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: In a microservices architecture, there is a lack of formal process to trace and document design decisions and their impact on continuous integration. This leads to challenges in the continuous integration process, slowing down the entire development process. There is also a lack of standard process that utilizes design patterns and anti-patterns to improve continuous integration.

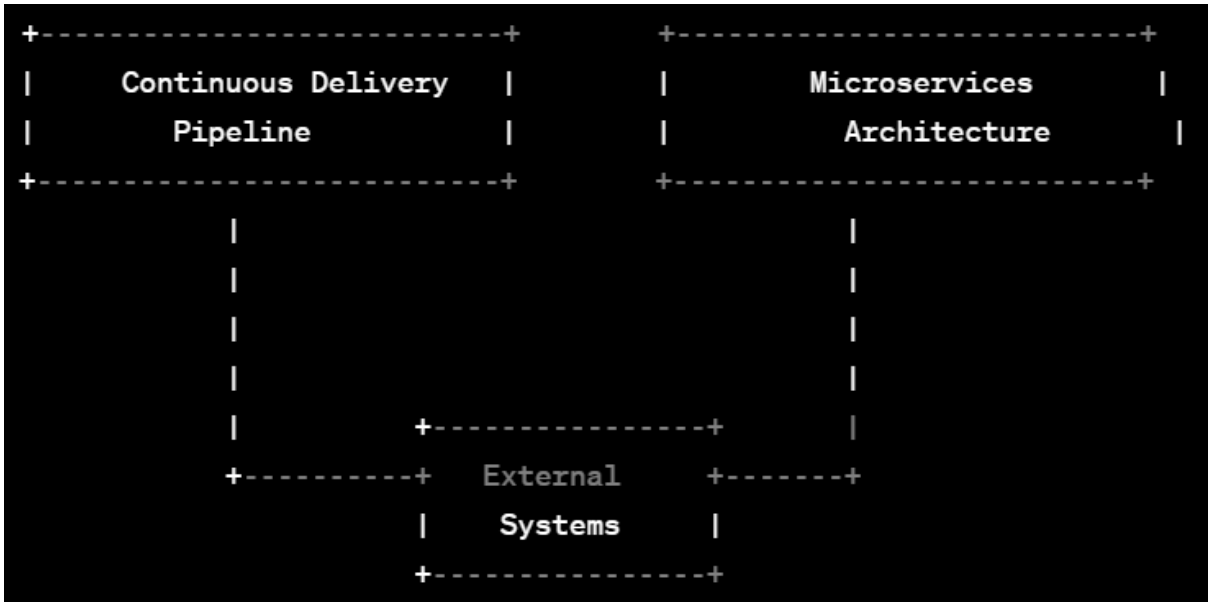
Solution: A software system architecture solution should be designed to address the identified requirements:

1. Shorten the release cycle by implementing a continuous delivery approach that includes automated testing and deployment pipelines.
2. Increase the frequency of merging changes by implementing a distributed version control system.
3. Identify and enforce the use of architectural patterns that support DevOps operations and increase team velocity.
4. Identify and avoid anti-patterns that minimize the benefits of DevOps processes.
5. Promote automated testing by using patterns that support and simplify automated testing.
6. Extend the current DevOps process by proposing an extended process that utilizes the knowledge of patterns and anti-patterns and enforces their use.
7. Support auto-provisioning and de-provisioning of services by designing stateless services and using external persistent storage for services that maintain state.

The solution should also include a formal process for documenting design decisions and their impact on continuous integration, as well as a mechanism for utilizing design patterns and anti-patterns to improve the continuous integration process. The system architecture should also support the use of multiple programming languages and data storage and retrieval technologies by different development teams.

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 the proposed solution:



In this system context diagram, we have two main components, Continuous Delivery Pipeline and Microservices Architecture, which are represented by rectangular boxes. The Continuous Delivery Pipeline is responsible for automating the delivery of software changes to production, while the Microservices Architecture is responsible for developing software applications as a group of small services, each running in its own process.

