



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Chapter 06

SOFTWARE ARCHITECTURE — PART 2

Recap: What We Learned in Part 1

In Part 1, we explored:

- **Why architecture matters** — the bridge between Agile delivery and sustainable systems
- **The 4+1 View Model** — Logical, Process, Development, Physical, and Scenarios
- **Non-functional requirements** shape architectural decisions

Today: From *understanding* to *applying* architecture with proven **patterns**.

The Pattern Promise

When a developer says "**we use MVC**", other developers instantly understand:

- How responsibilities are divided
- Where to find business logic vs. UI code
- How to add new features without breaking existing ones

*Patterns are not just code structures — they are **shared vocabulary** that enables efficient communication.*

Learning Objectives

By the end of this lecture, you will be able to:

1. **Explain what architectural patterns are** and why they provide value as proven solutions
2. **Apply MVC and layered architecture** to structure interactive applications
3. **Evaluate when to use distributed patterns** like client-server and microservices
4. **Design an evolved architecture** for Road Profile Viewer

What Is an Architectural Pattern?

***An architectural pattern** is a description of a system organization that has been successfully used in different software systems.*

Patterns are discoveries, not inventions.

Why use patterns?

1. **Proven:** Thousands of systems have validated these approaches
2. **Communicable:** "We use MVC" instantly conveys structure
3. **Analyzable:** Known patterns have known trade-offs
4. **Reusable:** Framework support, documentation, community knowledge

Three Categories of Architectural Patterns

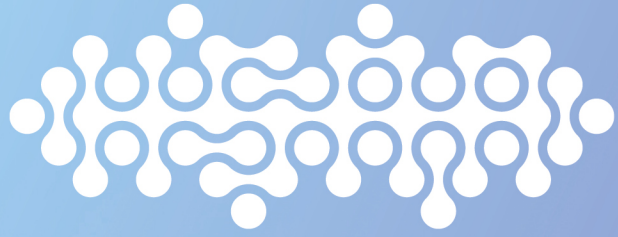
Category	Purpose	Examples
Interactive Systems	Organize human-computer interactions	MVC, MVP, MVVM
System Structure	Organize components and objects	Layered, Pipes/Filters, Repository
Distributed Systems	Enable distributed resources	Client-Server, Microservices, Event-Driven

Today we'll explore one pattern from each category.

Quick Poll: Which Category Interests You Most?

- A)** Interactive Systems (MVC, MVP, MVVM)
- B)** System Structure (Layered Architecture)
- C)** Distributed Systems (Microservices)
- D)** All of them equally!

Raise your hand or use the poll link!

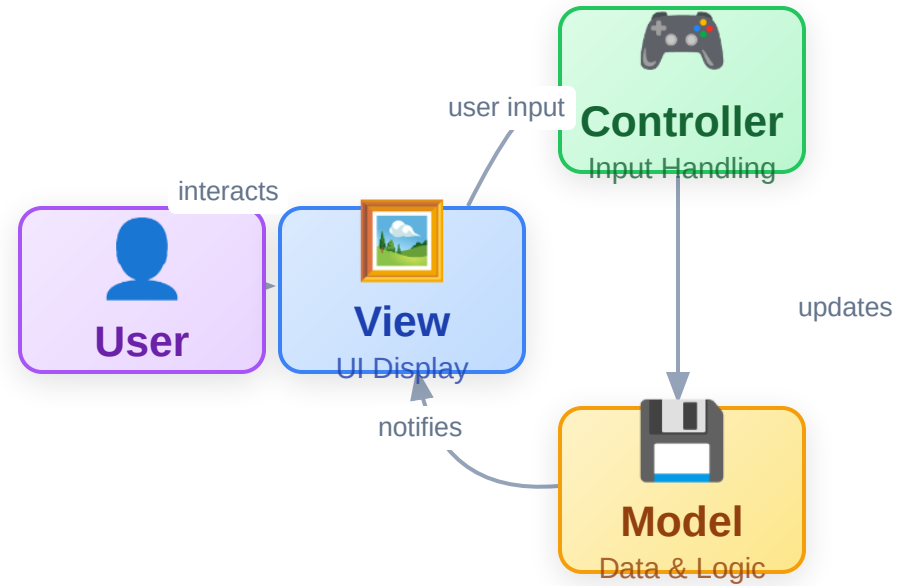


Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Interactive Systems: MVC

1

Model-View-Controller (MVC)



