

Architecture Patterns: Interview Questions and Model Answers

Sep 27, 2026 · @CD Smith

How to answer architecture questions

Strong answers are short, structured, and end on a trade-off. The model answers below are written to be spoken in 30–90 seconds; adapt them with examples from your own work — a real story beats a textbook answer every time.

A structure that works for almost any architecture question:

1. **Define it** in one sentence.
2. **Say when you'd use it** — tie it to a constraint (team size, scale, consistency, domain complexity).
3. **Name the main trade-off** — what it costs you.
4. **Give a concrete example**, ideally from your own experience.
5. **Show judgment** — when you would *not* use it, or what you'd start with instead.

If you don't know something, say what you do know and how you'd reason about it. Interviewers reward clear thinking over memorized terms.

1. Layered (N-tier)

Q: What's the difference between a layer and a tier?

A layer is a logical separation of code by responsibility — presentation, business logic, data access. A tier is a physical deployment boundary — a separate machine or process, like a web server, an app server, and a database server. You can have three layers running in one tier, which is how most small web apps are deployed. The distinction matters because crossing a tier adds network latency and failure modes, while crossing a layer is just a method call.

Q: What is the architecture sinkhole anti-pattern, and how do you deal with it?

It's when requests pass straight through layers without any logic being applied — a controller calls a service that just calls a repository. A little of that is normal, but if most requests are pass-throughs, the layers are adding ceremony without value. A common rule of thumb is that if more than about 20% of requests are sinkholes, you should reconsider. Options are making some layers *open* so they can be skipped for simple reads, or moving to a vertical-slice organization where each feature owns its own path.

Q: What are closed vs open layers?

A closed layer must be passed through — the presentation layer can only talk to the business layer, never directly to data access. That enforces isolation, so changing the data layer doesn't affect the UI. An open layer can be bypassed, which is useful for shared services or simple reads but weakens isolation. I'd default to closed and open a layer deliberately, with a documented reason.

Q: What's the main weakness of a traditional layered architecture?

The business layer usually depends on the data layer, so your core logic is coupled to the database and hard to test without it. Changes also tend to ripple through every layer — adding a field touches the UI, service, repository, and schema. That's the problem Clean and Hexagonal Architecture solve by inverting the dependency so the data layer depends on the domain.

2. Monolith and modular monolith

Q: Would you start a new product as a monolith or with microservices?

Almost always a monolith — specifically a modular one. Early on, the domain boundaries aren't clear yet, and drawing service boundaries in the wrong place is very expensive to undo across a network. A monolith lets a small team move fast, debug easily, and use normal database transactions. I'd enforce module boundaries from day one so that if one area later needs independent scaling or a separate team, extracting it is a refactor rather than a rewrite.

Q: What is a modular monolith, and how do you keep the modules from becoming tangled?

It's a single deployable application divided into modules around business capabilities — Orders, Billing, Catalog — where each module owns its data and exposes a small public API. The boundaries have to be enforced, not just agreed on. In .NET I'd put each module in its own project, make internals `internal`, give each module its own schema, and add architecture tests (for example with NetArchTest) that fail the build if one module references another's internals. Modules talk through public interfaces or in-process events.

Q: What is a distributed monolith, and why is it bad?

It's a system split into services that still have to be changed and deployed together — because they share a database, call each other synchronously in long chains, or share domain models. You pay all the costs of microservices — network latency, partial failures, complex ops — and get none of the benefits like independent deployment. The usual cause is splitting by technical layer instead of business capability. The telltale sign is needing a coordinated release across several services for one feature.

Q: When does a monolith become a problem?

When the pain is organizational or operational rather than just size: many teams blocking each other on releases, one area needing very different scaling, long build and test times, or one faulty feature taking down the whole app. Size alone isn't the trigger — plenty of large, successful systems are monoliths. I'd look at deployment frequency and team friction before deciding to split.

3. Microservices

Q: How do you decide where to draw service boundaries?

I draw them around business capabilities, using Domain-Driven Design bounded contexts — places where a term like "Order" has one consistent meaning and one team owns it. A good boundary means most changes stay inside one service and services rarely need to chat synchronously. Bad signs are two services that always change together or constantly call each other for data. Conway's Law also applies: boundaries should line up with team ownership, because your architecture will end up mirroring your org chart anyway.

Q: How do you handle a transaction that spans multiple services — say, placing an order that reserves stock and charges a card?

