1. Monolith Architecture

Introduction

Monolithic architecture is the **traditional approach** to building software applications, where all components—user interface, business logic, data access, and background jobs—are combined into a **single, unified codebase** that is built, deployed, and scaled as one unit. Think of it like a self-contained house where the kitchen, bedrooms, living room, and utilities are all under one roof and share the same foundation.

This pattern isn't new—it's how software has been built since the beginning of computing. The term "monolith" gained prominence not because the pattern was created, but because developers needed a name for it when contrasting it with newer distributed approaches like microservices.

The word "monolith" often carries negative connotations today, but this is misleading. **A well-designed monolith is often the best choice** for many applications. The problems people associate with monoliths usually stem from poor code organization, not the monolithic deployment model itself.

References

1. Newman, S. (2019). *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith.* O’Reilly Media.
2. Newman, S. (2021). *Building Microservices: Designing Fine-Grained Systems* (2nd ed.). O’Reilly Media.
3. Vernon, V., & Jaskuła, T. (2021). *Strategic Monoliths and Microservices: Driving Innovation Using Purposeful Architecture.* Addison-Wesley Professional.

Explanation

Core Concept

A monolith is characterized by its **deployment model**, not its internal structure. The key defining features are:

1. **Single Codebase**: All code lives in one repository
2. **Single Build Process**: One compilation/build creates the entire application
3. **Single Deployment Unit**: You deploy everything together as one artifact
4. **Shared Memory Space**: Components communicate through direct method calls, not network requests
5. **Single Database** (typically): All components share the same database instance

Not all monoliths are created equal. The internal organization can vary significantly:

1. Unstructured Monolith (Big Ball of Mud)

- No clear architectural pattern
- Components tightly coupled with tangled dependencies
- Code mixed together without clear boundaries
- **This is the "bad monolith"** people warn about
- Often the result of technical debt accumulation

2. Layered Monolith

- Uses horizontal layers (presentation, business, data)
- Clear separation of concerns by technical function
- Still deployed as one unit
- Better than unstructured, but can still have coupling issues
- This is what we covered in the layered architecture explanation

3. Modular Monolith

- Organized around **business capabilities** or **domains**
- Each module is self-contained with its own logic and data
- Modules communicate through well-defined interfaces
- **This is the "good monolith"** that combines monolithic deployment with microservice-like modularity
- Often the sweet spot for many applications

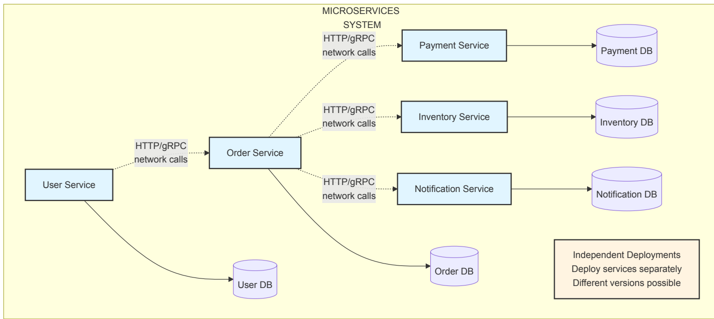
4. Modular Monolith with Internal Architecture

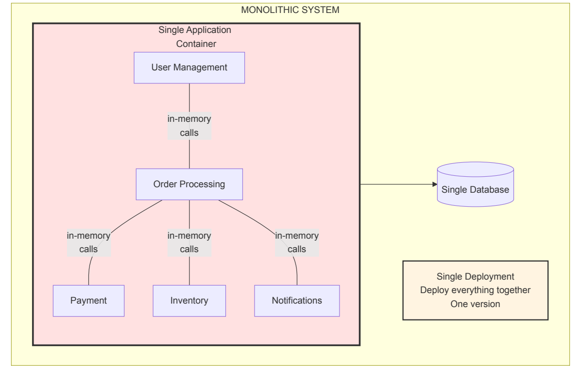
- Each functionality internally uses layered, hexagonal, or other patterns

- Best of all worlds: modular organization, clean architecture, single deployment
- Most complex to build but easiest to maintain and potentially migrate later

---

Monolith vs Microservices





How a Monolith Works: Request Flow Example

User | Load Balancer | Monolith Application | Order Module | Payment Module | Inventory Module | Notification Module | Database | Redis Cache

POST /orders

Route to instance

createOrder()

In-memory call
(no network overhead)

BEGIN TRANSACTION

INSERT INTO orders

checkInventory()

Direct method call
(same process)

SELECT FROM inventory

Stock available

reserveItems()

UPDATE inventory

Items reserved

processPayment()

Direct method call
(same process)

INSERT INTO payments

Payment successful

UPDATE orders SET status

COMMIT TRANSACTION

sendOrderConfirmation()

Async background job
(still same application)

Queue email job

Order created

201 Created

Order confirmation

Background worker
picks up job later

Dequeue email job

Send email via SMTP

User | Load Balancer | Monolith Application | Order Module | Payment Module | Inventory Module | Notification Module | Database | Redis Cache

Advantages

Simplicity and Ease of Development

- One codebase to understand and navigate
- No need to manage multiple repositories or services
- Single IDE project—everything is visible
- Straightforward debugging with complete stack traces
- New developers can understand the entire system

Strong Data Consistency

- ACID transactions across all operations
- No distributed transaction complexity
- Database guarantees data integrity
- Easy to maintain referential integrity
- No eventual consistency headaches

