1. For the text below, summarise the problem and suggest a software system architecture solution. The main stakeholders in this case are: - Software system architects Problem, context, and related work Decomposing the system into microservices is not an easy task, especially when the resulting system needs to meet certain quality requirements. Our focus is on the performance of microservices. Creating architecture based on the microservices architectural style considers partitioning the software into parts that execute and communicate concurrently. Choosing the right granularity of partitioning is difficult, and some architectural designs might result in heavy communication between only specific microservices, creating bottlenecks for the rest of the system. System performance is a critical quality property for edge-computing applications. Thus, before implementing such applications, it is essential to predict the performance of architectural decisions in an attempt to ensure that it meets the requirements. One way to do so is to validate the suitability of the design decisions early during the design activity based on their impact on the performance. There are several ways to do so, starting from self-assessment to building prototypes. There exist approaches that statically analyse code and lift it to design. There are approaches that test performance using a large number of tests. Both have issues: imprecision of static analysis, while for the dynamic analysis, the system has to be implemented. Our goal is to assist architects in evaluating their architectural design and reason if their architectural decisions are going to have a negative effect on performance. There are no approaches to support architects to automatically evaluate the validity of their decisions during the design phase of microservices systems. To evaluate a microservices system design model and eliminate undesired anti-patterns, an architect must validate his decisions manually by comparing them with known patterns and best practices. Even though some tools can be used to automatically assess the conformance of microservices systems to known patterns and best practices and detect violations and anti-patterns, these tools depend on the existence of the source code. Some of these tools perform static analysis on the source code itself, as in (Ilaria Pigazzini, Francesca Arcelli Fontana, Valentina Lenarduzzi, and Davide Taibi. 2020. Towards microservice smells detection. In Proceedings of the 3rd International Conference on Technical Debt (TechDebt '20). Association for Computing Machinery, New York, NY, USA, 92–97), while others perform static analysis on the design models generated from the source code, as in (E. Ntentos, U. Zdun, K. Plakidas and S. Geiger, "Semi-automatic Feedback for Improving Architecture Conformance to Microservice Patterns and Practices," 2021 IEEE 18th International Conference on Software Architecture (ICSA), Stuttgart, Germany, 2021, pp. 36-46). In either case, the architect will not be able to assess the architectural model until later stages of the development life cycle, namely after the actual implementation of the system. Furthermore, existing approaches do not provide any insight into the correlation between patterns and anti-patterns on one side and the system performance on the other. Even though some researchers have reported on the harmfulness of microservices design and implementation anti-patterns, they based their assessment on the experience of interviewed practitioners, as in (D. Taibi and V. Lenarduzzi, "On the Definition of Microservice Bad Smells," in IEEE Software, vol. 35, no. 3, pp. 56-62, May/June 2018) and (Microservices Anti Patterns: A Taxonomy, Davide Taibi, Valentina Lenarduzzi, Claus Pahl). Furthermore, they have not specified the factors taken into account for the evaluation, i.e., whether the harmfulness assessment was due to the anti-pattern impact on the performance, the maintainability, or other aspects. To the best of our knowledge, no research has studied the effect of these anti-patterns on performance. Moreover, regarding testing the performance of microservices systems, to the best of our knowledge, there is no approach to provide insights into the system's expected performance in the early stages of the development life cycle. Existing performance testing approaches are designed to test the performance in the pre-production and production stages of the continuous delivery pipeline, i.e., testing is done after implementing the system, as in