There is an arrow connecting the Continuous Delivery Pipeline to the Microservices Architecture, indicating that the pipeline uses the services provided by the architecture. The external systems, if any, are represented by a rounded rectangular box with a dotted border, and they are connected to the Microservices Architecture.

There are no external stakeholders identified in this diagram, so we do not include any in the diagram.

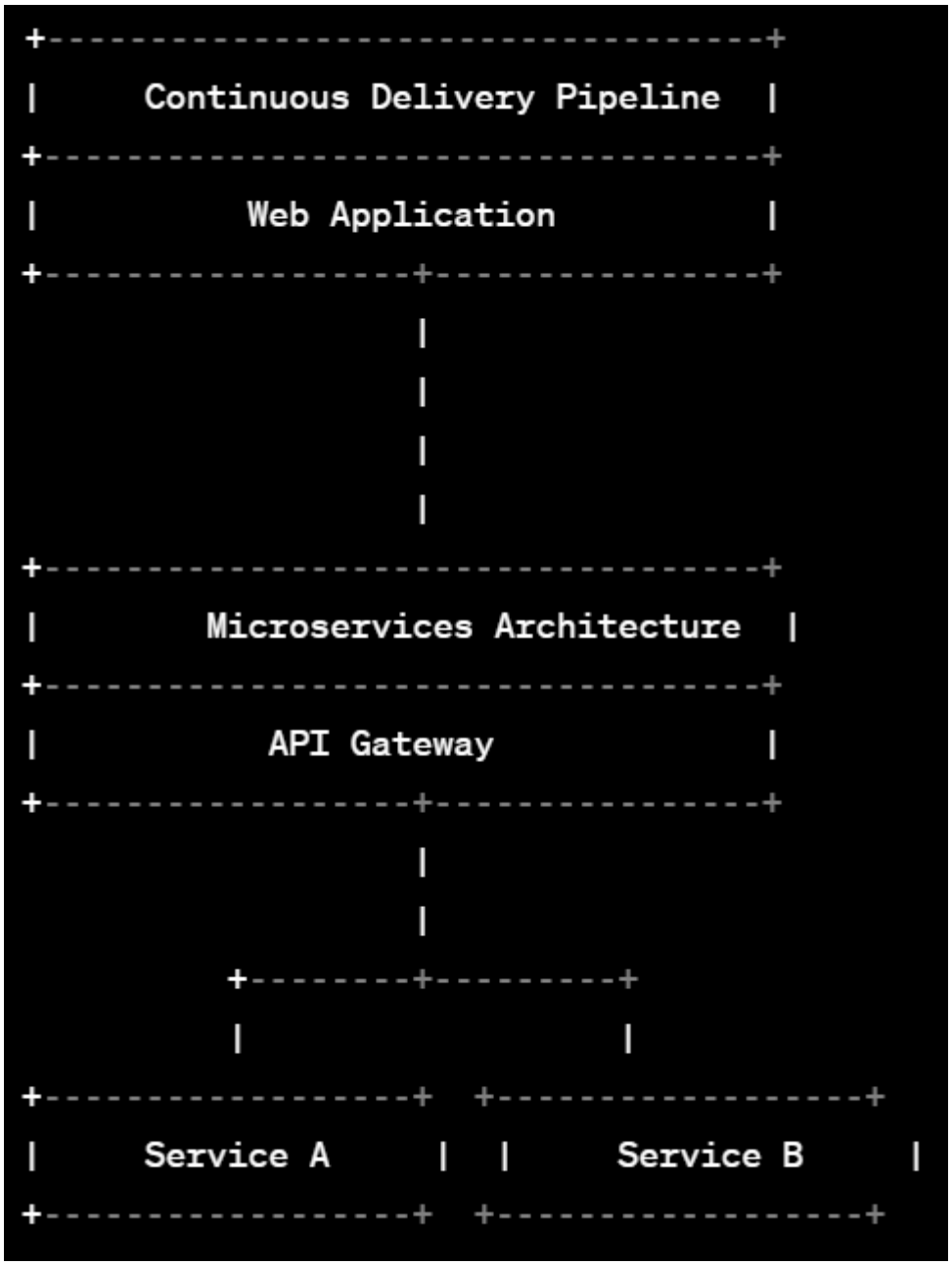
The abstract data types transmitted between components and stakeholders are not specified in this diagram.

