

Architectural Styles

COMP2511, CSE, UNSW



UNSW
SYDNEY

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:

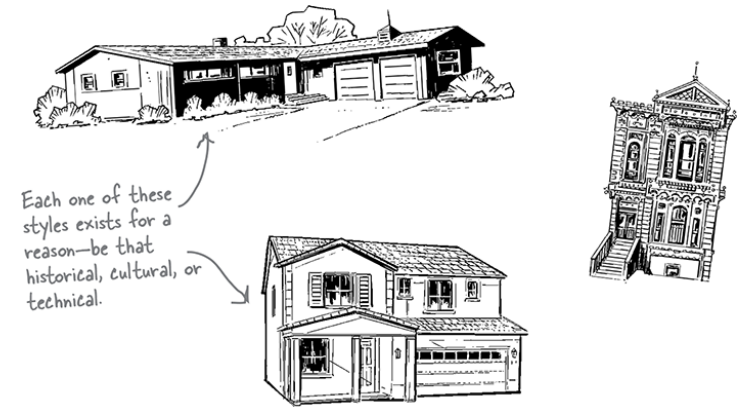
- Predefined **patterns** and philosophies guiding how software systems are structured and deployed.

❖ 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.

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

Partitioning by Technical Concerns

Technical Partitioning:

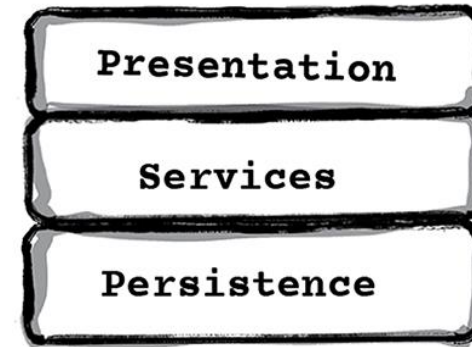
- Code organized by functional roles or technical layers.

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:

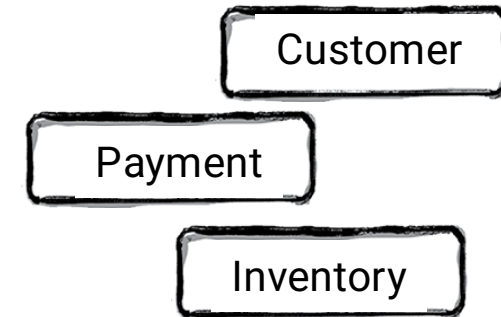
- Code organized around business domains or problem areas.

