COMP2511

WEEK 9

Would you rather have 4 or 7 eyes?

ADMIN STUFF

- Assignment-ii due next week Wednesday 3pm
  - Make sure your merge requests are linked in your blogs under relevant tasks!
- Week 10 Sample Exam (in-person labs ONLY)

AGENDA

- Architectural Characteristics
- Architectural Styles
  - Layered
  - Monolithic
  - Microservices
  - Event-Driven

Content of slides inspired by Esha Tripathi's (COMP2511 Tutor) Slides.

Architectural Characteristics

ARCHITECTURAL CHARACTERISTICS

“Image you’ve built a perfect online banking app. Every feature works flawlessly. BUT THEN when 100 users log in, the system crashes. Did you succeed?”

ARCHITECTURAL CHARACTERISTICS

“Image you’ve built a perfect online banking app. Every feature works flawlessly. BUT THEN when 100 users log in, the system crashes. Did you succeed?”

- Non-functional requirements (architectural characteristics) are just as important as functional ones.
- Success in building software isn’t just about getting the features to work.
- It’s about designing a system that **performs well** under expected conditions.

ARCHITECTURAL CHARACTERISTICS

“What makes software good?”

ARCHITECTURAL CHARACTERISTICS

“What makes software good?”

- *Can be maintained well*
- *Follows design principles*
- *Handles many users without crashing → Scalability*
- *Doesn’t crash, works every time → Reliability*
- *Good performance → Performance/Responsiveness*
- *Secure against vulnerabilities → Security*

ARCHITECTURAL CHARACTERISTICS

Architectural characteristics, also known as non-functional requirements, define **fundamental qualities** software architecture must support.

- Influence critical design decisions **structure, behaviour**, and **trade-offs** in software design.

- Impacts the system's **long-term success**, not just initial delivery.

ARCHITECTURAL CHARACTERISTICS

https://www.educaplay.com/learning-resources/26531280-cloud_system_qualities_match.html

Architectural Styles

ARCHITECTURAL STYLES

Architectural styles are predefined patterns and philosophies guiding how software systems are structured and deployed.

- Each style comes with **trade-offs** in scalability, performance, complexity, etc.
- Choosing the right style is a core part of software architecture. Shapes how the system **behaves** and evolves.
- Common architectural styles include:
  - **Layered** architectures (usually monolithic - deployed as a single unit)
  - **Modular monolithic** architectures (monolithic)
  - **Microservices** architectures (distributed, deployed as smaller services that are independent and communicate with each other)
  - **Event-driven** architectures (distributed)
- In practice, a *hybrid* of architectural styles may be used.

