

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 - DevOps engineers - Developers

Problem, context, and related work

Microservices architecture is an approach to developing software applications as a group of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API (M. Fowler. (2014). *Microservices*, a definition of this new architectural term, [Online]. Available: <https://martinfowler.com/articles/microservices.html>). Development teams build these services around business capabilities and deployable by automated deployment machinery. Developers may write these services in different programming languages and may use different data storage and retrieval technologies. Different development teams may develop and maintain these services. Microservices architecture has become popular in the industry because of its benefits, such as promoting scalability (W. Hasselbring, "Microservices for scalability: Keynote talk abstract", Mar. 2016), agility and reliability in software systems (W. Hasselbring and G. Steinacker, "Microservice architectures for scalability, agility and reliability in e-commerce", in 2017). Microservices are gaining momentum across the industry. The International Data Corporation (X. Larrucea, I. Santamaria, R. Colomo-Palacios, and C. Ebert, "Microservices", *IEEE Software*, vol. 35, no. 3, pp. 96–100, 2018) predicts that by the end of 2021, 80% of the cloud-based systems will use Microservices. By decomposing software systems into microservices, it becomes important to use a software development process that supports achieving the goals of the architecture. DevOps as a process supports the development and deployment of independent modules and components while automating repetitive tasks. DevOps process serves the goals of microservices architecture by supporting parallel development and reducing operational overhead. DevOps is a set of practices that aim to decrease the time between changing a system and transferring that change to the production environment. It also insists on maintaining software quality for both code and the delivery mechanisms. DevOps considers any technique that enables these goals to be part of the process (A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables devops: Migration to a cloud-native architecture", *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.). One practice that is used to realize DevOps process is Continuous Integration (CI). Continuous integration is a software development practice where members of a team integrate their work frequently (i.e., several commits to a code repository and integrations per day) (M. Fowler. (2006). *Continuous integration*, [Online]. Available: <https://martinfowler.com/articles/continuousIntegration.html>). An automated build process verifies each integration (including test) to automatically detect integration errors. Many teams find that the approach of frequent integrations verified by automated tests leads to significantly reduced integration problems and allows a team to develop cohesive software faster. Some architecture styles are more suitable for continuous engineering than others. In systems with microservices, architecture development teams, by nature of this architectural style, are encouraged to work more independently. Continuous Integration process plays a central role in DevOps process. Microservices' architecture produces services that are developed and deployed independently. The architecture affects the way these services are integrated and how the code is managed between different development iterations. Because of that direct effect, system designers and developers rely on known architectural design patterns as trusted solutions for common challenges and problems. We aim to use architectural design patterns to improve the process of continuous integration. While performing continuous integration, challenges may appear. These challenges have a negative impact on the process of continuous integration. This impact can be small or it can be preventive for the whole continuous integration process. This impact may increase build time or increase the time that the team requires analyzing and resolving problems. The negative impact of these challenges may slow down the entire development process. When the team faces issues within the continuous integration process,

finding the causes of these problems starts. With consideration of the boundaries of different stages of DevOps life cycle, identifying the root causes of these problems is limited within these stages. System designers and developers identify the root cause of continuous integration problems at a later point in time during the project or product development lifespan. There is no formal process that enables tracing the design decisions that are problematic in the context of continuous integration. Teams depend on their knowledge, experience and sometimes trial and error to identify the design decisions that are causing problems for continuous integration. Another problem is that there is no formal process on how problematic design decisions can be changed to better suit continuous integration. In the context of information system, a team is responsible for developing a software solution. The team follows DevOps as process with four weeks sprint. The code base is maintained in code versioning repository and several code integrations are done every day. The product is released into test environment every two four weeks and to production every two months. The activity of integrating changes into the code base is taking longer time and code reviews sessions are conducted to optimize the output from developers and architectures to further support the continuous integration process. These sessions are becoming more frequent and more demanding in terms of time. The team aims to reduce the time to market by reducing the time needed to complete the sprints.

