

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. Ntontos, 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 were 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 face challenges in designing microservices systems that meet performance requirements, especially when it comes to choosing the appropriate granularity of partitioning, which can result in bottlenecks and anti-patterns. Existing approaches for assessing the validity of architectural decisions are limited in that they depend on the existence of the source code, making it difficult to evaluate the performance of different architectural alternatives early in the development lifecycle.

Proposed Solution: A software system architecture solution that assists architects in evaluating the performance of their microservices edge-based systems early during the design phase. The system should provide insights into the impact of different architectural decisions on performance, identify anti-patterns that negatively affect performance, and recommend hardware and software deployment configurations. It should also compare the performance of different architectural alternatives and accept multiple SUTs as input, presenting the results in an intuitive way for the user to compare and choose the best option. The system should be integrated into a continuous engineering approach and a reverse engineering approach to identify the parts of the SUT affected by source code changes and trigger tests only when necessary.

Requirements:

- R.01: The system should evaluate the SUT's performance using relevant performance metrics for microservices edge-based systems.
- R.02: The system should measure the values of the performance metrics for the SUT and test it based on the metrics listed in R.01.
- R.03: The system should display the measured numerical values of the performance metrics for the SUT.
- R.04: The system should visualize the measured values of the performance metrics for the SUT in a graphical manner.
- R.05: The system should provide feedback on the user's architectural decisions based on the measured performance.
- R.06: The system should analyze the measured performance properties of the SUT and interpret the results to provide hints about possible reasons for low performance.
- R.07: The system should guide the user in choosing appropriate hardware and software deployment configurations based on the SUT's performance.

- R.08: The system should compare the performance of different architectural alternatives for the SUT.
- R.09: The system should accept multiple SUTs as input and compare their performance.
- R.10: The system should be integrated into a continuous engineering approach.
- R.11: The system should be integrated into a reverse engineering approach to identify the parts of the SUT affected by source code changes.
- R.12: The system should trigger tests only for the parts of the SUT affected by source code changes.

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.

The proposed conceptual solution for assisting architects in evaluating the performance of their microservices edge-based systems during the design phase can be represented using a C4 system context diagram, as shown below:



In this diagram, the "Microservices Performance Tool" is the main component that assists architects in evaluating the performance of their microservices edge-based systems. It measures the performance of the system based on the relevant performance metrics for microservices edge-based systems, as defined in R.01, and displays the measured numerical values and visualizes them in a graphical manner, as specified in R.03 and R.04, respectively.

The architects use the tool to experiment with different architectural alternatives for their microservices edge-based system, as stated in R.05 and R.08. The tool provides feedback on the user's architectural decisions based on the measured performance, as defined in R.05, and analyzes the measured performance properties of the system to provide hints about possible reasons for low performance, as defined in R.06. The tool also guides the user in choosing appropriate hardware and software deployment configurations based on the system's performance, as specified in R.07.

The "Architect 1-2-N" components represent the external stakeholders who use the tool to evaluate the performance of their microservices edge-based system during the design phase. They experiment with different architectural alternatives using the tool and use its feedback to make informed decisions about the system's architecture.


The "Microservices Edge-Based System 1-2-N" components represent the external system that the architects are designing. The tool measures its performance based on the performance metrics listed in R.01 and provides feedback on the user's architectural decisions based on the measured performance.

