica may be a fraction of a second behind (Section 7.4).

Caching. Menus, product pages and configuration change rarely and are read constantly; caching them in memory or in a shared cache removes most database load (Section 7.6). Static assets go to a CDN.

Background workers. Slow or bursty work (emails, PDF invoices, imports, payouts) goes onto a queue and is processed by worker processes. Workers usually run the same codebase with a different start command, so the system stays one codebase but the web and worker roles scale separately (Section 9.3).

Bigger machines and partitioning. Vertical scaling of the database is often the simplest next step. For very large systems, data can be partitioned by tenant or region, with a full copy of the monolith per partition (the Z axis of the scale cube). Stack Overflow is a well-known example of how far a largely monolithic design can go: according to its engineers' own write-up, on one day in February 2016 its load balancers handled about 209 million HTTP requests, with the main Q&A application (one multi-tenant application for all its Q&A sites) running on nine primary web servers, with four SQL Server machines serving the whole network.

A monolith scaled out



One codebase, many copies: stateless web replicas behind a load balancer, shared cache and sessions, a primary database with read replicas, and workers fed by a queue.

WORKED EXAMPLE

Worked example: how far will one monolith go?

A new food-delivery start-up has 7 engineers and expects, by the end of year two, a peak of 1,200 HTTP requests per second at Friday dinner time. A load test of the modular monolith shows one replica (4 vCPU) handles 180 requests per second within the 300 ms p95 target. About 85% of requests are reads (menus, order tracking). Figures are illustrative.

Replicas for peak	1,200 / 180 = 6.7, so 7 replicas at peak
Headroom	Keep utilisation well below saturation (Section 2.4) and survive one of three availability zones failing: plan for about 50% headroom (with only two zones it would be about 100%), so 7 x 1.5 = about 11 replicas at peak, scaled down off-peak
Database writes	15% of 1,200 = 180 requests per second, each doing about 3 writes: about 540 writes per second, well within a single managed PostgreSQL primary on a mid-sized instance
Database reads	1,020 read requests per second; a cache for menus (which change rarely) absorbs most, and two read replicas take order history and reports, accepting a second or so of replication lag
Background work	Receipt emails, courier payouts and restaurant reports move to a worker pool fed by a queue, so a slow email provider cannot slow checkout
Conclusion	One codebase, deployed as one unit and run as about 11 web replicas plus workers, meets the two-year target. The architecture budget goes on module boundaries and statelessness, not on splitting.

The limit that arrives first is rarely CPU. More often it is team coordination as the engineering group grows, or a single module (such as dispatch) with very different scaling or release needs.

Every figure here should be replaced by measurements from your own load tests; capacity planning is covered in Section 9.7.

Signs that it is time to split

Splitting should be a response to evidence, not to fashion. The signs that one deployment unit has become the constraint are mostly organisational and operational:

Team contention. Several teams change the same code and wait for each other; merge conflicts, coordination meetings and broken builds caused by other teams rise as the organisation grows. If modules are already well bounded and contention persists, deployment independence may be what is missing.

Deployment queues. Releases are batched into trains because every change needs the whole system re-tested; a small fix waits days behind unrelated work; one team's failed change blocks everyone's release.

Wildly different scaling or availability needs. One module (say, real-time courier tracking) needs fifty times the capacity of the rest at peak, or must stay up when everything else is down. Scaling the whole monolith for one module's needs becomes expensive.

Different security or compliance scope. Card data or health records may justify a separate, tightly controlled unit to keep the rest of the system out of audit scope.

Different technology needs. A machine-learning model best served from Python inside a Java monolith, for example.

Some reasons are not good reasons on their own: "monoliths are legacy", a desire to use a new technology, or slow code (which is a design problem that splitting will not fix and may make worse by adding network calls).

How to split when the time comes

Split incrementally, one module at a time, starting with the module that has both the clearest boundary and the strongest reason to leave. Modularise first if the code is not yet modular: extracting a service from a tangle drags the tangle along. Then extract along bounded contexts (Section 5.7): put a network API in front of the module's existing API, move its data to its own database, replace in-process events with a message broker and replace any shared transactions with explicit sagas. Route traffic gradually to the new service and keep the old path until the new one has proved itself.

This is the strangler fig approach and its relatives, covered in Section 11.6. The destination is often not "all microservices" but a monolith with a few extracted services where the evidence demanded it; the service-based architecture of Section 6.2 is a common, pragmatic end state.

