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

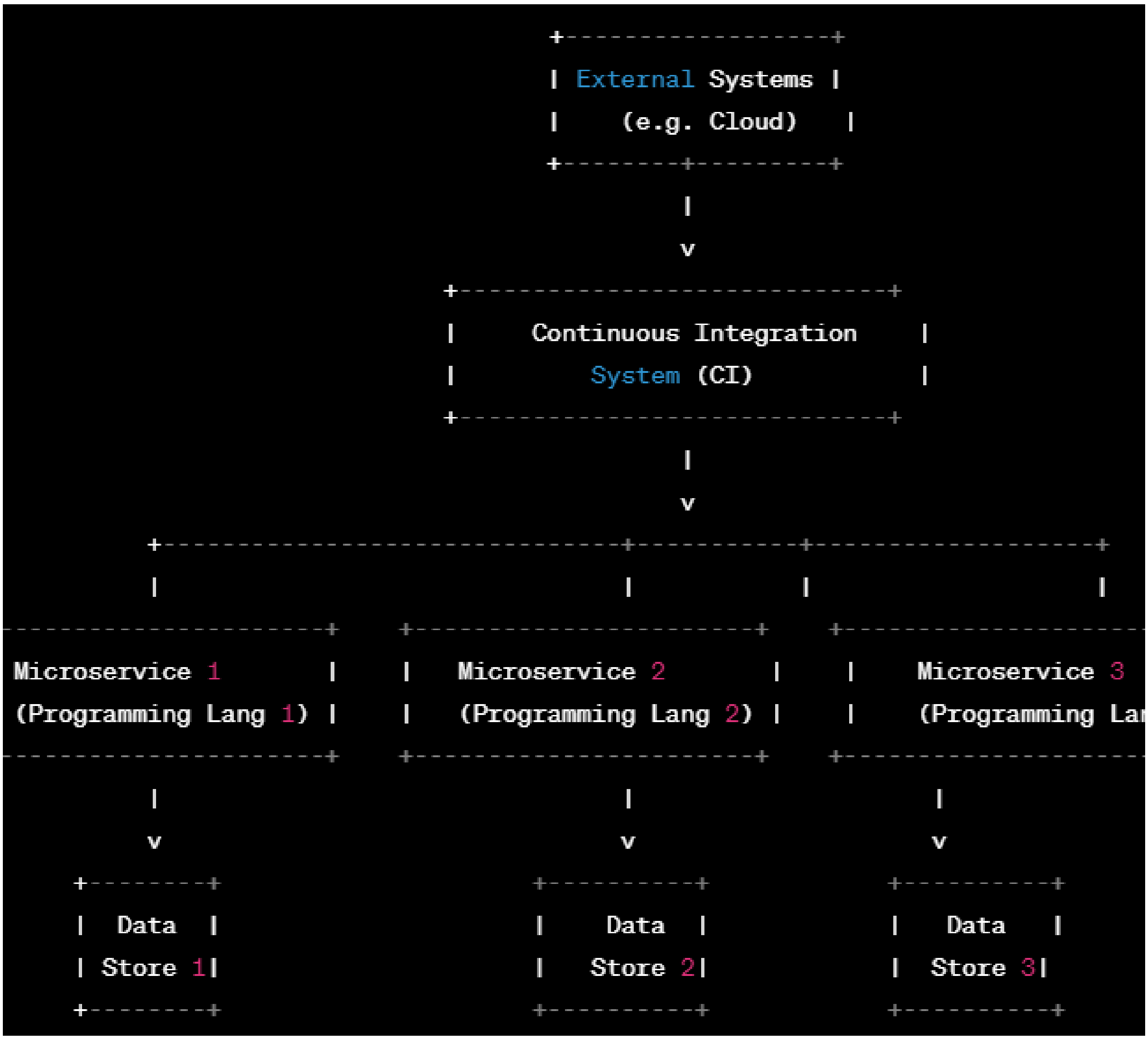
development process. When the team faces issues within the continuous integration process, finding the causes of these problems starts. With consideration of the boundaries of different stages of DevOps life cycle, identifying the root causes of these problems is limited within these stages. System designers 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: Microservices architecture is gaining popularity in the industry, but there are challenges in achieving the goals of the architecture, particularly in the context of continuous integration. Design decisions made during the system design stage or during implementation stages can have a negative impact on the continuous integration process. There is no formal process to trace these design decisions and their impact on continuous integration.