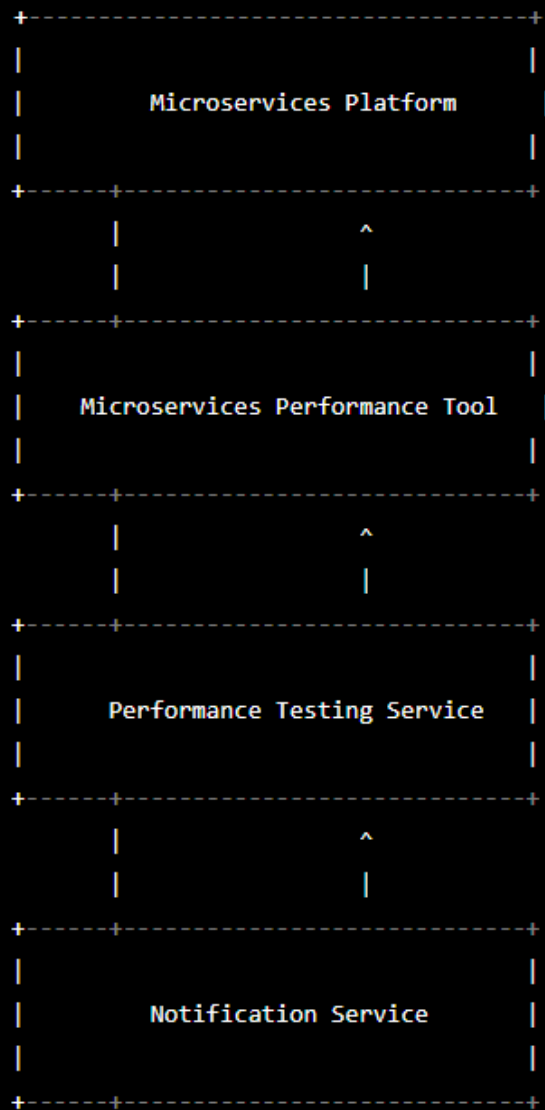
The relevant external systems and stakeholders are included in the diagram as separate components. The high-level abstract data types transferred between components and stakeholders are the microservices system design models, the measured performance metrics, and the feedback on the user's architectural decisions. These data types are transmitted between the architects and the Microservices Performance Tool component. This information is included in the connections and interfaces between the components.

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.

The proposed solution can be further detailed using a C4 container diagram, as shown below:

lua

 Copy code



The main stakeholder of the solution, the software system architects, will use the Microservices Performance Tool to evaluate and experiment with different microservices architectures during the design phase. The tool will provide feedback on the architects' decisions based on measured performance metrics.

The following quality properties are important for the solution:

- Performance
- Scalability
- Maintainability

To address these quality properties, the solution will use the microservices architectural style, which will allow for independent deployment and scaling of microservices. The system will be designed using best practices and design patterns, such as the Circuit Breaker pattern and the

API Gateway pattern, to enhance the system's performance and scalability. The solution will also use design tactics, such as caching and load balancing, to further improve the system's performance and scalability.

The components of the system will include the Microservices Platform, which will provide the infrastructure for deploying and running microservices. The Performance Testing Service will measure the performance of the microservices system and provide the measured performance metrics to the Microservices Performance Tool. The Notification Service will notify the architects of any performance issues detected during testing.

The data types transferred between components will include the microservices system design models, the measured performance metrics, and notifications of performance issues.

The Microservices Performance Tool will require an interface for receiving measured performance metrics from the Performance Testing Service and an interface for providing feedback on the architects' design decisions based on the measured performance metrics. The Performance Testing Service will require an interface for receiving the microservices system design model and an interface for providing the measured performance metrics. The Notification Service will require an interface for sending notifications to the architects.