Research Questions • Ch1 Documenting Design decisions and their impact on Continuous Integration. While exploring the patterns and anti-patterns for Req.3 and Req.4, it is clear that the identified patterns and anti-patterns span over different aspects of the software development process. The patterns can be categorized into the following categories (C. Richardson, Microservices Patterns: With examples in Java. Manning Publications, 2018) based on their impact and application: • Application architecture patterns (Microservices or Monolithic) • Data management patterns • Testing patterns • Deployment patterns • External API patterns • Service discovery patterns • Reliability patterns • Observability patterns However, one key aspect is lacking from traceability point of view: design decisions are made during system design stage or during implementation stages. These decisions are made without visibility of their impact on the continuous integration process that takes place just after implementation stage. During a key stage of the software development process, the gap that exist is capturing and documenting these design decisions in the form of patterns and anti-patterns along with their impact on continuous integration. Such a documentation must clearly state the positive or negative consequences of using these patterns and anti-patterns on the continuous integration process.

• Ch2 Extending the DevOps process to utilize design patterns and anti-patterns. While reviewing literature for answers in response to Req. 6, it is clear that although there are few mechanisms to derive numerical indicators of the architectural smells which can be derived automatically from code, the establishment and utilization of these indicators is a decision left to the team to make. There is a gap of not having a standard process that utilizes design decisions -in the form of patterns and anti-patterns- to improve continuous integration process. By having an extended process as standard, improving the continuous integration process becomes part of the DevOps process, and that is enforced by using design patterns and anti patterns.

Requirements for the solution • Req. 1: Shorten the current release cycle: The goal here is to shorten the internal release cycle from two weeks to one week to receive feedback on changes more frequently and rapidly. • Req. 2: Increase the frequency of merging changes into the code repository, which reduces the number of merge conflicts. • Req. 3: Identify and enforce using architectural patterns that support the DevOps operations (e.g. using external configurations of microservices or Blue-Green Deployment Pattern) to increase teams' velocity. • Req. 4: Identify and avoid the anti-patterns that the teams are using which minimize the benefits of DevOps processes (e.g. to stop using continuous build anti pattern or to avoid selective automation for developed features). • Req. 5: to promote automated testing among the team, that is by using patterns that support and simplify

automated testing (e.g. using page objects to model web pages) and avoid using anti patterns that hinder automated testing. • Req. 6: Extend the currently used DevOps process and propose an extended process that utilizes the acquired knowledge of patterns and anti patterns and enforces using the knowledge as part of the process. • Req. 7: to support auto provisioning and de-provisioning of services, these services should aim to be stateless and be treated as transient. In case a service has to maintain state, it should maintain it in external persistent storage.

Problem: Challenges in implementing a microservices architecture and DevOps process, resulting in slow continuous integration and prolonged code review sessions, hindering the team's goal of shortening the release cycle and increasing velocity.