Solution: A software system architecture solution can be designed to address the challenges of continuous integration in microservices architecture. The solution should aim to shorten the release cycle, increase the frequency of merging changes, identify and enforce architectural patterns that support DevOps operations, identify and avoid anti-patterns that hinder DevOps processes, promote automated testing, and support auto provisioning and de-provisioning of services. The solution should also involve documenting design decisions and their impact on continuous integration and extending the DevOps process to utilize design patterns and anti-patterns. This can be achieved through the use of a formal process that utilizes design decisions as patterns and anti-patterns to improve continuous integration as part of the DevOps process. The solution should also consider the use of external persistent storage for maintaining state of services.

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.

Here is a C4 system context diagram for the proposed solution:



```
                                     +-------------------+
                                     | External Systems |
                                     |    (e.g. Cloud)    |
                                     +--------+---------+
                                              |
                                              v
                              +-----------------------------+
                              |     Continuous Integration     |
                              |         System (CI)            |
                              +-----------------------------+
                                              |
                                              v
          +--------------------------------+-----------+-------------------+
          |                                |           |                   |
----------------------+    +------------------------+    +---------------------
Microservice 1        |    |   Microservice 2        |    |    Microservice 3
(Programming Lang 1)  |    |  (Programming Lang 2)   |    |    (Programming La
----------------------+    +------------------------+    +---------------------
          |                                |                    |
          v                                v                    v
     +--------+                     +----------+            +----------+
     |  Data  |                     |    Data  |            |   Data    |
     | Store 1|                     |   Store 2|            |  Store 3|
     +--------+                     +----------+            +----------+
```

In this diagram, we have:

- External Systems: These are external systems that may interact with the CI system, such as a cloud infrastructure.
- Continuous Integration System (CI): This is the central system that manages the continuous integration process.
- Microservices 1-3: These are the individual microservices developed by different teams using different programming languages.
- Data Store 1-3: These are the data stores used by the microservices.

The arrows between the components indicate the flow of data between them:

- The external systems send data to the CI system for processing and analysis.
- The CI system sends data to each microservice, triggering their build and deployment processes.
- The microservices send data to their respective data stores, where the data is stored and retrieved as needed.

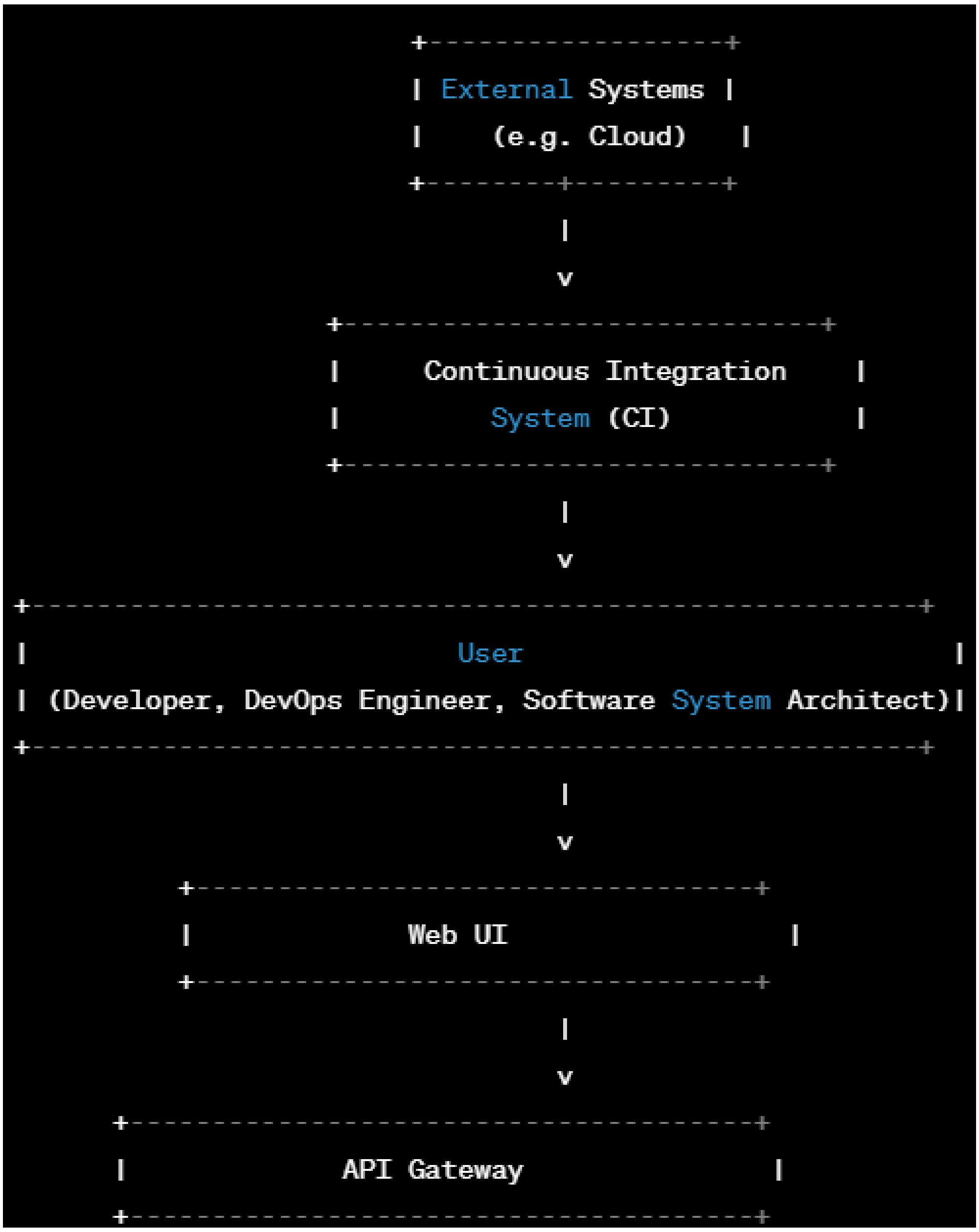
The abstract data types transmitted between components and stakeholders include:

- Build and deployment configurations: These are configurations that specify how the microservices should be built and deployed.
- Code changes: These are the changes made to the code base by developers.
- Test results: These are the results of automated tests run by the CI system to ensure that the microservices are functioning correctly.
- Operational metrics: These are metrics related to the performance and reliability of the microservices, such as response times and error rates.

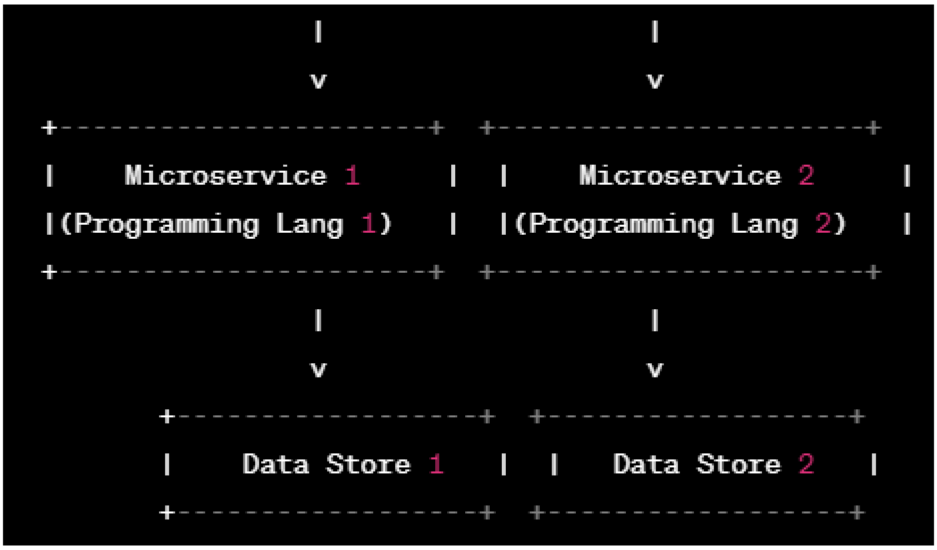
Overall, the solution is designed to support continuous integration and DevOps processes by enabling rapid and reliable deployment of microservices. By identifying and utilizing architectural patterns that support DevOps and automated testing, and avoiding anti-patterns that hinder these processes, the solution aims to improve the efficiency and effectiveness of the development team.

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.

