



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Chapter 06

SOFTWARE ARCHITECTURE — PART 1

Your Sprints Are Fast... But Can Your System Scale?

You've learned Agile and Scrum:

- Embrace change through short iteration cycles
- Deliver working software every 1-4 weeks
- Track work with GitHub Issues and Projects

Your Road Profile Viewer is working great! CI is green. Product Owner is happy.

Then the stakeholder asks:

"Can we deploy this to 1,000 concurrent users across three European offices?"

The Problem: Architecture vs. Agility

Suddenly, you realize:

- Your Dash app runs on a single laptop
- SQLite can't handle concurrent writes from multiple locations
- The entire application is one Python process
- There's no way to scale components independently

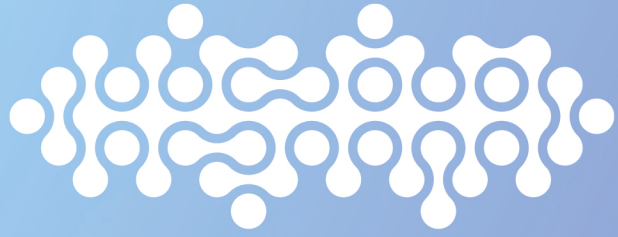
The Agile process is working. The architecture is not.

Real-World Example: Twitter's Fail Whale

Twitter 2008-2010: Monolithic Ruby on Rails application couldn't handle explosive user growth.

The "Fail Whale" became iconic — a symbol of what happens when architecture can't support success.

Lesson: Even successful products fail without scalable architecture.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Foundations of Software Architecture

1

What Is Software Architecture?

Software architecture describes how a system is organized as a set of communicating components and how those components behave and interact.

Architecture answers:

- How are responsibilities divided among components?
- How do components communicate with each other?
- How is the system deployed across hardware?
- What happens when one component fails?

The Bridge: Agile Meets Architecture

Agile tells us:

- *How to organize work*
- Short iteration cycles
- Respond to change

Architecture tells us:

- *How to organize the system*
- Component structure
- Communication patterns

Both must work together. Agile architecture *evolves* through iterative refinement — make enough decisions to start, then refine as you learn.

Learning Objectives

By the end of this lecture (Part 1), you will be able to:

1. **Name the motivation and goals** for architectural design — why architecture matters and what an architecture model enables
2. **Explain the dependency** between system structure and non-functional requirements — how performance, security, and maintainability shape choices
3. **Apply the 4+1 View Model** to describe a system from multiple perspectives — Logical, Process, Development, Physical, and Scenarios

Architecture as a Creative Process

Unlike algorithms with provably optimal solutions, architecture involves **trade-offs** that depend on:

- **System type:** Real-time embedded $\neq$ web application
- **Architect experience:** What has worked (and failed) in similar systems
- **Specific requirements:** Your project's unique constraints

*"There is no formulaic architectural design process."
— Sommerville, Software Engineering*

Think of architectural design as a series of decisions to be made.

Key Design Questions

1 Is there a generic architecture template?

2 How will it be distributed across hardware?

3 What architectural patterns might be used?

4 What is the fundamental structuring approach?

5 How will components be decomposed?

6 What strategy controls component operation?

7 What organization delivers NFRs best?

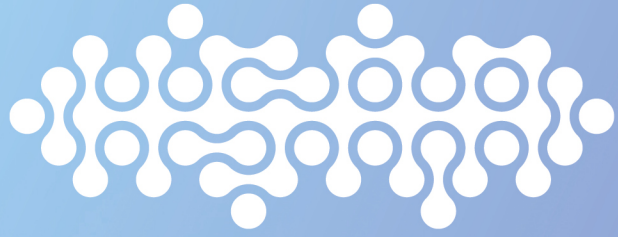
8 How should the architecture be documented?