MVC Component Responsibilities

Component	Responsibility	Road Profile Viewer
Model	Data and business logic; knows nothing about UI	<code>RoadProfile</code> class, <code>GeometryCalculator</code>
View	Displays data; receives user input	Dash layout, Plotly charts
Controller	Handles user input; updates Model and View	Dash callbacks

Why MVC?

Benefits:

- **Separation of concerns:** UI designers work on Views, developers on Models
- **Testability:** Models can be unit tested without UI
- **Flexibility:** Same Model can have multiple Views (web, mobile, CLI)

Trade-offs:

- More files to manage
- Learning curve for new developers
- Overkill for very simple apps

MVC in Django (Python)

Note: Django uses **MTV** (Model-Template-View) terminology, not MVC. Django's "View" acts as the Controller!

```
# models.py – Model (same as MVC)
class RoadProfile(models.Model):
    name = models.CharField(max_length=100)
    data = models.JSONField()

# views.py – Django "View" = MVC Controller (handles requests)
def profile_detail(request, profile_id):
    profile = RoadProfile.objects.get(id=profile_id)
    return render(request, 'profile.html', {'profile': profile})

# templates/profile.html – Django "Template" = MVC View (presentation)
<h1>{{ profile.name }}</h1>
```

MVC in Flask (Python)

```
# models.py (Model)
@dataclass
class RoadProfile:
    id: str
    name: str
    coordinates: list[tuple[float, float]]

# routes.py (Controller)
@app.route('/profiles/<profile_id>')
def show_profile(profile_id):
    profile = ProfileRepository.get(profile_id)
    return render_template('profile.html', profile=profile)

# templates/profile.html (View) - Jinja2 template
```

Think-Pair-Share: MVC in Your Experience

2 minutes: Think about web applications you've used.

2 minutes: Discuss with a neighbor.

Share: What patterns do you recognize?

Discussion questions:

1. Can you identify Model, View, Controller in apps you've built?
2. Have you seen apps where concerns were mixed? What problems arose?

Road Profile Viewer: Applying MVC

Model (`models/profile.py`):

```
class RoadProfile(BaseModel):  
    """Domain model - knows nothing about UI or storage."""  
    id: str  
    name: str  
    x_coordinates: list[float]  
    y_coordinates: list[float]  
  
    def max_slope(self) -> float:  
        """Business logic lives in the Model."""  
        ...
```

Road Profile Viewer: View Component

View (`presentation/charts.py`):

```
def create_profile_figure(profile: RoadProfile) -> go.Figure:
    """Pure visualization - no business logic."""
    return go.Figure(
        data=go.Scatter(
            x=profile.x_coordinates,
            y=profile.y_coordinates,
            mode='lines', name=profile.name
        ),
        layout=go.Layout(title=f"Road Profile: {profile.name}")
    )
```

Road Profile Viewer: Controller Component

Controller (presentation/callbacks.py):

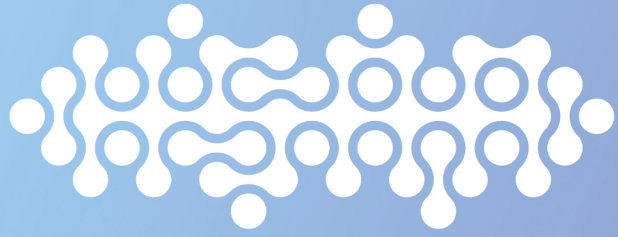
```
@callback(  
    Output('profile-chart', 'figure'),  
    Input('profile-dropdown', 'value')  
)  
def update_chart(profile_id: str):  
    """Controller: coordinates Model and View."""  
    if not profile_id:  
        return go.Figure() # Empty figure  
  
    profile = repository.get(profile_id) # Get Model  
    return create_profile_figure(profile) # Create View
```

