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.

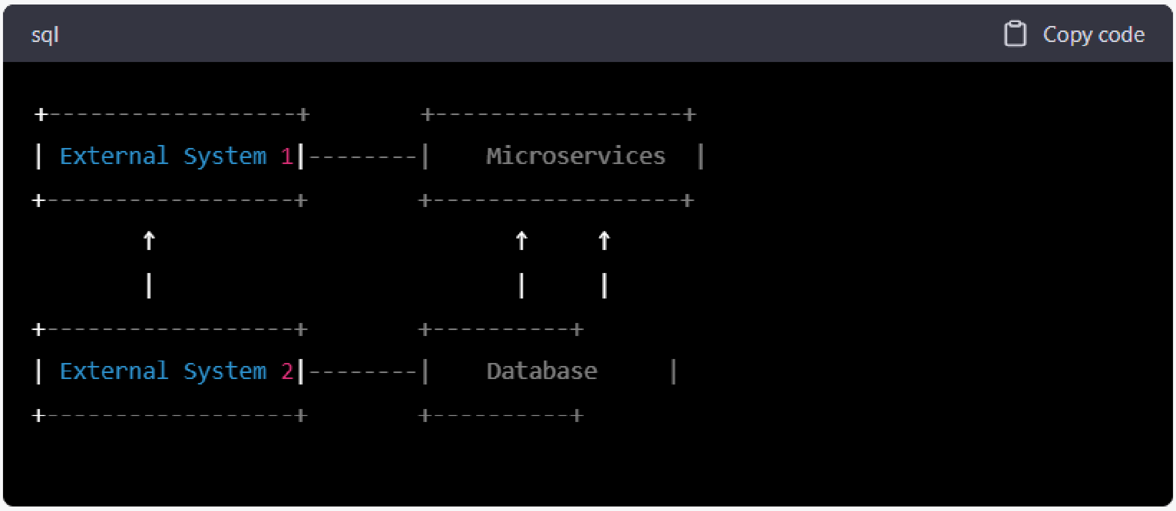
Problem: Teams using microservices architecture face challenges in the continuous integration process, including identifying problematic design decisions and changing them to better suit continuous integration.

Solution: A software system architecture solution would involve documenting design decisions and their impact on continuous integration, as well as extending the DevOps process to utilize design patterns and anti-patterns. The solution should meet the following requirements:

- Req. 1: Shorten the release cycle from two weeks to one week
- Req. 2: Increase the frequency of merging changes into the code repository
- Req. 3: Identify and enforce using architectural patterns that support DevOps operations
- Req. 4: Identify and avoid anti-patterns that minimize the benefits of DevOps processes
- Req. 5: Promote automated testing using patterns that support and simplify automated testing and avoid anti-patterns that hinder automated testing
- Req. 6: Extend the DevOps process to utilize design patterns and anti-patterns
- Req. 7: Support auto-provisioning and de-provisioning of services, with stateless services and external persistent storage for services that maintain state.

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 for a conceptual solution:

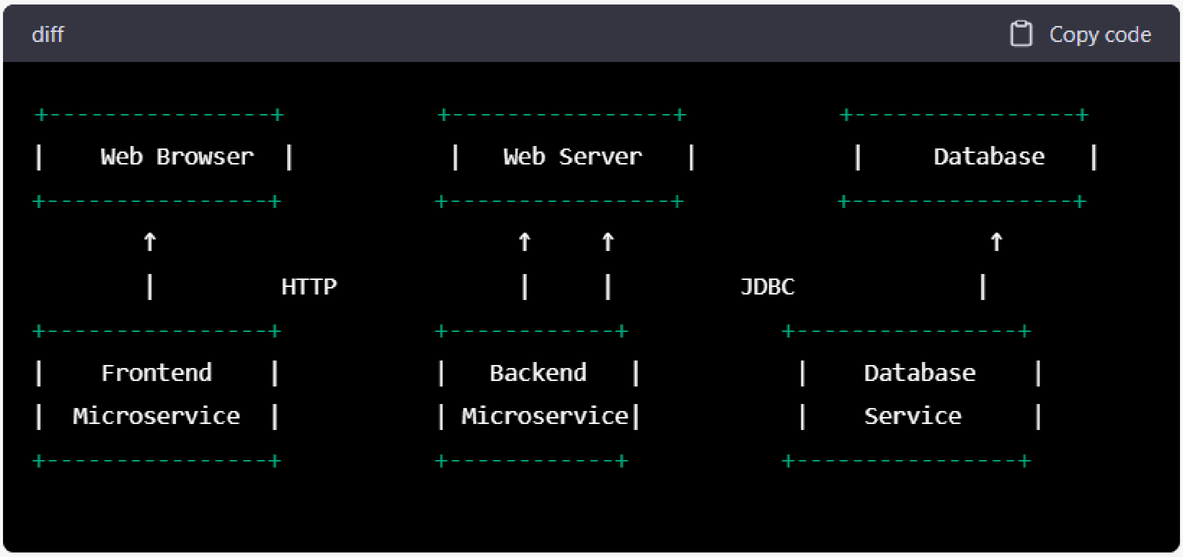


```
+-------------------+         +-------------------+
| External System 1|---------|   Microservices   |
+-------------------+         +-------------------+
        ↑                            ↑     ↑
        |                            |     |
+-------------------+         +----------+
| External System 2|---------|   Database       |
+-------------------+         +----------+
```

- Relevant external systems: External System 1, External System 2
- Relevant external stakeholders: None depicted
- High level abstract data types:
    - Data from External System 1
    - Data from External System 2
    - Data stored in the Database
    - Data communicated between Microservices
- Data types transmitted between components:
    - Microservices receive and send data to each other, and may access the database for data storage and retrieval
    - External System 1 sends data to Microservices
    - External System 2 sends data to the Database, and Microservices may retrieve data from the Database.

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 for a conceptual solution:



```diff
+----------------+      +----------------+      +----------------+
|  Web Browser   |      |  Web Server    |      |    Database    |
+----------------+      +----------------+      +----------------+
        ↑                     ↑     ↑                    ↑
        |        HTTP         |     |       JDBC         |
+----------------+      +------------+      +----------------+
|   Frontend     |      |  Backend   |      |   Database     |
|  Microservice  |      |Microservice|      |   Service      |
+----------------+      +------------+      +----------------+
```