Think-Pair-Share: Road Profile Viewer

Question	Road Profile Viewer Answer
Template?	Web dashboard applications
Distribution?	Currently single-machine
Patterns?	MVC for UI, layered architecture
Decomposition?	Split visualization, data access, business logic
Control?	Request-response for web, callbacks for Dash
NFR priority?	Maintainability over performance (student project)

Note: MVC and layered architecture patterns will be explained in detail in Part 2.

Discuss with your neighbor: What would change for 1,000 users?



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Non-Functional Requirements

2

Non-Functional Requirements Drive Architecture

Architectural decisions depend heavily on non-functional requirements, and conversely, influence the system's non-functional properties.

NFR	Architectural Implication
Performance	Localize critical operations; minimize network hops
Security	Layered architecture with innermost critical assets
Availability	Redundant components; enable hot swapping
Maintainability	Fine-grained, self-contained components

Performance-Driven Architecture

When performance is the primary concern:

Core Principles:

1. Localize critical operations
2. Minimize network hops
3. Co-locate data and computation
4. Use larger, monolithic components

Why?

- Network call: 1-100ms
- Local function call: nanoseconds
- Fewer components = less overhead

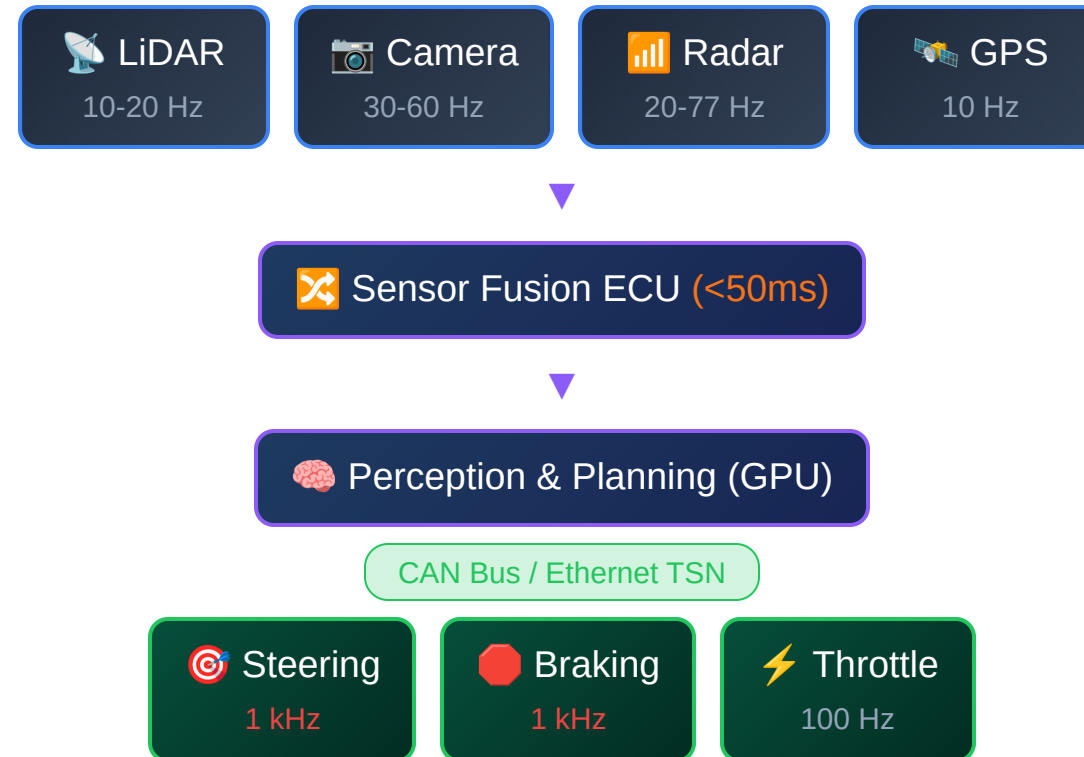
Trade-off: Performance-optimized architectures sacrifice modularity and testability.