(Avritzer, Alberto & Ferme, Vincenzo & Janes, Andrea & Russo, Barbara & Schulz, Henning & van Hoorn, André. (2018). A Quantitative Approach for the Assessment of Microservice Architecture Deployment Alternatives by Automated Performance Testing), (Avritzer, Alberto & Menasché, Daniel & Rufino, Vilc & Russo, Barbara & Janes, Andrea & Ferme, Vincenzo & van Hoorn, André & Schulz, Henning. (2019). PPTAM: Production and Performance Testing Based Application Monitoring), (Rolando Brondolin and Marco D. Santambrogio. 2020. A Black-box Monitoring Approach to Measure Microservices Runtime Performance. ACM Trans. Archit. Code Optim. 17, 4, Article 34 December 2020, 26 pages) and (A. Janes and B. Russo, "Automatic Performance Monitoring and Regression Testing During the Transition from Monolith to Microservices," 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Berlin, Germany, 2019, pp. 163-168). Thus, there is no mean for architects to experiment with different combinations of architectural decisions, e.g., designs, resources, and deployment strategies, to evaluate and compare their effects on the performance before actually implementing the software. The problems and limitations discussed above in the state-of-the-art identify the gap we aim to bridge in our work. Our conceptual solution aims to help architects make adequate architectural design decisions that would lead to achieving the desired performance of their microservices systems. Our goal is to provide this support early during the design phase of the system. We aim to achieve this by supporting architects in experimenting with multiple possible design models, providing insights into how different design decisions might affect the system's performance, and thus guide architects to fix design issues and to choose the design that results in the desired performance. Research Questions • RQ.01: What are the performance properties for microservices edge-based systems? • RQ.02: Is there a mapping between architectural decisions for microservices edge-based systems and performance properties? • RQ.03: Is there a mapping between microservices anti-patterns and performance properties on the edge? Requirements for the solution • R.01: The system should assist the user in evaluating the performance of the microservices edge-based SUT (System Under Test). The system should use an identified set of performance metrics relevant to microservices edge-based systems (e.g., response time, reaction time, worst-case execution time, average execution time, throughput, reliability, availability, CPU utilization, memory utilization, network utilization, overall performance under different loads) to evaluate the SUT performance. • R.02: The system should measure the values of the performance metrics for the microservices edge-based SUT. The system should test the SUT based on the performance metrics listed in R.01. • R.03: The system should display the measured numerical values of the performance metrics, listed in R.01, for the SUT. • R.04: The system should visualize the measured values of the performance metrics, listed in R.01, for the SUT in a graphical manner. The user should be able to view a graphical representation of the resulted performance evaluation (e.g., line chart). • R.05: The system should assist the user in validating the architectural decisions for the microservices edge-based SUT, based on its performance, to guide the user to choose adequate ones. The system should provide feedback about the user's architectural decisions (e.g., microservices design patterns and anti-patterns, microservices size, communication patterns, and dependencies, etc.) based on the measured performance, in R.01. • R.06: The system should analyze the measured performance properties of the SUT and interpret the results to provide hints about possible reasons in case of detected low performance. The system should be able to map performance properties to the relevant architectural decisions in order to provide the user with a set of possible reasons for a certain low-performance value (e.g., detected anti-pattern or bottleneck). • R.07: The system should guide the user to choose adequate hardware and software deployment configurations for the microservices edge-based SUT based on its performance. The user uses the system to choose the best deployment configuration (e.g., CPU allocation, memory allocation, number of container replicas assigned to each microservice, dedicated or shared edge nodes, ... etc.) for the SUT according to the performance metrics in R.01. • R.08: The system should compare the performance of different architectural alternatives of the microservices edge-based SUT. The system compares the performance of different versions of the SUT, which are built according to different architectural decisions. • R.09: The system should accept more than one SUT as an input and compare their performance. The user can test more than one alternative of the SUT at the same time. The system should present the results in an intuitive way for the user to compare the SUTs performance in order to choose the adequate one according to his/her performance criteria. • R.10: The system should be integrated into a continuous engineering approach. The system should provide continuous feedback for engineers while the SUT's source code is still being updated. • R.11: The system should be integrated into a reverse engineering approach to identify the impact of source code changes on the logical architecture and identify the parts that were affected by the changes. The system should identify which parts where affected by the source code updates in order to trigger tests for those parts only. • R.12: The system should only retest the parts of the SUT that were affected by source code updates and trigger the test only when necessary. The system should decide automatically if and which parts of the SUT should be retested upon source code update.

Problem: Architects need to ensure that the microservices systems they design meet performance requirements, but existing approaches only allow performance evaluation after the system is implemented, and there is no way to experiment with different architectural decisions beforehand.

