

Architecture Patterns

Playbook



LEVEL UP
CODING

LEVEL UP CODING

Copyright Notice

© 2025 Level Up Coding. All rights reserved.

This ebook was created to help engineers sharpen their system design skills.

Copying, distribution, or repackaging of this ebook, in whole or in part, is not permitted without consent.

If you're writing about system design, keep it original, don't copy from here.

For educational use only.

If you'd like to share it, please share **blog.levelupcoding.com** so others can grab their own copy, rather than distributing this file.

Thanks for respecting our work.

Published by Level Up Coding

www.levelupcoding.com

Table of Contents

Monolithic	1
Modular Monolith	4
Microservices	7
Event-Driven Architecture (EDA)	10
Serverless Architecture	13
Domain-Driven Design (DDD)	16
Clean Architecture	19
Strangler Fig Pattern	22
Backend-For-Frontend (BFF)	25
Command Query Responsibility Segregation (CQRS)	28

Monolithic

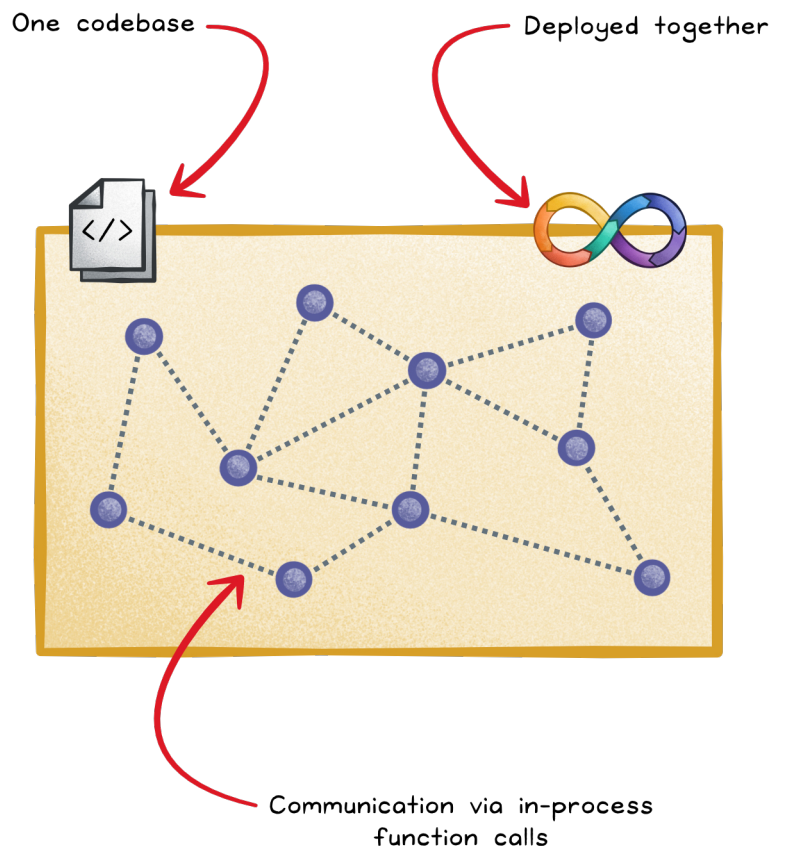
- i** A single, unified application where all features share one codebase, database, and deployment. It scales by cloning the entire application, and any change requires redeploying the whole system.

Benefits	Tradeoffs	When To Use
• Unified codebase deployment • Low-latency function calls • Simplified debugging process • Strong data consistency • Minimal infrastructure needs • Suited for small teams	• No independent scaling • Full redeploy required • Increases complexity over time • Risk of tight coupling • Single point of failure • Tech stack lock-in • Team coordination challenges	• Fast early development • Unclear or evolving domain boundaries • Modest traffic • Minimal DevOps capacity • Requires strong consistency



How It Works

Monolithic architecture builds the entire application as one cohesive unit. All features (e.g. product catalog, payments, user accounts) live in one codebase, run in one process, and are deployed together. All internal communication is handled via in-process function calls, not over the network. Often, the app uses a single shared database, and scaling is done by cloning the entire application (vertical or horizontal scaling). A change to any part of the system requires a redeployment of the whole application.



Benefits

- Monolithic applications **simplify development and deployment** by consolidating everything into one codebase and deployment pipeline.
 - Because internal module calls are in-process, monolithic systems benefit from **faster performance and lower latency** compared to distributed systems.
 - **Debugging and testing are more straightforward**, as developers can trace requests through the entire stack within a single environment.
 - A shared database across the system makes it **easier to maintain strong data consistency and support ACID transactions across modules**.
 - With fewer moving parts, monolithic systems impose **lower operational overhead**, requiring less infrastructure and tooling to manage.
 - This architecture suits small teams particularly well, enabling **faster collaboration and iteration** without the need for service boundaries or coordination across repos.
-

Tradeoffs

- Monolithic systems **cannot scale components independently**, so the entire application must be scaled even if only one part is under load.
- **Deployment becomes a bottleneck**, as even small changes require rebuilding and releasing the full application, introducing risk and slowing down iteration.
- Over time, as more features are added, the codebase **can become complex and tightly coupled**, making it harder to maintain and evolve.
- **A failure in any module has the potential to crash the entire system**, since all components share the same process.
- **Technology choices are locked in across the system**, making it difficult to adopt new frameworks or languages for specific areas.
- As teams grow, **coordinating work in a single shared codebase can become a productivity drag**, leading to merge conflicts and slower delivery cycles.



When to Use

- Monolithic architecture is ideal for early-stage products or MVPs, **where speed of development and simplicity of deployment matter most.**
 - It works well when **domain boundaries are unclear or evolving**, as the flexibility of a single codebase allows for rapid iteration and restructuring.
 - Applications with **modest traffic levels** can be efficiently served by scaling the whole system **without the need for granular optimization.**
 - Teams with **limited DevOps experience or small engineering headcount** benefit from the lower infrastructure complexity of a monolith.
 - Use a monolith when your system **requires strong consistency guarantees or tightly coupled operations across modules.**
 - When **most user flows touch many parts of the application simultaneously**, keeping everything in one process avoids unnecessary network complexity.
-



Real-World Examples

A small content publishing platform, such as a blog or forum, can start as a monolithic application that combines publishing, commenting, and user management into a single codebase.

Each feature interacts through in-process calls and shares the same database, making it simple to build, test, and deploy. When a user posts a comment, the system handles authentication, data validation, and updates in a single transaction.

This tight integration allows small teams to iterate quickly without managing service boundaries or infrastructure overhead. With modest traffic, the platform can scale vertically or run a few instances behind a load balancer, delivering performance and reliability without the need for distributed complexity.