The Chapter 5 styles compared

Characteristic	Layered	Modular monolith	Pipes and filters	Microkernel	Hexagonal / clean
Main partitioning	Technical	Domain	Processing steps	Core plus plug-ins	Inside versus outside
Simplicity	High	Medium to high	High	Medium	Low to medium
Overall cost	Cheap	Cheap	Cheap	Cheap	Cheap to medium
Modifiability for business change	Low to medium	High	High within the flow	High in planned directions	High for infrastructure
Testability	Medium	Medium to high	High	Medium to high	High
Deployability	Low	Low to medium	Low to medium	Medium	Set by deployment style
Scalability	Low	Low to medium	Low to medium	Low	Set by deployment style
Fault tolerance	Low	Low	Low to medium	Low to medium	Set by deployment style
Typical fit	Small to medium apps, one team	Growing products, several teams	ETL, media, logs, compilers	Products with many variants	Logic that must outlive frameworks

The ratings summarise the judgements made in each section and assume typical implementations. Domain-driven design does not appear as a column because it is a method for finding boundaries, which any of these styles can use. Real systems combine styles: a modular monolith whose modules are hexagonal inside, with a pipes-and-filters import job and a microkernel for tax rules, is entirely normal.

DECISION GUIDE

Decision guide: choosing among the Chapter 5 styles

Use it for a new system or a major restructuring that will run as one deployment unit. Work down the questions; several styles can apply at different levels.

Q1 **Is the system mainly a flow of data through a series of transformations (import, clean, enrich, export), rather than users interacting with shared state?**

- **Yes** Use pipes and filters for that flow (Section 5.4). Keep each filter stateless and the data format between filters explicit.
- **No** Go to the next question.

Q2 **Does the product have a stable core workflow but many variants of particular steps (per country, customer, provider or file format), or do third parties need to extend it?**

- **Yes** Use a microkernel (Section 5.5) for those variation points, ideally after building two or three variants so the contract is grounded in real cases.
- **No** Go to the next question.

Q3 **Is it small (one team, a few dozen screens), with simple rules, and unlikely to grow much?**

- **Yes** Use a layered architecture (Section 5.2) with closed layers enforced by an architecture test. Do not add more structure than the problem needs.
- **No** Go to the next question.

Q4 **Will several teams work on it, or does the domain span several areas with their own language (menus, orders, dispatch)?**

- **Yes** Use a modular monolith (Section 5.3) with modules drawn along bounded contexts (Section 5.7), each owning its data and API.
- **No** A layered or modular structure both work; prefer modules by domain if growth is likely, because restructuring later is costly.

Q5 **Inside each module or application, is the business logic substantial, or does it depend on external services likely to change?**

- **Yes** Structure the inside with ports and adapters (Section 5.6): domain at the centre, adapters for the web, database and providers.
- **No** Keep the framework's default structure inside the module; add ports only at boundaries that are likely to change.

Checklist: a monolith that stays healthy and splittable

Most of these cost little when done early and a great deal when retrofitted.

- [ ] **Modules follow business domains**
  Top-level packages are named after bounded contexts (orders, dispatch, payments), not technical layers.
- [ ] **Each module has a small public API**
  Everything else is internal; an architecture test fails the build on a forbidden import.
- [ ] **Each table has one owning module**
  Separate schemas or naming conventions make ownership visible; no module reads another's tables.
- [ ] **The application is stateless**
  Sessions, uploads and caches live in shared stores, so any replica can serve any request and replicas can be added at will.
- [ ] **Slow work runs in background workers**
  Emails, reports, payouts and imports go through a queue, so web replicas stay fast and the two can be scaled separately.
- [ ] **Build and tests stay fast**
  Track build and test times; keep the main pipeline under about 10 to 15 minutes by testing modules separately and in parallel.
- [ ] **Deployment is automated and frequent**
  Small, frequent releases with automated rollback; feature flags decouple deployment from release.
- [ ] **Observability by module**
  Logs, metrics and traces are tagged with the module, so you can see which module is slow, failing or expensive.
- [ ] **Signs of strain are measured**
  Lead time, deployment queue length, merge conflicts per team and resource use per module are reviewed regularly, so the decision to split is based on evidence.