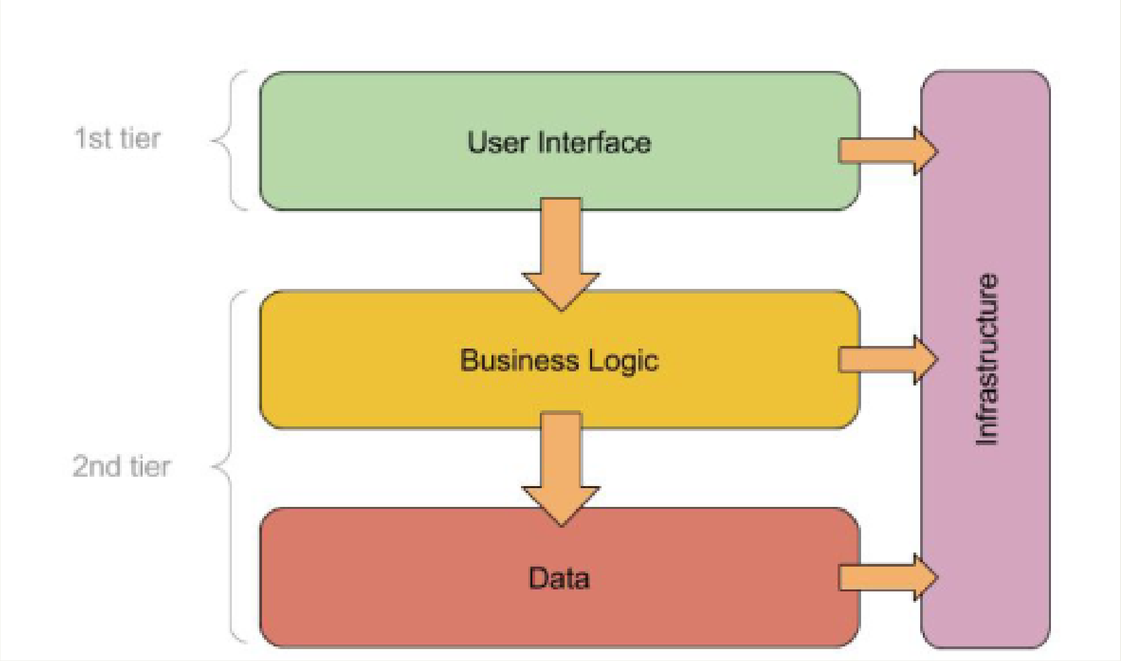
LAYERED ARCHITECTURE

A software design pattern that organises an application into distinct layers, each with a specific responsibility.

Domain logic spans multiple layers:

- Presentation (UI components)
- Workflow (business logic components)
- Persistence (database schemas and operations)

Easy to understand and lets you build simple systems fast.



LAYERED ARCHITECTURE

You’re building a Food Delivery App (like Uber Eats), your team is small, and your product is simple. Users can:

- login & signup
- browse restaurants
- place orders
- track delivery

You want to get this app up and running quickly. At this stage, you don’t know all the features your app will need, but you know you’ll need at least:

- UI
- Backend with business logic
- Database for persistence

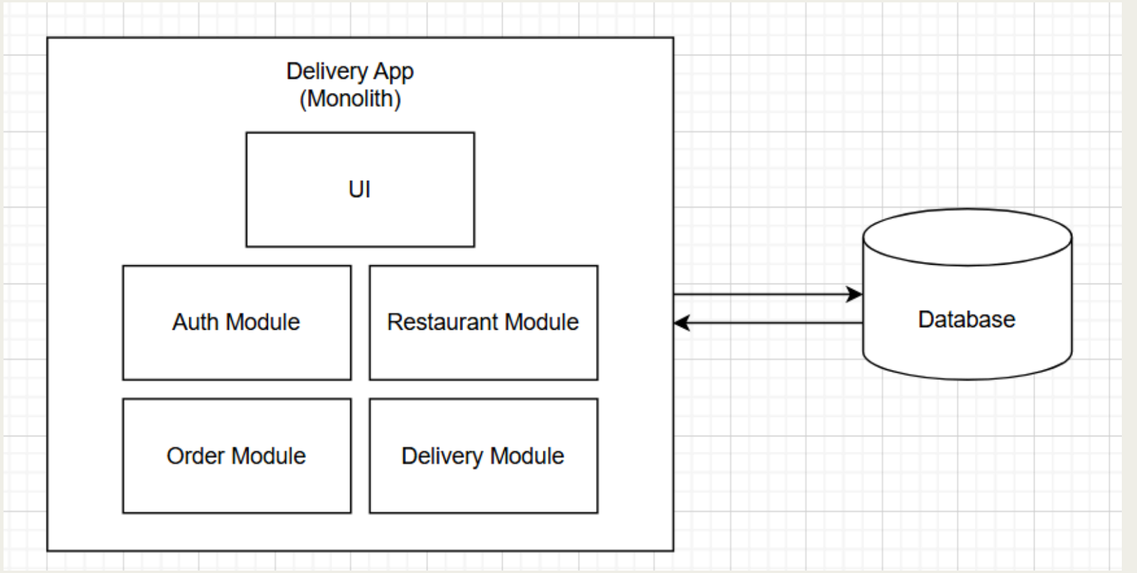
LAYERED ARCHITECTURE

Strengths	Challenges
• Feasibility: Quick and cost-effective • Simplicity: Clear structure and easy maintenance • Technical partitioning: Easier to **reuse shared components** like a common data access layer across teams • Data-intensive: Isolate all data logic in a **central database**, optimised for performance • Performance: more efficient when they **don't need to access data over the network**	• Deployability: As monoliths grow, even small updates require redeploying the entire app • Highly coupled • Scalability: Can't **scale individual features**, hit limits like memory or CPU because everything runs in one big unit • Elasticity: Single process **struggles with traffic spikes**, as it can't scale parts of the app independently • Testability: Larger and more connected the monolith, the **harder it is to test** reliably

MODULAR MONOLITHIC

All your code lives in one codebase, deployed as a single unit, communicate to one shared database and organised in domain-based modular structure.