You can't use a single ACID transaction across services, so I'd use the Saga pattern: a sequence of local transactions, each with a compensating action if a later step fails — if payment fails, release the reserved stock. Sagas can be choreographed, where each service reacts to events, or orchestrated, where a central coordinator tells each service what to do. I prefer orchestration once there are more than three or four steps, because the flow is visible in one place. To publish events reliably, I'd use the Outbox pattern so the database write and the event can't get out of sync.

Q: What is the Outbox pattern and what problem does it solve?

It solves the dual-write problem: if a service saves to its database and then publishes a message, a crash between the two leaves them inconsistent. With an outbox, the service writes the business change and the outgoing message to an outbox table in the same local transaction. A separate relay process reads that table and publishes to the broker, retrying until it succeeds. That gives at-least-once delivery, so consumers must be idempotent.

Q: How do you make services resilient when a dependency is slow or down?

Timeouts on every network call, retries with exponential backoff and jitter for transient failures, and a circuit breaker so we stop hammering a failing service and fail fast instead. Bulkheads isolate resources so one slow dependency can't exhaust all threads. Where possible I'd switch to asynchronous messaging so a downstream outage just builds a queue. In .NET, Polly — now built into

`Microsoft.Extensions.Http.Resilience` — covers these policies.

Q: How do you debug a request that fails somewhere across ten services?

Distributed tracing with a correlation ID that flows through every call and message, so I can see the whole request path and where time was spent — OpenTelemetry is the standard. Add structured logging that includes the trace ID, and centralized logs and metrics. Without that observability in place first, I wouldn't adopt microservices at all.

4. Service-Oriented Architecture (SOA)

Q: What's the difference between SOA and microservices?

Both split a system into services, but they optimize for different things. SOA aims for enterprise-wide reuse: coarse-grained shared services connected through an Enterprise Service Bus that handles routing, transformation, and orchestration — "smart pipes, dumb endpoints." Microservices aim for independent change: small services owned by one team, each with its own data, communicating over lightweight protocols — "smart endpoints, dumb pipes." SOA services often share databases; microservices deliberately don't.

Q: What problems did the Enterprise Service Bus cause?

Business logic crept into the bus — transformations, routing rules, orchestration — so it became a central, hard-to-change system owned by a separate integration team. Every change needed that team, the bus became a bottleneck and a single point of failure, and it was hard to test. That experience is a big reason microservices push logic back into the services and keep the messaging layer simple.

Q: Is SOA still relevant?

Yes, especially in integration work. Many enterprises still run SOAP services, ESBs, and systems like ERPs and mainframes that must be integrated. The principles — clear service contracts, loose coupling, reusable capabilities — are still sound. What's changed is the tooling: lightweight brokers, API gateways, and integration platforms have mostly replaced heavy ESBs. In an integrations role, I'd expect to connect to SOA-era systems and wrap them behind cleaner APIs or an anti-corruption layer.

5. Event-driven architecture

Q: What's the difference between an event and a command?

A command is a request for something to happen, sent to one specific handler that can reject it — `PlaceOrder`. An event is a fact that already happened, named in the past tense, broadcast to any number of subscribers who can't reject it — `OrderPlaced`. Commands couple the sender to the receiver; events decouple them, because the publisher doesn't know who's listening. That distinction also drives the transport: commands usually go to a queue, events to a topic.

Q: Most brokers guarantee at-least-once delivery. How do you handle duplicate messages?

I make consumers idempotent, so processing the same message twice has the same effect as once. The common approach is to give each message a unique ID and record processed IDs in the same transaction as the business change, skipping any already seen. Some operations are naturally idempotent, like setting a status rather than incrementing a counter. That's also why "exactly-once" in practice means at-least-once delivery plus idempotent processing.

Q: What do you do with a message that keeps failing?

Retry a limited number of times with backoff for transient errors, then move it to a dead-letter queue so it stops blocking the rest. The DLQ needs monitoring and alerting, and a way to inspect, fix, and replay messages. I'd also distinguish transient failures — worth retrying — from permanent ones like a malformed payload, which should go straight to the DLQ.

Q: Choreography or orchestration — which would you choose?

Choreography — each service reacting to events on its own — is very loosely coupled and works well for simple flows with few steps. As a flow grows, it becomes hard to see the overall process and to handle failures, because the logic is spread across services. Orchestration puts the workflow in one coordinator, which is easier to understand, monitor, and change, at the cost of a central component. My rule: choreography for simple fan-out notifications, orchestration for multi-step business processes like order fulfillment.