Performance Efficiency

- In-memory method calls (nanoseconds vs milliseconds for network calls)
- No serialization/deserialization overhead
- Shared database connection pool
- No network latency between components
- Lower resource usage overall

Simpler Deployment

- Single artifact to build and deploy
- One deployment pipeline
- No complex orchestration (Kubernetes, service mesh)
- Easier rollback—just redeploy previous version
- No partial deployment failures

Easier Testing

- End-to-end tests run against one application
- No need to mock network calls between services
- Integration testing is straightforward
- Complete system runs on developer laptop
- Faster test execution

Lower Operational Complexity

- One application to monitor
- Single log aggregation point
- Simpler infrastructure requirements
- One technology stack to maintain
- Fewer moving parts to break

Cost Effective

- Lower infrastructure costs (fewer servers, less network traffic)
- Smaller team can manage it
- Less specialized DevOps knowledge needed
- Simpler hosting (can run on a single server initially)
- No need for advanced orchestration tools

Easier Refactoring

- IDE refactoring tools work across entire codebase
- Find all usages immediately
- Safe renames and moves
- Compile-time safety ensures changes don't break things
- No coordination needed between teams for breaking changes

Better for Small to Medium Teams

- One team can own the entire application
- No communication overhead between service teams
- Shared context and knowledge
- Faster feature development (no cross-service coordination)

Disadvantages

Scaling Limitations

- Must scale the entire application, not individual components
- Can't scale CPU-intensive parts independently from memory-intensive parts
- Resource waste—scaling for one bottleneck scales everything
- All instances are identical (no specialization)

Long Build and Deployment Times

- Large codebase takes longer to compile
- Complete test suite must run before each deployment
- Deployment affects entire application
- Small changes require full rebuild
- CI/CD pipelines become slower as application grows

Technology Lock-in

- Entire application uses same technology stack
- Hard to introduce new languages or frameworks
- Stuck with initial technology choices
- Legacy code must be updated with framework upgrades
- Innovation is constrained

Team Coordination Challenges

- As teams grow, coordination becomes necessary
- Multiple teams working on same codebase causes conflicts
- Merge conflicts increase with team size
- Harder to establish clear ownership
- Release coordination becomes complex

Deployment Risk

- One bug can bring down entire application
- No isolated failures—everything fails together
- Deployment affects all users simultaneously
- Higher risk with each release
- Harder to do gradual rollouts or canary deployments

Tight Coupling Risk

- Without discipline, modules become tightly coupled

- Shared database makes boundaries fuzzy
- Easy to create unintended dependencies
- Technical debt accumulates faster
- Harder to extract parts later

Limited Fault Isolation

- Memory leak in one module affects entire application
- CPU spike in one feature slows down everything
- No bulkheads to contain failures
- One bad deployment takes everything down

Onboarding Complexity (for large monoliths)

- New developers face huge codebase
- Understanding the entire system takes time
- Hard to find specific functionality
- Risk of unintended side effects when making changes

Database Bottleneck

- Single database becomes the scaling bottleneck
- Hard to optimize for different access patterns
- Can't use different databases for different needs
- Database schema changes affect entire application

Slower Innovation

- Harder to experiment with new technologies
- Longer feedback loops
- Risk-averse culture develops (fear of breaking things)
- Difficult to rewrite parts incrementally

Architecture Health Indicators

Healthy architecture

- Feature velocity remains high
- Deployment confidence is strong
- Team collaboration is smooth
- Performance meets requirements
- Bugs are easy to find and fix

Warning signs

- Deployment taking > 1 hour
- Frequent production incidents
- Hard to understand what code does
- Developers afraid to make changes
- Teams constantly blocking each other
- Unable to scale to meet demand

When to Switch to Microservices

When to migrate:

- Team size > 15-20 developers working on same codebase
- Clear, stable domain boundaries identified
- Need to scale different components independently
- Different parts have different availability requirements
- Want to use different technology stacks

Approach

- Migrate only critical parts
- Leave as much logic as possible in the monolith

Common hybrid pattern

```
Core Monolith + Specialized Services:

[Main Monolith]
    ├── User Management
    ├── Orders
    ├── Products
    └── Admin

[Extracted Services]
    ├── Image Processing Service (CPU-intensive)
    ├── Search Service (Elasticsearch, different tech)
    ├── Recommendation Engine (Python/ML)
    └── Real-time Chat (WebSocket, needs high availability)
```

Migration Anti-Patterns

Big Bang Migration

- "Let's rewrite everything as microservices!"
- Result: 2+ years, failed project, team exhaustion
- Better: Gradual extraction over time

Microservices Too Early

- Startup with 3 developers builds 12 microservices
- Result: Operational nightmare, slow development
- Better: Start with monolith, split when it hurts

Premature Optimization

- "We might need to scale this part"
- Extract before you have data showing it's necessary
- Result: Complexity without benefit
- Better: Wait for actual problems

Ignoring Data Migration

- Focus on code, forget about database
- Shared database defeats purpose of microservices
- Result: "Distributed monolith" (worst of both worlds)
- Better: Plan database separation from start

No Clear Boundaries

- Split without understanding domain
- Services depend on each other heavily
- Result: Microservices that can't be deployed independently
- Better: Ensure loose coupling before splitting

When Monolith is Enough

Some applications should **never** leave the monolith:

- **Internal tools** (low traffic, small team)
- **Simple CRUD apps** (admin panels, forms)
- **Mature products** (no growth, stable features)
- **Small SaaS** (< 100k users, small team)
- **E-commerce stores** (until huge scale)

Migration Decission Flow