Here is a C4 container solution diagram for the proposed solution:



```
                        +------------------+
                        | External Systems |
                        |    (e.g. Cloud)   |
                        +--------+---------+
                                 |
                                 v
                   +-----------------------------+
                   |     Continuous Integration    |
                   |         System (CI)           |
                   +-----------------------------+
                                 |
                                 v
+------------------------------------------------------+
|                          User                          |
| (Developer, DevOps Engineer, Software System Architect)|
+------------------------------------------------------+
                                 |
                                 v
          +----------------------------------+
          |             Web UI                 |
          +----------------------------------+
                                 |
                                 v
      +--------------------------------------+
      |             API Gateway               |
      +--------------------------------------+
```



```
                  |                   |
                  v                   v
  +----------------------+  +----------------------+
  |    Microservice 1     |  |    Microservice 2     |
  |(Programming Lang 1)   |  |(Programming Lang 2)   |
  +----------------------+  +----------------------+
                  |                   |
                  v                   v
         +------------------+  +------------------+
         |    Data Store 1   |  |   Data Store 2   |
         +------------------+  +------------------+
```

In this diagram, we have:

- External Systems: These are the external systems that may interact with the CI system, such as a cloud infrastructure.
- Continuous Integration System (CI): This is the central system that manages the continuous integration process.
- User: This is the main stakeholder of the system who will interact with the web UI and API gateway.
- Web UI: This component provides the user interface for interacting with the CI system.
- API Gateway: This component provides a single entry point for external requests to the system.
- Microservices 1-2: These are the individual microservices developed by different teams using different programming languages.
- Data Store 1-2: These are the data stores used by the microservices.

The quality properties that are important for this solution include:

- Reliability: The system should be reliable and ensure that all components are available and functioning as expected.
- Scalability: The system should be able to handle increased load and scale as necessary.
- Maintainability: The system should be easy to maintain and update.

To address these quality properties, we can use an architectural style that supports microservices and DevOps processes. We can also use design patterns such as the Circuit Breaker pattern and the Gateway pattern to ensure reliability and scalability. The components can be structured as follows:

- Web UI and API Gateway: These components are responsible for handling external requests and routing them to the appropriate microservice.
- Microservices: These components are responsible for building and deploying individual microservices.
- Data Stores: These components are responsible for storing and retrieving data used by the microservices.

The data types transferred between components include:

- Build and deployment configurations: These are configurations that specify how the microservices should be built and deployed.
- Code changes: These are the changes made to the code base by developers.
- Test results: These are the results of automated tests run by the CI system to ensure that the microservices are functioning correctly.
- Operational metrics: These are metrics related to the performance and reliability of the microservices, such as response times and error rates.

The connections and interfaces between components include:

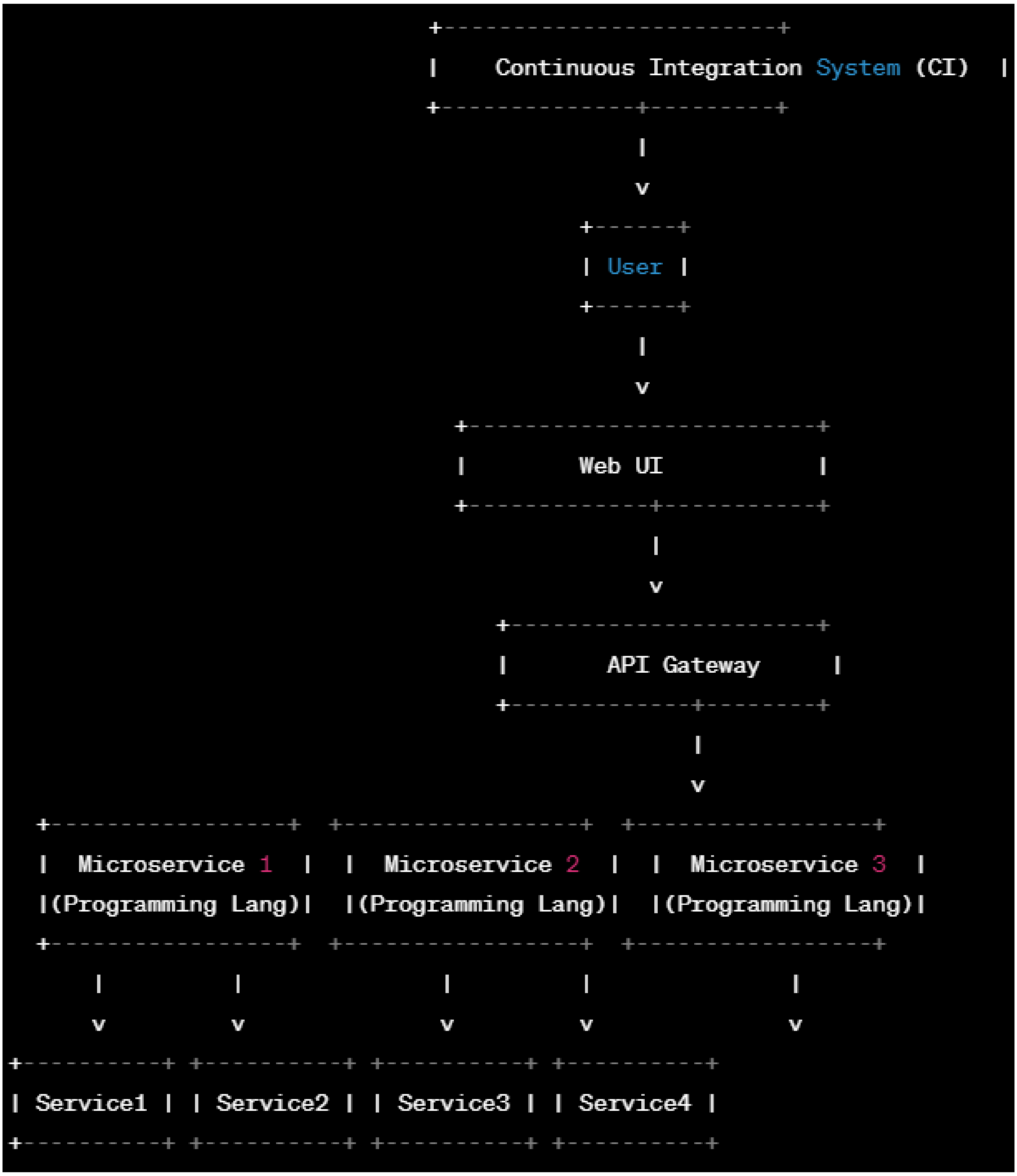
- Web UI to API Gateway: The Web UI provides requests to the API Gateway, which in turn routes them to the appropriate microservice.

- API Gateway to Microservices: The API Gateway provides requests to the appropriate microservice for processing.
- Microservices to Data Stores: The microservices retrieve and store data in the appropriate data store.

The interfaces between components include:

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.

Here is a C4 component solution diagram for the proposed solution:



```
                              +------------------------+
                              |    Continuous Integration System (CI)  |
                              +--------------+---------+
                                             |
                                             v
                                         +------+
                                         | User |
                                         +------+
                                             |
                                             v
                                +-------------------------+
                                |        Web UI           |
                                +-------------+-----------+
                                              |
                                              v
                                   +----------------------+
                                   |       API Gateway     |
                                   +-------------+--------+
                                                 |
                                                 v
  +-----------------+  +-----------------+  +-----------------+
  |  Microservice 1  |  |  Microservice 2  |  |  Microservice 3  |
  |(Programming Lang)|  |(Programming Lang)|  |(Programming Lang)|
  +-----------------+  +-----------------+  +-----------------+
      |         |              |         |              |
      v         v              v         v              v
+----------+ +----------+ +----------+ +----------+
| Service1 | | Service2 | | Service3 | | Service4 |
+----------+ +----------+ +----------+ +----------+
```

In this diagram, we have:

- Continuous Integration System (CI): This is the central system that manages the continuous integration process.
- User: This is the main stakeholder of the system who will interact with the Web UI and API Gateway.
- Web UI: This component provides the user interface for interacting with the CI system.
- API Gateway: This component provides a single entry point for external requests to the system.
- Microservices: These are the individual microservices developed by different teams using different programming languages.

- Services: These are the individual components of the microservices that perform specific functions.

The quality properties that are important for this solution include:

- Reliability: The system should be reliable and ensure that all components are available and functioning as expected.
- Scalability: The system should be able to handle increased load and scale as necessary.
- Maintainability: The system should be easy to maintain and update.

To address these quality properties, we can use an architectural style that supports microservices and DevOps processes. We can also use design patterns such as the Circuit Breaker pattern and the Gateway pattern to ensure reliability and scalability. The components can be structured as follows:

- Web UI and API Gateway: These components are responsible for handling external requests and routing them to the appropriate microservice.
- Microservices: These components are responsible for building and deploying individual microservices.
- Services: These components are responsible for performing specific functions within each microservice.

