Architectural Styles

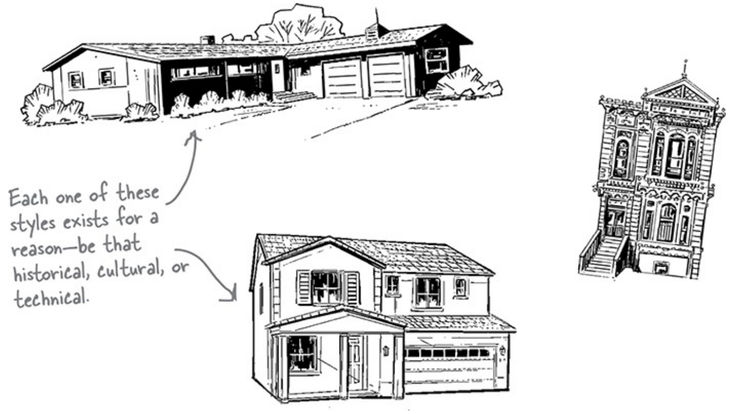
COMP2511, CSE, UNSW

These lecture slides are from the book “*Head First Software Architecture*”,

by Raju Gandhi, Mark Richards, Neal Ford, O'Reilly Media, Inc., March 2024

Introduction to Architectural Styles

- **Architectural Styles**:
  - Predefined patterns and philosophies guiding how software systems are structured and deployed.
- Importance of Understanding Styles:
  - Facilitates better design decisions.
  - Aligns software architecture with project needs.
- **Example**:
  - Residential housing styles influenced by geography, climate, personal preference. Similarly, software architecture varies by project requirements.



Categorizing Architectural Styles

Two main categories for architectural styles:

1. Partitioning
   - Technical vs. Domain-based.

2. Deployment
   - Monolithic vs. Distributed.

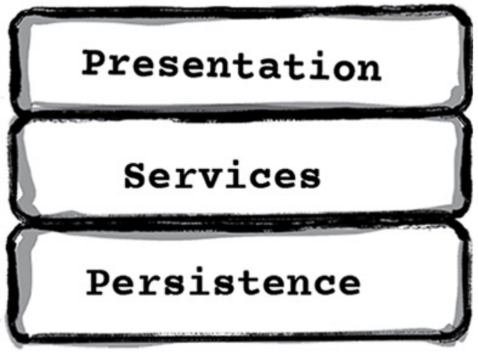
❖ Why Categorize?
   - Helps systematically analyse and select appropriate architecture.

Deployment model		Partitioning	
		Technical	Domain
	Monolith	Layered Microkernel	Modular monolith
	Distributed	Event-driven	Microservices

Partitioning by Technical Concerns

Technical Partitioning:

- Code organized by functional roles or technical layers.



Characteristics:

- Clear separation of responsibilities.
- Easier specialization of teams.

Example: A standard web application:

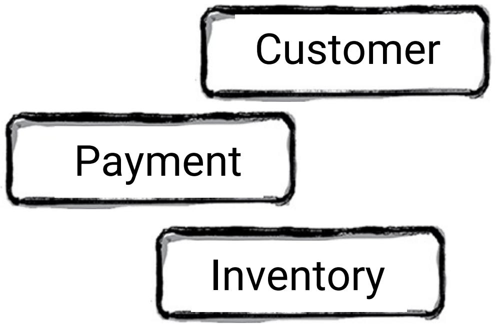
- Presentation Layer (UI);
- Business Logic Layer (Services)
- Data Persistence Layer (Database)

> Real-world Analogy:
> Roles in a fancy restaurant (host, server, chef, busser) clearly divided by technical concerns (greeting, cooking, cleaning).

Partitioning by Domain Concerns

Domain Partitioning:

- Code organized around business domains or problem areas.



Characteristics:

- Alignment with business goals.
- Easier maintenance of related features.
- Strong domain modeling.

Example: An e-commerce platform:

- Customer Domain (user accounts, user interface)
- Inventory Domain (product catalog, stock management)
- Payment Domain (billing, transactions)

> ➢ Real-world Analogy:
> Food court restaurants, each specialised in distinct cuisines (pizza, salads, burgers).

