University of St.Gallen

TAPAS Project –
Mid-Term Presentation

St.Gallen, 23rd October 2023
Group 5

From insight to impact.

The Team

Kaan Aydin

Michael Brülisauer

Daniel Leal

Stephan Nef

Agenda

- Software Architecture
- Selected Architecture Decisions
- Performance & Testing
- Demonstration
- Key Takeaways & Contributions

Agenda

Software Architecture

- Event Storming Board
- Architecture Characteristics
- Architectural Styles
- Service Design
- Application Flow

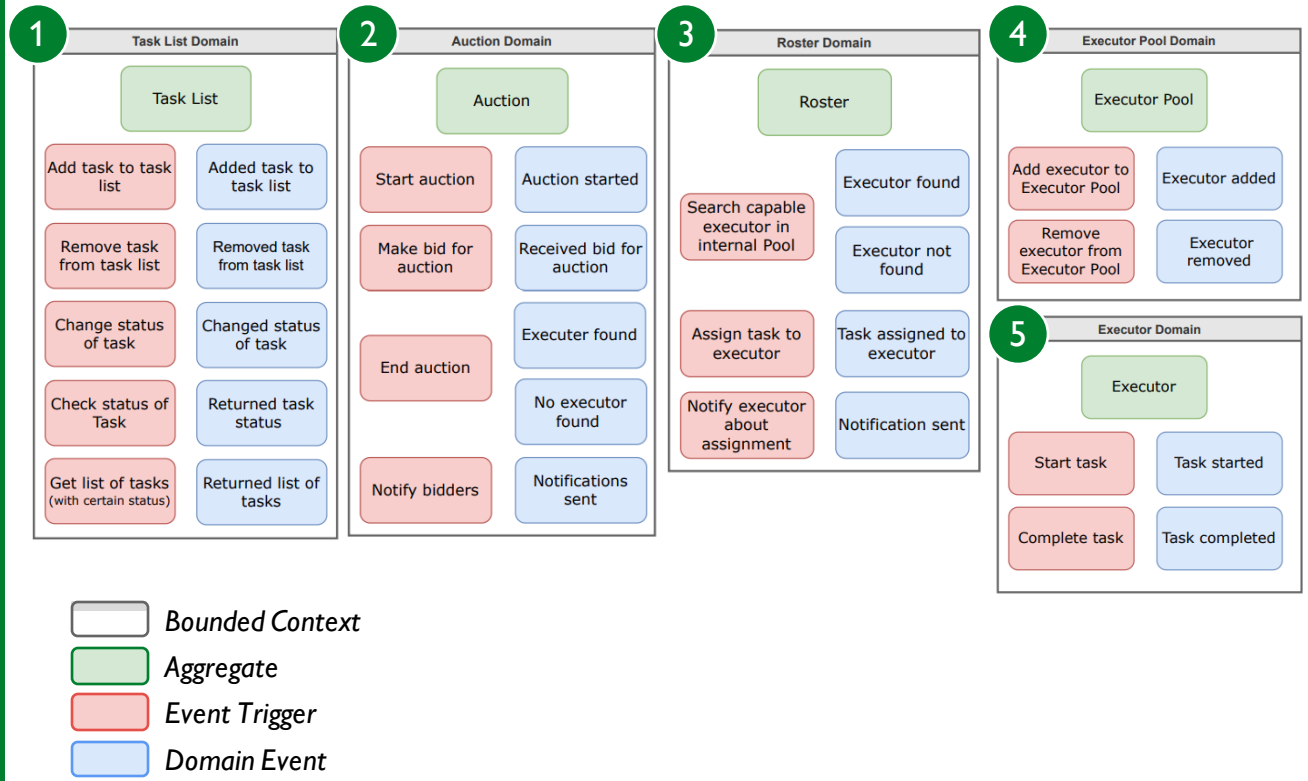
Selected Architecture Decisions

Performance & Testing

Demonstration

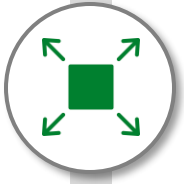
Key Takeaways & Contributions

Event Storming Board | We identified 5 bounded contexts



Architecture Characteristics | Given the user and systems requirements, we selected seven driving characteristics

Top 3



Scalability

Each organization can have thousands of users adding Tasks to the organization's Task List

Elasticity

The performance of the system should degrade gracefully in the presence of heavy load.