Real-Time Systems Example

System Type	Timing Requirement	Architectural Implication
Hard real-time	Deadlines must <i>never</i> be missed	Predictable execution, no GC, dedicated hardware
Soft real-time	Occasional misses acceptable	Priority scheduling, bounded queues
Near real-time	Fast response (< 100ms)	Caching, async processing, load balancing

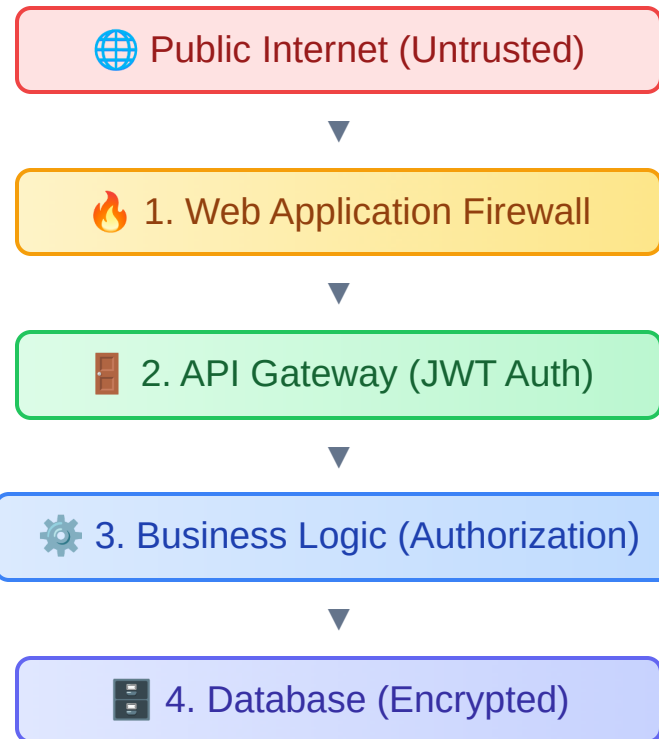
Example: Autonomous Vehicle — Braking must respond in < 1ms. HTTP microservices would be fatal!

Autonomous Vehicle Architecture



Security-Driven Architecture

Defense in Depth: Multiple layers of protection. An attacker must breach ALL layers.



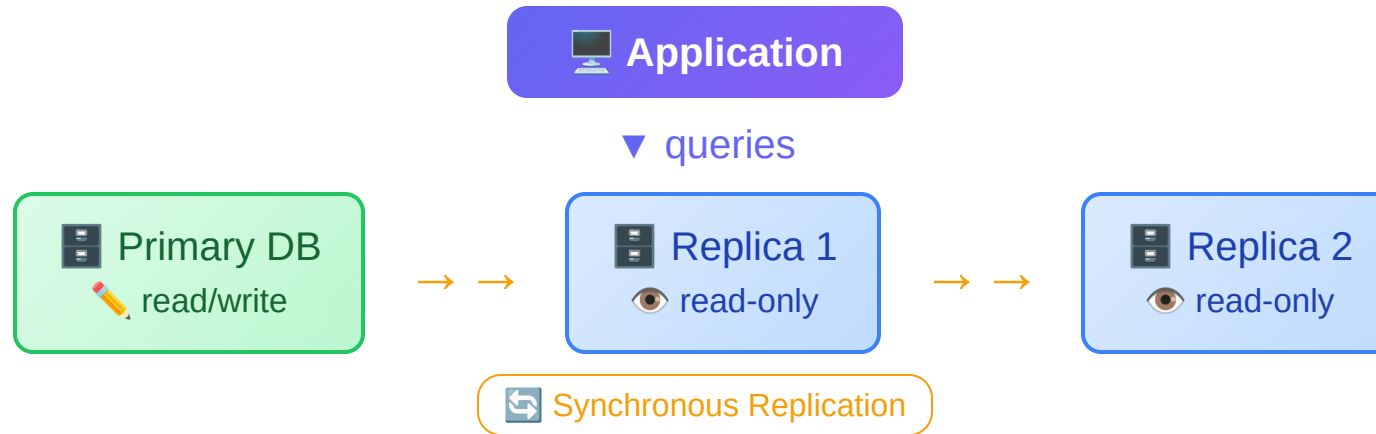
Availability-Driven Architecture

Availability = How often your system is operational.