Overall, this system context diagram provides a high-level overview of the proposed solution, highlighting the two main components and their relationship. It does not go into detail about the interfaces between the components or the specific data types transmitted between them.

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 the proposed solution:



In this container diagram, we have identified two main components, Continuous Delivery Pipeline and Microservices Architecture, and their associated subcomponents. The Continuous Delivery Pipeline is represented by a rectangular box with the label "Web Application," while the Microservices Architecture is represented by a rectangular box with the label "API Gateway" and two subcomponents, "Service A" and "Service B."

The main stakeholders will use the web application to trigger the delivery of software changes to production through the Continuous Delivery Pipeline.

The important quality properties for this solution are scalability, agility, and reliability. To address these properties, we propose using a microservices architecture style, which promotes scalability, agility, and reliability by breaking down the system into small, independent services. We also propose using design patterns such as external configurations of microservices and blue-green deployment pattern to support DevOps operations and increase team velocity.

The data types transferred between components include code changes, test results, and configuration data.

The Web Application requires an interface to trigger the deployment process and receives deployment status updates. The API Gateway requires an interface to receive requests from the Web Application and send them to the appropriate service. The Service A and Service B components require an interface to receive requests from the API Gateway and send responses.

The required interfaces are:

- Continuous Delivery Pipeline Web Application requires an interface to trigger the deployment process and receive deployment status updates from the pipeline.
- API Gateway requires an interface to receive requests from the Web Application and send them to the appropriate service.
- Service A and Service B require an interface to receive requests from the API Gateway and send responses.

The provided interfaces are:

- Continuous Delivery Pipeline Web Application provides an interface to trigger the deployment process and receive deployment status updates from the pipeline.
- API Gateway provides an interface to receive requests from the Web Application and send them to the appropriate service.
- Service A and Service B provide an interface to receive requests from the API Gateway and send responses.

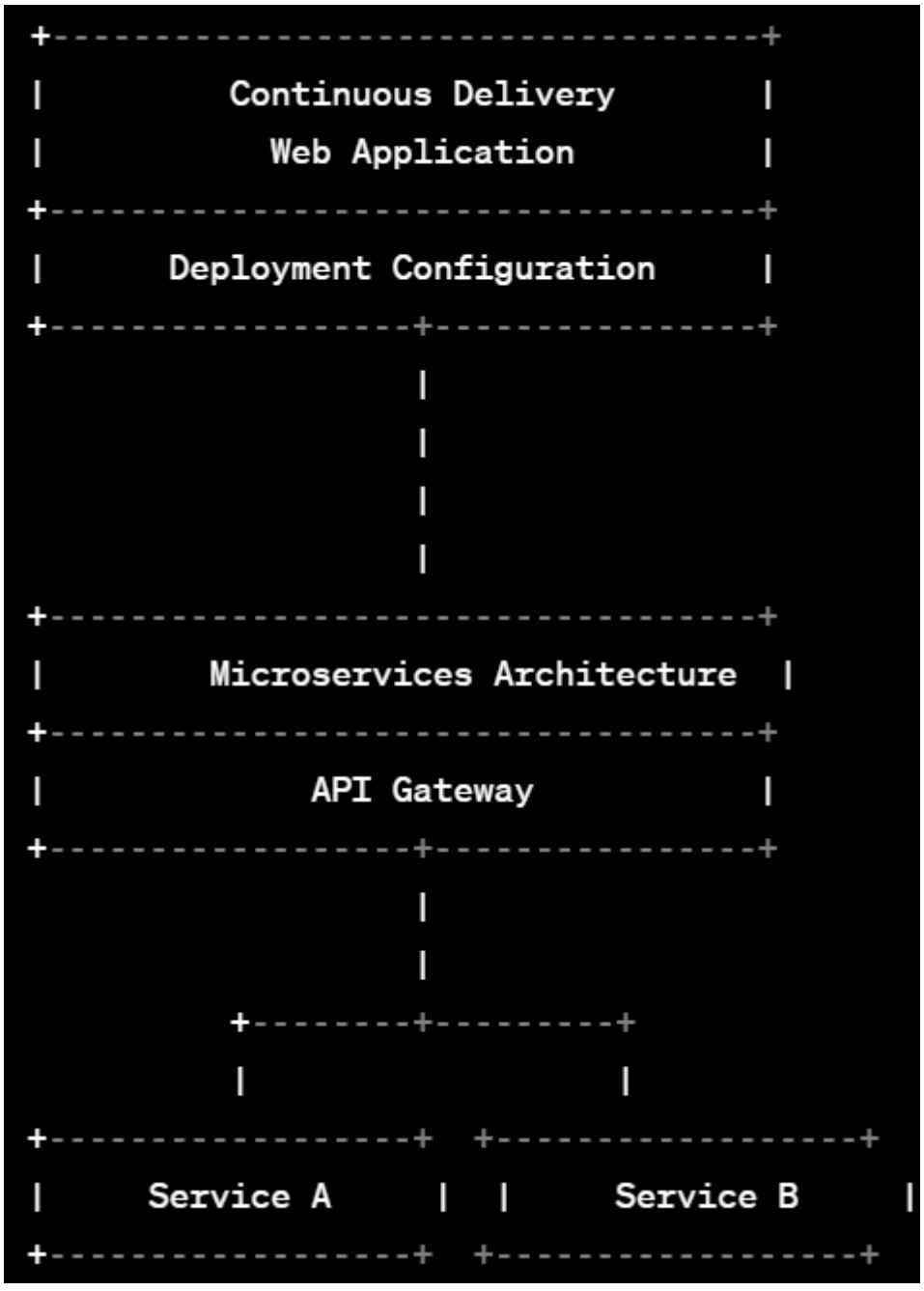
The data types transmitted between components include JSON payloads for requests and responses, YAML configuration files, and binary files for deploying the application.