Modularity

We can add new Executors while the system is running w/o affecting remaining parts of the system

Interoperability

Tasks for which no internal Executors are available need to be executed by external Executors

Recoverability

The assignment of tasks to executors should be preserved in case (parts of) the system fail

Fault Tolerance

The system's operation should not be disrupted in case an Executor malfunctions or fails

Evolvability

Evolvability is key to support incremental changes throughout the semester across dimensions

Architectural Styles | We evaluated several architectural along their strengths & weaknesses

ADR 2

	Layered	Modular Monolith	Microkernel	Microservices	Service-based	Service-oriented	Event-driven	Space-based
Partitioning type	Technical	Domain	Domain and technical	Domain	Domain	Technical	Technical	Domain and technical
# quanta	1	1	1	1 to many	1 to many	1	1 to many	1 to many
Agility	+	++	+++	+++++	++++	+	+++	++
Abstraction	+	+	+++	+	+	+++++	++++	+
Configurability	+	+	++++	+++	++	+	++	++
Deployability	+	++	+++	+++++	++++	+	+++	+++
Elasticity	+	+	+	+++++	++	+++	++++	+++++
Evolvability	+	+	+++	+++++	+++	+	+++++	+++
Fault tolerance	+	+	+	+++++	++++	+++	+++++	+++
Integration	+	+	+++	+++	++	+++++	+++	++
Interoperability	+	+	+++	+++	++	+++++	+++	++
Overall cost	+++++	+++++	+++++	+	++++	+	+++	++
Performance	++	++	+++	++	+++	++	+++++	+++++
Scalability	+	+	+	+++++	+++	+++	+++++	+++++
Simplicity	+++++	+++++	++++	+	+++	+	+	+
Testability	++	++	+++	+++++	++++	+	++	+
Workflow	+	+	++	+	+	+++++	+++++	+

Both Microservices & Event-Driven rank highly based on our selected characteristics

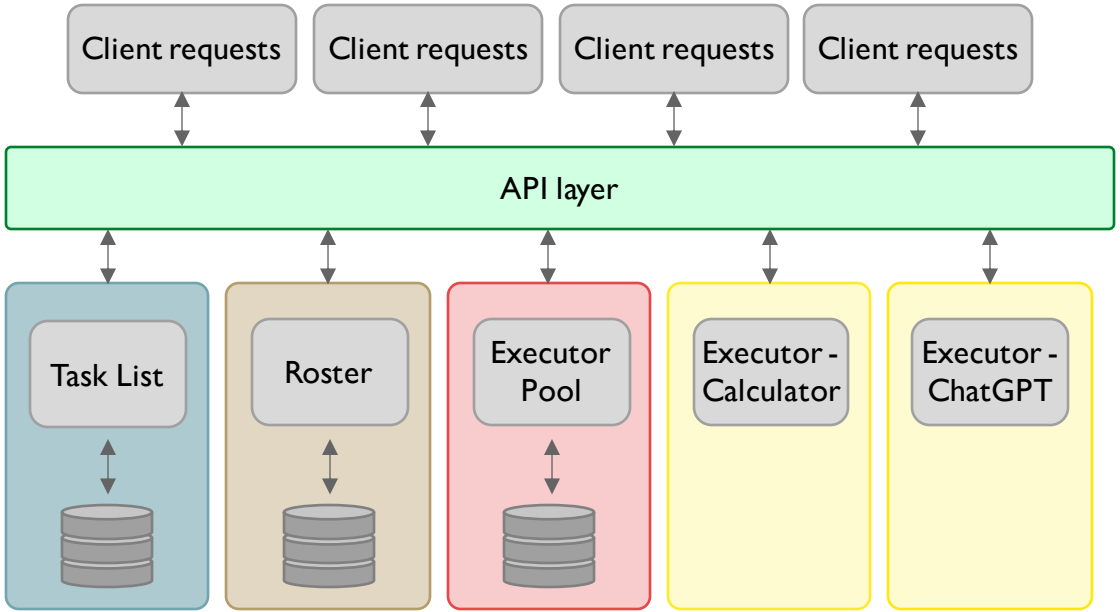
However, event-driven adds significant overhead due to async comms and it is technically partitioned

Microservices allow us to add changes to our system more flexibly & we will use API layers to handle request efficiently

Hence, we chose Microservices for our architectural style

Service Design | Overview of our Microservices

ADRs 3, 4, 5, 6, 7



We implemented all bounded contexts as dedicated services within our TAPAS application