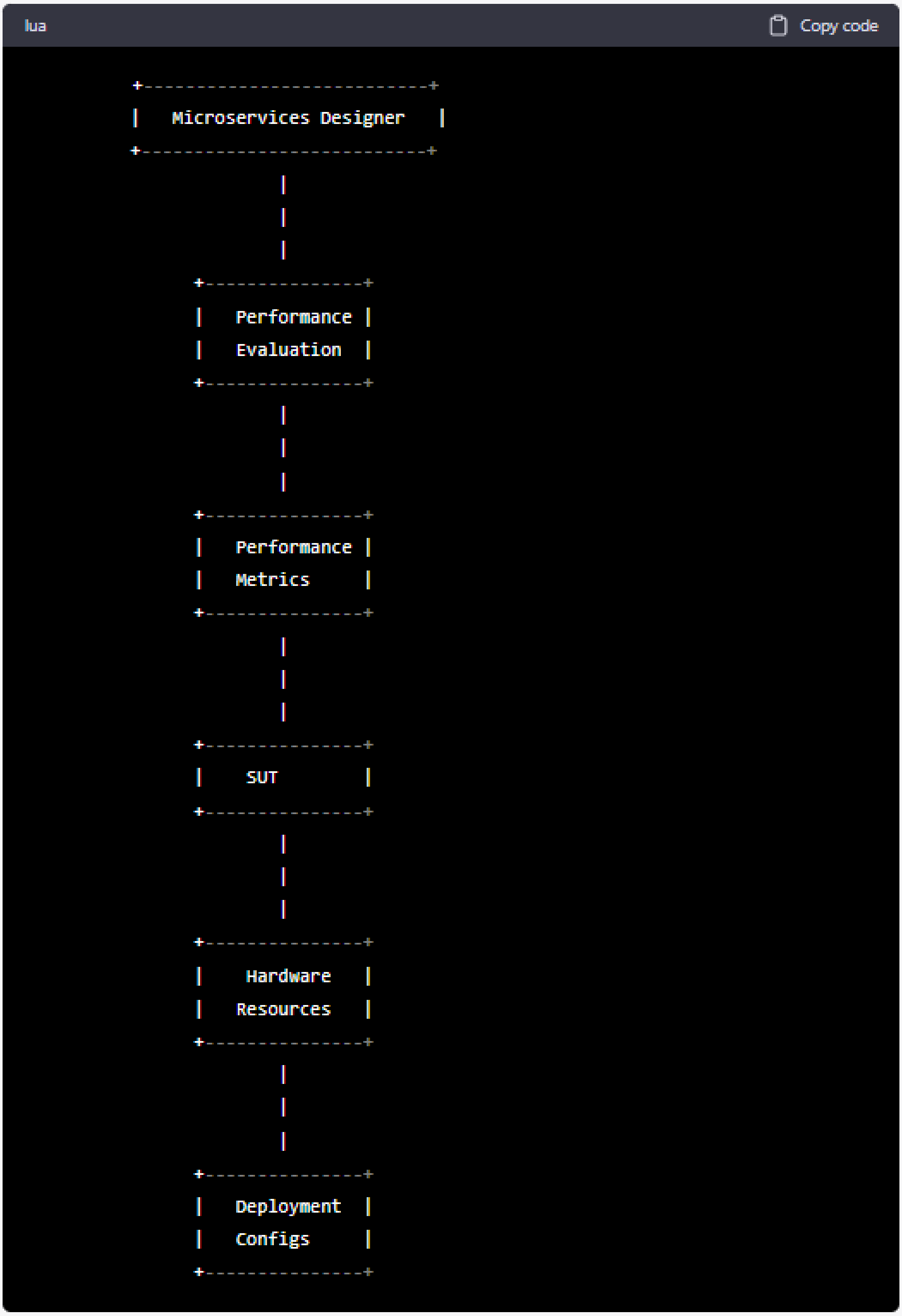
Solution: A software system architecture solution that assists architects in experimenting with different architectural decisions and evaluating their impact on performance early in the design phase of the microservices system. The system should meet the following requirements:

- Assist the user in evaluating the performance of the microservices edge-based System Under Test (SUT) using a set of relevant performance metrics (R.01).
- Measure the values of the performance metrics for the SUT (R.02).
- Display and visualize the measured performance metrics in a graphical manner (R.03, R.04).
- Assist the user in validating architectural decisions for the microservices edge-based SUT based on its performance (R.05).
- Analyze the measured performance properties of the SUT and provide hints about possible reasons for low performance (R.06).
- Guide the user to choose adequate hardware and software deployment configurations for the microservices edge-based SUT based on its performance (R.07).
- Compare the performance of different architectural alternatives of the microservices edge-based SUT (R.08).
- Accept more than one SUT as input and compare their performance (R.09).
- Be integrated into a continuous engineering approach and a reverse engineering approach to identify the impact of source code changes on the logical architecture and trigger necessary tests (R.10, R.11, R.12).