We need to avoid anti-patterns such as the continuous build anti-pattern and selective automation for developed features to minimize the negative impact of DevOps processes.

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 the proposed solution:



In this component diagram, we have identified the components that make up the Continuous Delivery Pipeline and Microservices Architecture, and their associated subcomponents.

The main stakeholders will use the web application to trigger the delivery of software changes to production through the Continuous Delivery Pipeline. The Deployment Configuration component is used to configure the pipeline and the Microservices Architecture.

The important quality properties for this solution are scalability, agility, and reliability. To address these properties, we propose using a microservices architecture style, which promotes scalability, agility, and reliability by breaking down the system into small, independent services. We also propose using design patterns such as external configurations of microservices and blue-green deployment pattern to support DevOps operations and increase team velocity.

The data types transferred between components include code changes, test results, and configuration data.

The Continuous Delivery Web Application component requires an interface to trigger the deployment process and receives deployment status updates. The Deployment Configuration component requires an interface to manage the pipeline configuration. The API Gateway requires an interface to receive requests from the Web Application and send them to the appropriate service. The Service A and Service B components require an interface to receive requests from the API Gateway and send responses.

The required interfaces are:

- Continuous Delivery Web Application requires an interface to trigger the deployment process and receive deployment status updates from the pipeline.
- Deployment Configuration component requires an interface to manage the pipeline configuration.
- API Gateway requires an interface to receive requests from the Web Application and send them to the appropriate service.
- Service A and Service B require an interface to receive requests from the API Gateway and send responses.

The provided interfaces are:

- Continuous Delivery Web Application provides an interface to trigger the deployment process and receive deployment status updates from the pipeline.
- Deployment Configuration component provides an interface to manage the pipeline configuration.
- API Gateway provides an interface to receive requests from the Web Application and send them to the appropriate service.
- Service A and Service B provide an interface to receive requests from the API Gateway and send responses.

The data types transmitted between components include JSON payloads for requests and responses, YAML configuration files, and binary files for deploying the application.

We need to avoid anti-patterns such as the continuous build anti-pattern and selective automation for developed features to minimize the negative impact of DevOps processes.

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.