Comparing Technical vs. Domain Partitioning

Technical Partitioning	Domain Partitioning
Layered by technical roles	Organized by business areas
Easier for specialised teams	Aligned closely with business needs
Risk of over-generalisation	Risk of duplicating common functionalities

Example Scenario: A banking application:

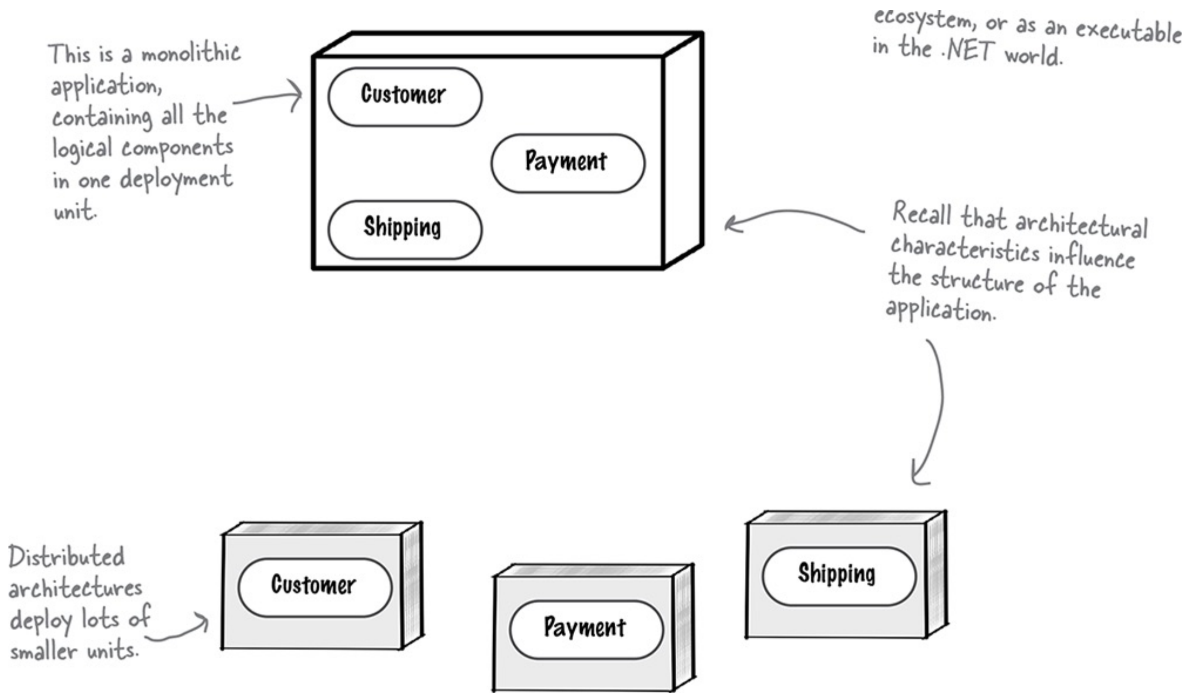
- ***Technical***: Separate teams for frontend, backend, DB administration.
- ***Domain***: Separate teams for loans, investments, account management.

Deployment Models Overview

1. Monolithic Architecture
   - Single deployable unit.

2. Distributed Architecture
   - Multiple deployable units communicating over networks.

Choice affects scalability, complexity, and cost.



Monolithic Architecture – Overview and Pros

Monolithic:

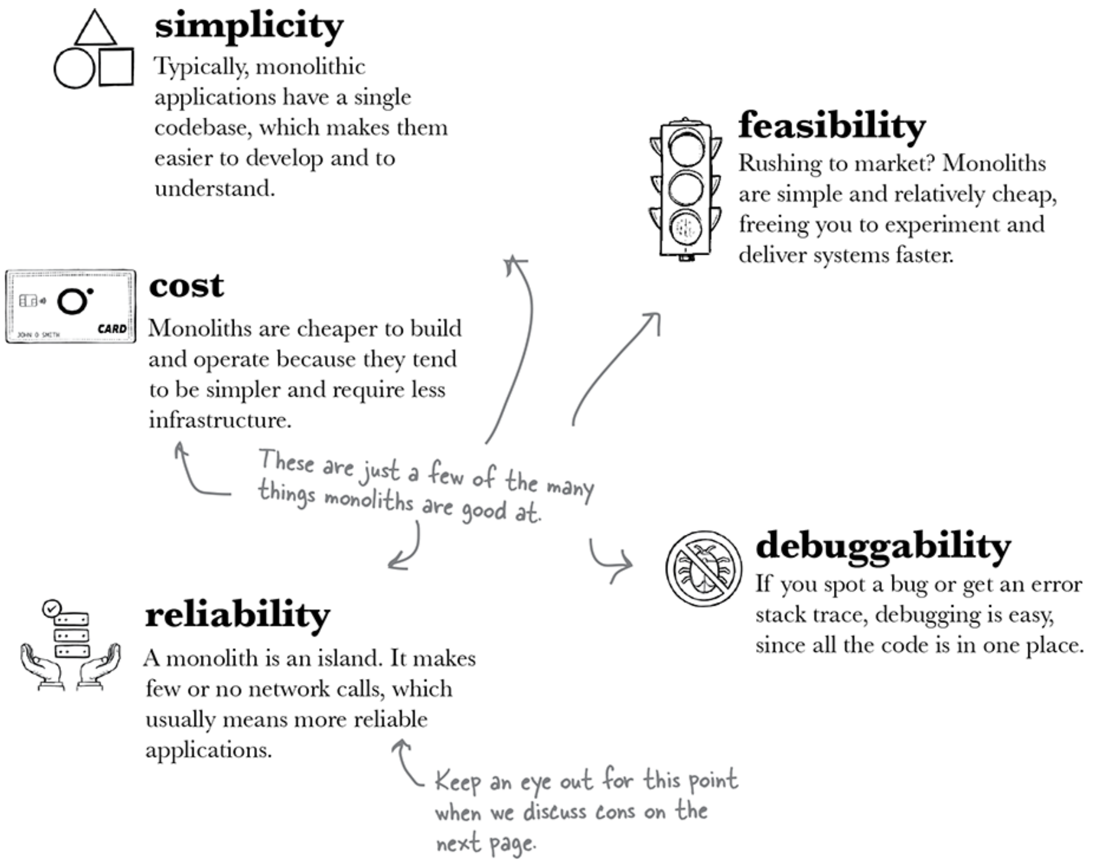
- Entire application deployed as one single executable or package.

Pros:

- Easier initial development.
- Simplified debugging.
- Lower initial deployment cost.

Examples:

- A single .jar (Java) or .exe (.NET) containing all app logic and resources.
- Smartphone as a single device doing many functions (calling, browsing, tracking).



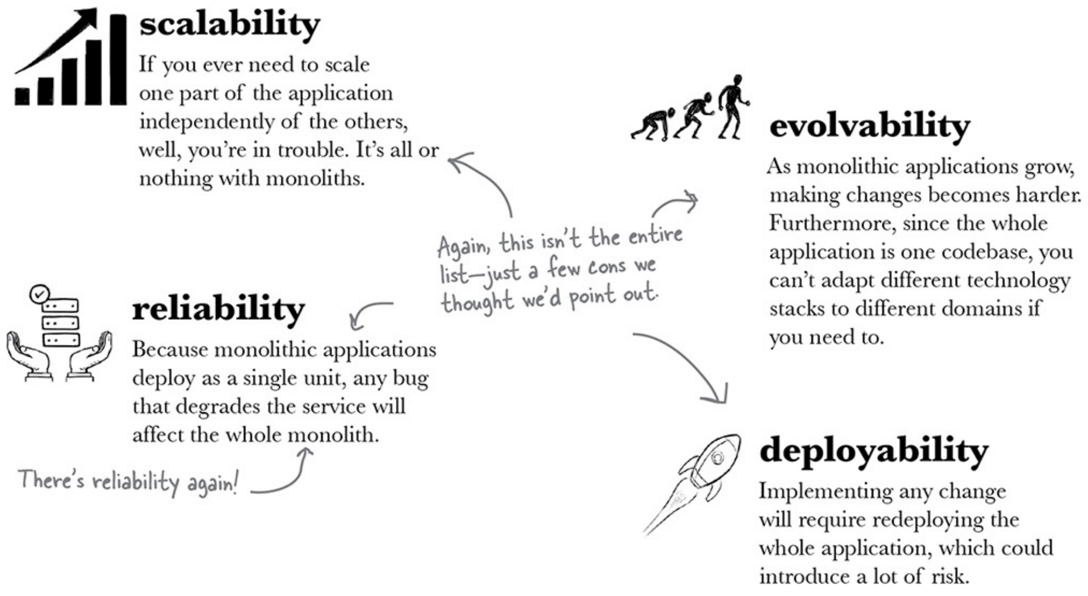
Monolithic Architecture - Limitations

Cons:

- Difficult to scale independently.
- Single bug can disrupt entire system.
- Inflexible when adapting to changing demands.

Example:

- Scaling a monolithic online store application
- Scaling means duplicating the entire application, increasing resource consumption significantly.



Distributed Architecture - Overview

Distributed:

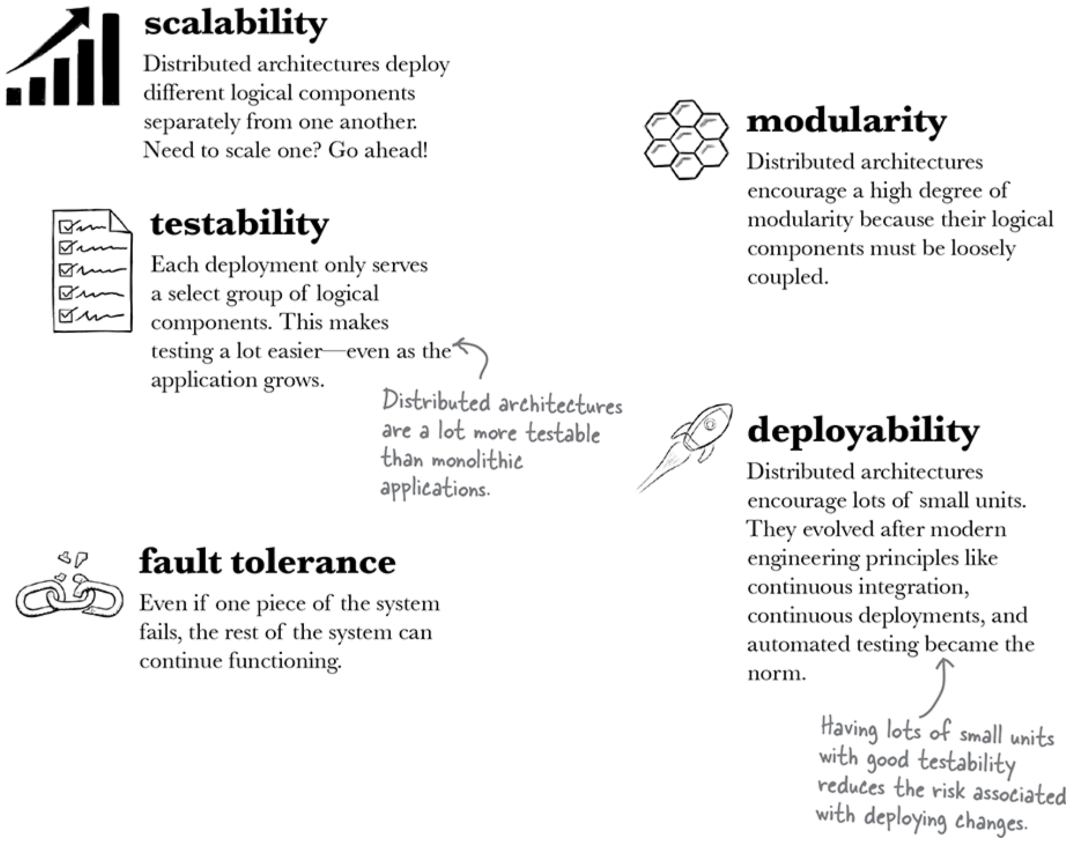
- Application components deployed separately, each as individual processes/services.

Pros:

- Independent scalability of components.
- Encourages modular design.
- Fault isolation—failures affect only single units.

Example:

- Microservices architecture for Netflix or Amazon, allowing independent scaling of services like user management, video streaming, and recommendation systems.