MVC Benefits Achieved

With proper MVC separation:

- `RoadProfile` can be tested **without Dash**
- Charts can be reused in other contexts (PDF export, etc.)
- Callbacks are **thin** — just coordination, no business logic

*The Model doesn't know about the View.
The View doesn't know about the Controller.
Everyone talks to the Model.*



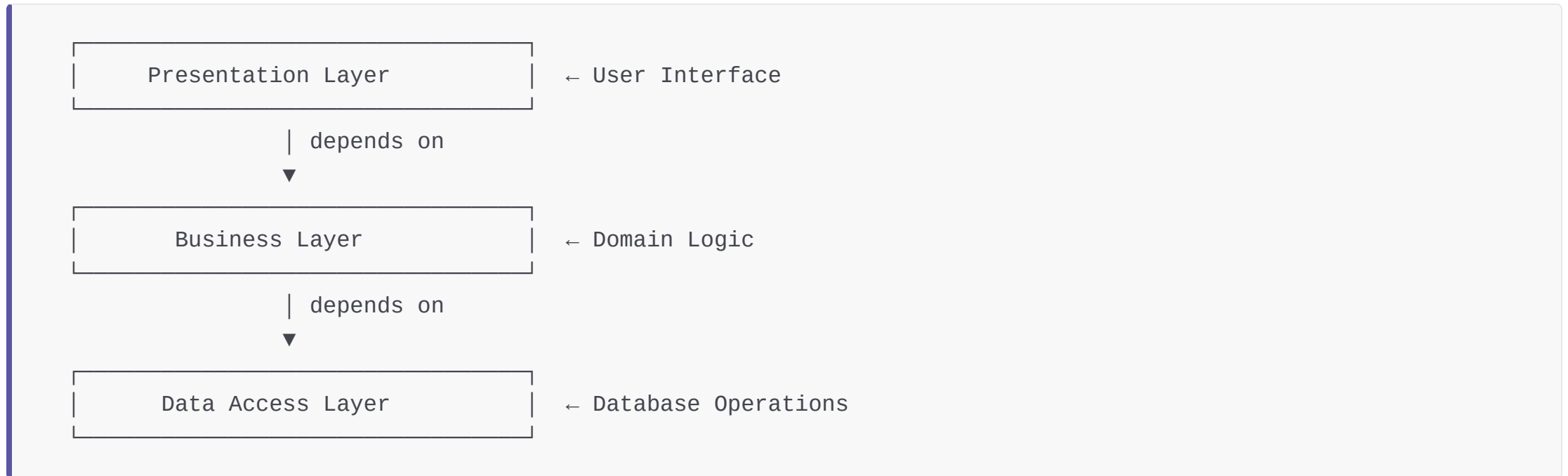
Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

System Structure: Layered Architecture

2

Layered Architecture

Layered architecture organizes code into horizontal layers, where each layer only depends on the layer below it.



Layered Architecture: Key Rules

The Three Golden Rules:

1. **Downward dependencies only:** Presentation → Business → Data
2. **No skipping layers:** Presentation should NOT call Data directly
3. **Layers are cohesive:** All UI code in Presentation, all domain logic in Business

Why? Changes in one layer don't cascade through the entire system.

Layer Responsibilities

Layer	Responsibility	Typical Components
Presentation	Handle user interaction, display data	Web pages, API endpoints, CLI
Business	Domain logic, validation, calculations	Services, domain models, validators
Data Access	Persist and retrieve data	Repositories, ORM models, cache

Netflix: A Real-World Example

Netflix demonstrates layered principles at scale:

PRESENTATION: Web UI, Mobile Apps, TV Apps, API Gateway

|

BUSINESS: User Service, Content Service, Recommendation

|

DATA: Cassandra, MySQL, ElasticSearch, S3

Netflix: Presentation Layer

Component	Description
Web UI	React Single Page Application
Mobile Apps	iOS & Android native apps
TV Apps	Smart TVs, Roku, PlayStation
API Gateway (Zuul)	Routes requests to services

Each presentation client can be updated independently!