Target	Max Downtime/Year	Requirements
99% ("two nines")	3.65 days	Basic monitoring
99.9% ("three nines")	8.76 hours	Redundant components, auto failover
99.99% ("four nines")	52.6 minutes	Multiple data centers
99.999% ("five nines")	5.26 minutes	Active-active replication

Core principle: Redundancy. No single point of failure.

Database Failover Pattern



If primary fails → replica can be promoted!

Netflix: Chaos Engineering

Netflix requires **99.99% availability** — only 52 minutes downtime/year.

Their approach:

- **Multiple AWS regions:** Active-active in US-East + US-West
- **Chaos Monkey:** Randomly terminates production instances
- **Chaos Gorilla:** Kills entire availability zones
- **Chaos Kong:** Simulates full region failure

"We could align teams around infrastructure resilience by isolating problems and pushing them to the extreme."

Trade-off: High availability = expensive (redundancy costs money).

Maintainability-Driven Architecture

80% of software cost is maintenance, not initial development.

Core principle: Separation of Concerns

Signs of POOR maintainability:

- "God Class" with 500+ lines
- Change to DB requires touching UI
- Can't test without starting server

Signs of GOOD maintainability:

- Each module has one purpose
- Changes are localized
- Can test components in isolation

Single Responsibility Principle (SRP)

```
# ✓ GOOD: Separated responsibilities
class ProfileRepository:    # Only: data access
    def get(self, id): ...
    def save(self, profile): ...

class ProfileService:      # Only: business logic
    def validate_and_save(self, profile):
        self._validate(profile)
        self.repository.save(profile)

def create_profile_chart(profile): # Only: visualization
    fig, ax = plt.subplots()
    ax.plot(profile.distances, profile.elevations)
    return fig
```

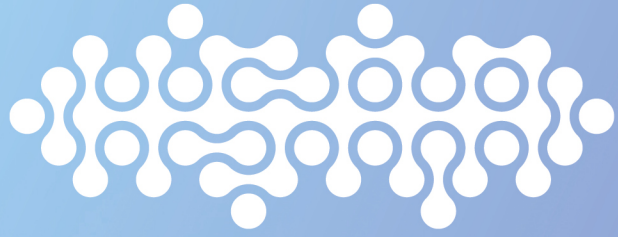
Each module has exactly ONE reason to change.

Dependency Injection: Key to Testability

```
# ✗ Hard to test: Service creates its own dependencies
class ProfileService:
    def __init__(self):
        self.repository = ProfileRepository("production.db") # Hardcoded!

# ✓ Easy to test: Dependencies are injected
class ProfileService:
    def __init__(self, repository: ProfileRepository):
        self.repository = repository # Caller decides!

# In tests:
mock_repo = MockProfileRepository() # Fake, no real DB
service = ProfileService(mock_repo)
```



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Architectural Trade-offs

3

Trade-offs: You Cannot Optimize Everything

Architectural goals often conflict:

Goal A	Conflicts With	Why
Performance	Maintainability	Large components fast but hard to change
Security	Usability	More layers = more friction
Availability	Cost	Redundant systems are expensive
Flexibility	Simplicity	More config = more bugs

How to Handle Trade-offs

- 1. Prioritize:** Which NFRs matter most for YOUR system?
- 2. Partition:** Use different architectures for different parts
 - Performance-critical core + maintainable plugins
- 3. Document:** Record WHY you made each trade-off

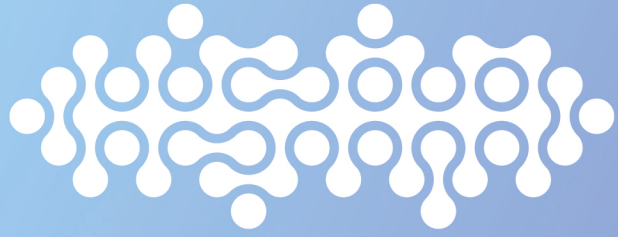
For Road Profile Viewer: Maintainability and simplicity > performance optimization.
Performance can wait until you actually have 1,000 users!