WATCH OUT

Common mistakes

- **Treating "monolith" as a structure.** It is a deployment decision. The internal quality is a separate decision, and it is the one that decides whether you can change or split the system later.
- **Splitting to fix code quality.** Services drawn around a tangled codebase become a distributed monolith: all the coupling, plus the network.
- **Scaling the wrong thing.** Adding replicas while sessions are stored in memory, or while the database is the bottleneck, buys nothing.
- **Never measuring the strain.** Without lead-time, deployment and contention data, the decision to split becomes a matter of opinion and fashion.

CHECK YOUR UNDERSTANDING

▼ **Why is a wrong boundary between services more expensive than a wrong boundary between modules?**

Moving a module boundary is a refactoring inside one codebase and one database transaction; moving a service boundary changes network APIs, data stores, deployments and often team ownership, and must be done without downtime.

▼ **What must be true of a monolith before you can scale it by adding replicas?**

It must be stateless: sessions, files and caches must live in shared stores so any replica can serve any request.

▼ **Name two signs that a monolith has become the constraint, and one reason for splitting that is not good enough on its own.**

Signs: persistent team contention and deployment queues, or one module with very different scaling or compliance needs. Not enough on its own: fashion, wanting a new technology, or slow code.

▼ **How do the MonolithFirst and "Don't start with a monolith" positions overlap?**

Both say the hard problem is finding and keeping good boundaries; a monolith with enforced module boundaries addresses Tilkov's concern while keeping Fowler's lower cost.

▼ **Try it: Six engineers expect a peak of 900 requests per second, 80% reads. One replica of their modular monolith handles 150 within target. How many replicas are needed with 30% headroom? They want microservices: what is your advice?**

900 / 150 = 6 replicas at the limit. Running at 70% means 900 / (150 x 0.7) = 8.6, so 9 replicas. That is easily run behind a load balancer, and with 80% reads, caching and read replicas will protect the database. There is no evidence of team contention, deployment queues or very different scaling needs, and six engineers cannot carry the operational cost. Stay with the monolith, and split later along bounded contexts if evidence appears.

SOURCES

- Martin Fowler: MonolithFirst (2015): https://martinfowler.com/bliki/MonolithFirst.html
- Martin Fowler: Microservice Premium (2015): https://martinfowler.com/bliki/MicroservicePremium.html
- David Heinemeier Hansson: The Majestic Monolith (2016): https://signalvnoise.com/svn3/the-majestic-monolith/
- Stefan Tilkov: Don't start with a monolith (2015): https://martinfowler.com/articles/dont-start-monolith.html
- Nick Craver: Stack Overflow, the architecture (2016 edition): https://nickcraver.com/blog/2016/02/17/stack-overflow-the-architecture-2016-edition/

Further reading

- *Fundamentals of Software Architecture (2nd edition)*, Mark Richards and Neal Ford (2025), O'Reilly Media. A chapter per style, including layered, modular monolith, pipeline and microkernel, each with a characteristics rating; the model for this chapter's comparisons.
- *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal (1996), John Wiley and Sons. The classic catalogue of architectural patterns, including Layers, Pipes and Filters and Microkernel, each with context, forces and consequences.
- *An Introduction to Software Architecture*, David Garlan and Mary Shaw (1994), Carnegie Mellon University, Software Engineering Institute (CMU/SEI-94-TR-021). The short paper that defined architectural styles as components, connectors and constraints, with pipes and filters and layered systems among its examples.
  https://www.sei.cmu.edu/library/an-introduction-to-software-architecture/
- *Software Architecture: The Hard Parts*, Neal Ford, Mark Richards, Pramod Sadalage and Zhamak Dehghani (2021), O'Reilly Media. Develops the architecture quantum and the analysis of coupling and data ownership that decides where a system can be split.
- *Monolith to Microservices*, Sam Newman (2019), O'Reilly Media. A balanced guide to when, whether and how to decompose a monolith, with a strong case for the modular monolith.
- *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Eric Evans (2003), Addison-Wesley. The origin of bounded contexts, context maps and the ubiquitous language used to find module and service boundaries.
- *Hexagonal Architecture*, Alistair Cockburn (2005), alistair.cockburn.us. The original description of ports and adapters, short and still the clearest statement of the idea.
  https://alistair.cockburn.us/hexagonal-architecture/
- *Deconstructing the Monolith: Designing Software that Maximizes Developer Productivity*, Kirsten Westeinde (2019), Shopify Engineering. A first-hand account of turning a very large Rails application into a modular monolith instead of microservices.
  https://shopify.engineering/deconstructing-monolith-designing-software-maximizes-developer-productivity

Glossary

Adapter
Code that translates between a port and a specific technology, such as an HTTP controller, a SQL repository or a payment provider's SDK. One port can have many adapters.