2. Suggest architecture of a conceptual solution for this problem. We need a C4 system context diagram (https://c4model.com/#SystemContextDiagram). Please use text (arrows, dots, different shapes, and different colours) to visualise the solution. Consider the following: - Identify relevant external systems (if any) and add them to the diagram. - Identify relevant external stakeholders (if any) and add them to the diagram. - Identify high level abstract data types transferred between

components and stakeholders. - Summarise data types transmitted between components. Add this information to the connections and interfaces between components.

C4 System Context Diagram:



```
          +----------------------------+
          |   Microservices Designer   |
          +----------------------------+
                        |
                        |
                        |
                +---------------+
                |   Performance |
                |   Evaluation  |
                +---------------+
                        |
                        |
                        |
                +---------------+
                |   Performance |
                |   Metrics     |
                +---------------+
                        |
                        |
                        |
                +---------------+
                |    SUT        |
                +---------------+
                        |
                        |
                        |
                +---------------+
                |    Hardware   |
                |   Resources   |
                +---------------+
                        |
                        |
                        |
                +---------------+
                |   Deployment  |
                |   Configs     |
                +---------------+
```

Legend:

- Rectangle: Software system
- Rounded rectangle: External System
- Person shape: External Stakeholder
- Arrow: Communication channel between components/stakeholders
- Different colors: Used to group components based on functionality
- Dots: Used to indicate that there are more components that could fit on the diagram

Abstract Data Types:

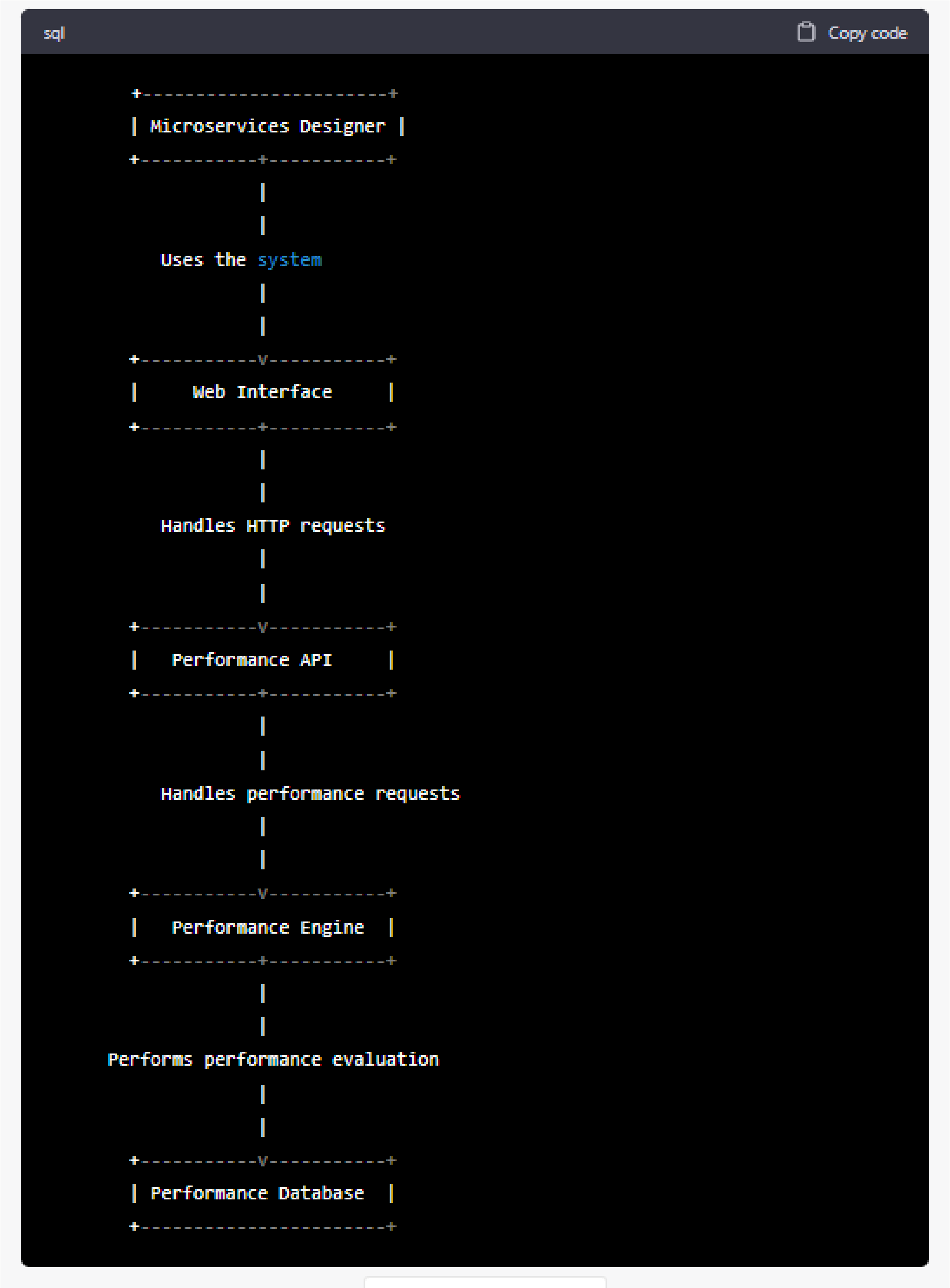
- Performance metrics: Response time, reaction time, worst-case execution time, average execution time, throughput, reliability, availability, CPU utilization, memory utilization, network utilization, overall performance under different loads.
- Deployment configs: CPU allocation, memory allocation, number of container replicas assigned to each microservice, dedicated or shared edge nodes, etc.
- SUT: System Under Test