Modular Monoliths

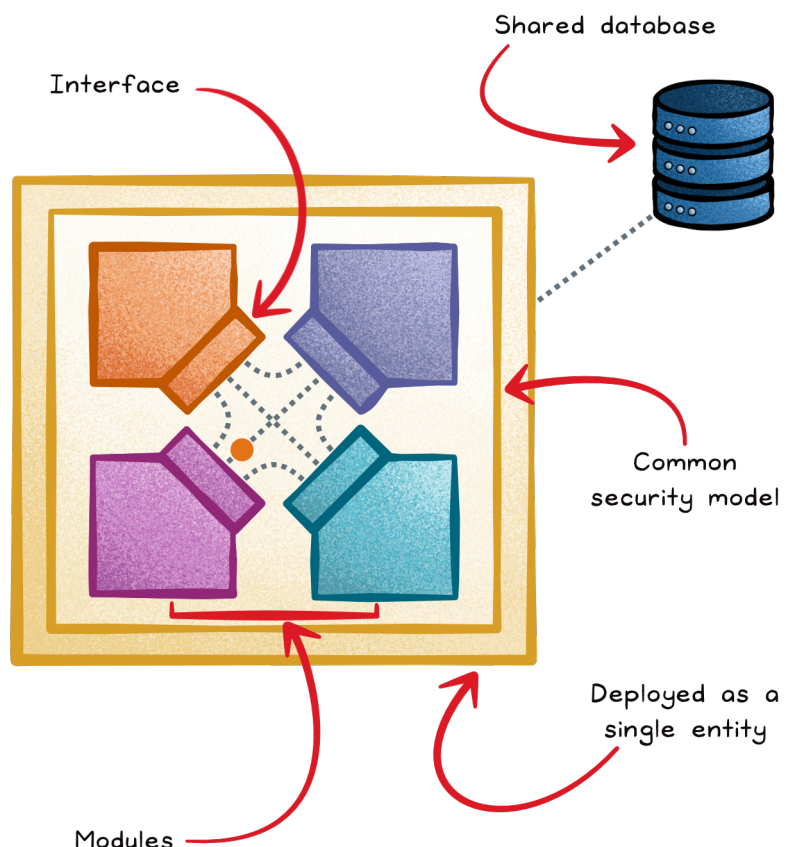
i Unlike a traditional monolith, a modular monolith organizes the codebase into well-defined internal modules, designed to deliver the simplicity of a monolith with the separation benefits of microservices, without distributed complexity.

Benefits	Tradeoffs	When To Use
• Simple infrastructure management • Improved performance • Strong data consistency • Clean modular codebase • Streamlined testing and debugging • Fast iteration • Smooth microservices transition • Lower infrastructure costs	• No independent scaling • System-wide failure risk • Limited tech flexibility • Module boundaries require discipline • Team coordination challenges • Shared release bottleneck • Risky full redeploys	• Prioritize speed and simplicity • Flexible refactoring • Limited DevOps capacity • Require transactional consistency • Moderate scaling suffices • Defer microservices complexity



How It Works

A modular monolith is a single-deployment application structured as a set of internal modules aligned to business domains. Modules expose clear interfaces and own their data (e.g. separate tables/schemas) within a shared database. All inter-module communication is in-process, not over the network. The entire system is built and released as one unit, delivering monolith simplicity while enabling clean boundaries and a viable path to later service extraction.



Benefits

- **Deployment and infrastructure management are simpler**, since the entire system is a single unit without the need for service discovery or orchestration tools.
 - **Performance is improved** due to in-process communication between modules, eliminating the network latency of distributed calls.
 - **Strong data consistency is easier to maintain** across modules by using ACID transactions within a shared database.
 - Modular boundaries within the monolith **keep the codebase clean and navigable**, making it easier to onboard new engineers and evolve the system over time.
 - **Testing and debugging are more straightforward**, as all code runs in one environment and errors can be traced without coordinating across multiple services.
 - Developers can **iterate quickly in early stages**, since all modules are part of one build and one deployment pipeline.
 - The modular structure makes it **easier to eventually extract services** if scaling or autonomy needs arise, which offers a smooth path to microservices.
 - **Infrastructure costs are lower** because the system runs as a single application, avoiding duplication across services.
-

Tradeoffs

- You **can't scale modules independently**, so resource-heavy components require scaling the whole system together.
- **A failure in one module can affect the entire application**, since all modules share the same runtime environment.
- You are generally **constrained to a single tech stack**, which can be limiting when different modules would benefit from different tools or languages.
- **Module boundaries require discipline, since there's no physical isolation**, over time developers may bypass interfaces, increasing coupling and reducing **maintainability**.
- Large teams working on a single codebase can face **coordination challenges, version control conflicts, and longer build times**.
- All modules share a release cycle, so **delays in one module's development can impact unrelated areas** unless mitigated by feature flags or branching strategies.
- **Even minor code changes require redeploying the entire application**, which can make releases riskier as the system grows.



When to Use

- You're building a new product or MVP and want to **prioritize speed and simplicity over premature scalability**.
 - Your domain boundaries are still forming, and you want the **flexibility to refactor easily before committing to service separation**.
 - **Your team is small or lacks DevOps support**, making the overhead of microservices impractical.
 - You **require transactional consistency across multiple operations** (e.g. placing an order and updating inventory atomically).
 - Your system has **moderate scaling needs** that can be handled through vertical scaling or whole-app replication.
 - You want to **defer the complexity of microservices until a proven need arises**, while keeping future migration options open.
-



Real-World Examples

A fintech company creates a customer-facing portal for managing accounts, investments, and loan applications. The system is built as a modular monolith with modules such as Authentication, Account Management, Portfolio Analytics, and Loan Processing. Because users often perform operations that span multiple modules in a single session (e.g. checking balances, applying for a loan, and reviewing investments), the architecture enables real-time consistency and avoids the need for distributed transactions. As usage grows, the compute-heavy analytics module becomes a candidate for extraction into a separate service, while the rest remains integrated.

Microservices

i Microservices are a distributed architecture style that breaks an application into independently deployable services, each mapped to a specific business function.