Netflix: Business Layer (Microservices)

Service	Responsibility
User Service	Authentication & profiles
Content Service	Movies & shows metadata
Recommendation	ML-powered suggestions
Playback Service	Streaming logic

Each service can **scale independently** based on demand!

Netflix: Data Layer

Database	Use Case
Cassandra	User data & viewing history
MySQL	Billing & accounts
ElasticSearch	Search indexing
S3	Video files (petabytes!)

Key insight: Different data stores for different needs!

Layered Architecture: Key Benefits

Independent scaling:

- Need more recommendations?
- Scale just that service
- Other layers unchanged

Technology flexibility:

- Swap SQLite for PostgreSQL
- Change only Data Access layer
- Business logic untouched

Each layer can evolve independently.

Road Profile Viewer: Current State (Mixed Concerns)

```
@app.callback(...)
def update_chart(profile_id):
    if not profile_id:
        return {} # UI logic

    conn = sqlite3.connect('profiles.db') # Data access
    cursor = conn.execute('SELECT * FROM profiles WHERE id = ?', (profile_id,))
    row = cursor.fetchone()

    x_coords = json.loads(row[1]) # Business logic
    y_coords = json.loads(row[2])
    max_slope = calculate_max_slope(x_coords, y_coords)

    return create_figure(x_coords, y_coords, max_slope) # UI again
```

Problem: All layers mixed in one function!

Road Profile Viewer: Data Layer

`data_access/repositories.py` :

```
class ProfileRepository:
    def __init__(self, db_path: str = 'profiles.db'):
        self.db_path = db_path

    def get(self, profile_id: str) -> Profile | None:
        conn = sqlite3.connect(self.db_path)
        cursor = conn.execute(
            'SELECT * FROM profiles WHERE id = ?', (profile_id,)
        )
        row = cursor.fetchone()
        return Profile.from_row(row) if row else None
```

Road Profile Viewer: Business Layer

business/services.py :

```
class ProfileService:
    def __init__(self, repository: ProfileRepository):
        self.repository = repository

    def get_profile_with_analysis(self, profile_id: str) -> dict:
        profile = self.repository.get(profile_id)
        if not profile:
            raise ProfileNotFoundError(profile_id)

        return {
            'profile': profile,
            'max_slope': self._calculate_max_slope(profile),
        }
```

Road Profile Viewer: Presentation Layer

`presentation/callbacks.py` :

```
service = ProfileService(ProfileRepository())

@app.callback(...)
def update_chart(profile_id):
    if not profile_id:
        return empty_figure()

    try:
        data = service.get_profile_with_analysis(profile_id)
        return create_figure(data)
    except ProfileNotFoundError:
        return error_figure("Profile not found")
```

Clean! Each layer has one responsibility.

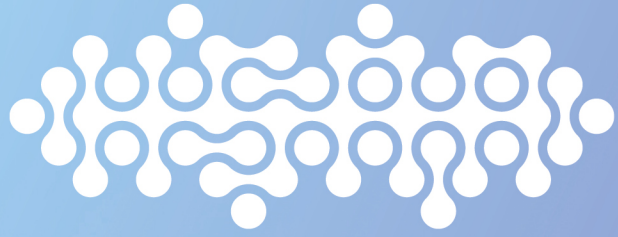
Think-Pair-Share: Where Would You Add Caching?

Scenario: Profile calculations are slow. You want to cache results.

Discuss:

1. Which layer should the cache live in?
2. Should the Business layer know about the cache?
3. How would you implement it without changing other layers?

2 minutes think, 2 minutes pair, then share!