- Each domain is represented as a module
- Different parts of the app talk to each other through direct function calls or public controller interfaces.
- In-process calls (e.g., function calls)
- Synchronous, tightly coupled
- Fast, simple



MODULAR MONOLITHIC

As you start to develop your food delivery app with more functionalities, you realise that each of your technical layers begin to bleed into each other.

You also notice that the system naturally separates into distinct functionalities: one for **restaurants** (managing menus and availability), one for **orders** (handling cart logic and payments), and one for **delivery** (tracking drivers and locations).

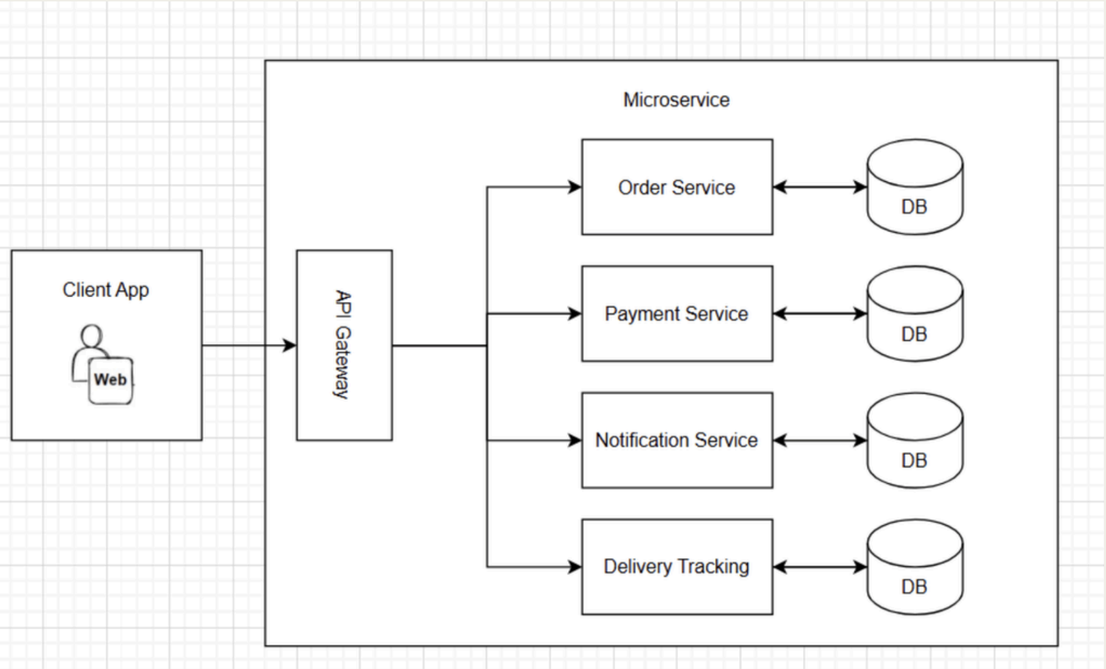
MODULAR MONOLITHIC

Strengths	Challenges
• Maintainability - Easier to maintain code when localised to each domain • Performance - As there are **no network calls** and all data processing happens in single place • Testability - Since scope of changes is **limited to one module teams** can build an entire testing suite for a domain • Deployability - Single unit, easier CI/CD • **Good for a small team** and simple requirements	• Hard to **scale** parts independently and modify as codebase grows • Hard to reuse logic across modules, cannot share common functionality without **high coupling** • Single set of architectural characteristics for the entire app, no per-module characterisation • **Fragile modularity**, easy to break boundaries • Slower development as team grows

MICROSERVICES

Microservices are single-purpose, independently deployed units. Ideal for environments requiring frequent changes and scalability.

- A microservice performs one specific function very well and communicates with other microservices over a network.
- Each service own their own data. Direct access to data is restricted to the owning microservice.
- Balance of granularity disintegrators (smaller services) and integrators (larger services).