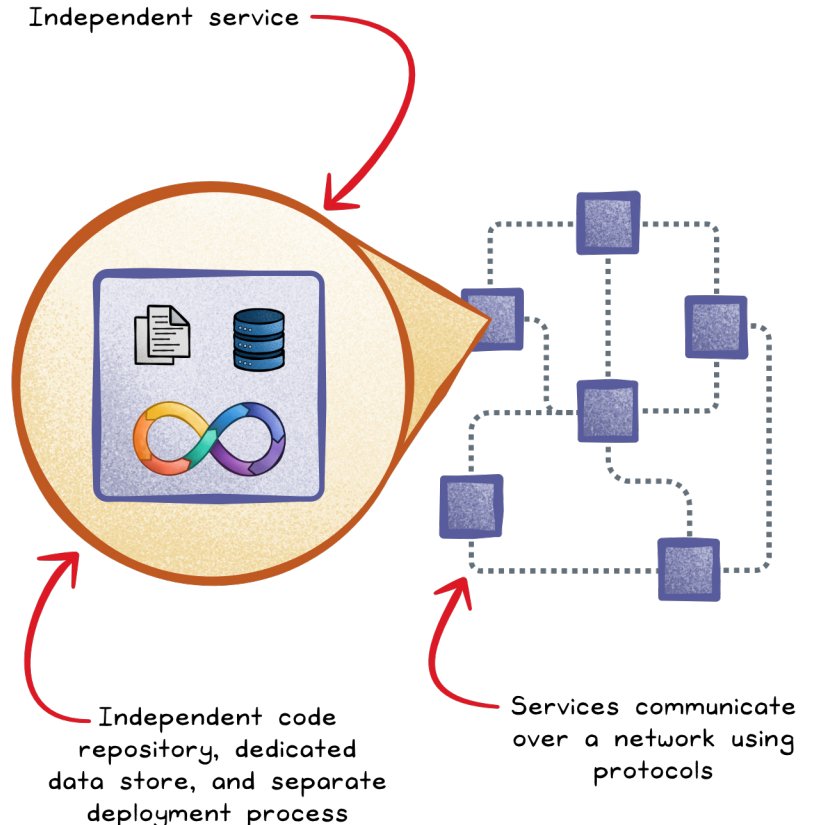
Benefits	Tradeoffs	When To Use
• Independent deploys • Faster and safer release cycles • Improves resilience • Reduces dependencies and speeds up delivery • End-to-end ownership of services empowers small teams • Flexible technology choices	• Increases system complexity • Adds operational overhead • Adds network latency • Harder to maintain data integrity • Testing and debugging efforts increase	• Some components have heavy scaling demands • Rapid iteration and frequent deployments is needed • There are clear functional boundaries • Teams need autonomy • High availability is critical



How It Works

Microservices architecture splits a system into independent services, each with its own codebase, data store, and deployment lifecycle. These services communicate over a network using protocols like HTTP, gRPC, or asynchronous queues (e.g. Kafka, NATS, RabbitMQ). Each service typically aligns with a bounded context and is owned end-to-end by a dedicated team. Deployments are often containerized and orchestrated (e.g. Docker + Kubernetes) to enable isolated scaling and orchestration.

Independent service



Benefits

- With clear service boundaries and strong CI/CD practices, teams can deploy services independently, **enabling faster, safer release cycles**.
 - **Resilience improves as failures stay contained**; for instance, if the analytics service fails, users can still browse and enroll in courses without interruption.
 - Teams can iterate on their own services without needing to coordinate across the entire codebase, **reducing dependencies and speeding up delivery**.
 - End-to-end ownership of services empowers small teams to **make decisions quickly and avoid bottlenecks**.
 - A polyglot approach allows each service to **use the most suitable technology stack or language for its purpose**.
-

Tradeoffs

- **System complexity grows** as teams manage inter-service communication, distributed failure modes, and asynchronous coordination.
- Running multiple services introduces **operational overhead**, from setting up service discovery and container orchestration to maintaining distributed logging, tracing, and CI/CD pipelines.
- Network calls replace in-process function calls, **adding latency and requiring strategies** like caching, batching, and circuit breakers to maintain performance.
- **Data integrity becomes harder to maintain**, as distributed transactions often rely on eventual consistency or orchestration patterns such as Sagas instead of simple ACID guarantees.
- **Testing and debugging efforts increase**, since integration and end-to-end tests must span multiple services, with issues potentially spread across log streams and network layers.



When to Use

- **Some components, such as search or real-time analytics, face heavier scaling demands** and benefit from being able to scale independently from the rest of the system.
 - **Rapid iteration requires frequent deployments**, with teams updating their domains without waiting on others or risking changes to unrelated services.
 - **Clear functional boundaries** make it easy to align services with business domains like payments, content delivery, or identity.
 - Multiple teams need the **freedom to work autonomously**, and a shared monolith is creating bottlenecks in velocity, ownership, and code quality.
 - **High availability is critical**, and the system must degrade gracefully, containing failures to a single service rather than allowing them to cascade through the application.
-



Real-World Examples

An e-commerce platform splits into services like product catalog, cart, orders, payments, accounts, and recommendations. Each is deployed, scaled, and updated independently. During holiday traffic spikes, order and payment services scale out without affecting catalog or search. Teams ship changes like new payment options without touching unrelated code. If recommendations fail, checkout continues, preserving resilience and velocity.

Event-Driven Architecture (EDA)

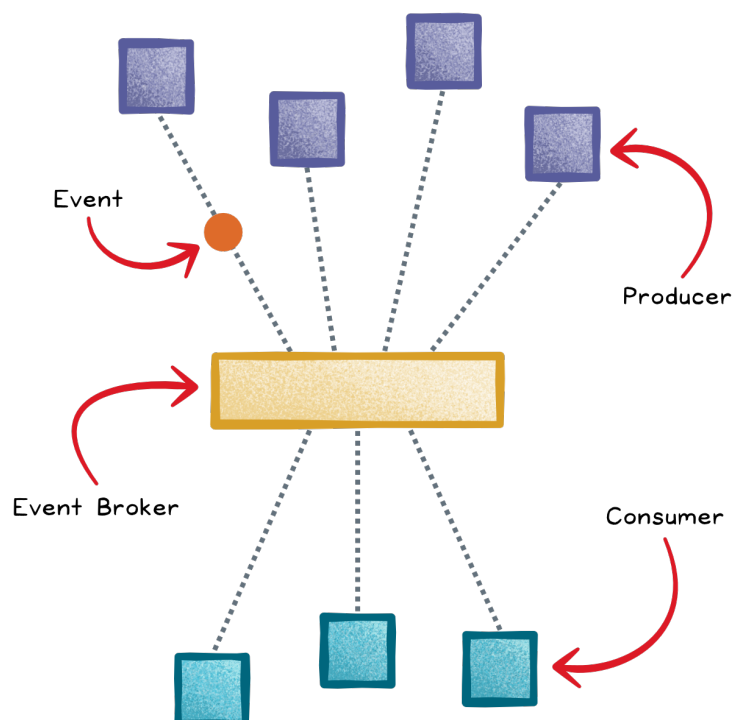
i A reactive architecture where services communicate by emitting and responding to events. Producers and consumers are decoupled, enabling flexible, loosely coupled systems.