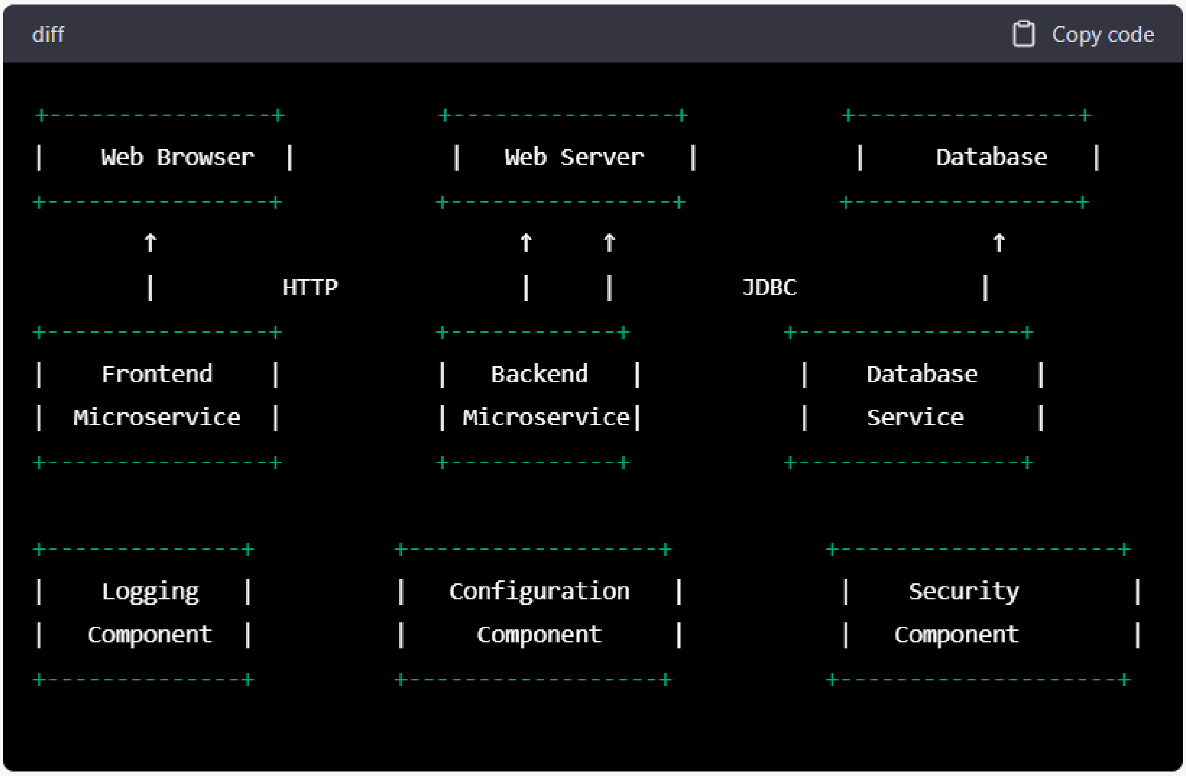
+----------------+ +------------+ +----------------+

- Main stakeholder use:
    - Web Browser users interact with the frontend microservice to access the system, perform actions and retrieve information
- Important quality properties:
    - Scalability: Use microservices architecture style to enable independent scaling of different components
    - Agility: Utilize DevOps practices, Continuous Integration (CI) and automated testing to support rapid development and deployment of changes
    - Reliability: Utilize reliability patterns such as Circuit Breakers and Retries to handle failures gracefully and increase availability
- Components and their relationships:
    - Web Browser: interacts with the Frontend microservice via HTTP
    - Frontend Microservice: interacts with Backend Microservice via HTTP and retrieves data from the Database Service via JDBC
    - Backend Microservice: interacts with the Database Service via JDBC
    - Database Service: handles data storage and retrieval via JDBC to the Database
- Data types transferred between components:
    - JSON data transmitted between Web Browser and Frontend Microservice via HTTP
    - JSON data transmitted between Frontend Microservice and Backend Microservice via HTTP
    - SQL queries and results transmitted between Backend Microservice and Database Service via JDBC
- Interfaces between components:
    - Frontend Microservice provides HTTP interface to Web Browser and consumes HTTP interface from Backend Microservice
    - Backend Microservice provides HTTP interface to Frontend Microservice and consumes JDBC interface from Database Service
    - Database Service provides JDBC interface to Backend Microservice
- Anti-patterns to avoid:
    - Continuous build anti-pattern: avoid delaying builds and integrating code frequently
    - Selective automation anti-pattern: avoid automating only specific parts of the development process
    - Tight coupling anti-pattern: avoid tightly coupling microservices with each other or with the database, as it can lead to cascading failures and decreased scalability. Utilize loose coupling patterns such as API Gateway and Service Registry.