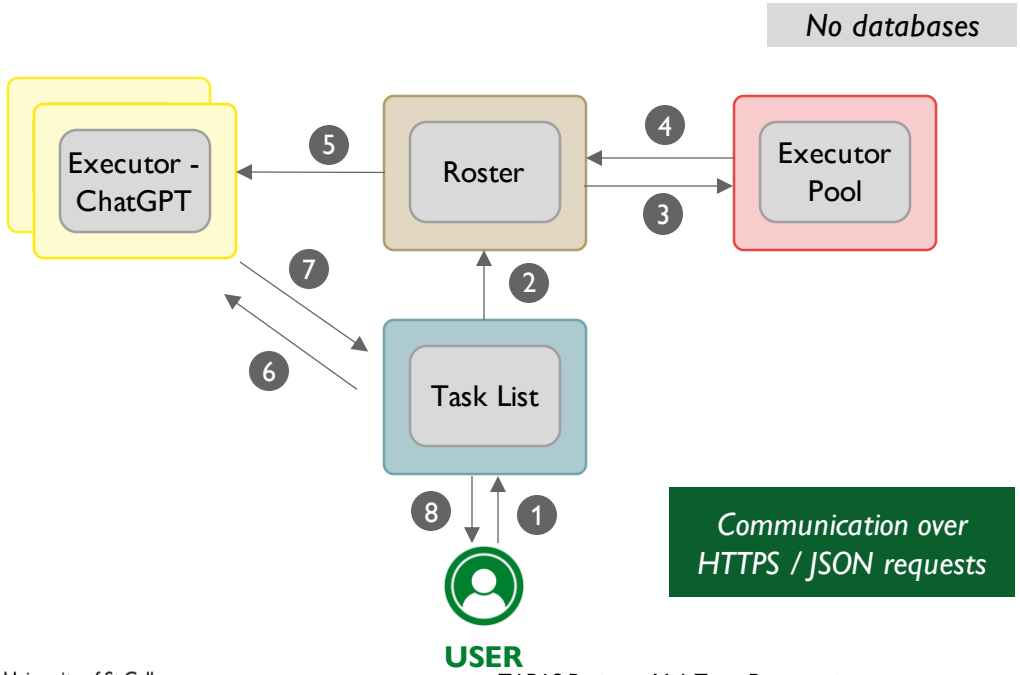
Advantages are higher modularity, granularity, interoperability, and scalability

Disadvantages are development complexity, service coordination & resource utilization

In this setup, we need to ensure that services can communicate with each other via an API layer with HTTP requests

Application Flow | How does an added task flow through the application and gets executed

Graphical Illustration



Explanation

1. User adds a task to the task list
2. Task List sends added task to Roster
3. Roster sends request to Pool with the Task Type
4. Pool returns to Roster a list of executors with the right type
5. Roster sends task location (i.e., URI) to the Executor, starting execution
6. Executor reads user input for task from Task List
7. Executor executes task and updates task output & status in Task list
8. User can view the task output