Summary of data types transmitted between components/stakeholders:

- Microservices Designer -> Performance Evaluation: architectural decisions, microservices design patterns, microservices size, communication patterns, and dependencies.
- Performance Evaluation -> Performance Metrics: performance data (response time, throughput, etc.).
- Performance Evaluation -> SUT: configuration changes (e.g., deployment configs).
- SUT -> Performance Metrics: performance data (response time, throughput, etc.).
- Hardware Resources -> SUT: computing resources.
- Deployment Configs -> SUT: deployment configurations.

3. Suggest architectural components that we need to solve this problem. We need a C4 container solution diagram (https://c4model.com/#ContainerDiagram). Please use text (arrows, dots, different shapes, and different colours) to visualise the solution. Consider the following: - Define how the main stakeholder will use the solution. Add it to the diagram.. - Which quality properties are important? Suggest adequate architectural solutions to address them (architectural style, design patterns, design tactics, components and their relationships). Structure the components according to it. - Define data types transferred between components. - Summarise data types transmitted between components. Add this information to the connections and interfaces between components. - Define interfaces between components. Add information about required and provided interfaces to the components, along with the data types. - Document which anti-patterns we need to avoid when implementing this solution.

C4 Container Solution Diagram:



```
        +------------------------+
        | Microservices Designer |
        +-----------+-----------+
                    |
                    |
           Uses the system
                    |
                    |
        +-----------v-----------+
        |     Web Interface     |
        +-----------+-----------+
                    |
                    |
           Handles HTTP requests
                    |
                    |
        +-----------v-----------+
        |   Performance API     |
        +-----------+-----------+
                    |
                    |
           Handles performance requests
                    |
                    |
        +-----------v-----------+
        |   Performance Engine  |
        +-----------+-----------+
                    |
                    |
      Performs performance evaluation
                    |
                    |
        +-----------v-----------+
        | Performance Database  |
        +-----------------------+
```

Legend:

- Rectangle: Container
- Arrow: Communication channel between containers
- Different colors: Used to group containers based on functionality
- Dots: Used to indicate that there are more containers that could fit on the diagram

Stakeholder:

- Microservices Designer: uses the Web Interface to experiment with different architectural decisions and evaluate their impact on performance, based on feedback from the Performance Engine.

Quality Properties:

- Performance: achieved through microservices architectural style, design patterns (e.g., service discovery, circuit breaker), and design tactics (e.g., caching, load balancing).
- Maintainability: achieved through the use of appropriate design patterns and modularity.