MICROSERVICES

Over time, complexity grows:

- As the app grows, you add more features:
  - Real-time tracking
  - scheduled deliveries
  - payment gateway integration

Your monolithic app starts to feel fragile and hard to scale. So the team decides to split things up. You split features like **Payments, Orders, Tracking**, and **Notifications** into independent services, each owned by a dedicated team.

MICROSERVICES

Strengths	Challenges
• Maintainability - each micro-service is **single purpose** and separately deployed so easier to find and fix code • Testability - each micro-service testing **scope is smaller** than larger monolith • Deployability - **separate** and faster deployment • Scalability - scale only functionalities you need to meet • Faster, parallel development • **Fault isolation** - if the payment service crashes, the rest of the app can still work • Smaller, focused codebases	• More complex to set up and in workflow management • Harder to debug: tracing an issue across services is tricky • Communication costs: **services talk over the network** → slower and more error-prone than in-process calls. • Databases separated by microservces, not suitable for data **cannot be broken apart**

Case Study: How Requirements Shape Architecture

- Architectural requirements drive changes all the time. GitLab and Atlassian (Confluence) both started as monoliths, but their goals were different.
- This led them down different paths.
- We use this case study as examples to the following questions:
  - Why might a team choose a Modular Monolith as an intermediate step before adopting Microservices?
  - Describe how communication differs between components in Monolithic and Microservices architectures.

GitLab: Modular Monolith	Atlassian: Decomposing Confluence Cloud
GitLab engineers found that their large Rails monolith slowed development and created tight coupling between teams. Instead of moving immediately to microservices, they implemented a **modular monolith**.	Atlassian rearchitected Confluence Cloud to address global scale and reliability requirements. They gradually decomposed the monolith into **independent services**, allowing teams to deploy and scale autonomously.
Key drivers: • Improve development velocity and predictability • Reduce coupling and improve maintainability • Allow teams to work independently	**Key drivers:** • Handle multi-tenant workloads globally • Improve reliability and uptime • Enable faster delivery through team autonomy
Architectural characteristics: Maintainability, Modifiability, Deployability	**Architectural characteristics:** Scalability, Availability, Reliability, Team Autonomy
1. Keeps one app but organizes code into clear modules 2. Reduces coupling, adds useful abstractions 3. Lets parts of the app be deployed separately if needed	1. Made the app stateless and multi-tenant for global scale 2. Gradually split into microservices for resilience & team autonomy 3. Routing layer allows smooth migration without breaking anything
Microservices might not *always* be the goal. A modular monolith can deliver high maintainability and team productivity when designed carefully.	Atlassian moved incrementally, balancing refactoring with continuous delivery. This is a good example of evolutionary architecture in practice.

Sources: Gitlab's ADR, Atlassian's Engineering Blog

MICROSERVICES

Where might Microservices be a poor choice?

If we had started with microservices in the early stage (Small team, simple requirements), that would’ve been a mistake:

- Adds **unnecessary complexity** (tooling, ops, communication)
- **Higher operational cost**: infra, monitoring, deployment
- **Slower development** at first due to over-engineering
- Monolith is **simpler, cheaper, easier** to manage for such cases

→ Microservices are great for scale, but not for MVPs or small teams

MICROSERVICES

Why might a team choose a Modular Monolith as an intermediate step before adopting Microservices?

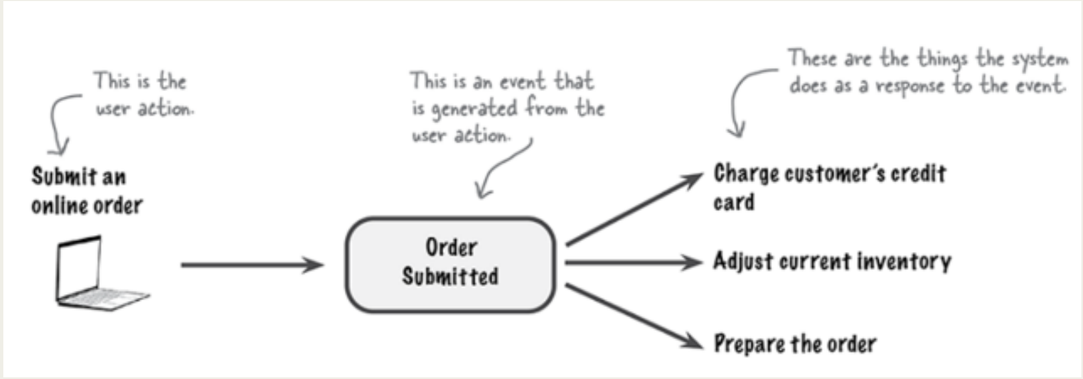
So instead, many teams go for a Modular Monolithic: a structures monolithic codebase with clear boundaries between modules like orders, payments, and users