Q: How do you guarantee ordering of events?

Global ordering is expensive and rarely needed; usually you need ordering per entity — all events for one order in sequence. With Kafka, I'd use the order ID as the partition key so its events land in one partition and are consumed in order. Azure Service Bus offers sessions for the same idea. Consumers should still tolerate out-of-order arrival where possible, for example by checking a version number.

6. Hexagonal / Clean / Onion

Q: What is the Dependency Rule in Clean Architecture?

Source-code dependencies can only point inward, toward the domain. The innermost layer — entities and business rules — knows nothing about databases, web frameworks, or UI. When the core needs something from the outside, like saving an order, it defines an interface, and an outer layer implements it. That's the Dependency Inversion Principle applied to the whole application, and it's what lets you test business logic without infrastructure.

Q: Explain ports and adapters.

A port is an interface the application core owns, describing what it needs or offers — for example `IOrderRepository` or `IPaymentGateway`. An adapter is the concrete implementation that connects that port to a real technology — an EF Core repository or a Stripe client. Driving (primary) adapters call into the app, like an API controller or a message consumer; driven (secondary) adapters are called by the app, like repositories and HTTP clients. Swapping a payment provider means writing a new adapter without touching the core.

Q: How is Clean Architecture different from a traditional layered architecture?

In a traditional layered app, dependencies flow down: the business layer depends on the data layer, so the domain is coupled to the database. Clean Architecture inverts that — the data layer depends on interfaces defined by the domain. The result is a core you can unit-test with in-memory fakes, and infrastructure you can replace. The structure looks similar on disk; the difference is which way the arrows point.

Q: How would you structure a .NET solution using Clean Architecture?

Four projects: **Domain** for entities, value objects, and domain events with no dependencies; **Application** for use cases, command/query handlers, and the interfaces (ports) it needs; **Infrastructure** for EF Core, external API clients, and messaging, implementing those interfaces; and **Web API** as the entry point that wires everything up with dependency injection. Application references Domain; Infrastructure and API reference Application — never the reverse.

Q: When would you *not* use Clean Architecture?

For simple CRUD services or short-lived tools with little business logic. The extra projects, interfaces, and mapping add cost without much benefit when there are no real rules to protect. In those cases a simple layered or vertical-slice approach is more honest. I'd reach for Clean Architecture when the domain logic is the valuable, long-lived part of the system.

7. CQRS

Q: What is CQRS, and why would you use it?

CQRS separates the model that changes data — commands — from the model that reads it — queries. The write side focuses on enforcing business rules; the read side returns data shaped exactly for each screen, often with no domain logic at all. I'd use it when reads and writes have very different shapes or loads, like a system with complex order validation but dashboards read thousands of times more often. It lets each side be optimized and scaled on its own.

Q: Does CQRS require event sourcing or two databases?

No — that's a common misconception. The simplest CQRS is just separate command and query handlers over the same database: commands go through the domain model, queries use lightweight SQL or Dapper projections straight to DTOs. Separate read stores and event sourcing are optional steps you take only when scale or requirements justify them. I'd start with the simple version and add complexity only where there's a clear need.

Q: With separate read and write stores, a user saves something and doesn't see it on the next screen. How do you handle that?

That's eventual consistency showing through. Options: return the updated state directly from the command so the UI can display it; have the UI optimistically show the change; read from the write store for that user's own recent changes (read-your-own-writes); or pass a version number and have the query wait briefly until the read model catches up. The right choice depends on how much the lag actually matters to the user — often a few hundred milliseconds is fine.

Q: What's the relationship between CQRS and Command-Query Separation?

Command-Query Separation, from Bertrand Meyer, is a method-level rule: a method either changes state or returns data, never both. CQRS applies the same idea at the architectural level, splitting whole models and sometimes whole data stores. CQS is a good default in any codebase; CQRS is a deliberate architectural choice for specific parts of a system.

8. Event sourcing

Q: What is event sourcing, and how is it different from a normal database?

A normal database stores current state — each update overwrites the previous value. Event sourcing stores the sequence of events that led to that state, append-only, and derives current state by replaying them. A bank account is the classic example: rather than storing a balance, you store deposits and withdrawals, and the balance is computed. The event log becomes the source of truth, and you get a complete audit trail for free.

Q: If you have millions of events, isn't rebuilding state slow?