Benefits	Tradeoffs	When To Use
• Loosely coupled • Scalable and elastic • Real-time responsiveness • Greater extensibility • Efficient resource use • Resilient	• Higher architectural complexity • Debugging and observability are harder • Eventual consistency • Latency variability • Error handling requires compensation logic • Infrastructure overhead	• Has asynchronous workflows • Needs near-real-time responsiveness • Bursty or variable load patterns • Needs a unified event channel for different systems • You anticipate rapid changes



How It Works

Event-Driven Architecture (EDA) is an architectural style where components publish immutable events (e.g. "Order Placed") to an event broker (e.g. Kafka, RabbitMQ). Producers don't know consumers; subscribers consume events asynchronously via topics/queues, enabling fan-out and loose coupling. Delivery is typically at-least-once, so systems use idempotency, retries, and dead-letter queues. This decoupling supports near-real-time reactions and modular system evolution.



Benefits

- Services can **evolve independently** thanks to loose coupling between components.
 - Each consumer **independently scales** to match its own workload
 - Events trigger immediate reactions, supporting **real-time**, user-facing experiences.
 - New consumers can be added without altering existing producers, **increasing extensibility**.
 - **Resources are used efficiently** by acting only when events occur.
 - Failed consumers don't block producers, with retries and queues ensuring **resilience**.
-

Tradeoffs

- **Architectural complexity grows** as event flows increase in number.
- Asynchronous communication and decoupling make **debugging and observability more challenging**.
- Some scenarios can't tolerate the **eventual consistency** that event-driven systems impose.
- Network and broker delays can **introduce variability in latency**.
- **Handling errors often requires compensation logic** such as sagas or retries.
- A reliable event broker and supporting tooling introduces **infrastructure overhead**.



When to Use

- You have **asynchronous workflows** (e.g. processing an order across multiple services).
 - You need to **react in real-time** (e.g. dashboards, IoT, fraud detection).
 - Your system has **bursty or variable load** patterns that need buffering and elastic scaling.
 - You're building a **microservices system** and want to avoid tight coupling.
 - You need to **integrate diverse systems** through a unified event channel.
 - You anticipate rapid changes, requiring **high extensibility** without service coordination.
-



Real-World Examples

In a supply chain system, every package movement, such as scanning a shipment at a warehouse, loading it onto a truck, or crossing a delivery checkpoint, emits an event (e.g. "Package Shipped" or "Arrived at Depot"). These events trigger real-time updates across multiple services: the tracking service updates the customer-facing interface, the inventory system adjusts in-transit stock levels, and the alerting service notifies customers or partners of delays or delivery windows. This event-driven model ensures high visibility and coordination across a distributed logistics network, enabling faster response to issues like delays or rerouting needs.

Serverless Architecture

i A managed backend model where compute, storage, and databases scale automatically. No server provisioning, patching, or capacity planning required.

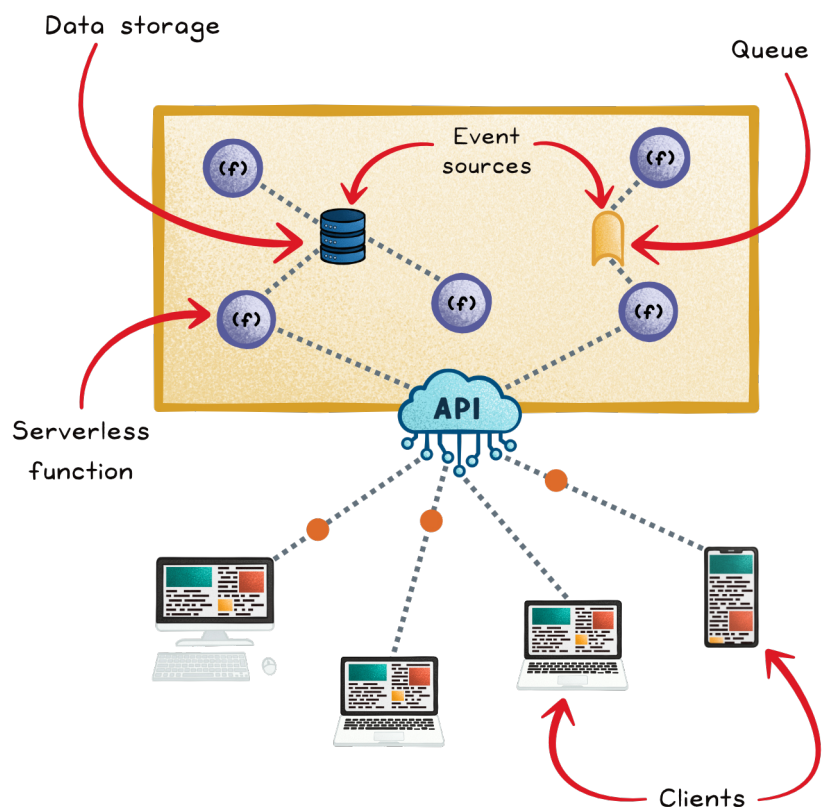
Benefits	Tradeoffs	When To Use
• No need to provision and maintain infrastructure • Services scale automatically and granularly • Only charged for what you use • Rapid, low-risk releases • Availability and fault tolerance often built-in	• Vendor lock-in • Removes control over runtime, hardware, and networking • Hard caps on memory, execution time, and concurrency • Cold starts & tail latency • Increased observability complexity	• Has event-driven workflows or background tasks • Bursty, unpredictable, or infrequent traffic • Benefits from auto-scaling and cost-efficiency • Needs to speed up time-to-market • Needs scheduled jobs or cron tasks • Requires quick iteration



How It Works

Serverless architecture is a cloud-native approach where applications are built primarily on fully managed, on-demand services.

The cloud provider takes care of infrastructure setup, scaling, and operations, so developers can focus solely on business logic and integration rather than infrastructure.



Benefits

- **Developers no longer provision or maintain infrastructure**, which reduces operational overhead and lets teams focus on business logic.
 - **Servers, databases, APIs, and other managed services scale automatically and granularly with demand**, ensuring the system can absorb sudden traffic spikes without manual intervention.
 - Pay-per-use billing means **you are charged only for resources consumed**, often lowering costs for bursty or sporadic workloads.
 - Independent resources and managed services enable **rapid, low-risk releases, so new features reach production faster**.
 - **High availability and fault tolerance** are often built-in by the provider.
-

Tradeoffs

- Deep integrations with a single cloud's resources and tooling can create significant **vendor lock-in** and migration costs.
- Teams relinquish low-level control over runtime, hardware, and networking, **limiting optimization options and support for custom dependencies**.
- **Hard caps on memory, execution time, and concurrency** that constrain high-volume or long-running workloads.
- **Cold starts** and runtime initialization add **unpredictable latency**, raising **p95/p99** and impacting user-facing responsiveness after **idle** periods or sudden bursts.
- Distributed, event-driven execution makes local **testing, debugging, and observability harder**, requiring specialized tooling and increasing operational overhead.



