

Software Architecture Patterns: Interview Guide

Sep 27, 2026 · @CD Smith

How to use this guide

No pattern is "best" — interviewers want to hear you match a pattern to constraints (team size, scale, domain complexity, consistency needs) and name its trade-offs. Each section below covers what the pattern is, when to use it, pros and cons, and the points worth raising in an interview.

A reliable answer shape: **start simple, name the forces that would push you to something more complex, and state the cost of that move.** Patterns also combine — a modular monolith can use Clean Architecture inside, CQRS for one busy module, and events between modules.

1. Layered (N-tier) architecture

Code is organized into horizontal layers — typically Presentation → Business/Application → Data Access → Database — and each layer only calls the one directly below it. It is the default shape of most enterprise apps and the one most developers learn first.

Use it for CRUD-heavy line-of-business apps, small-to-medium teams, and projects where speed of delivery and familiarity matter more than flexibility. It is a sensible starting point when you don't yet know where complexity will land.

- **Pros:** easy to understand and onboard; clear separation of concerns; each layer can be tested and replaced in isolation; well supported by frameworks.
- **Cons:** business logic tends to leak into controllers or the data layer; changes often ripple through every layer; the business layer usually depends on the database, making the domain hard to test without it; can decay into a "big ball of mud".
- **Interview highlights:** know *closed* layers (must go through each layer) vs *open* layers (may skip); name the **architecture sinkhole anti-pattern** — requests that just pass through layers with no logic; contrast *layers* (logical) with *tiers* (physical deployment); explain why Clean/Hexagonal inverts the dependency on the database.

2. Monolith and modular monolith

A monolith is one deployable unit containing all features, usually sharing one database. A **modular monolith** keeps the single deployment but enforces strong internal boundaries: each module (Orders, Billing, Catalog) owns its data and exposes a public API, and other modules may not reach into its internals.

Use a monolith for new products, small teams, and domains you don't fully understand yet. Choose a modular monolith when the codebase is growing and you want microservice-style boundaries without the operational cost — it is increasingly the recommended default, and it makes a later split into services far easier.

- **Pros:** simple to build, test, debug and deploy; in-process calls are fast and reliable; ACID transactions across features; one codebase to reason about.
- **Cons:** the whole app scales and deploys as one unit; a bug or memory leak can take everything down; builds and tests slow as it grows; without discipline, modules become tangled.
- **Interview highlights:** "monolith" is not a dirty word — say *monolith first*; distinguish a modular monolith from a **distributed monolith** (services that must deploy together — the worst of both worlds); mention enforcing boundaries with separate projects/assemblies, internal visibility, and architecture tests; each module owning its schema is what enables a future extraction.

3. Microservices

The system is split into small, independently deployable services, each owning one business capability and its own data store. Services talk over the network — synchronously (HTTP/gRPC) or asynchronously (messages/events) — and each can use its own tech stack.

Use it when many teams need to ship independently, when parts of the system have very different scaling or availability needs, and when the domain boundaries are well understood. It solves an organizational scaling problem as much as a technical one; for a small team it usually adds more cost than value.

- **Pros:** independent deployment and scaling; fault isolation; team autonomy and clear ownership; freedom to pick the right tech per service.
- **Cons:** distributed-system complexity (latency, partial failure, retries, versioning); no cross-service ACID transactions; data consistency becomes eventual; heavy ops burden — CI/CD, observability, service discovery, containers/orchestration.

- **Interview highlights:** **database per service**; boundaries follow **DDD bounded contexts**, not technical layers; **Conway's Law**; distributed transactions via the **Saga pattern** (choreography vs orchestration) and reliable publishing via the **Outbox pattern**; resilience patterns — circuit breaker, retry with backoff, timeouts, bulkhead (e.g. Polly in .NET); **API Gateway** and **Backend-for-Frontend**; the **Strangler Fig** pattern for migrating off a monolith; distributed tracing (OpenTelemetry) and correlation IDs.