Aggregate
A cluster of domain objects changed as one unit through a single root entity, which enforces the rules that must always hold for the cluster.

Anti-pattern
A commonly used solution that looks reasonable but reliably causes more problems than it solves.

Anticorruption layer
A translation layer that a downstream context builds so that another context's or system's model cannot leak into its own.

Architectural pattern
A documented, reusable solution to a recurring structural problem in a stated context, naming the parts, their responsibilities and how they relate. Often used as a synonym for style.

Architectural style
A named family of architectures defined by the kinds of element and connector it uses and the constraints on how they may be combined, such as layered or pipes and filters.

Architecture characteristic
Richards and Ford's term for a quality attribute the architecture must support, such as scalability, deployability or fault tolerance.

Architecture quantum
An independently deployable part of a system with high functional cohesion, including everything it needs to run, such as its database; the scope within which one set of architecture characteristics applies.

Architecture sinkhole
An anti-pattern in layered systems where requests pass through layers that only forward them, adding code and delay without adding behaviour.

Architecture test
An automated test that fails the build when code breaks a structural rule, such as a forbidden dependency between layers or modules; tools include ArchUnit and import-linter.

Back pressure
A way for a slow consumer to make a fast producer slow down, instead of letting buffers and queues grow without limit.

Background worker
A process that takes slow or bursty jobs, such as emails or reports, from a queue and processes them outside the request that created them.

Batch processing
Processing a bounded set of data in one run, such as all of yesterday's orders, and finishing.

Big ball of mud
A system with no recognisable structure, in which almost everything depends on everything else; the name comes from Brian Foote and Joseph Yoder (1997).

Bounded context
An explicit boundary within which one domain model and its ubiquitous language apply consistently; outside it, the same words may mean something else.

Closed layer
A layer that every request must pass through: the layer above may not skip it to call a lower layer directly.

Composition root
The single place, usually the program's start-up code, where concrete adapters are created and plugged into the application.

Context map
A picture of a system's bounded contexts and the relationships between them, such as customer-supplier or anticorruption layer, covering both integration and team relationships.

Core domain
The subdomain that gives the business its competitive advantage and therefore deserves the best people, the richest modelling and code the company owns.

Core system
In a microkernel architecture, the minimal part of the application that every installation needs: the main workflow, shared services and the points where plug-ins attach.

Dead-letter queue
A separate place where messages that repeatedly fail processing are parked for inspection, so they do not block the rest of the flow.

Dependency rule
Robert C. Martin's rule for clean architecture: source code dependencies point only inwards, so inner rings never name anything declared in outer rings.

Deployment unit
The thing that is built, released and started as one piece: one executable, one container image or one web archive, for example.

Design pattern
A reusable solution to a recurring problem at the scale of a few classes or objects, such as Strategy or Observer; smaller in scope than an architectural pattern.

Distributed architecture
An architecture made of several separately deployed units that cooperate by sending messages over a network.

Distributed monolith
A system deployed as separate services that are so tightly coupled they must be changed and released together, giving the costs of distribution without its benefits.

Domain event
A record of something meaningful that happened in the business domain, named in the past tense, such as OrderPlaced or CourierAssigned.

Domain partitioning
Dividing a system's top-level structure by business capability or subdomain, such as ordering, payments and delivery.

Domain-driven design (DDD)
An approach introduced by Eric Evans in 2003 that builds software around a model of the business domain, developed with domain experts, and divides large domains into explicitly bounded models.

Driven (secondary) adapter
An adapter that implements an interface the application calls when it needs something from outside, such as storing data, charging a card or sending a message.

Driving (primary) adapter
An adapter that receives requests from outside, such as a web controller, a test harness or a scheduled job, and calls the application through a driving port.

Entity
A domain object defined by an identity that persists over time while its attributes change, such as an order or a customer.

Entry point (Python packaging)
Metadata in an installed Python package that advertises an object, such as a plug-in class, under a named group so that other code can discover it at run time.

ETL
Extract, transform, load: a pipeline that takes data from source systems, cleans and reshapes it, and loads it into a target store such as a data warehouse.

Event storming
A workshop format created by Alberto Brandolini in which domain experts and developers place domain events on a timeline to explore a business process and find boundaries.