Agenda

Software Architecture

Selected Architecture Decisions

- Data Ownership & Access
- Code Sharing
- Dynamic Quantum Coupling

Performance & Testing

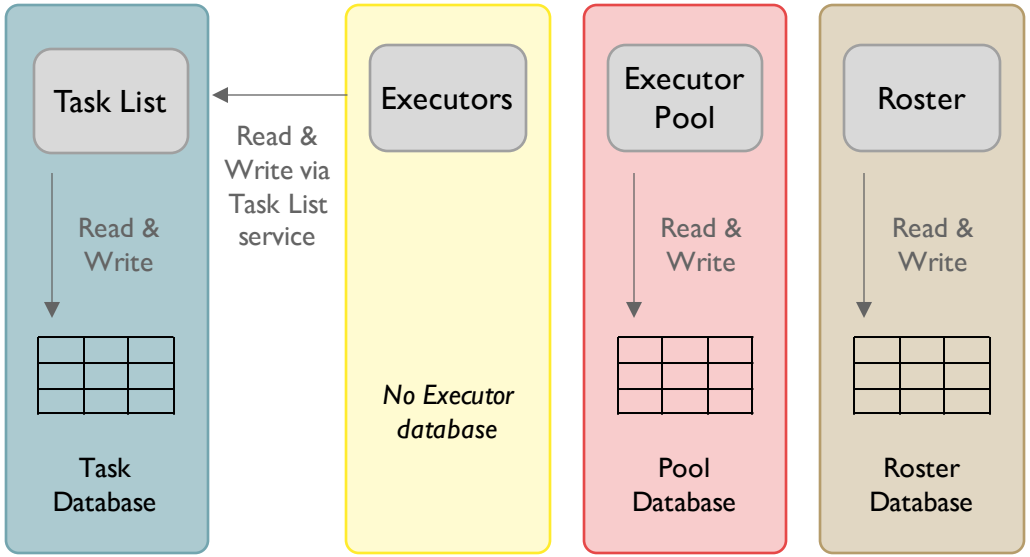
Demonstration

Key Takeaways & Contributions

Data Ownership & Access | Context, Decision & Consequences

ADR 9

Graphical Illustration



Description

- We have three services that require a database: task list, executor pool & roster
- Executors perform read (for task input) and write (for task output & status) to the task database via task list service

- **Single Ownership** for Task List, Executor Pool & Roster - maintain isolation of architectural quantum
- **Delegated Joint Ownership** for Executors & Task List - maintain single ownership for Task List

Code Sharing | Context, Decision & Consequences

ADRs 12, 13

Shared Library for Communication

Currently, we use a lot of **standard HTTP client / server** code to enable communication between microservices, leading to a lot code duplication

We will create **a separate helper class library** that will help us to create more standardized HTTP requests easily

- ✓ Code bundling leads to less duplication
- ✗ Increased complications & efforts for version mgmt.

Shared Library for Executors

Our executors contain **a similar code base** to replicate the functionality of an executor, for instance domain logic or communication to the other microservices

We will build a **separate executor library** that implements the common functionality across executors (i.e., everything that is not executor-specific)

- ✓ Changes on common functionality of executors implemented at a single place
- ✗ Increased complications & efforts for version mgmt.

Dynamic Quantum Coupling | Context, Decision & Consequences

ADR 11

Communication

In our current architecture, microservices communicate via **synchronous HTTP calls**, leading to dynamic quantum entanglement

Moving to **async communication** between our microservices (especially for the Executor) will lead to:

- ✓ Highly decoupled systems
- ✓ Higher performance & scale
- × Harder code to build / debug

Consistency

We maintain consistency across our microservices in our architecture **through events** (e.g., status updates)

To maintain eventual consistency, we need to **synchronize data into consistent state** (e.g., Task updates)

- ✓ More resilient to NW failures
- ✓ Higher scalability
- × Harder code to manage / debug

Coordination

Currently, we have not implemented an orchestrator – once a task is added to the list, the info / action moves from one service to the next

We will focus on **choreography** (vs. orchestration) as our fundamental coordination pattern

- ✓ Higher scalability, fault tolerance and responsiveness
- × More difficult state mgmt.

Agenda

Software Architecture

Selected Architecture Decisions

Performance & Testing

- JMeter
- ChaosMonkey

Demonstration

Key Takeaways & Contributions

JMeter | We ran the following tests

Analysis Application:
https://jmeter.streamlit.app/

Local Environment *(with docker)*

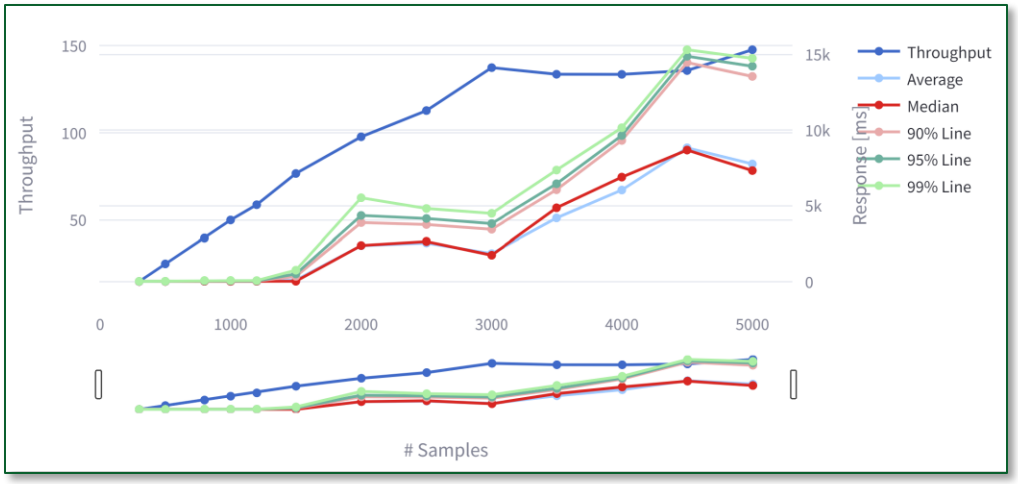
1. Throughput Analysis with POST tasks requests
   - ☆ Without cache clearance
2. Throughput Analysis with POST and GET tasks request
   - ☆ Without cache clearance
   - ☆ With cache clearance
3. Throughput Analysis with GET roster request
   - Without cache clearance