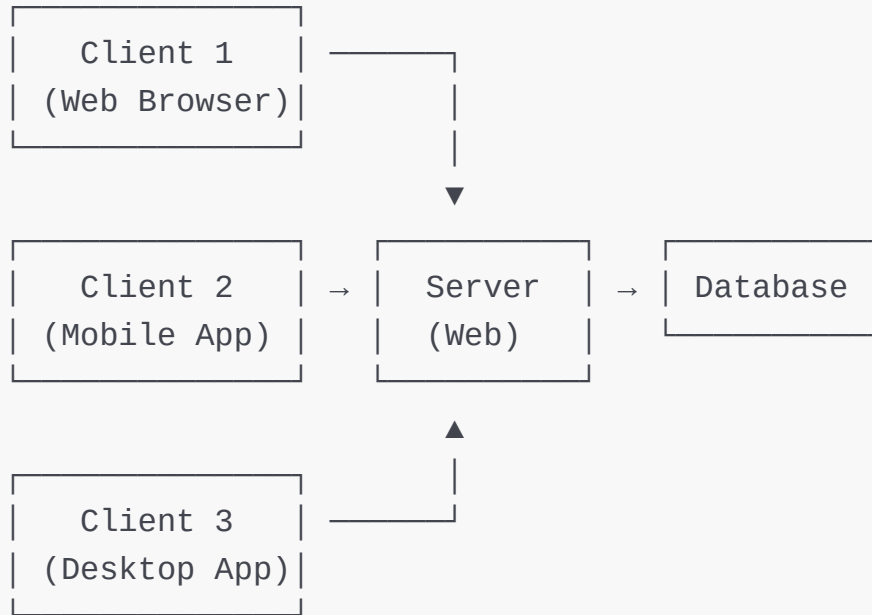


Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Distributed Systems

3

Client-Server Architecture



Client-Server Characteristics

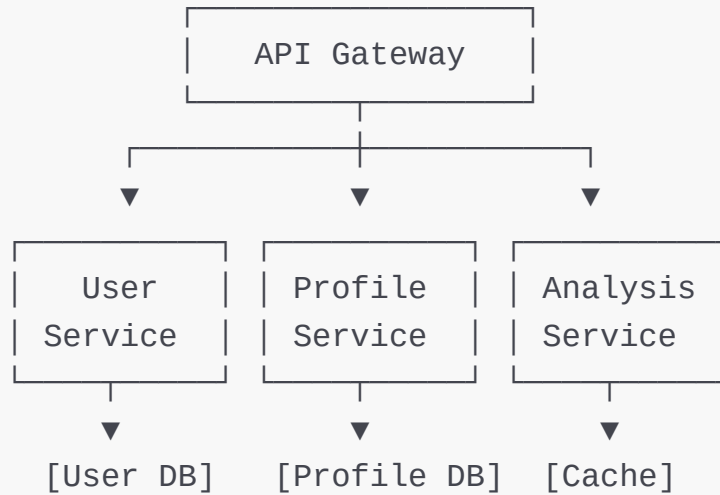
Component	Responsibility
Clients	Initiate requests, display results, handle user input
Servers	Process requests, manage data, enforce business rules
Protocol	Usually HTTP/HTTPS for web applications

Road Profile Viewer is already client-server:

- Client: Web browser running JavaScript/Dash frontend
- Server: Python Dash/Flask process handling callbacks

Microservices Architecture

Microservices decompose a system into small, independently deployable services:



Monolith vs. Microservices

Monolith:

- All-or-nothing deployment
- Scale everything together
- One language/framework
- One bug can crash system
- Simple deployment

Microservices:

- Independent deployment
- Scale services individually
- Best tool for each job
- Failures are isolated
- Complex deployment

Amazon's Two-Pizza Teams

In 2002, Jeff Bezos mandated:

"All teams must expose their functionality through service interfaces."

The "two-pizza team" rule: Teams small enough to feed with two pizzas.