When to Use

- Your application has **event-driven workflows or background tasks** that can run **asynchronously**.
 - You need to handle **bursty, unpredictable, or infrequent traffic** without provisioning for peak load.
 - You're building backend APIs for web or mobile applications that benefit from **auto-scaling and cost-efficiency**.
 - You want to **reduce time-to-market by composing applications with managed services** instead of building everything in-house.
 - You need to run **scheduled jobs or cron-like tasks** without keeping servers always-on.
 - You're **prototyping or iterating quickly** and want to **avoid infrastructure setup and maintenance overhead**.
-



Real-World Examples

A video-sharing platform like Vimeo or TikTok can build its upload and processing pipeline entirely on serverless services. When a user uploads a video, the request is routed through a serverless API Gateway, which authenticates the user and writes initial metadata to a serverless database (e.g. DynamoDB, Firestore). The video file itself is stored in serverless object storage (e.g. Amazon S3, Azure Blob Storage). Multiple serverless functions then process the video and notify the user once it's done. Because uploads are unpredictable and processing is bursty, serverless allows the platform to scale up aggressively and then scale to zero when idle, without reserving compute capacity.

Domain-Driven Design (DDD)

i DDD is a strategy for modeling complex systems where code structure, service boundaries, and team ownership mirror distinct business functions.

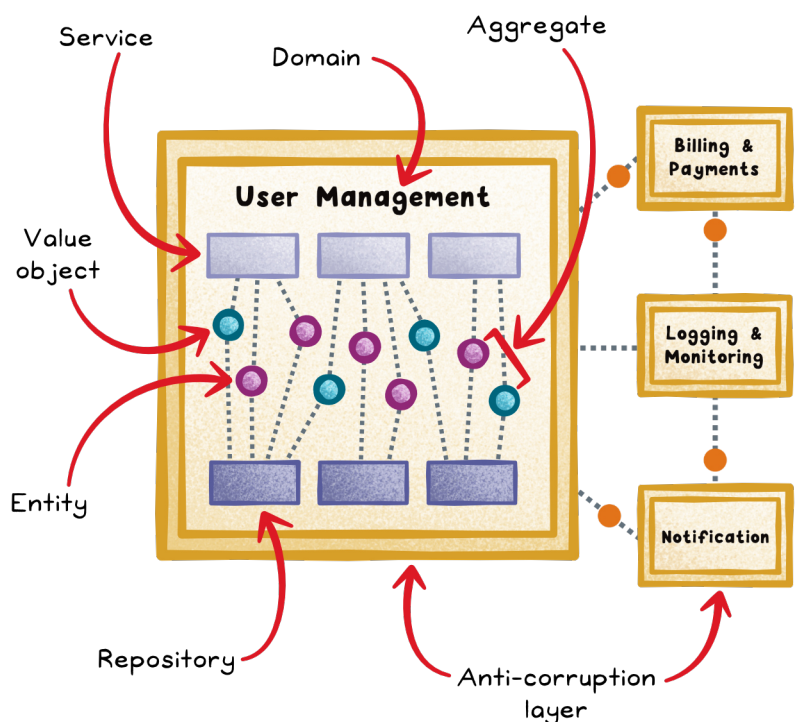
Benefits	Tradeoffs	When To Use
• Stays aligned to business needs • Clearer communication between tech and business • Modular, decoupled components • Encapsulates core domain complexity • Cleaner and more purposeful code • Clear ownership across domains	• Steep learning curve • Significant upfront effort • Over-engineered for simple applications • Integration between bounded contexts adds complexity • Requires regular access to domain experts	• The domain is complex • Correctness and flexibility are critical • System spans multiple teams



How It Works

Domain-Driven Design (DDD) is an architectural approach that centers software around the business domain.

Teams model the domain with a shared ubiquitous language, split the system into bounded contexts (strategic design), and implement each context with tactical patterns such as Entities, Value Objects, Aggregates (with invariants), Repositories, Domain Services, and Domain Events. Context maps (e.g. Anti-corruption Layer, Customer/Supplier, Conformist, Shared Kernel) define how contexts integrate.



DDD can be applied to a modular monolith or microservices; the focus is on clear boundaries and truthful models, not a specific deployment style.

Benefits

- **Software stays aligned with business needs**, as domain models directly reflect business logic and rules.
 - **Communication between tech and business is clearer**, thanks to the use of a shared, ubiquitous language.
 - Bounded contexts create **modular, decoupled components** that are easier to maintain and evolve.
 - **Core domain complexity is encapsulated**, making it easier to respond to changes in business rules or workflows.
 - Isolating domain logic from technical concerns makes systems more testable, with **cleaner and more purposeful code**.
 - Team structures can mirror bounded contexts, allowing for **clear ownership** and faster development across domains.
-

Tradeoffs

- There's a **steep learning curve**, especially for teams unfamiliar with DDD terminology and patterns.
- Requires **significant upfront effort** to model the domain and build shared language with domain experts.
- **Over-engineered for simple applications** where CRUD functionality and minimal business logic are sufficient.
- **Integration between bounded contexts adds complexity**, particularly around data consistency and eventual consistency.
- **Success depends on regular access to domain experts** because without them, the domain model can drift from reality.
- **Cultural shift may be required** as developers must think like problem-solvers in the domain, not just coders.



When to Use

- **The domain is complex**, with many business rules, workflows, or changing requirements over time.
 - You are working on **a core business system where correctness and flexibility are critical** to long-term success.
 - The system **spans multiple teams**, and you need clear boundaries to avoid bottlenecks and interference.
-



Real-World Examples

In a banking platform, a customer initiates a funds transfer. The request is handled inside the Accounts/Ledger bounded context, which enforces rules (e.g. sufficient funds, overdraft limits) and applies the debit/credit atomically in one transaction. After committing, Accounts/Ledger publishes a FundsTransferred integration event. Other contexts react independently: Risk Assessment evaluates fraud, Alerts notifies the customer, and the Customer Portal/Reporting updates its read model for balances and statements. No context writes directly to another's data; they integrate through events and clear contracts. The result is autonomy with coherence; each domain enforces its own rules while participating in end-to-end financial workflows.

Clean Architecture

i An architectural style that protects core logic by ensuring all dependencies point inward. Frameworks, databases, and UIs become replaceable.