Two answers. For a single entity, streams are usually short, and when they aren't, you take periodic snapshots — store the state at event 1,000, then replay only events after it. For queries, you never replay on read; you maintain projections — read models updated as events arrive — and query those. Projections are disposable: if one is wrong, you fix the code and rebuild it from the log.

Q: How do you change the structure of an event once it's been stored?

Stored events are immutable, so you never rewrite history. The common approaches are versioning events (`OrderPlacedV2`) and handling both, or upcasting — transforming old events into the new shape as they're read. Additive changes, like a new optional field with a default, are easiest. This is one of the hardest parts of event sourcing, so event schemas deserve the same care as a public API.

Q: How do you handle concurrent updates to the same entity?

Optimistic concurrency on the stream. When appending, the writer states the version it expects — "I read up to event 42." If someone else has written event 43 in the meantime, the append fails and the command is retried against the latest state. It avoids locks while guaranteeing no update is silently lost.

Q: When would you avoid event sourcing?

For most CRUD applications. It adds significant complexity — projections, eventual consistency, schema evolution, and tricky compliance issues like GDPR deletion, which often needs crypto-shredding. I'd reserve it for domains where history is genuinely valuable: finance, auditing, compliance, or where the business asks questions like "what did this look like last Tuesday?" And I'd apply it to one bounded context, not the whole system.

9. Serverless

Q: What is a cold start, and how do you reduce it?

A cold start happens when the platform has no warm instance ready, so it has to provision one, load the runtime, and initialize your code before handling the request — adding anywhere from a hundred milliseconds to several seconds. To reduce it: keep deployment packages small, minimize work in startup code, use pre-warmed or provisioned instances (Azure Functions Premium plan, AWS provisioned concurrency), and in .NET consider Native AOT for much faster startup. For latency-critical paths, I might choose containers instead.

Q: Why should serverless functions be stateless and idempotent?

Stateless because any invocation can land on a fresh instance, and instances are recycled at any time — anything kept in memory can vanish, so state belongs in external storage. Idempotent because triggers like queues retry on failure or timeout, so the same event can run more than once. If a function charges a card, running it twice must not charge twice — I'd use an idempotency key checked against storage.

Q: How would you implement a long-running, multi-step workflow with functions?

Not by chaining functions directly or running one long function — you hit timeouts and lose visibility. I'd use a durable orchestration service: Azure Durable Functions or AWS Step Functions. They persist workflow state between steps, handle retries, waiting for external events, fan-out/fan-in, and human approval steps, while each step stays a short, simple function.

Q: When is serverless more expensive than containers?

With steady, high-volume traffic. Pay-per-execution is great for spiky or low traffic because idle time is free, but at a constant high load, you're effectively paying a premium per request compared to always-on containers or VMs sized for that load. I'd model cost at expected volume before choosing, and it's common to mix: serverless for event-driven jobs and glue, containers for the core high-traffic API.

10. Microkernel (plug-in)

Q: Give an example of a microkernel architecture and explain how it works.

VS Code is a good one. The core editor handles files, text editing, and the extension host, and exposes an extension API. Language support, themes, debuggers, and linters are all extensions the core discovers and loads at runtime through a registry. The core stays small and stable while the ecosystem grows around it — and extensions run in a separate process, so a buggy one can't crash the editor.

Q: What is the hardest part of designing a plug-in architecture?

The plug-in contract. Once third parties build against it, every breaking change breaks their plug-ins, so it has to be designed carefully and versioned. I'd keep it small, expose capabilities through well-defined extension points, version it explicitly, and support older versions for a deprecation period. The other hard part is isolation — deciding whether plug-ins run in-process for speed or out-of-process for safety.

Q: How does microkernel relate to SOLID?

It's the Open/Closed Principle at system scale: the core is closed for modification but open for extension. New behavior arrives as new plug-ins rather than edits to the core. At the code level, the Strategy pattern is a small-scale version — selecting an implementation of an interface at runtime. In .NET, you'd typically discover plug-ins via dependency injection, assembly scanning, or `AssemblyLoadContext` for isolated loading.

11. MVC / MVVM

Q: What's the difference between MVC and MVVM?

In MVC, the Controller receives input, updates the Model, and chooses which View to render — the controller actively drives things, which suits request/response web apps like ASP.NET Core MVC. In MVVM, the ViewModel exposes state and commands, and the View binds to them declaratively; when the ViewModel's state changes, the View updates automatically. MVVM suits rich client UIs with data binding — WPF, .NET MAUI, SwiftUI. The key MVVM rule is that the ViewModel never references the View.