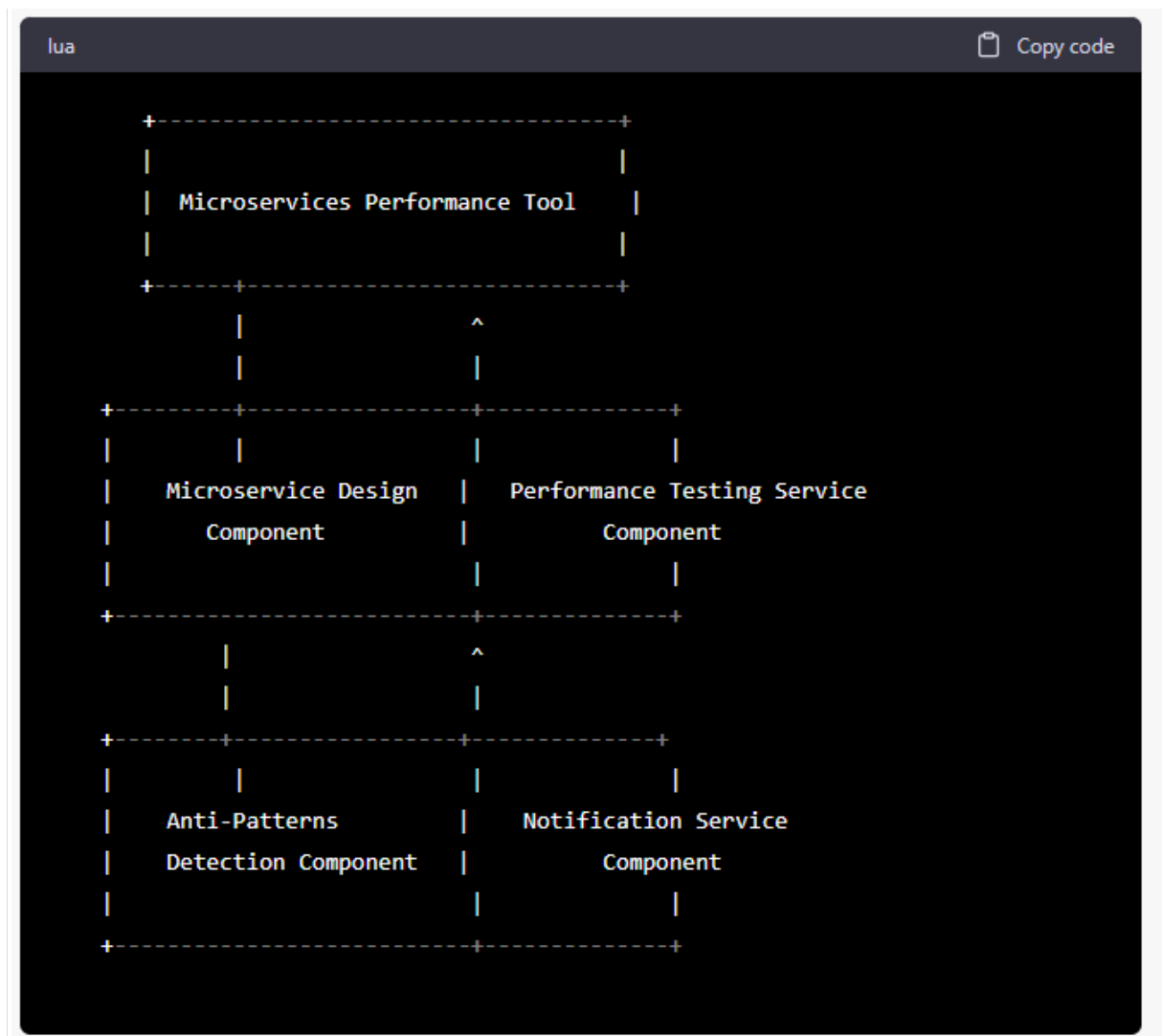
The anti-patterns to avoid when implementing this solution include the following:

- Monolithic architecture
- Tight coupling between microservices
- Heavy communication between microservices
- Lack of fault tolerance and resiliency in 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.

To further detail the proposed solution, a C4 component diagram is shown below:



The main stakeholder, the software system architects, will use the Microservices Performance Tool to experiment with different microservices architectures during the design phase. The Microservice Design Component will provide the tool with microservice design models. The Performance Testing Service Component will measure the performance of the microservices system and provide the measured performance metrics to the Microservices Performance Tool. The Anti-Patterns Detection Component will detect anti-patterns in the microservices design and provide feedback to the Microservices Performance Tool. The Notification Service Component will notify the architects of any performance issues detected during testing.

The following quality properties are important for the solution:

- Performance
- Scalability
- Maintainability

To address these quality properties, the solution will use the microservices architectural style, which will allow for independent deployment and scaling of microservices. The system will be designed using best practices and design patterns, such as the Circuit Breaker pattern and the API Gateway pattern, to enhance the system's performance and scalability. The solution will also

use design tactics, such as caching and load balancing, to further improve the system's performance and scalability.

The data types transferred between components will include the microservices system design models, the measured performance metrics, notifications of performance issues, and detected anti-patterns.