4. Service-Oriented Architecture (SOA)

SOA predates microservices: coarse-grained, reusable enterprise services (Customer, Billing, Inventory) are exposed through standard contracts — historically SOAP/WSDL — and connected by an **Enterprise Service Bus (ESB)** that handles routing, transformation, and orchestration.

Use it in large enterprises integrating many heterogeneous systems (ERP, CRM, legacy mainframes) where reuse and standardized contracts across departments matter more than deployment speed. You will meet it most often in integration work with older systems.

- **Pros:** strong reuse of shared services; standardized, formal contracts; good for integrating legacy and vendor systems; centralized governance and security.
- **Cons:** the ESB becomes a bottleneck and single point of failure; "smart pipes" hide business logic in middleware; heavyweight protocols and governance slow change; services often share databases.
- **Interview highlights:** the classic comparison — SOA uses **smart pipes, dumb endpoints** (logic in the ESB) while microservices use **smart endpoints, dumb pipes**; SOA optimizes for **reuse** across the enterprise, microservices for **independent change**; SOA services are coarser and often share data; modern integration platforms and message brokers have largely replaced the traditional ESB.

5. Event-driven architecture

Components communicate by producing and reacting to **events** — facts that something happened (`OrderPlaced` , `PaymentFailed`) — through a broker such as Kafka, RabbitMQ, Azure Service Bus, or AWS SNS/SQS. Producers don't know who consumes their events, so components are loosely coupled in time and in knowledge.

Use it when one action must trigger several independent reactions, for high-throughput or real-time processing (IoT, clickstreams, notifications), for integrating systems asynchronously, and to decouple microservices. Avoid it where a user needs an immediate, strongly consistent answer.

- **Pros:** loose coupling — add consumers without touching producers; absorbs load spikes via queues; scales well; naturally resilient if a consumer is down temporarily.
- **Cons:** eventual consistency; flows are hard to follow and debug; duplicate, lost, and out-of-order messages must be handled; event schemas become contracts that are hard to change.
- **Interview highlights:** **events vs commands** (a fact to many vs an instruction to one); **pub/sub vs point-to-point queues**; **choreography vs orchestration**; delivery guarantees — at-most-once, at-least-once, and why exactly-once is really **idempotent consumers** plus deduplication; **dead-letter queues**; the **Outbox pattern** to avoid dual writes; event notification vs event-carried state transfer; partitioning for ordering (e.g. Kafka keys).

6. Hexagonal / Clean / Onion architecture

Three closely related patterns with one core idea: **the domain sits at the center and depends on nothing; all dependencies point inward**. Hexagonal (Ports and Adapters, Alistair Cockburn) defines *ports* — interfaces the core owns — and *adapters* that implement them for the database, UI, message bus, or external APIs. Onion (Jeffrey Palermo) and Clean Architecture (Robert C. Martin) express the same rule as concentric rings: Entities → Use Cases → Interface Adapters → Frameworks/Drivers.

Use it for applications with meaningful business rules that should outlive frameworks and databases, where testability matters, or where you expect to swap infrastructure (a new payment provider, a different datastore). It is overkill for thin CRUD apps.

- **Pros:** business logic is testable without a database or web server; infrastructure is swappable; framework upgrades don't touch the core; clear, enforceable boundaries.
- **Cons:** more projects, interfaces, and mapping code; indirection can feel like ceremony; easy to over-engineer small apps; teams often argue over which ring code belongs in.
- **Interview highlights:** it is the **Dependency Inversion Principle** applied at architecture scale; the **Dependency Rule** (source dependencies point inward only); **primary/driving adapters** (API controllers, UI) vs **secondary/driven adapters**

(repositories, API clients); compare with Layered — here the data layer depends on the domain, not the reverse; the typical .NET solution split is Domain / Application / Infrastructure / Web API, often paired with MediatR-style handlers for use cases.

7. CQRS (Command Query Responsibility Segregation)

CQRS separates the **write model** (commands that change state and enforce business rules) from the **read model** (queries shaped for display). At its simplest, that is just separate command and query handlers over one database; at its fullest, writes go to a normalized store and events update denormalized read stores tuned for each screen.