Q: Are MVC and MVVM software architectures?

They're presentation patterns — they organize the UI layer, not the whole system. They say nothing about how you structure business logic, data access, or services. In practice they sit in the outermost layer of a larger architecture: an ASP.NET Core MVC controller is a driving adapter in a Clean Architecture app, calling into the application layer's use cases.

Q: Why is MVVM good for testability?

Because the ViewModel holds all presentation logic — formatting, validation, enabling buttons, loading state — as plain code with no UI dependencies. You can unit-test it directly: set inputs, invoke a command, assert on the exposed state, with dependencies like services mocked. The View becomes a thin declarative layer with little logic to test.

Q: What is "Massive View Controller" and how do you avoid it?

It's a common problem in MVC — especially classic iOS UIKit — where the controller accumulates networking, validation, formatting, and navigation until it's thousands of lines. The fix is to move responsibilities out: business logic into services or use cases, presentation logic into ViewModels or presenters, navigation into coordinators. In web MVC the equivalent is a fat controller; keep controllers thin and delegate to an application layer.

12. Pipe-and-filter

Q: Give a real example of pipe-and-filter you've used.

The ASP.NET Core middleware pipeline is one most .NET developers use daily: each middleware — exception handling, authentication, routing, CORS — receives the request, does its job, and passes it to the next, then the response flows back out. Others are Unix pipes like `cat log | grep ERROR | sort`, and ETL jobs that extract, validate, transform, and load records in stages. Each filter does one thing and doesn't know about the others.

Q: What is backpressure and why does it matter in a pipeline?

Backpressure is a way for a slow stage to signal upstream stages to slow down. Without it, a fast producer overwhelms a slow consumer — buffers grow until memory runs out or data is dropped. Solutions include bounded queues that block when full, pull-based consumption where each stage requests data when ready, or scaling out the slow stage. In .NET, `System.Threading.Channels` with a bounded capacity gives you backpressure built in.

Q: How do you handle a bad record in the middle of a large batch pipeline?

Don't fail the whole run for one bad record. Validate early, route records that fail validation or transformation to an error or dead-letter output with the reason attached, and let the rest continue. Then alert on error counts and make it easy to fix and reprocess. I'd also make stages idempotent so a failed run can be safely re-run from a checkpoint instead of from scratch.

Cross-pattern scenario questions

These open-ended questions test judgment across patterns. There's no single right answer — interviewers listen for clarifying questions, trade-offs, and a sensible starting point.

Q: How would you migrate a large legacy monolith to microservices?

Incrementally, using the Strangler Fig pattern — never a big-bang rewrite. First, I'd put a routing layer or API gateway in front of the monolith. Then pick one well-bounded, relatively low-risk capability, build it as a new service with its own data, and route that traffic to it, keeping an anti-corruption layer where it talks to the old system. Repeat, shrinking the monolith over time. Before any of that, I'd make sure CI/CD and observability are in place, and I'd honestly ask whether modularizing the monolith would solve the problem more cheaply.

Q: Design an integration that syncs orders from our system to a third-party ERP that's sometimes unavailable.

I'd decouple with messaging. When an order is saved, write an `OrderPlaced` event to an outbox in the same transaction; a relay publishes it to a queue. A consumer transforms it into the ERP's format through an adapter and calls the ERP with timeouts, retries with backoff, and a circuit breaker. If the ERP is down, messages wait in the queue; persistent failures go to a dead-letter queue with alerting. The consumer is idempotent — keyed on order ID — so retries never create duplicates in the ERP. I'd also add reconciliation to catch anything that slipped through.

Q: Your read traffic is growing far faster than writes, and reports are slowing down transactions. What do you do?

Start with the cheapest fixes: indexes, query tuning, and caching hot reads. Next, move reporting queries to a read replica so they stop competing with transactions. If read and write shapes keep diverging, introduce CQRS for that area — commands against the primary, queries against denormalized read models updated via events. Each step adds complexity, so I'd only take the next one when metrics show the previous one isn't enough.

Q: What's the most important factor when choosing an architecture?

The constraints of the specific situation — team size and skills, how well the domain is understood, scale and latency needs, consistency requirements, and how fast the business needs to change. Architecture is about trade-offs; every pattern buys one quality by spending another. I'd pick the simplest thing that meets today's needs, keep boundaries clean so it can evolve, and record the decision and its reasoning in an Architecture Decision Record so future teams know why.