Distributed Architecture - Challenges

Cons:

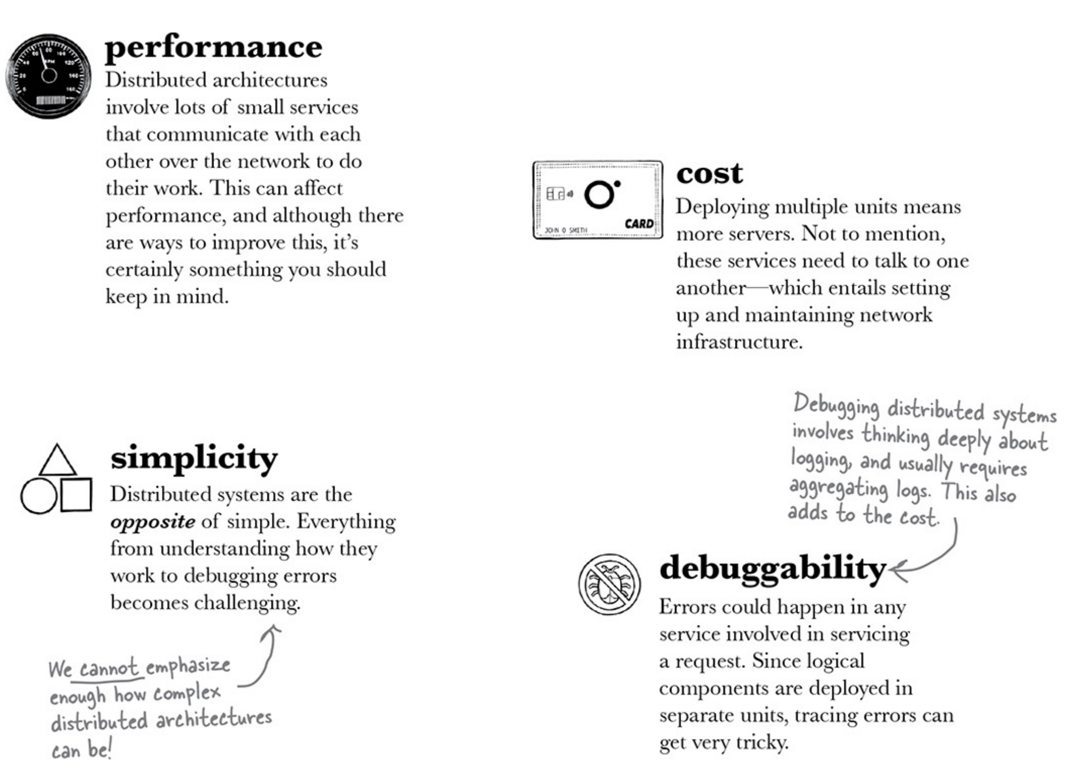
- High complexity due to network dependence.
- Increased maintenance and debugging complexity.
- Higher infrastructure and operational costs.

Example:

- Managing distributed transactions across services—complex coordination required, increased risk of partial failures.

Real-world Analogy:

- Earlier days—separate devices for GPS, web browsing, and phone calls each required separate maintenance and integration.



performance
Distributed architectures involve lots of small services that communicate with each other over the network to do their work. This can affect performance, and although there are ways to improve this, it's certainly something you should keep in mind.

cost
Deploying multiple units means more servers. Not to mention, these services need to talk to one another—which entails setting up and maintaining network infrastructure.

simplicity
Distributed systems are the ***opposite*** of simple. Everything from understanding how they work to debugging errors becomes challenging.

We cannot emphasize enough how complex distributed architectures can be!

Debugging distributed systems involves thinking deeply about logging, and usually requires aggregating logs. This also adds to the cost.

debuggability
Errors could happen in any service involved in servicing a request. Since logical components are deployed in separate units, tracing errors can get very tricky.