☆ *Deep-dive on next pages*

VM Environment

4. Throughput Analysis with POST and GET tasks request
   - Without cache clearance

JMeter | POST Task Request, Local, Without Cache Clearance



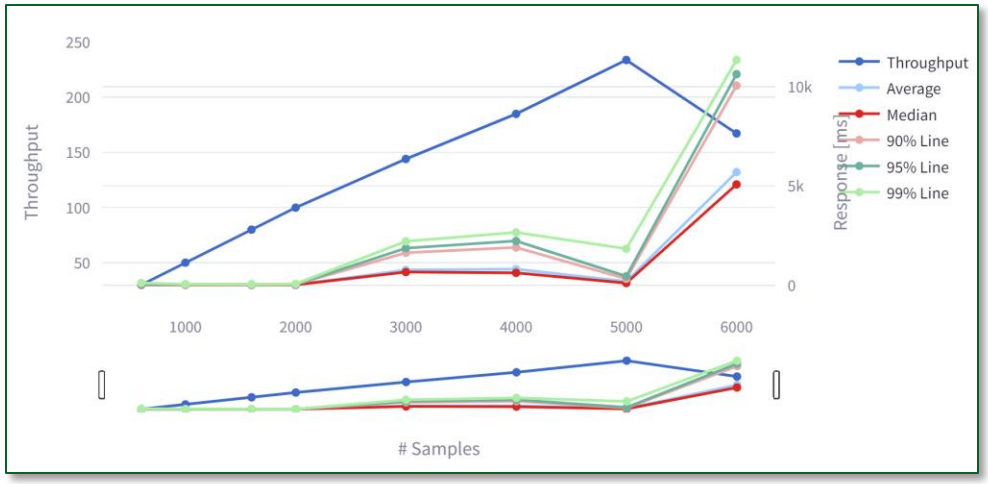
Test parameters

Threads: 300-5000 | Ramp-Up Period: 20 sec | Type: GREETING

Throughput stagnates with 3'000 threads

8% Errors with 5'000 threads, no issues before

JMeter | POST & GET Task Request, Local, Without Cache Clearance



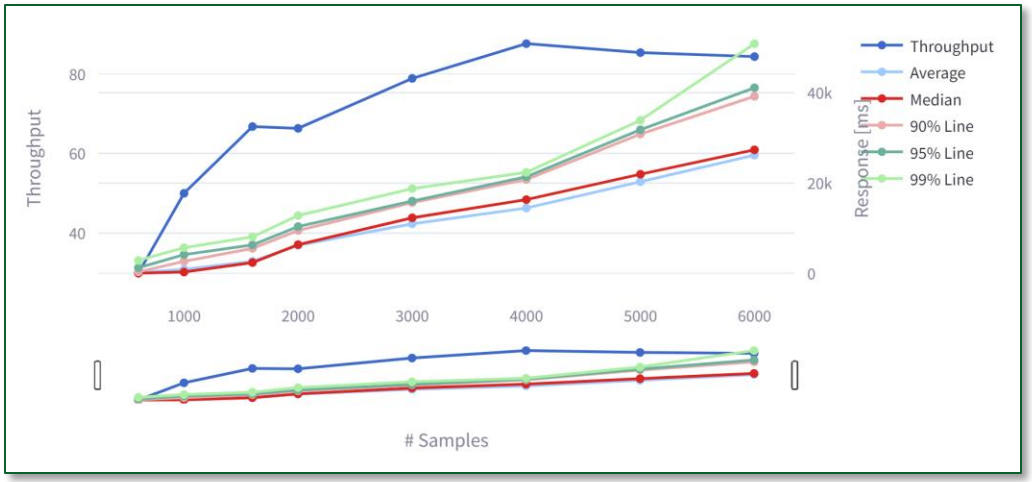
Test parameters

Threads: 300-6000 | Ramp-Up Period: 20 sec | Type: GREETING

Linear behaviour until 5'000 threads seems a bit suspicious

2.5% Errors starting at 1'600 Samples and increase fast from 4'000 samples onwards