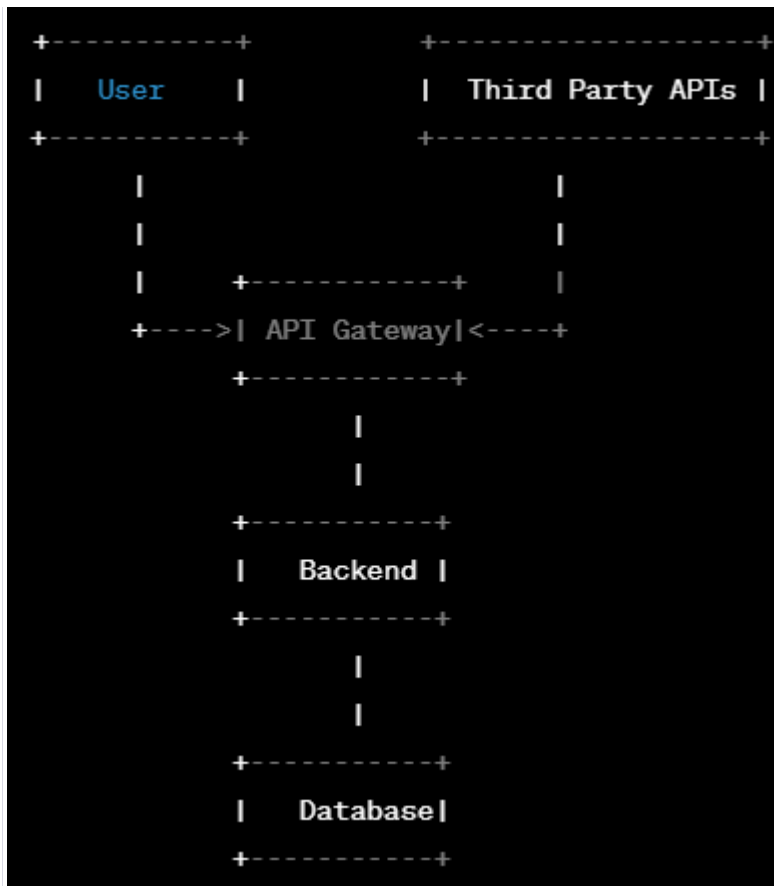
Solution: A software system architecture solution that addresses the challenges of implementing a microservices architecture and DevOps process and supports the team's requirements. The solution should include:

- Documentation of design decisions and their impact on continuous integration, including patterns and anti-patterns.
- Utilization of design patterns and anti-patterns to improve continuous integration and DevOps process.
- Shorten the release cycle from two weeks to one week.
- Increase the frequency of merging changes into the code repository.
- Identification and enforcement of architectural patterns that support DevOps operations and avoidance of anti-patterns.
- Promotion of automated testing through the use of patterns that support and simplify automated testing and avoidance of anti-patterns that hinder it.
- Extension of the DevOps process to utilize acquired knowledge of patterns and anti-patterns and enforce their use.
- Support for auto-provisioning and de-provisioning of services, aiming for stateless services and external persistent storage for stateful services.

The solution can be implemented through the use of appropriate tools and technologies, including automated testing tools, continuous integration tools, and containerization technologies such as Docker and Kubernetes. The system architecture should include service discovery mechanisms, load balancers, and monitoring tools to support scalability, reliability, and observability. The architecture should also be modular, allowing for independent development and deployment of microservices, and flexible enough to accommodate changes and updates in the future.

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.

Based on the problem and requirements described in question 1, the following is a proposed solution architecture in the form of a C4 system context diagram:



In this architecture, the user interacts with the system through the API Gateway component, which acts as a facade for the backend microservices. The API Gateway is responsible for routing requests to the appropriate microservice and returning responses to the user.

The Backend component contains the microservices that make up the system. These microservices are responsible for handling specific business capabilities and communicating with each other through lightweight mechanisms, such as HTTP resource APIs. The microservices are developed and deployed independently, and they use different programming languages and data storage technologies.

The Database component represents the persistent data storage used by the system. It can be a relational or non-relational database, depending on the specific needs of the microservices.

The system context also includes third-party APIs, which are used by the backend microservices to perform specific functions, such as sending email or processing payments.