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

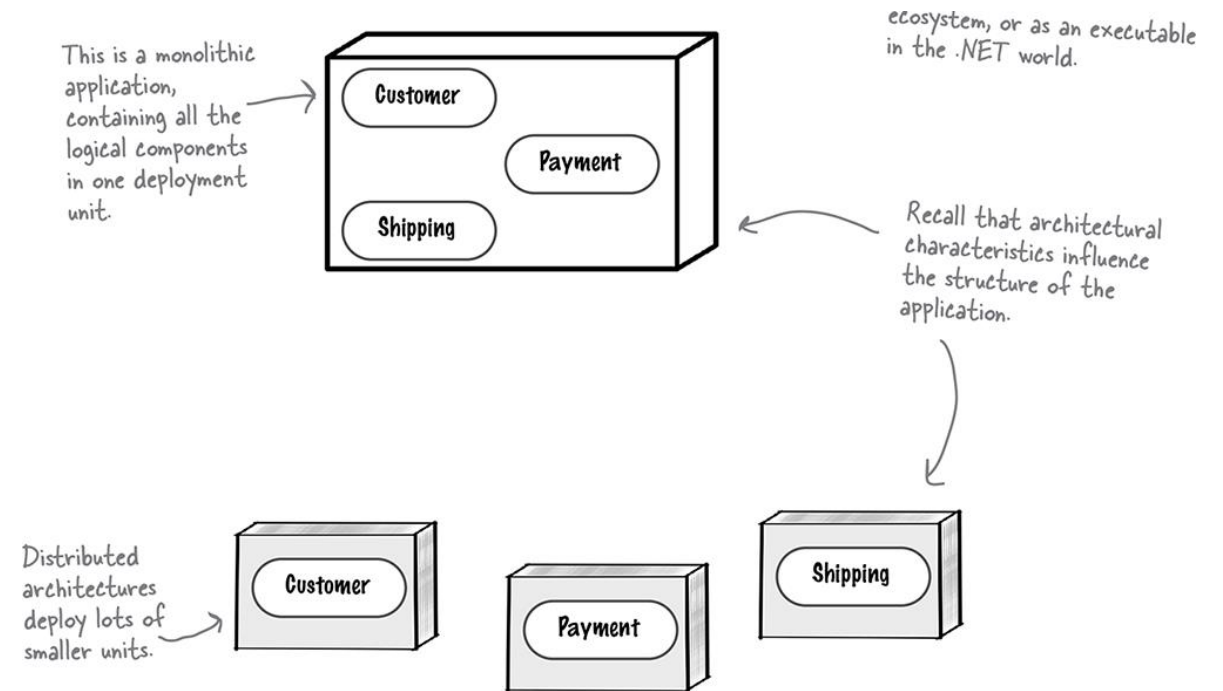
1. Monolithic Architecture

- 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).



simplicity

Typically, monolithic applications have a single codebase, which makes them easier to develop and to understand.



cost

Monoliths are cheaper to build and operate because they tend to be simpler and require less infrastructure.



feasibility

Rushing to market? Monoliths are simple and relatively cheap, freeing you to experiment and deliver systems faster.



reliability

A monolith is an island. It makes few or no network calls, which usually means more reliable applications.



debuggability

If you spot a bug or get an error stack trace, debugging is easy, since all the code is in one place.

These are just a few of the many things monoliths are good at.

Keep an eye out for this point when we discuss cons on the next page.

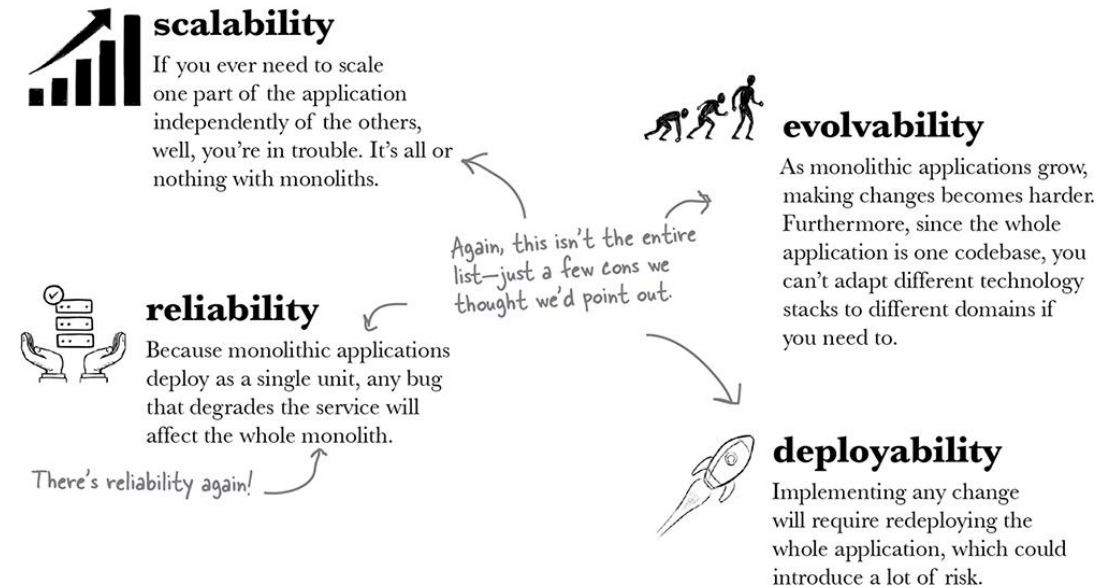
Monolithic Architecture - Limitations

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.

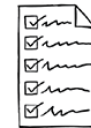
Example:

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



scalability

Distributed architectures deploy different logical components separately from one another. Need to scale one? Go ahead!



testability

Each deployment only serves a select group of logical components. This makes testing a lot easier—even as the application grows.

Distributed architectures are a lot more testable than monolithic applications.



fault tolerance

Even if one piece of the system fails, the rest of the system can continue functioning.



modularity

Distributed architectures encourage a high degree of modularity because their logical components must be loosely coupled.



deployability

Distributed architectures encourage lots of small units. They evolved after modern engineering principles like continuous integration, continuous deployments, and automated testing became the norm.

Having lots of small units with good testability reduces the risk associated with deploying changes.

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.

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.