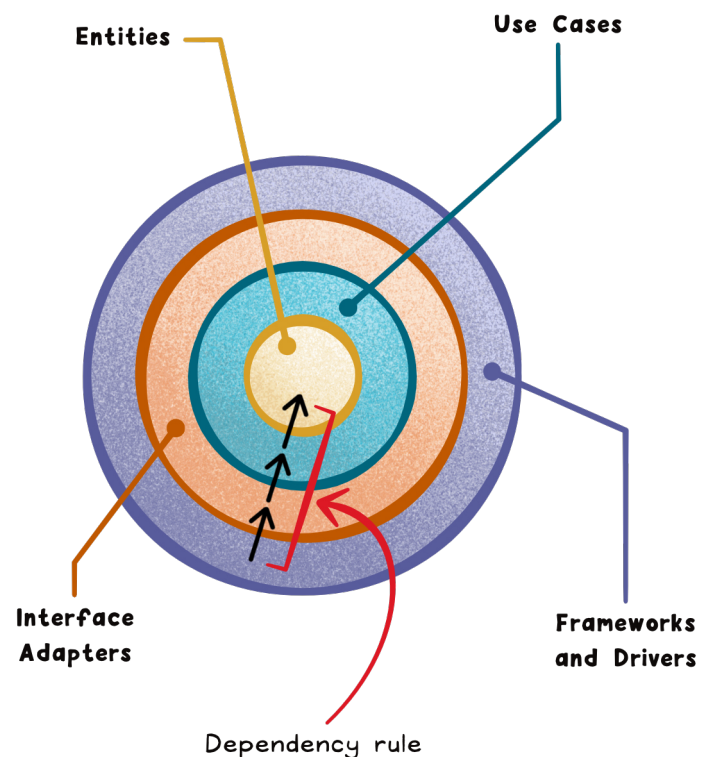
Benefits	Tradeoffs	When To Use
• Highly maintainable over time • Business rules can be unit tested in isolation • Technology choices remain flexible • Teams can work in parallel across layers • Consistent business behavior is enforced across multiple interfaces • Supports long-term evolution	• Adds extra boilerplate and indirection • Steep learning curve • Development may be slower at first • Debugging and performance can be less straightforward • Over-engineers small or short-lived applications	• Application is complex and long-lived • Undergoes many changes over time • Rich domain logic • Multiple user interfaces or external integrations • Teams need clear separation of responsibilities



How It Works

Clean Architecture is a layered design approach that organizes software into concentric rings, keeping core business logic independent from external details like databases, frameworks, or UI. At its core are Entities (enterprise-wide business rules), surrounded by Use Cases (application-specific logic), then Interface Adapters (translating between domains and externals), and finally Frameworks & Drivers (UI, databases, APIs).

The key principle is the Dependency Rule: dependencies always point inward, ensuring that high-level policy doesn't rely on low-level details. This creates a highly decoupled,



modular, and technology-agnostic system where the core remains stable even as the surrounding layers evolve.

Benefits

- Changes to the UI, database, or frameworks can be made without affecting the core business logic, making the system **highly maintainable over time**.
 - **Business rules can be unit tested in isolation**, enabling robust, fast, and framework-agnostic test suites.
 - **Teams can work in parallel across layers** (e.g. backend vs UI) with less coordination, improving development velocity and scalability.
 - **Consistent business behavior is enforced across multiple interfaces**, reducing bugs and duplicated logic.
 - **The architecture supports long-term evolution** and can act as a foundation for migrating to microservices or distributed systems later on.
-

Tradeoffs

- Enforcing layer separation introduces **extra boilerplate and indirection**, increasing the volume of code and development effort for even simple features.
- **The learning curve is steep**; teams unfamiliar with the principles may misapply them and create unnecessary abstractions.
- **Development may be slower at first**, especially for prototypes or MVPs, due to the need to define interfaces, adapters, and separate concerns rigorously.
- **Debugging and performance tuning may be less straightforward** due to additional layers and abstraction boundaries.
- It **may be over-engineered for small or short-lived applications** with minimal business logic.



When to Use

- The application is **complex, long-lived**, and will undergo **many changes** over time.
 - The system has **rich domain logic** that needs to be protected from external concerns.
 - The system must support **multiple user interfaces or external integrations**.
 - **Teams require clear separation of responsibilities** to support parallel development.
 - You **anticipate swapping or evolving parts of your tech stack** and want to isolate the core from those changes.
 - **High reliability, maintainability, or testability is a requirement**, such as in regulated industries or mission-critical systems.
-



Real-World Examples

An internal employee scheduling system used by HR and managers can evolve rapidly as company processes change. Clean Architecture isolates core rules like shift eligibility, overtime limits, and employee availability in the inner layers. UI changes (e.g. moving from web to a Slack bot interface) or integrations (e.g. replacing an HR database or payroll system) affect only the adapter layer. This makes the system resilient to internal system churn, and easier to extend for new functionality like compliance tracking or automated shift suggestions.

Strangler Fig Pattern

- i** A stepwise migration strategy where modern services are built in parallel and gradually intercept traffic from the legacy system, replacing it piece by piece.

Benefits	Tradeoffs	When To Use
• Business operations aren't disrupted • Reduces migration and rewrite risks • New components ship as soon as they're ready • Quicker feedback loops • Focuses on smaller, manageable pieces • Legacy pain points are prioritized	• Add complexity and maintenance overhead • Data synchronization can be challenging • Risks re-implementing old technical debt • Requires deep understanding of legacy code • Transition can take years • Proxies can be a bottleneck or single point of failure	• A migration or rewrite is necessary • The system is large and complex • Downtime must be avoided • Certain components should be prioritized first • Domains or features can be isolated

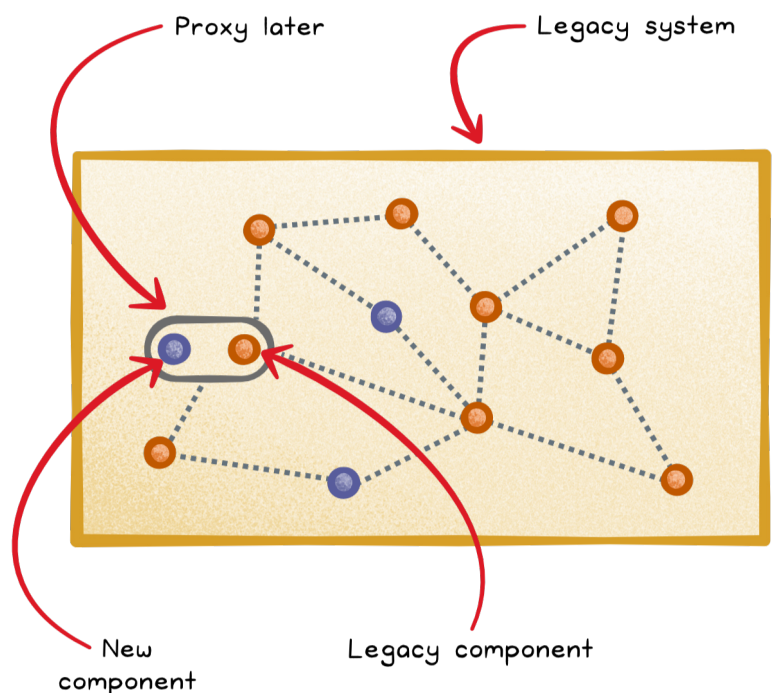


How It Works

The Strangler Fig Pattern is a modernization approach where a legacy system is gradually replaced by a new system through incremental refactoring.