The high-level abstract data types transmitted between components and stakeholders are:

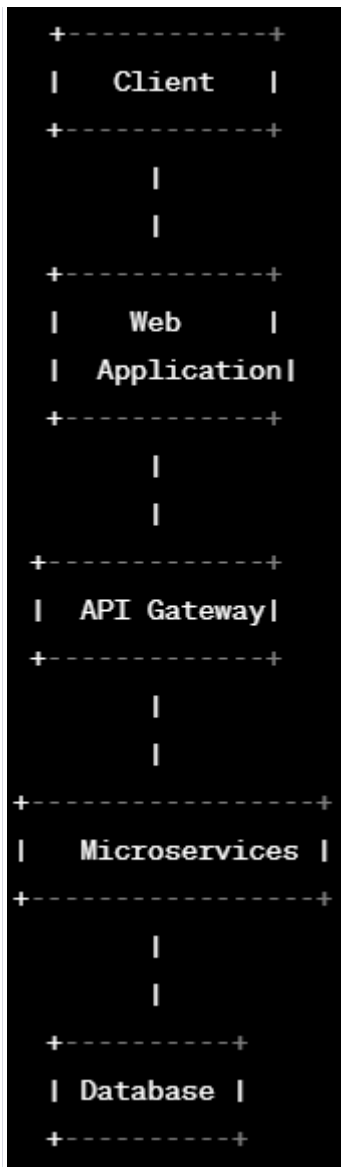
- User: JSON or XML data representing user requests and responses
- API Gateway: JSON or XML data representing the requests and responses between the user and the backend microservices
- Backend Microservices: JSON or XML data representing the requests and responses between the microservices, as well as any data processed or stored by the microservices

- Database: Structured data such as SQL or NoSQL data formats representing the data stored in the database
- Third-Party APIs: JSON or XML data representing the requests and responses between the microservices and the third-party APIs.

The connections between the components indicate the interfaces through which data is transmitted. The connections between the User and the API Gateway, as well as the connections between the API Gateway and the Backend Microservices, transmit JSON or XML data. The connections between the Backend Microservices and the Database transmit structured data, and the connections between the Backend Microservices and Third-Party APIs transmit JSON or XML data.

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.

Based on the problem and requirements described in question 1, the following is a proposed solution architecture in the form of a C4 container diagram:



The main stakeholder in this solution is the development team responsible for building, testing, and deploying the microservices-based application. They will use the solution to develop, deploy, and maintain microservices that can be integrated seamlessly using a DevOps process with Continuous Integration.

The important quality properties for this solution are scalability, reliability, maintainability, and testability. To address these, we suggest using a microservices architecture style and design patterns such as External Configuration for Microservices, Circuit Breaker pattern, and Blue-Green Deployment pattern. Additionally, we can use containerization technologies such as Docker and Kubernetes to manage and deploy the microservices.

The data types transferred between components include JSON or XML data representing the requests and responses between the client and the web application, the API Gateway and the microservices, and structured data such as SQL or NoSQL data formats representing the data stored in the database.

The connections between the components indicate the interfaces through which data is transmitted. The connections between the client and the web application, as well as the connections between the API Gateway and the microservices, transmit JSON or XML data. The connections between the microservices and the database transmit structured data.

The interfaces between the components are defined as follows:

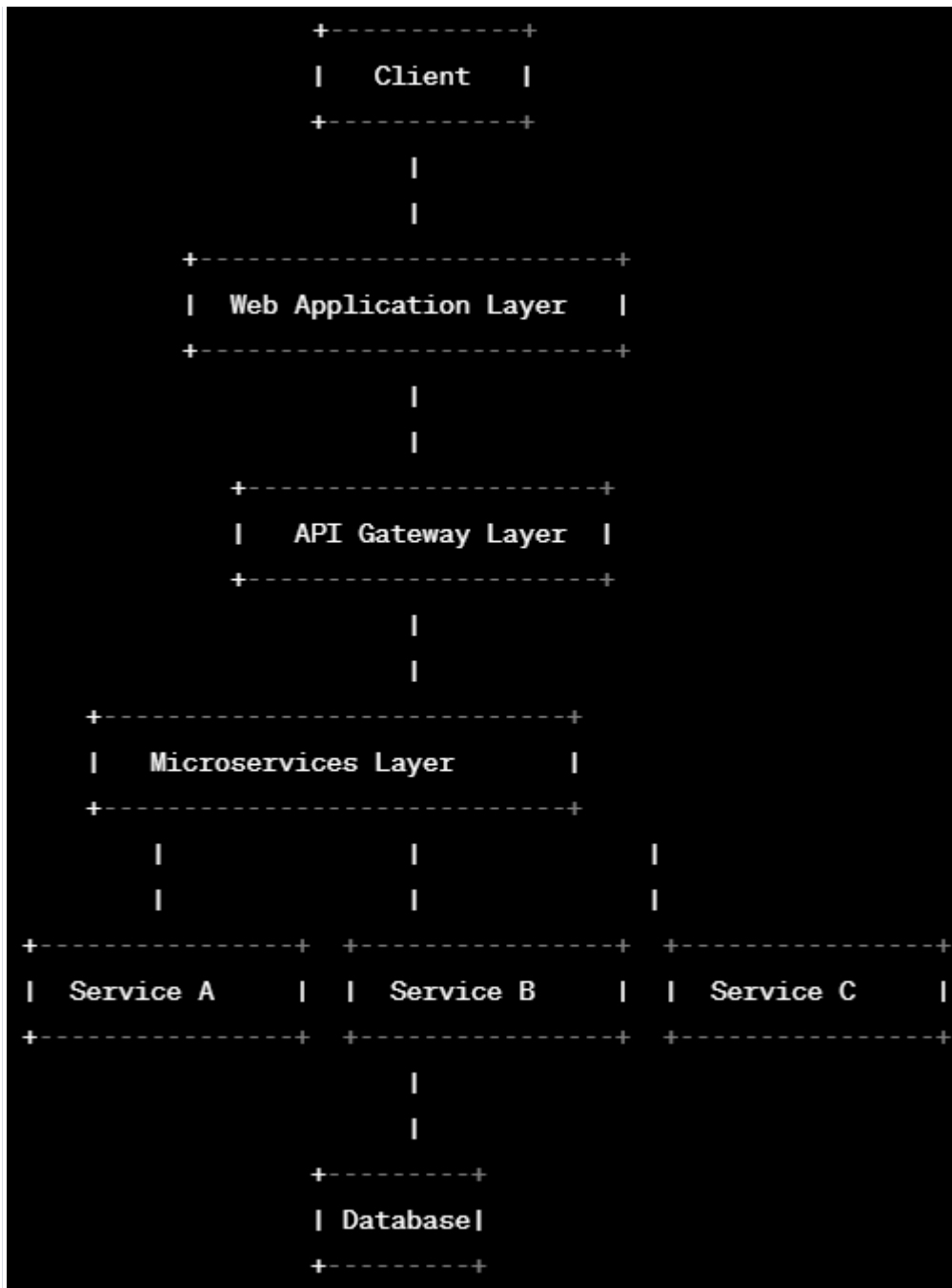
- Client: Provides user interface, consumes data from the web application, and provides JSON or XML data.
- Web Application: Provides interface for the client, consumes data from the API Gateway, and provides JSON or XML data.
- API Gateway: Provides interface for the web application and the microservices, consumes JSON or XML data from the web application, and provides JSON or XML data to the microservices.
- Microservices: Provides interface for the API Gateway, consumes JSON or XML data from the API Gateway, and provides structured data to the database.
- Database: Provides interface for the microservices, consumes structured data from the microservices.

To avoid anti-patterns when implementing this solution, we need to avoid using Continuous Build Anti-Pattern and selective automation for developed features. We should also avoid tightly coupling the microservices, as this can lead to increased complexity and hinder scalability. Instead, we should design the microservices to be loosely coupled and independent, promoting easy integration and independent deployment.

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.

Based on the problem and requirements described in question 1, the following is a proposed solution architecture in the form of a C4 component diagram:



The main stakeholder in this solution is the development team responsible for building, testing, and deploying the microservices-based application. They will use the solution to develop, deploy, and maintain microservices that can be integrated seamlessly using a DevOps process with Continuous Integration.