JMeter | POST & GET Task Request, Local, With Cache Clearance



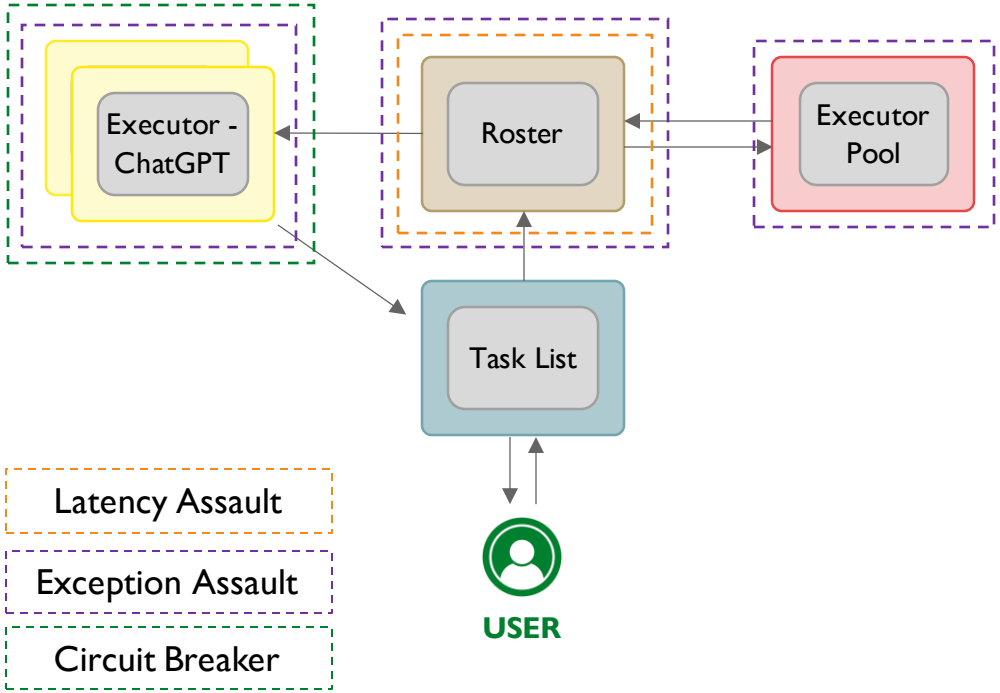
Test parameters

Threads: 300-6000 | Ramp-Up Period: 20 sec | Type: GREETING

Cache clearance before the test slows down the response time around factor 4

- Cold Start of Docker
- Database vs. Cache

ChaosMonkey | TAPAS Chaos Testing



Roster

- Latency Assault: Delays in task execution.
- Exception Assault: Delay & possible breakdown of application.
- Most vulnerable service. Robust fault-tolerance needed to prevent application breakdown.

Executor Pool

- Exception Assault: Failure to match and assign executors. Causes Task backlog. Possible eventual breakdown of application.

Executor Calculator

- Exception Assault: Failure to execute tasks. Task List is also not notified of failure.
- Circuit Breaker: Fallback method to notify task list of task execution failure.

Agenda

Software Architecture

Selected Architecture Decisions

Performance & Testing

Demonstration

Key Takeaways & Contributions

Demonstration | We will show you 2 different executors in action

Calculator

«Sassy» Coding Copilot

Agenda

Software Architecture

Selected Architecture Decisions

Performance & Testing

Demonstration

Key Takeaways & Contributions

- Troubles & Learning
- Contributions

Key Takeaways | Troubles & Learning

Troubles

- Hexagonal and the Microservices Architecture
- Deployment on Docker
- Compilation Issues
- Performance Testing, e.g., JMeter
- ….

Learning

- Communication between microservices
- Getting more familiar with testing
- Experience with non-relational databases
- Improved collaboration with GitHub / Trello
- …

Contributions | We were able to distribute the group work quite fairly within our team

	Kaan	Michael	Daniel	Stephan
Task List				
Executor Pool	✓	✓	✓	
Executors	✓			✓
Roster	✓	✓		
Testing		✓	✓	✓
Performance		✓	✓	✓
Documentation	✓		✓	✓

Any questions?