Quick Poll: Your Priorities

Which NFR matters **MOST** for your student projects?

- A) **Performance** — must be fast
- B) **Security** — must protect data
- C) **Availability** — must be always up
- D) **Maintainability** — must be easy to change

Discuss: Why might the "right" answer differ for a startup vs. a bank vs. a game?



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

The 4+1 View Model

4

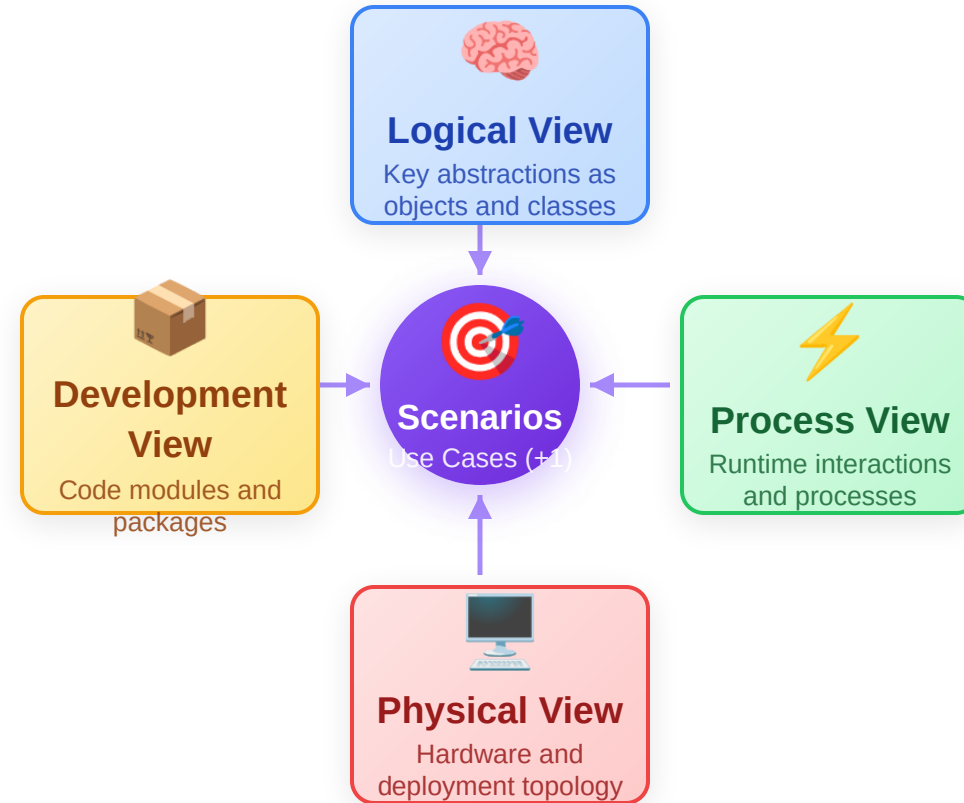
Why One Diagram Is Never Enough

Different stakeholders need different information:

- **Developers:** What classes exist? What packages contain them?
- **Operations:** What servers run what services? How do they communicate?
- **Project managers:** What teams own what components?
- **Security auditors:** Where does sensitive data flow?