The Microservices Performance Tool will require interfaces for receiving microservice design models from the Microservice Design Component, measured performance metrics from the Performance Testing Service Component, detected anti-patterns from the Anti-Patterns Detection Component, and notifications of performance issues from the Notification Service Component. The Performance Testing Service Component will require interfaces for receiving microservice design models from the Microservice Design Component and providing measured performance metrics to the Microservices Performance Tool. The Anti-Patterns Detection Component will require an interface for receiving microservice design models from the Microservice Design Component and providing detected anti-patterns to the Microservices Performance Tool. The Notification Service Component will require an interface for sending notifications to the architects.

The anti-patterns to avoid when implementing this solution include the following:

- Monolithic architecture
- Tight coupling between microservices
- Heavy communication between microservices
- Lack of fault tolerance and resiliency in microservices.

5. Please suggest a toolchain to implement the solution. Consider which tools we can reuse. For the components we need to develop, please suggest a programming language. Please use text (arrows and dots) to visualise the solution. Define data types transferred between components. Summarise data types transmitted between components. Add this information to the connections and interfaces between components.

To implement the solution, we can reuse existing tools and libraries where possible. The following toolchain can be used:

1. Programming language: Java
2. Spring Boot framework: for building and deploying microservices
3. Docker: for containerization and deployment of microservices
4. Kubernetes: for managing the deployment of microservices and scaling them based on demand
5. Prometheus: for collecting metrics data from microservices
6. Grafana: for visualizing the metrics data collected by Prometheus
7. Git: for version control and continuous integration and deployment
8. Jenkins: for automating the build, test, and deployment processes

The components we need to develop include:

1. Performance Testing Service: responsible for measuring the performance of the microservices under test and collecting the performance metrics data.
2. Performance Metrics Storage Service: responsible for storing the performance metrics data collected by the Performance Testing Service.
3. Performance Metrics Visualization Service: responsible for visualizing the performance metrics data collected by the Performance Testing Service using Grafana.
4. Architecture Validation Service: responsible for validating the architectural decisions made by the software system architects and providing feedback on how these decisions may impact the performance of the microservices.
5. Deployment Configuration Service: responsible for assisting software system architects in choosing the best deployment configuration for the microservices based on their performance metrics data.

The data types transferred between components include:

1. Performance Metrics Data: includes performance metrics such as response time, reaction time, worst-case execution time, average execution time, throughput, reliability, availability, CPU utilization, memory utilization, network utilization, and overall performance under different loads.
2. Architecture Design Data: includes microservices design patterns and anti-patterns, microservices size, communication patterns, and dependencies, etc.
3. Deployment Configuration Data: includes deployment configuration data such as CPU allocation, memory allocation, number of container replicas assigned to each microservice, dedicated or shared edge nodes, etc.

The summarized data types transmitted between components are:

1. Performance Testing Service -> Performance Metrics Storage Service: Performance Metrics Data
2. Performance Testing Service -> Performance Metrics Visualization Service: Performance Metrics Data
3. Architecture Validation Service -> Performance Metrics Storage Service: Performance Metrics Data
4. Deployment Configuration Service -> Performance Metrics Storage Service: Performance Metrics Data
5. Architecture Validation Service <-> Software System Architects: Architecture Design Data
6. Deployment Configuration Service <-> Software System Architects: Deployment Configuration Data

The interfaces between components and the required/provided data types are:

1. Performance Testing Service:
 - Required Interface: None
 - Provided Interface: Performance Metrics Data
2. Performance Metrics Storage Service:
 - Required Interface: Performance Metrics Data
 - Provided Interface: None
3. Performance Metrics Visualization Service:
 - Required Interface: Performance Metrics Data

- Provided Interface: None
- 4. Architecture Validation Service:
 - Required Interface: Architecture Design Data, Performance Metrics Data
 - Provided Interface: None
- 5. Deployment Configuration Service:
 - Required Interface: Deployment Configuration Data, Performance Metrics Data
 - Provided Interface: None
- 6. Software System Architects:
 - Required Interface: Architecture Design Data, Deployment Configuration Data
 - Provided Interface: None

To implement the solution, we can use Java as the programming language since it is widely used and has many libraries and frameworks available. Spring Boot can be used to build and deploy microservices, while Docker and Kubernetes can be used for containerization and deployment. Prometheus can be used to collect performance metrics data, while Grafana can be used to visualize this data. Git can be used for version control, and Jenkins can be used for continuous integration and deployment.