Introduction to Fallacies of Distributed Computing

- Originated at Sun Microsystems in 1994
- Common false assumptions about networks
- Crucial for architects of distributed systems
- 11 total fallacies (8 classical + 3 additional)

Fallacy #1 - The Network Is Reliable

- Reality: Networks can and do fail

- Impact: Services might be healthy but unreachable

- Mitigation:
  - Use timeouts
  - Retry policies

- Example: Service A sends request to Service B → no response due to intermittent network issue

Fallacy #2 - Latency Is Zero

- Reality: Remote calls take milliseconds, not microseconds
- Impact: Chained service calls can add significant delay
- Mitigation:
  - Monitor 95th-99th percentile latency
  - Minimise unnecessary calls
- Example: 10 chained calls with 100ms each = 1s delay

Fallacy #3 - Bandwidth Is Infinite

- Reality: Bandwidth is limited, especially under load
- Impact: Excessive inter-service communication slows the system
- Mitigation:
    - minimizing the passing of large, complex data structures
- Example: Returning 500KB when only 200B needed → 1Gbps load for 2k req/s

Fallacy #4 - The Network Is Secure

- Reality: More endpoints = higher attack surface

- Impact: Inter-service communication can be vulnerable

- Mitigation:
  - Zero-trust architecture
  - Secure each endpoint

- Example: Internal services hacked due to open port

Fallacy #5 - Topology Never Changes

- Reality: Network topology evolves frequently
- Impact: Latency assumptions break
- Mitigation:
  - Coordinate with network teams
  - Use adaptive timeout policies
- Example: Sunday network upgrade → production timeouts Monday

Fallacy #6 - There Is Only One Administrator

- Reality: Multiple admins across departments

- Impact: Miscommunication and missed changes

- Mitigation:
  - Maintain a clear contact directory
  - Standardize change coordination

- Example: Change in one subnet unknowingly affects dependent service

Fallacy #7 - Transport Cost Is Zero

- Reality: Infrastructure and routing costs add up

- Impact: Distributed systems are more expensive

- Mitigation:
  - Assess total cost of ownership (TCO)
  - Consider hybrid designs

- Example: Simple REST call needs new proxies, firewalls, gateway

Fallacy #8 - The Network Is Homogeneous

- Reality: Different vendors, firmware, configurations

- Impact: Compatibility and packet loss

- Mitigation:
  - Test network assumptions regularly
  - Avoid hard dependencies on vendor features

- Example: Packet loss between Cisco and Juniper segments

Fallacy #9 - Versioning Is Easy

- Reality: Supporting multiple versions is hard
- Impact: Contract proliferation, test complexity
- Mitigation:
  - Limit concurrent versions
  - Use deprecation plans
- Example: Team supports 7 versions of same API endpoint

Fallacy #10 - Compensating Updates Always Work

- Reality: Rollbacks can fail too
- Impact: Data inconsistency
- Mitigation:
  - Design for idempotency
  - Include recovery mechanisms
- Example: Order placed, and rollback fails → duplicated state

Fallacy #11 - Observability Is Optional

- Reality: Without observability, debugging is impossible

- Impact: Silent failures across services

- Mitigation:
  - Centralized logging
  - Distributed tracing
    - E.g., *OpenTelemetry*: open-source framework for collecting, processing, and exporting telemetry data (traces, metrics, and logs) from cloud-native applications and infrastructure.

- Example: Request times out without any log trail

Fallacy - Summary and Implications

- Fallacies reveal key weaknesses in distributed systems
- Addressing them improves resilience and clarity
- Must be communicated to development and operations teams
- Good architecture anticipates and mitigates these assumptions

Comparing Monolithic vs. Distributed

Monolithic	Distributed
Simpler development & debugging	Complex system integration
Lower initial costs	Higher upfront infrastructure cost
Scaling is all-or-nothing	Individual services scalable
Single failure disrupts whole system	Fault tolerance through isolation

Discussion - Regulatory and Compliance Needs

Consider special needs like:

- Regulatory compliance (e.g., financial industry).
- Security requirements.

Monolithic:

- Easier control and monitoring in regulated environments.

Distributed:

- Can complicate compliance but increases modularity and maintainability.

Example:

- Banking systems might use monolithic for core banking due to tight regulatory controls, however distributed services for customer engagement modules.

Key Takeaways

- Numerous architectural styles exist; each with unique characteristics and trade-offs.
- Partitioning styles: Technical vs. Domain.
- Deployment models: Monolithic vs. Distributed.
- Choice of style influenced by:
  - Project goals.
  - Scalability requirements.
  - Complexity management.
  - Cost implications.