"It is impossible to represent all relevant information about a system's architecture in a single diagram."
— Sommerville

The 4+1 View Model



4+1 Views Overview

View	Shows	Key Question
Logical	Objects and classes	"What are the main concepts?"
Process	Runtime interactions	"What runs when?"
Development	Code modules	"How is code organized?"
Physical	Hardware deployment	"What runs where?"
Scenarios (+1)	Use cases	"How does a request flow through?"

Logical View

Shows the system's key abstractions — the domain concepts.

Road Profile Viewer - Logical View:

RoadProfile ← ProfileRepository

↑

└── GeometryCalculator

↑

└── ChartBuilder

Key concepts: RoadProfile (data), Repository (storage), Calculator (math), Builder (visualization)

Process View

Shows what happens at runtime — processes, threads, interactions.

```
User → DashApp → Callback → Database
      ↓
    Generate Chart
      ↓
Return Updated Figure
      ↓
    Display to User
```

Helps analyze: "Every user interaction triggers a DB query — is that a bottleneck?"

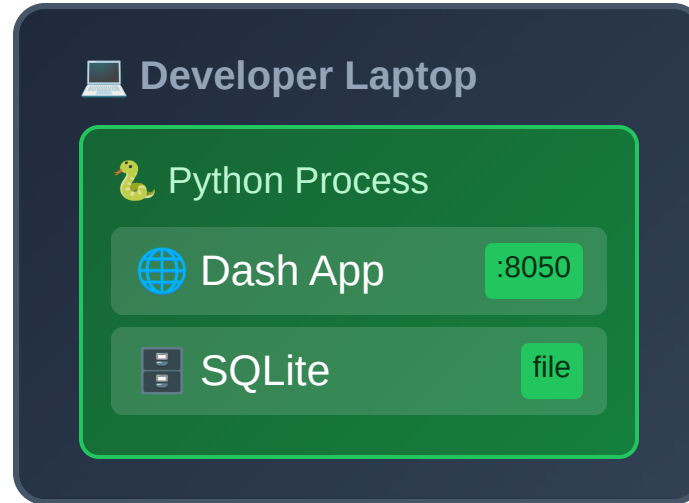
Development View

Shows how code is organized into packages and modules.

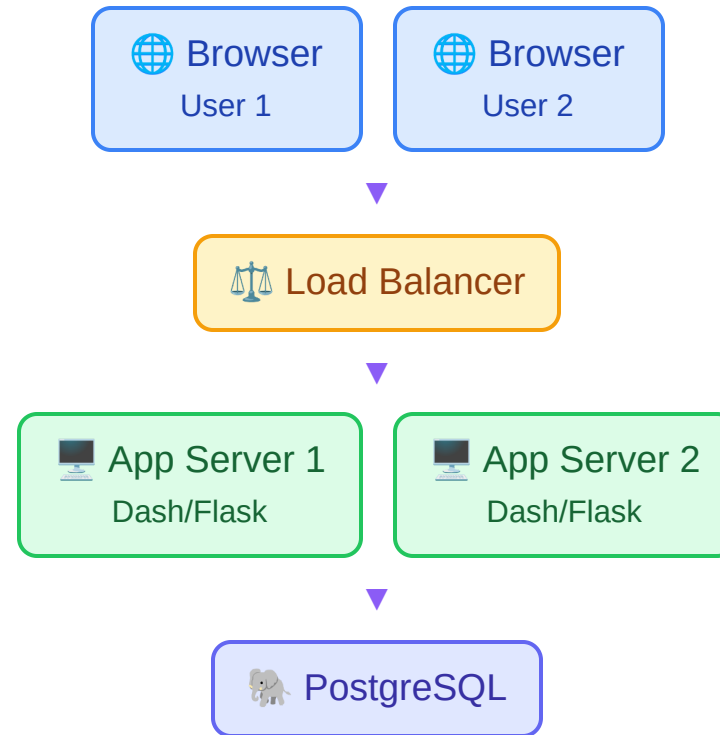
```
road-profile-viewer/  
├─ src/  
│   ├─ models/           # Domain models  
│   ├─ repositories/     # Data access  
│   ├─ services/         # Business logic  
│   └─ presentation/     # Dash UI  
├─ tests/  
└─ pyproject.toml
```

Helps coordinate: "Anna works on `services/`, Ben works on `presentation/`."