Result: Amazon Web Services — originally internal infrastructure, now an **\$80+ billion business**.

When to Use Microservices

DO use when:

- Large teams (can't coordinate)
- Different scaling needs
- Technology diversity required
- Teams need autonomy

DON'T use when:

- Small teams (3-5 people)
- Early-stage products
- Simple domains
- Need to iterate quickly

Road Profile Viewer: *Stay with a monolith! You have 3 developers and a simple domain.*

Quick Poll: Monolith or Microservices?

Scenario: Your Road Profile Viewer team grows from 3 to 30 developers. You now have 5 feature teams working on different parts.

What architecture would you recommend?

- A)** Keep the monolith, just organize code better
- B)** Split into microservices
- C)** Start with a "modular monolith" and split later

AWS Cloud Architecture

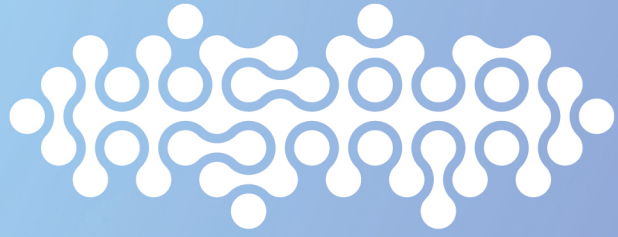
Cloud platforms provide building blocks for distributed architectures:

```
Users → CloudFront (CDN) → Load Balancer → ECS/Fargate → RDS
      (Edge Cache)      (Routing)      (Containers) (Database)
```

For Road Profile Viewer (future):

- **Render.com** or **Railway.app** — simple PaaS, free tier
- **Supabase** — managed PostgreSQL with free tier

You don't need AWS complexity for a student project!



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Applying Patterns to Road Profile Viewer

4

Current State: From Monolith to Modules

In Lecture 4 (Refactoring), we split a monolithic `main.py` into modules:

```
road-profile-viewer/  
├─ src/  
│   ├── geometry.py      # Ray intersection calculations  
│   ├── road.py          # Profile generation  
│   ├── visualization.py  # Chart creation  
│   └─ main.py           # Application entry point  
└─ tests/
```

Good start! But we can go further with proper layered architecture.

Proposed Evolved Architecture

```
road-profile-viewer/
├── src/
│   ├── domain/                # Core business concepts
│   │   ├── models.py          # RoadProfile, Measurement
│   │   └── services.py        # GeometryCalculator
│   ├── infrastructure/        # External systems
│   │   └── repositories.py    # ProfileRepository
│   ├── presentation/          # User interface
│   │   ├── callbacks.py       # Controllers
│   │   └── charts.py          # Views
│   └── api/                   # Optional: REST API
│       └── routes.py           # FastAPI endpoints
```

Adding a REST API Layer

Why add an API when Dash already works?

- **Multiple frontends:** Same backend for web, mobile, CLI
- **Testing:** APIs are easier to test than UI callbacks
- **Integration:** Other systems can access your data

```
@app.get("/api/profiles/{profile_id}")
def get_profile(profile_id: str) -> ProfileResponse:
    profile = repository.get(profile_id)
    if not profile:
        raise HTTPException(status_code=404)
    return ProfileResponse(**profile.dict())
```

Future Scalability

If you ever need to scale Road Profile Viewer:

Change	What to Modify
Database	Replace SQLite with PostgreSQL — change only <code>infrastructure/</code>
Caching	Add Redis cache — new file in <code>infrastructure/</code>
Deployment	Containerize with Docker — no code changes
Load balancing	Add multiple instances — infrastructure only

The layered architecture makes changes localized!

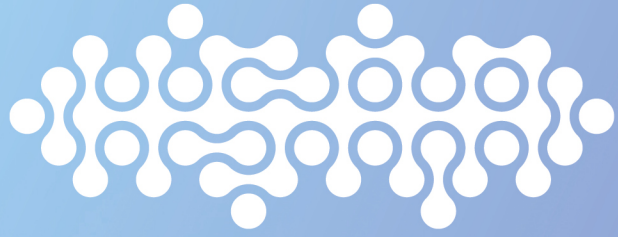
Interactive Exercise: Design the Next Feature

Scenario: The Product Owner wants to add "export profile to PDF".