Data Types:

- Architectural design decisions: e.g., microservices design patterns, microservices size, communication patterns, and dependencies.
- Performance metrics: response time, reaction time, worst-case execution time, average execution time, throughput, reliability, availability, CPU utilization, memory utilization, network utilization, overall performance under different loads.
- Configuration settings: e.g., deployment configs.

Summary of Data Types Transmitted Between Containers:

- Microservices Designer -> Web Interface: architectural design decisions.
- Web Interface -> Performance API: performance requests.
- Performance API -> Performance Engine: performance requests and configuration settings.
- Performance Engine -> Performance API: performance evaluation results.
- Performance Engine -> Performance Database: performance data for storage.

Interfaces:

- Web Interface:
    - Required: HTTP request interface for architectural design decisions.
    - Provided: HTTP response interface for performance requests.
- Performance API:
    - Required: HTTP request interface for performance requests.
    - Provided: HTTP response interface for performance evaluation results.
- Performance Engine:
    - Required: configuration settings interface.
    - Provided: performance evaluation results interface.
- Performance Database:
    - Required: performance data storage interface.

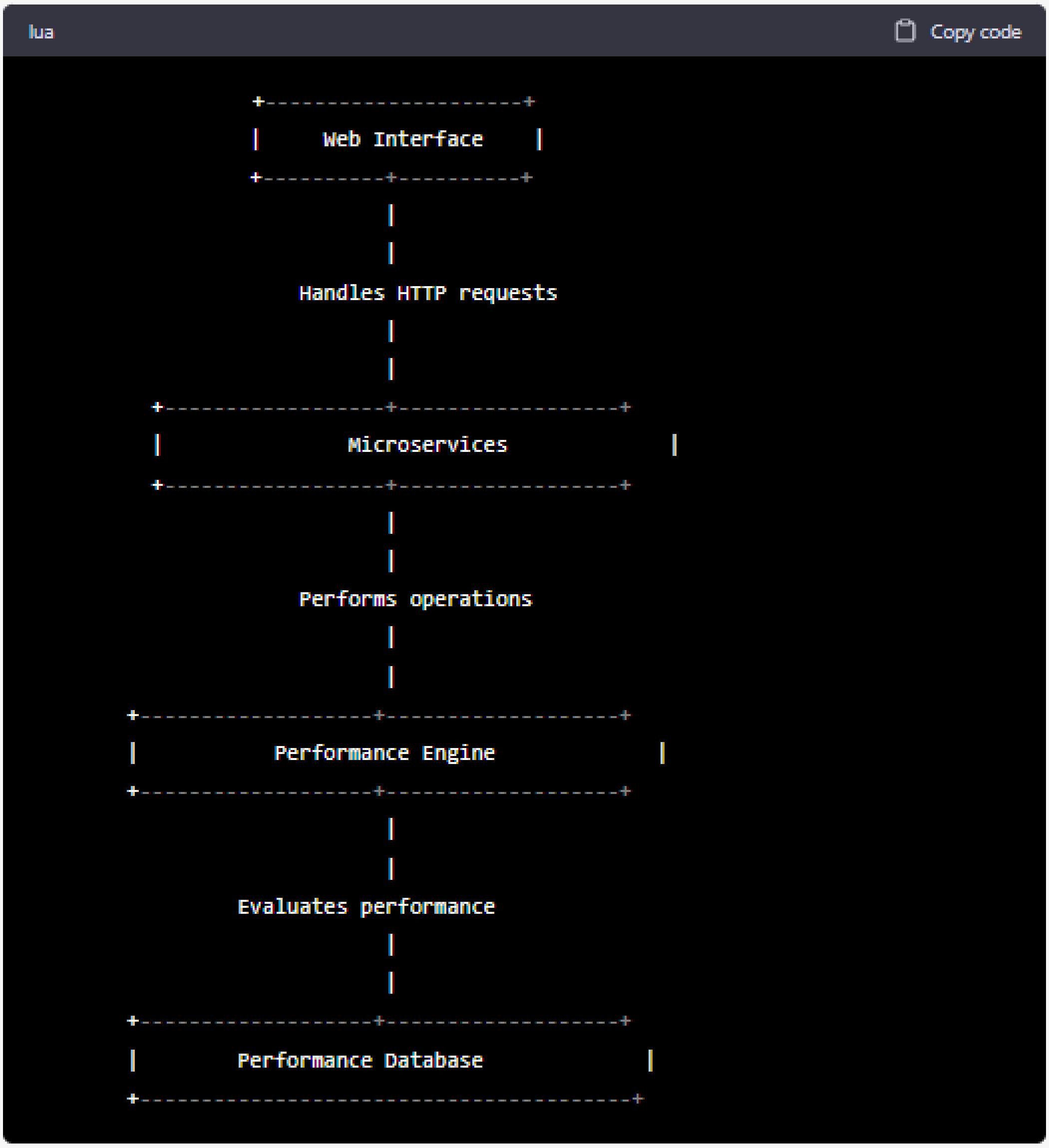
Anti-patterns to Avoid:

- Monolithic design: should use a microservices architectural style to achieve performance and maintainability goals.
- Tight coupling: should use appropriate design patterns to avoid tight coupling between microservices.

- Overly chatty communication: should use appropriate design patterns and tactics to avoid excessive communication between microservices.

4. Suggest architectural components that we need to solve this problem. We need a C4 component solution diagram (https://c4model.com/#ComponentDiagram). Please use text (arrows, dots, different shapes, and different colours) to visualise the solution. Consider following: - Define how the main stakeholder will use the solution. Add it to the diagram. - Which quality properties are important? Suggest adequate architectural solutions to address them (architectural style, design patterns, design tactics, components and their relationships). Structure the components according to it. - Define data types transferred between components. - Summarise data types transmitted between components. Add this information to the connections and interfaces between components. - Define interfaces between components. Add information about required and provided interfaces to the components, along with the data types. - Document which anti-patterns we need to avoid when implementing this solution.

C4 Component Solution Diagram:



```lua
          +----------------------+
          |     Web Interface    |
          +----------+-----------+
                     |
                     |
              Handles HTTP requests
                     |
                     |
  +------------------+------------------+
  |               Microservices             |
  +------------------+------------------+
                     |
                     |
              Performs operations
                     |
                     |
+-------------------+-------------------+
|           Performance Engine             |
+-------------------+-------------------+
                     |
                     |
         Evaluates performance
                     |
                     |
+-------------------+-------------------+
|        Performance Database             |
+----------------------------------------+
```

Legend:

- Rectangle: Component
- Arrow: Communication channel between components
- Different colors: Used to group components based on functionality
- Dots: Used to indicate that there are more components that could fit on the diagram

Stakeholder:

- Microservices Designer: uses the Web Interface to experiment with different architectural decisions and evaluate their impact on performance, based on feedback from the Performance Engine.

Quality Properties:

- Performance: achieved through microservices architectural style, design patterns (e.g., service discovery, circuit breaker), and design tactics (e.g., caching, load balancing).
- Maintainability: achieved through the use of appropriate design patterns and modularity.

Data Types:

- Architectural design decisions: e.g., microservices design patterns, microservices size, communication patterns, and dependencies.
- Performance metrics: response time, reaction time, worst-case execution time, average execution time, throughput, reliability, availability, CPU utilization, memory utilization, network utilization, overall performance under different loads.
- Configuration settings: e.g., deployment configs.

Summary of Data Types Transmitted Between Components:

- Microservices -> Performance Engine: architectural design decisions, performance metrics, configuration settings.
- Performance Engine -> Performance Database: performance data for storage.

Interfaces:

- Web Interface:
  - Required: HTTP request interface for architectural design decisions.
  - Provided: HTTP response interface for performance requests.
- Microservices:
  - Required: architectural design decisions interface.
  - Provided: operation execution interface for performing experiments.
- Performance Engine:
  - Required: configuration settings and performance metrics interface.
  - Provided: performance evaluation results interface.
- Performance Database:

- Required: performance data storage interface.

Anti-patterns to Avoid:

- Monolithic design: should use a microservices architectural style to achieve performance and maintainability goals.
- Tight coupling: should use appropriate design patterns to avoid tight coupling between microservices.
- Overly chatty communication: should use appropriate design patterns and tactics to avoid excessive communication between microservices.