Extension point
A named slot that a host application or plug-in declares so that other plug-ins can contribute behaviour to it, as in the Eclipse platform.

Filter
A self-contained processing step that reads input, then produces, transforms, tests or consumes it, without knowing which steps come before or after it.

Generic subdomain
A subdomain whose problems many businesses share and for which good solutions exist, such as payments or identity; usually bought or built thinly.

Idempotent
Describes an operation that has the same effect whether it is applied once or several times, so it can be retried safely.

In-process event
A message published inside one running application so that other modules can react to something that happened, without the publisher knowing who listens.

Layer
A group of components with the same technical responsibility, which offers services to the layer above and uses the layer below.

Layered architecture
A style that arranges components in horizontal layers, each with one technical role, where each layer uses only the layers beneath it.

Layers of isolation
The property that a change inside one layer does not ripple into other layers while its interface stays the same; it depends on layers being closed.

Microkernel architecture
An architectural style, also called a plug-in architecture, in which a small core system provides the general workflow and independent plug-ins add or change behaviour through published contracts.

Microservice premium
Martin Fowler's term (2015) for the extra cost and risk that microservices add, which only pays off once a system and organisation are complex enough.

Modular monolith
A single deployable application whose code is divided into modules along business boundaries, each with an explicit public interface and its own data.

Module API
The small, deliberately public part of a module that other modules may use; everything else is internal and free to change.

MonolithFirst
Martin Fowler's 2015 advice to start a new system as a well-modularised monolith and extract services later, when boundaries are understood and needed.

Monolithic architecture
An architecture in which the whole application is built and deployed as a single unit, whatever its internal structure.

Open host service
A context-map pattern in which an upstream context offers one well-defined, documented protocol or API for all its consumers.

Open layer
A layer that the layer above is allowed to bypass, typically a layer of shared services that only some requests need.

Pipe
A one-way channel that carries data from one filter to the next, usually with a buffer between them.

Pipes and filters
A style in which data flows through a chain of independent processing steps (filters) connected by one-way channels (pipes).

Plug-in
A separately built module that adds behaviour to a host application through a contract the host publishes, without the host's code being changed.

Plug-in contract
The published interface between a core system and its plug-ins: the operations, data types, error behaviour and version that both sides agree to.

Plug-in registry
The part of a core system that records which plug-ins are available, their names, versions and contracts, so the core can find them without naming them in code.

Ports and adapters (hexagonal architecture)
Alistair Cockburn's style in which an application talks to the outside world only through ports (interfaces it owns), and adapters translate between each port and a particular technology.

Published language
A documented, shared format for exchanging information between contexts, such as a versioned event schema, often offered through an open host service.

Read replica
A copy of a database that receives the primary's changes and serves read queries, reducing load on the primary at the cost of slightly stale data.

Reference architecture
A template architecture for a domain or platform that maps typical functions onto a standard set of parts, which projects adapt instead of designing from scratch.

Repository
An object that loads and saves whole aggregates and looks to the domain like a collection, hiding how and where they are stored.

Scale cube
Abbott and Fisher's model of three ways to scale: X axis (identical copies behind a load balancer), Y axis (split by function) and Z axis (split by data, such as by customer).

Semantic versioning
A version-numbering convention, MAJOR.MINOR.PATCH, in which the major number changes for breaking changes, the minor for backward-compatible additions and the patch for fixes.

Shared kernel
A small part of a model, and its code, that two contexts explicitly agree to share and may change only with both teams' consent.

Stateless (application)
An application that keeps no per-user data in its own memory between requests, so any copy can handle any request and copies can be added or removed freely.

Stream processing
Processing an unbounded flow of records continuously, each one shortly after it arrives.

Subdomain
A distinct area of a business's activity, such as menus, ordering, payments or dispatch. Subdomains belong to the problem, whether or not software exists.

Supporting subdomain
A subdomain that is specific to the business and necessary for it, but is not where it competes; it deserves sound but simpler design.

Technical partitioning
Dividing a system's top-level structure by technical role, such as user interface, business logic and data access.

Tier
A physical boundary at run time, such as a separate process or server. A layer is a logical boundary in the code; several layers can run in one tier.

Ubiquitous language
A vocabulary shared by developers and domain experts and used consistently in conversation, documents, tests and code within one bounded context.

Value object
A domain object defined only by its values, with no identity of its own, and usually immutable, such as an amount of money or an address.