In pairs, answer:

1. Which layer does PDF generation belong to?
2. Does it need a new service in the Business layer?
3. How would you expose it (button in UI? API endpoint? Both?)
4. What existing components would you reuse?

5 minutes, then we'll discuss!



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Architecture & Agile

5

The Apparent Conflict

Some developers believe Agile and architecture are in conflict:

Extreme 1:

"Agile means no upfront design — just start coding and refactor!"

Extreme 2:

"Architecture requires months of planning before writing any code!"

Both extremes are wrong.

Emergent Architecture in Scrum

The Agile approach: Emergent design with intentional structure.

1. **Make enough decisions to start:** Choose language, framework, basic structure
2. **Implement working software:** Build real features, not hypothetical infrastructure
3. **Refactor as patterns emerge:** When you see duplication, restructure
4. **Revisit decisions regularly:** Sprint retros can address architectural debt

Technical Spikes

A **technical spike** is a time-boxed investigation to reduce uncertainty.

Example spike story:

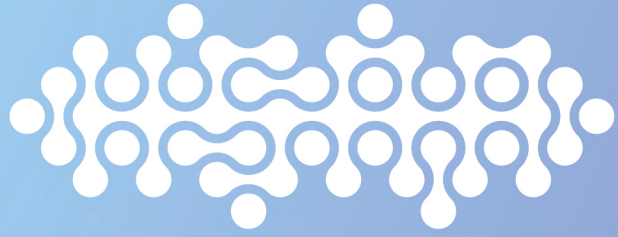
As a developer

I want to investigate PostgreSQL vs SQLite for our database

So that we can make an informed decision

Time-box: 2 days maximum

After the spike: Make an *informed* architectural decision, not a guess.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Summary

6

Summary Table

Concept	Key Point	Application
Patterns	Proven solutions to recurring problems	Use MVC for predictable structure
MVC	Separate Model, View, Controller	Pydantic models, charts, callbacks
Layered	Presentation → Business → Data	presentation/ , domain/ , infrastructure/
Distributed	Client-Server, Microservices	Stay monolithic for now
Agile	Emergent design, technical spikes	Make decisions, implement, refactor

Key Takeaways

1. **Patterns are proven solutions** — MVC, layered architecture, and client-server solve recurring problems
2. **MVC separates concerns** — Model knows data, View knows display, Controller coordinates
3. **Layers create boundaries** — Dependencies flow downward only
4. **Microservices aren't always the answer** — Small teams benefit from well-structured monoliths
5. **Architecture evolves with Agile** — Start simple, refactor as patterns emerge

Reflection Questions

1. **Pattern identification:** Which pattern does your Road Profile Viewer currently follow?
2. **Layer mapping:** Which layer does each of your Python files belong to?
3. **Dependency direction:** Do all your imports point "downward"?
4. **Scale trigger:** What metric would tell you it's time to move from SQLite to PostgreSQL?

What's Next: Apply to Your Project

Your action items:

1. **Identify your current pattern:** Which pattern does your code follow (if any)?
2. **Map your layers:** Which code is presentation? Business? Data access?
3. **Refactor one layer:** Pick the messiest area and apply separation
4. **Document your architecture:** Create a simple `ARCHITECTURE.md`
5. **Consider an API:** Would a REST API benefit your project?

Further Reading

Books:

- Ian Sommerville: *Software Engineering* (Chapter 6)
- Robert C. Martin: *Clean Architecture*
- Martin Fowler: *Patterns of Enterprise Application Architecture*

Online:

- [Martin Fowler on MVC](#)
- [Netflix Tech Blog](#)
- [The Twelve-Factor App](#)



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Questions?

THANK YOU