Physical View — Current (Simple)



Physical View — Scaled (Future)



Scenarios: The +1

Scenarios connect all four views by tracing a user action.

Scenario: User uploads a new road profile

1. **Physical:** Request arrives at load balancer → App Server 1
2. **Process:** Upload handler validates, parses, calls ProfileService
3. **Logical:** ProfileService creates RoadProfile, validates with Pydantic
4. **Development:** Code in `services/` calls `repositories/`
5. **Physical:** Database write goes to PostgreSQL
6. **Process:** Response flows back to user's browser

Quick Architecture Doc (Markdown)

You don't need formal UML for a student project!

ROAD PROFILE VIEWER ARCHITECTURE

LOGICAL VIEW: RoadProfile, ProfileRepository, GeometryCalculator

DEVELOPMENT VIEW: src/models/, src/services/, src/presentation/

PROCESS VIEW: Single-threaded Dash, callbacks, sync DB queries

PHYSICAL VIEW: Single machine, SQLite, port 8050

KEY SCENARIO: View Profile

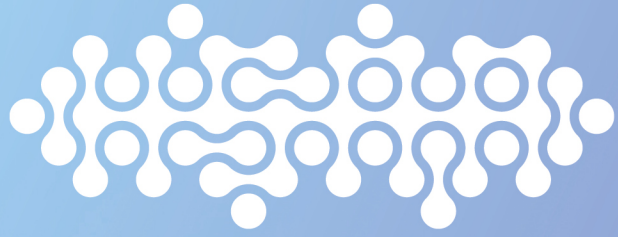
User selects → Callback → DB query → Chart → UI update

UML: Formal Notation

UML (Unified Modeling Language) — standardized visual notation.

UML Diagram Type	4+1 View	Shows
Class Diagram	Logical	Classes, attributes, relationships
Sequence Diagram	Process	Object interactions over time
Package Diagram	Development	Code organization
Deployment Diagram	Physical	Hardware nodes
Use Case Diagram	Scenarios	Actor-system interactions

For now: Simple markdown + Mermaid diagrams are sufficient.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Summary & Reflection

5

Summary

Concept	Key Point
Software Architecture	Fundamental organization as communicating components
Architectural Decisions	Creative process driven by NFRs
Non-Functional Requirements	Performance, security, availability, maintainability
4+1 Views	Logical, Process, Development, Physical + Scenarios
Trade-offs	Cannot optimize all qualities; must prioritize

Key Takeaways

1. **Architecture matters from day one** — even small projects benefit
2. **NFRs shape architecture** — performance, security, maintainability drive decisions
3. **Multiple views are necessary** — no single diagram captures everything
4. **Trade-offs are inevitable** — prioritize based on your system's needs
5. **Architecture enables communication** — diagrams help stakeholders understand

Reflection Questions

1. **Current architecture:** How would you describe your Road Profile Viewer's architecture?
What views can you identify?
2. **NFR priorities:** What are the top three NFRs for your project? How do these influence your choices?
3. **Trade-off scenario:** Easier to maintain OR faster to run — which would you choose?
Why?
4. **4+1 exercise:** Draw a simple diagram for each of the four views of your project.

What's Next: Part 2

Coming Up: Architectural Patterns

- **MVC (Model-View-Controller)** — separating concerns in interactive apps
- **Layered architecture** — Presentation, Business, Data layers
- **Distributed patterns** — Client-Server, Microservices
- **Applying patterns to Road Profile Viewer** — evolving your project

**You've learned WHY architecture matters.
Now learn HOW to apply proven patterns.**



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Questions?

PART 1 COMPLETE

Next: Part 2 — Architectural Patterns