Use it when reads and writes have very different shapes or loads (reads often outnumber writes 10:1 or more), when the write side has complex domain rules, or when you need multiple read views of the same data. Apply it to specific bounded contexts, not a whole system.

- **Pros:** read and write sides scale and optimize independently; simpler, focused models; fast queries from precomputed views; fits naturally with DDD and event sourcing.
- **Cons:** more moving parts; with separate stores, reads are **eventually consistent** (a user may not see their own change immediately); synchronization code to build and maintain; overkill for simple CRUD.
- **Interview highlights:** grew out of Bertrand Meyer's **Command-Query Separation** (a method either changes state or returns data, not both); CQRS does **not** require event sourcing or two databases — say so; handle stale reads with techniques like returning the new state from the command or read-your-own-writes; in .NET, the MediatR library is a common (lightweight) way to model commands and queries.

8. Event sourcing

Instead of storing an entity's current state, you store the full, append-only sequence of **events** that happened to it (`AccountOpened` , `MoneyDeposited` , `MoneyWithdrawn`).

Current state is rebuilt by replaying those events; read-optimized **projections** are built from the same stream. A bank ledger is the classic analogy.

Use it when a complete audit trail is a requirement (finance, healthcare, compliance), when you need to answer "what was the state at time X?", or when the history of changes is itself valuable business data. Avoid it for simple domains — it is one of the hardest patterns to retrofit or remove.

- **Pros:** perfect audit log and history; temporal queries and debugging by replay; new projections can be built from past events; pairs naturally with CQRS and event-driven integration.
- **Cons:** steep learning curve; **event schema versioning** is hard because events are immutable forever; querying current state requires projections; replaying long streams is slow without snapshots; handling data-deletion requests (e.g. GDPR) is tricky.
- **Interview highlights:** the event store is the **source of truth**; **snapshots** to speed up rebuilding; **upcasting** or versioned events to evolve schemas; **optimistic concurrency** via expected stream version; projections are disposable and rebuildable; distinguish domain events inside the service from integration events published outward; tools include EventStoreDB and Marten (on PostgreSQL, popular in .NET).

9. Serverless

Application logic runs as short-lived functions (AWS Lambda, Azure Functions, Google Cloud Functions) triggered by events — HTTP requests, queue messages, file uploads, timers — while the cloud provider manages servers, scaling, and patching. It usually leans on managed services (queues, databases, auth) for everything else, sometimes called Backend-as-a-Service.

Use it for spiky or unpredictable workloads, event-triggered glue and integration jobs, scheduled tasks, lightweight APIs, and prototypes where you want to pay only for actual use. It is a poor fit for long-running, latency-critical, or steady high-volume workloads.

- **Pros:** no server management; automatic scaling, including to zero; pay-per-execution; fast to ship small pieces.
- **Cons:** **cold starts** add latency; execution time and memory limits; **vendor lock-in**; harder local testing and debugging; costs can exceed containers at steady high load; distributed tracing across many functions is harder.
- **Interview highlights:** functions should be **stateless** and **idempotent** (triggers may retry); mitigate cold starts with provisioned/pre-warmed instances, smaller packages, or ahead-of-time compilation (e.g. .NET Native AOT); use durable orchestration (Azure Durable Functions, AWS Step Functions) for multi-step workflows; know the cost trade-off vs containers; serverless is a *deployment model* that often implements an *event-driven* architecture.

10. Microkernel (plug-in) architecture

A minimal **core system** provides the essential runtime and a plug-in contract; features are added as independent **plug-ins** that the core discovers and loads through a registry. VS Code, Eclipse, web browsers with extensions, and many rules engines and tax/insurance products follow this shape.

Use it for products that must be customized per customer or region, tools with third-party extension ecosystems, and systems whose rules change often while the core stays stable.

- **Pros:** highly extensible; features can be added, removed, or versioned independently; the core stays small and stable; plug-ins can be developed and tested in isolation.
- **Cons:** designing a stable plug-in contract up front is hard, and changing it breaks every plug-in; plug-in versioning and compatibility are ongoing work; a misbehaving plug-in can affect the host; usually not independently scalable.
- **Interview highlights:** the **plug-in contract/API** is the critical design decision; plug-ins shouldn't depend on each other; discuss isolation (separate process vs in-process loading); relate it to the **Open/Closed Principle** — open for extension, closed for modification; mention the **Strategy** pattern as its small-scale cousin.