Inspired by the strangler fig tree that grows around and eventually replaces its host, this pattern wraps the legacy system with a proxy or facade.

Over time, functionality is carved out into new components or services, which are deployed in parallel and gradually take over requests from the old system. Eventually,



the legacy system is phased out entirely, enabling a safer, low-risk evolution without service disruption.

Benefits

- **Business operations can continue without disruption**, as old and new systems run side-by-side during migration.
 - Risk is reduced by replacing **small, manageable pieces incrementally** rather than performing a risky “big bang” rewrite.
 - New features can be deployed faster, since each new component can **ship as soon as it’s ready**.
 - Teams gain the flexibility to **adapt and learn as they go**, improving future iterations of the migration.
 - Focus remains on one slice at a time, **avoiding the complexity of maintaining a full duplicate system**.
 - **Legacy pain points can be prioritized**, enabling early ROI with each improvement.
 - Over time, the new system **benefits from better architecture, cleaner code, and improved modularity**.
-

Tradeoffs

- Running a hybrid of old and new systems adds **complexity and maintenance overhead**.
- **Synchronizing data between systems can be challenging and error-prone**.
- There’s a **risk of re-implementing old technical debt** unless intentional refactoring is prioritized.
- Migration **requires deep understanding of legacy code**, which may be hard to find or retain.
- **The transition may take years**, and stalled efforts can leave teams with a more complex hybrid system.
- Carving out clean slices **can be hard in tightly coupled legacy systems**.
- The **proxy or facade can become a bottleneck or single point of failure** if not designed with resilience in mind.



When to Use

- Your system is **large, complex, and too risky to rewrite all at once**.
 - The system **must remain online continuously**, without significant downtime.
 - **Technical debt in the legacy system makes further development slow or brittle.**
 - **Certain modules need improved scalability or performance** without overhauling the entire system.
 - **You can isolate domain slices or features** that can be incrementally migrated.
 - **Modernization is a long-term strategy** and can be done in phases over time.
-



Real-World Examples

A government agency replaces parts of a legacy case tracking system by introducing modern web-based modules for document upload, status tracking, and communication. The transition happens slice by slice, allowing services to remain operational without citizen-facing downtime.

Backend-For-Frontend (BFF)

- i** In a BFF architecture, each frontend (web, mobile, etc.) has its own backend service that tailors API responses, isolates frontend-specific logic, and simplifies integration with internal systems.

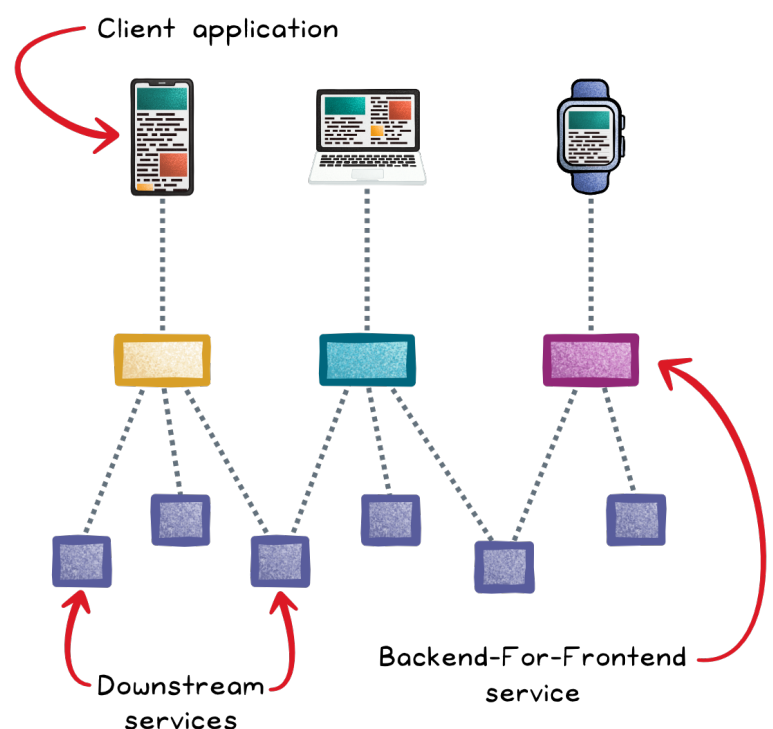
Benefits	Tradeoffs	When To Use
- Improves responsiveness and reduces bandwidth usage - Aggregates and transforms data on the server side - Frontends are simplified - Greater team autonomy - Error handling and security are customizable per client - Avoids wide technology and vendor lock-in	- Each BFF introduces another service - Risk of duplicated logic - More potential points of failure - Can introduce latency - Requires robust CI/CD practices for version management and deployment coordination - Can introduce unnecessary complexity	- Multiple frontend types are supported - Frontends require complex data aggregation or transformation - Different clients have different performance constraints or limitations - Frontend and backend teams want independence - Integrated to third-party clients or partners - Frontends need different data scopes, logic, and access controls



How It Works

Backend-For-Frontend (BFF) is an architectural pattern where a separate backend service is created for each specific frontend client such as web, mobile, or third-party applications.

BFF has also evolved to support specific product surfaces or feature areas, like Home or Search. Instead of using a single general-purpose API to serve all clients or surfaces, each BFF acts as a tailored intermediary that communicates with internal systems and returns optimized responses for its client.



This isolates frontend-specific logic in the BFF layer, decoupling the frontend from backend complexity and enabling faster, independent development of each client interface.