The data types transferred between components include:

- Build and deployment configurations: These are configurations that specify how the microservices should be built and deployed.
- Code changes: These are the changes made to the code base by developers.
- Test results: These are the results of automated tests run by the CI system to ensure that the microservices are functioning correctly.
- Operational metrics: These are metrics related to the performance and reliability of the microservices, such as response times and error rates.

The connections and interfaces between components include:

- Web UI to API Gateway: The Web UI provides requests to the API Gateway, which in turn routes them to the appropriate microservice.
- API Gateway to Microservices: The API Gateway provides requests to the appropriate microservice for processing.
- Microservices to Services: The microservices call specific services to perform functions within each microservice.

The interfaces between components include:

- Web UI to API Gateway: The Web UI provides requests to the API Gateway, which in turn routes them to the appropriate microservice.
- API Gateway to Microservices: The API Gateway provides requests to the appropriate microservice for processing.
-

continue

- Microservices to Services: The microservices call specific services to perform functions within each microservice.
- CI System to Microservices: The CI System provides build and deployment configurations to each microservice.
- Microservices to CI System: The microservices provide test results and operational metrics to the CI system.

Anti-patterns that should be avoided when implementing this solution include:

- Monolithic architecture: To support the microservices architecture and DevOps processes, it is important to avoid a monolithic architecture where all components are tightly coupled.
- Selective automation: To promote automated testing, it is important to avoid selective automation of developed features and ensure that all features are tested automatically.
- Continuous build anti-pattern: To avoid slowing down the development process, it is important to avoid the continuous build anti-pattern where every code change triggers a build, leading to longer build times and slower development. Instead, builds should be triggered only when necessary.

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 use the following toolchain:

- Version Control System: Git (reusable)
- Continuous Integration System: Jenkins (reusable)
- Containerization Platform: Docker (reusable)
- Orchestration Platform: Kubernetes (reusable)
- Service Registry and Discovery: Consul (reusable)
- Configuration Management: HashiCorp Vault (reusable)

For the components we need to develop, we can use the following programming languages:

- Microservices: Java
- Services: Python
- CI System: Bash scripts

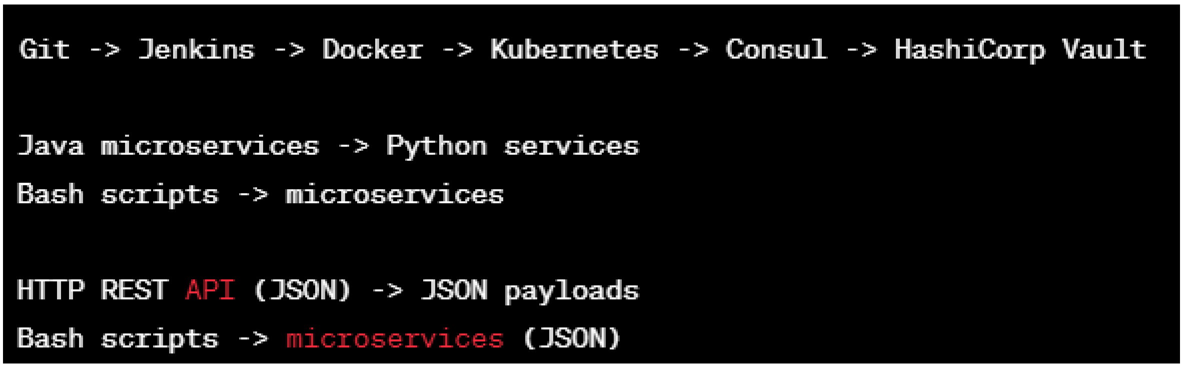
The data types transferred between components include:

- Request and response messages for API calls
- Configuration data for services and microservices
- Test results and operational metrics

The connections and interfaces between components can be summarized as follows:

- Microservices to Services: HTTP REST API using JSON
- CI System to Microservices: Bash scripts for build and deployment configurations
- Microservices to CI System: JSON payloads for test results and operational metrics

The toolchain and programming languages can be visualized using arrows and dots, as follows:



```
Git -> Jenkins -> Docker -> Kubernetes -> Consul -> HashiCorp Vault

Java microservices -> Python services
Bash scripts -> microservices

HTTP REST API (JSON) -> JSON payloads
Bash scripts -> microservices (JSON)
```