11. MVC / MVVM (presentation patterns)

These structure the *UI layer* rather than the whole system. **MVC** splits an app into the Model (data and rules), View (rendering), and Controller (handles input, updates the model, picks a view) — the basis of ASP.NET Core MVC, Rails, and Spring MVC. **MVVM** replaces the controller with a **ViewModel** that exposes state and commands the View binds to declaratively — used in WPF, .NET MAUI, and SwiftUI. MVP (Model-View-Presenter) sits between them, with a passive view driven by a presenter.

Use MVC for server-rendered web apps and request/response flows. Use MVVM for rich client UIs (desktop, mobile) with data binding, where you want UI logic testable without the UI.

- **Pros:** separates UI from logic; ViewModels and controllers are unit-testable; designers and developers can work in parallel; well supported by frameworks.
- **Cons:** MVC controllers tend to bloat ("Massive View Controller" is a well-known iOS joke); MVVM binding can be hard to debug and overkill for simple screens; neither organizes the domain or backend on its own.

- **Interview highlights:** be clear these are **presentation patterns**, not system architectures — they often sit inside the outer layer of Layered or Clean Architecture; in MVVM the ViewModel **never references the View** (binding and observable state, e.g. `INotifyPropertyChanged` or `IObservable`); know the flow direction of each; newer unidirectional patterns (Redux, The Composable Architecture) are a related topic.

12. Pipe-and-filter

Data flows through a chain of independent processing steps (**filters**) connected by **pipes**; each filter takes input, transforms it, and passes output on without knowing about the others. Unix shell pipelines, ETL jobs, compilers, media processing, and ASP.NET Core's middleware pipeline all follow this idea.

Use it for data transformation and integration pipelines, batch or stream processing, and any workflow that decomposes into sequential, reusable steps.

- **Pros:** filters are simple, reusable, and testable alone; steps can be reordered or added easily; stages can run in parallel or scale independently; easy to reason about.
- **Cons:** poor fit for interactive or highly stateful workflows; overhead from serializing data between stages; error handling and partial failures across the chain need care; a slow stage bottlenecks the whole pipeline.
- **Interview highlights:** filters should be **stateless** and share a common data format; discuss **backpressure** in streaming pipelines; handle bad records with a dead-letter path rather than failing the run; name real examples (Unix pipes, middleware, Apache Beam, Azure Data Factory); related to the **Chain of Responsibility** design pattern.

Quick comparison

Pattern	Best for	Main trade-off	One-line interview hook
Layered	CRUD business apps, small teams	Changes ripple across layers	Watch for the sinkhole anti-pattern
Modular monolith	New products, growing codebases	One deploy unit, needs discipline	Monolith first; avoid a distributed monolith
Microservices	Many teams, uneven scaling needs	Distributed-system complexity	Database per service, Sagas, Outbox

Pattern	Best for	Main trade-off	One-line interview hook
SOA	Enterprise and legacy integration	ESB bottleneck, heavy governance	Smart pipes vs smart endpoints
Event-driven	Async reactions, high throughput	Eventual consistency, hard to trace	Idempotent consumers, DLQs
Hexagonal / Clean	Rich domains, long-lived apps	Extra abstraction and mapping	Dependencies point inward
CQRS	Read/write asymmetry	Stale reads, more moving parts	Doesn't require event sourcing
Event sourcing	Audit trails, temporal queries	Schema evolution, learning curve	Events are the source of truth
Serverless	Spiky, event-triggered work	Cold starts, vendor lock-in	Stateless, idempotent functions
Microkernel	Extensible, customizable products	Plug-in contract is hard to change	Open/Closed at system scale
MVC / MVVM	UI structure	Bloated controllers / binding complexity	Presentation pattern, not system architecture
Pipe-and-filter	Data transformation pipelines	Poor for interactive flows	Stateless filters, backpressure