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 for a conceptual solution:



```
+----------------+           +----------------+           +----------------+
|   Web Browser   |           |  Web Server    |           |    Database    |
+----------------+           +----------------+           +----------------+
        ↑                          ↑     ↑                           ↑
        |         HTTP             |     |        JDBC              |
+----------------+           +------------+           +----------------+
|    Frontend    |           |   Backend   |           |   Database     |
|  Microservice  |           | Microservice|           |   Service      |
+----------------+           +------------+           +----------------+

+--------------+          +------------------+           +--------------------+
|    Logging   |          |   Configuration   |           |    Security        |
|   Component  |          |     Component     |           |   Component        |
+--------------+          +------------------+           +--------------------+
```

- Main stakeholder use:
  - Web Browser users interact with the frontend microservice to access the system, perform actions and retrieve information
- Important quality properties:
  - Scalability: Use microservices architecture style to enable independent scaling of different components
  - Agility: Utilize DevOps practices, Continuous Integration (CI) and automated testing to support rapid development and deployment of changes
  - Reliability: Utilize reliability patterns such as Circuit Breakers and Retries to handle failures gracefully and increase availability
- Components and their relationships:
  - Logging Component: handles logging and monitoring of system components
  - Configuration Component: provides configuration settings for system components
  - Security Component: handles security and access control for system components
  - Web Browser, Frontend Microservice, Backend Microservice, Database Service: as described in the C4 Container diagram
- Data types transferred between components:
  - JSON data transmitted between Web Browser and Frontend Microservice via HTTP

    - JSON data transmitted between Frontend Microservice and Backend Microservice via HTTP
    - SQL queries and results transmitted between Backend Microservice and Database Service via JDBC
- Interfaces between components:
    - Logging Component provides logging and monitoring interfaces to all other components
    - Configuration Component provides configuration interfaces to all other components
    - Security Component provides security and access control interfaces to all other components
- Required and provided interfaces and data types:
    - Logging Component provides logging and monitoring interfaces to all other components via HTTP and consumes JSON data
    - Configuration Component provides configuration interfaces to all other components via HTTP and consumes/produces JSON data
    - Security Component provides security and access control interfaces to all other components via HTTP and consumes/produces JSON data
- Anti-patterns to avoid:
    - Continuous build anti-pattern: avoid delaying builds and integrating code frequently
    - Selective automation anti-pattern: avoid automating only specific parts of the development process
    - Tight coupling anti-pattern: avoid tightly coupling microservices with each other or with the database, as it can lead to cascading failures and decreased scalability. Utilize loose coupling patterns such as API Gateway and Service Registry.

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.

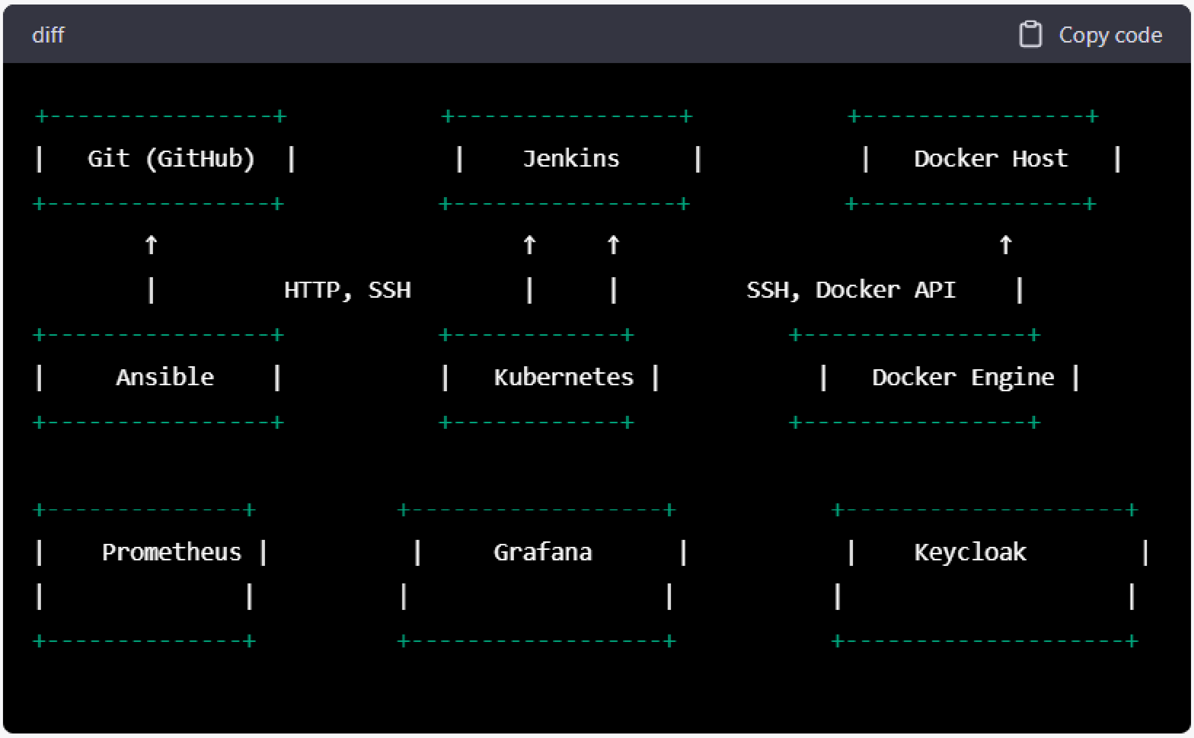
A suggested toolchain to implement the solution is as follows:

- Code Versioning: Git
- Continuous Integration: Jenkins
- Configuration Management: Ansible
- Container Orchestration: Kubernetes
- Logging and Monitoring: Prometheus and Grafana
- Security and Access Control: Keycloak

For the components that need to be developed, the following programming languages are suggested:

- Frontend Microservice: JavaScript and React
- Backend Microservice: Java and Spring Boot
- Database Service: SQL

C4 Component Solution Diagram for the toolchain:



```
+----------------+           +----------------+           +----------------+
|   Git (GitHub)  |           |    Jenkins     |           |   Docker Host   |
+----------------+           +----------------+           +----------------+
        ↑                          ↑     ↑                           ↑
        |         HTTP, SSH        |     |         SSH, Docker API    |
+----------------+           +------------+           +----------------+
|    Ansible     |           |  Kubernetes |           |  Docker Engine |
+----------------+           +------------+           +----------------+

+--------------+          +------------------+           +--------------------+
|    Prometheus |          |     Grafana      |           |    Keycloak        |
|              |          |                  |           |                    |
+--------------+          +------------------+           +--------------------+
```

- Data types transferred between components:
  - Git transfers code and configuration files in various formats (e.g., JSON, YAML)
  - Jenkins sends build status messages as JSON data
  - Ansible transfers configuration files in various formats (e.g., YAML)
  - Kubernetes and Docker Engine use the Docker Image Manifest v2 Schema for container images
  - Prometheus uses its own data model to collect and store time series data
  - Grafana consumes time series data from Prometheus and visualizes it
  - Keycloak uses its own data model to handle authentication and authorization
- Summarized data types transmitted between components:
  - Git sends code and configuration files
  - Jenkins sends build status messages
  - Ansible sends configuration files
  - Kubernetes and Docker Engine send container images
  - Prometheus sends time series data
  - Grafana consumes time series data
  - Keycloak handles authentication and authorization data
- Interfaces between components:
  - Git provides code versioning and configuration management interfaces via HTTP and SSH
  - Jenkins provides Continuous Integration interfaces via HTTP and consumes/produces JSON data
  - Ansible provides Configuration Management interfaces via SSH and consumes/produces YAML data
  - Kubernetes provides Container Orchestration interfaces via SSH and Docker API and consumes/produces Docker Image Manifest v2 Schema data

- Docker Engine provides Container Management interfaces via Docker API and consumes/produces Docker Image Manifest v2 Schema data
- Prometheus provides Monitoring interfaces via HTTP and consumes/produces its own data model
- Grafana provides Visualization interfaces via HTTP and consumes/produces time series data
- Keycloak provides Security and Access Control interfaces via HTTP and consumes/produces its own data model.