Benefits

- Each frontend receives only the data and structure it needs, **improving responsiveness and reducing bandwidth usage**, especially on constrained devices.
 - Performance improves as **BFFs can aggregate and transform data on the server side**, reducing frontend complexity and round trips.
 - **Frontends are simplified** by offloading formatting, orchestration, and anti-corruption to the BFF.
 - **Teams gain autonomy**, as frontend and BFF can evolve together without needing cross-team coordination.
 - **Error handling and security can be customized per client**, allowing BFFs to gracefully handle backend failures and enforce frontend-specific access control.
 - **Technology flexibility** allows each BFF to be implemented using the most suitable language or stack for its use case.
-

Tradeoffs

- **Each BFF introduces another service** to maintain, monitor, and deploy, increasing operational overhead.
- **There's a risk of duplicated logic across BFFs**, especially if clients share similar requirements.
- **More components mean more potential points of failure**; BFFs need resilience mechanisms like timeouts and circuit breakers.
- **An additional network hop may introduce slight latency** if not mitigated properly.
- **Version management and deployment coordination** between BFF and frontend clients can become complex without robust CI/CD practices.
- If overused or applied to uniform clients, **BFFs can introduce unnecessary complexity and maintenance burden**.



When to Use

- You support **multiple frontend types** (e.g. mobile, web, desktop) that need different data structures or interaction flows.
 - **Your frontends require complex data aggregation or transformation** that would be inefficient or repetitive on the client side.
 - **Different clients have different performance constraints or device limitations**, requiring customized optimizations.
 - **Frontend teams want independence from centralized backend changes** and need to move at their own release cadence.
 - **You're integrating third-party clients or partners** who need restricted, controlled access to backend systems.
 - Client-facing applications and internal admin tools need **different data scopes, logic, and access controls**.
-



Real-World Examples

A smart home system includes mobile apps, a web portal, and connected devices like thermostats and doorbell cameras. The Mobile BFF returns trimmed, compressed payloads and pushes live status via WebSockets/SSE; the Web BFF focuses on historical queries, pagination, and configuration flows. Devices interact through an IoT gateway/broker (e.g. MQTT); the BFF calls a device-control service/IoT API that publishes commands and aggregates state. Each BFF tailors APIs and resilience (timeouts/fallbacks) to its channel's constraints and interaction model.

Command Query Responsibility Segregation (CQRS)

 Separates read and write responsibilities into distinct models to scale independently and simplify complex domains.

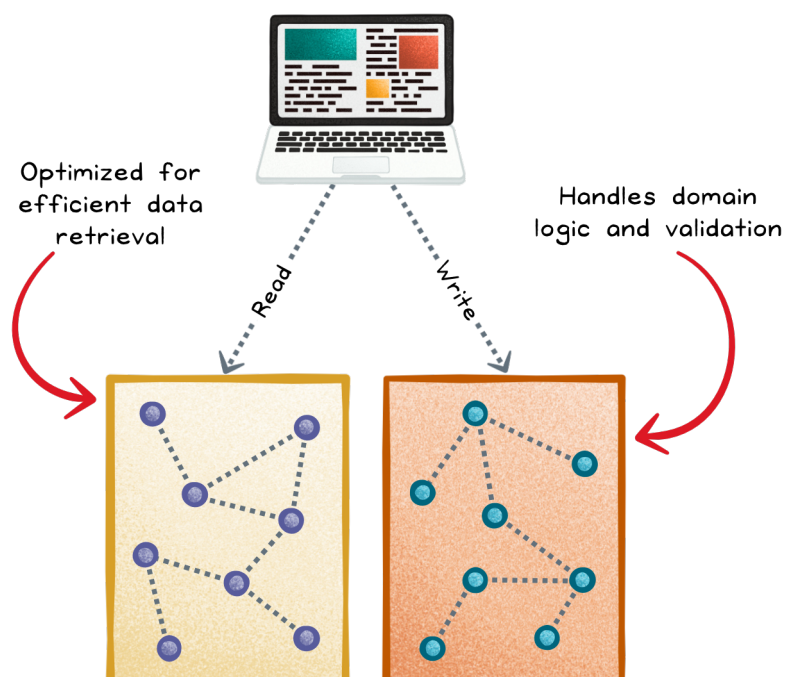
Benefits	Tradeoffs	When To Use
• Read and write workloads can scale independently • Data models are optimized for their purpose • Business logic is isolated • Enables low-latency responses • Improved system resilience • Flexible read evolution	• Two models must be built and maintained • Read models are eventually consistent • Synchronization between models introduces challenges • Increased infrastructure demand • Requires experience in distributed systems and eventual consistency	• Complex business rules or domain logic • Reads vastly outnumber writes • Event-driven or uses event sourcing • Multiple specialized read models are needed • Independent teams own commands and queries



How It Works

CQRS splits a system's write operations (commands) from read operations (queries) using separate models or components. The write model handles domain logic and validation, updating the authoritative state of the system.

The read model is a projection/materialized view optimized for efficient retrieval, typically updated asynchronously from the write model via events or replication.



This separation allows read and write concerns to be scaled, designed, and evolved independently, usually at the cost of eventual consistency and added complexity. CQRS is often combined with event sourcing but does not require it.

Benefits

- **Read and write workloads can be scaled independently**, improving system performance and resilience under load.
 - Developers can **optimize data models separately** for commands (normalized, transactional) and queries (denormalized, fast-access).
 - **Business logic is isolated** to the write side, while the read side stays simple and focused on **serving data efficiently**.
 - Precomputed views in the read model enable **low-latency responses and faster user experiences**.
 - **System resilience improves**, as failures on one side (e.g. read database outage) don't necessarily impact the other.
 - Enables **flexibility** in evolving read models (e.g. for new reports or screens) without touching core business logic.
-

Tradeoffs

- **Teams must build and maintain two models**, introducing significant design and operational complexity.
- Read models are **eventually consistent**, so clients may see stale data immediately after a write.
- **Synchronization between models** (via messaging or events) introduces challenges like message ordering, retries, and idempotency.
- **Infrastructure demands increase**, with potential multiple databases and messaging systems required.
- Requires a mature team with **experience in distributed systems and eventual consistency handling**.



When to Use

- Your system has **complex business rules or domain logic** that benefit from being modeled cleanly on the write side.
 - **Read operations vastly outnumber writes**, and you need to optimize for performance or latency.
 - Your architecture is **event-driven or uses event sourcing**, and you want to build **multiple specialized read models**.
 - **Independent teams are responsible for command and query concerns**, and you want to reduce cross-team contention.
-



Real-World Examples

In collaborative document editing tools, edits, comments, and permission changes are issued as commands against the write model, which validates rules and persists changes. The read model projects these changes into denormalized views for fast rendering and search, updating asynchronously from events. This supports high-concurrency collaboration with low-latency reads without locking the whole document, while the write side preserves correctness (e.g. via OT/CRDT conflict resolution).

That's a wrap on the

Architecture Patterns Playbook

For more system design insights and visual explainers,
join 400,000+ engineers already following us:

👉 Follow **Nikki Siapno** on **LinkedIn** (turn on notifications) 🔔

👉 Follow **Nikki Siapno** on **X/Twitter** (turn on notifications) 🔔

Stay sharp. Keep leveling up! 🚀