The important quality properties for this solution are scalability, reliability, maintainability, and testability. To address these, we suggest using a microservices architecture style and design patterns such as External Configuration for Microservices, Circuit Breaker pattern, and Blue-Green Deployment pattern. Additionally, we can use containerization technologies such as Docker and Kubernetes to manage and deploy the microservices.



The data types transferred between components include JSON or XML data representing the requests and responses between the client and the web application, the API Gateway and the microservices, and structured data such as SQL or NoSQL data formats representing the data stored in the database.

The connections between the components indicate the interfaces through which data is transmitted. The connections between the client and the web application, as well as the connections between the API Gateway and the microservices, transmit JSON or XML data. The connections between the microservices and the database transmit structured data.

The interfaces between the components are defined as follows:

- Client: Provides user interface, consumes data from the web application, and provides JSON or XML data.
- Web Application Layer: Provides interface for the client, consumes data from the API Gateway Layer, and provides JSON or XML data.
- API Gateway Layer: Provides interface for the web application layer and the microservices layer, consumes JSON or XML data from the web application layer, and provides JSON or XML data to the microservices layer.
- Microservices Layer: Provides interface for the API Gateway layer, consumes JSON or XML data from the API Gateway layer, and provides structured data to the database.
- Microservices: Provides specific functionality, consumes JSON or XML data from the microservices layer, and provides structured data to the microservices layer.
- Database: Provides interface for the microservices layer, consumes structured data from the microservices layer.

To avoid anti-patterns when implementing this solution, we need to avoid using Continuous Build Anti-Pattern and selective automation for developed features. We should also avoid tightly coupling the microservices, as this can lead to increased complexity and hinder scalability. Instead, we should design the microservices to be loosely coupled and independent, promoting easy integration and independent deployment. Additionally, we should avoid using anti-patterns such as God Component and Big Ball of Mud.

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 proposed in the previous questions, we can use a variety of tools and technologies. Here is a suggested toolchain for this solution:

- Programming Language: We can use a variety of programming languages for the microservices layer, depending on the preferences of the development team. Some popular options include Java, Python, and Node.js.
- Frameworks: We can use frameworks such as Spring Boot or Flask to simplify development and deployment of the microservices.
- Containerization: We can use Docker to containerize the microservices and Kubernetes to manage and deploy the containers.

- API Gateway: We can use an open-source API Gateway such as Kong or Tyk to manage the microservices and provide a unified interface to the web application layer.
- Database: We can use a variety of databases for the database layer, depending on the specific requirements of the application. Some popular options include MySQL, PostgreSQL, and MongoDB.
- Testing: We can use testing frameworks such as JUnit or pytest to write automated tests for the microservices.

The data types transferred between components include JSON or XML data representing the requests and responses between the client and the web application, the API Gateway and the microservices, and structured data such as SQL or NoSQL data formats representing the data stored in the database.

The connections between the components indicate the interfaces through which data is transmitted. The connections between the client and the web application, as well as the connections between the API Gateway and the microservices, transmit JSON or XML data. The connections between the microservices and the database transmit structured data.

The interfaces between the components are defined as follows:

- Client: Provides user interface, consumes data from the web application, and provides JSON or XML data.
- Web Application Layer: Provides interface for the client, consumes data from the API Gateway Layer, and provides JSON or XML data.
- API Gateway Layer: Provides interface for the web application layer and the microservices layer, consumes JSON or XML data from the web application layer, and provides JSON or XML data to the microservices layer.
- Microservices Layer: Provides interface for the API Gateway layer, consumes JSON or XML data from the API Gateway layer, and provides structured data to the database.
- Microservices: Provides specific functionality, consumes JSON or XML data from the microservices layer, and provides structured data to the microservices layer.
- Database: Provides interface for the microservices layer, consumes structured data from the microservices layer.