- It allows internal separation of concerns, easier to refactor, and makes future migration to microservices much smoother since modules map cleanly to future services
- It is less complex: single deployment, fewer moving parts
- Skill growth, lets teams adopt microservice thinking gradually
- Cost effective, avoids early microservice overhead

EVENT-DRIVEN

Structures systems to respond to events, which are significant changes in a system state.

- Components in a system communicate faster by producing and responding to events asynchronously
- Unlike request-driven systems, EDA components don't directly call each other, they emit events and other **listen** for them
  - Event: something that has already occurred and carries data about it. Events are immutable and often used as **triggers**
- EDA can choose between monolithic, domain-partitioned or database-per-service databases



EVENT-DRIVEN

As our food delivery app grows and becomes more successful, we find that our app is too slow. We want to handle more and more users, without slowing down or crashing our app.

Additionally, we find that having multiple databases separated by each service is a bit too granular. We want to have a shared database within a given domain.

EVENT-DRIVEN

Strengths	Challenges
• Maintainability: Services in EDA are **highly decoupled** and therefore easier to maintain • Performance: Due to **asynchronous communication** and events processed in parallel • Scalability: Due to component decoupling, individual components **scale independently** • Evolvability: Easy to add functionality from derived events • Fault tolerance: If one service goes down, it won't bring down others in the workflow	• Highly complex using asynchronous messages and parallel event processing • Testability: Complex event debugging, errors not immediate. Elaborate testing sequences • Does not work with workflows that require synchronous, dependent components

Case Study: Netflix’s Notification Dilemma

- You’re an engineer at Netflix.
- Over 220 million members are doing everything at once: updating profiles, changing plans, adding shows to ‘My List,’ or streaming the latest hit.
- How do you ensure every device shows the right information in **real-time?**

Source: Rapid Event Notification System at Netflix

Context:

- Netflix needed to deliver real-time, cross-device updates (e.g. “Continue Watching”, “My List”, profile updates).
- They built RENO: a Rapid Event Notification System using **event-driven architecture** (EDA) to push and pull updates asynchronously to millions of devices.

Key drivers:

- Real-time responsiveness across devices
- High scalability under fluctuating traffic
- Consistent experience across platforms

Architectural characteristics:

Scalability, Responsiveness, Elasticity, Availability

How is event-driven architecture used?

- **Single event source**: All actions flow through a unified system (Manhattan).
- **Asynchronous & scalable**: Events go into priority-based queues and are processed by dedicated clusters.
- **Push + Pull model**: Ensures updates reach online and offline devices.
- **Targeted delivery**: Only notify devices that need it, reducing traffic.
- **Plug-and-play**: New events/features can be added quickly.
- **Reliable & observable**: Less coupling, bulk-headed delivery, and real-time monitoring keep the system healthy.

EVENT-DRIVEN

What are the potential risks of using an Event-Driven architecture in a mission-critical system?

- **Eventual consistency**: data takes time to sync → fine for notifications, but bad for instant consistency needs
- **Error handling**: hard to trace bugs → when something breaks, it’s tough to figure out where. Errors can quietly get lost.
- **Operational overhead**: Complex deployment & monitoring
- **Event Storms & Failures**: Overloaded queues, poison messages
- **Governance Gaps**: Schema mismatches, data duplication

EVENT-DRIVEN

Which architectural pattern would you choose for a system requiring strong consistency across components? Justify your choice.

Monolithic or Layered (single deployment). Because:

- Centralized DB = strong consistency
- Supports ACID transactions
- Simpler to manage integrity (e.g., financial systems)

Why not others:

- Microservices: need complex distributed coordination
- Event-Driven: async & eventual consistency by nature

ITS LAB TIME

YASSS