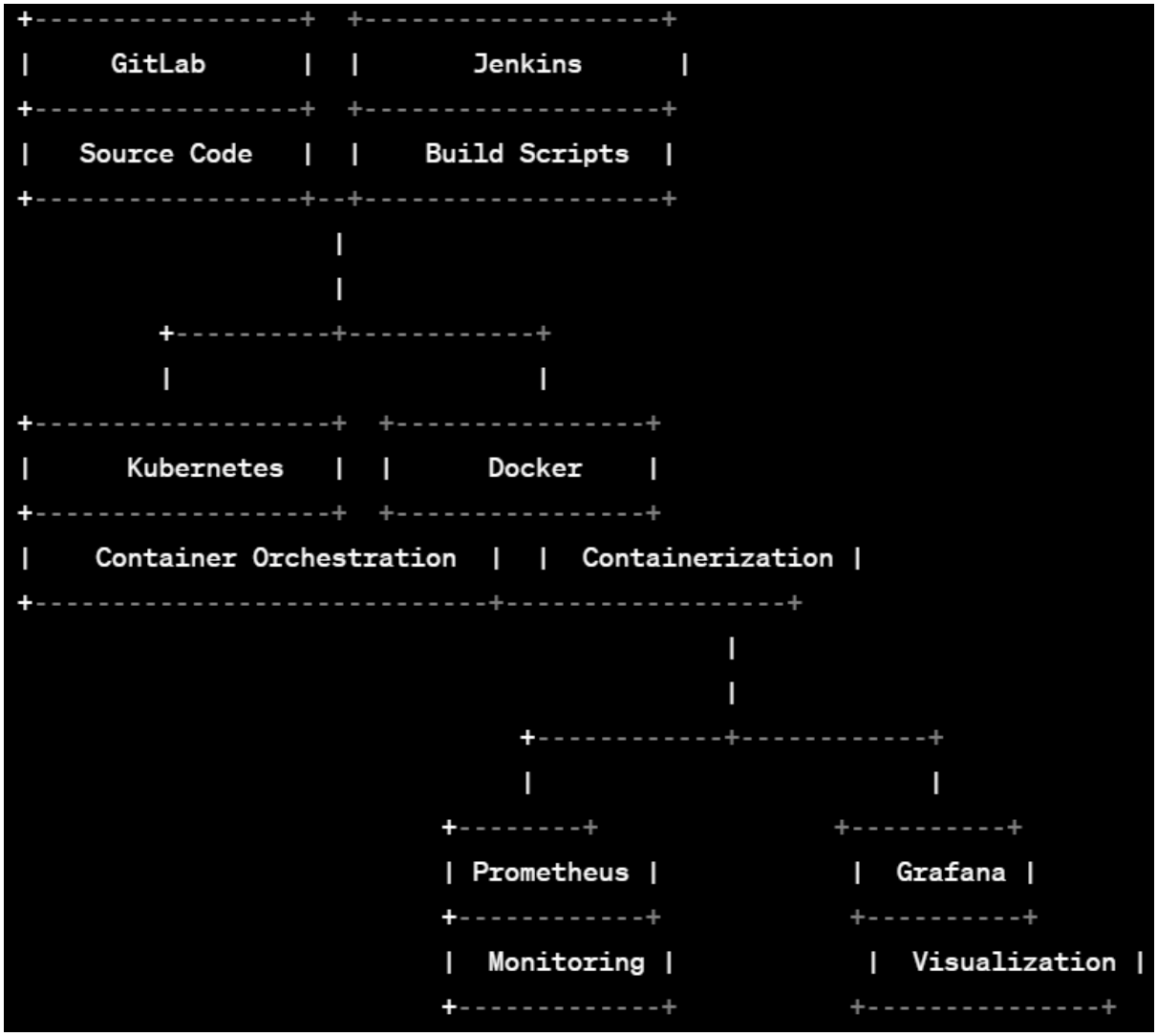
To implement the proposed solution, we suggest using the following toolchain:

- GitLab for source code management and version control.
- Jenkins for continuous integration and delivery.
- Kubernetes for container orchestration and management.
- Docker for containerization and packaging.
- Prometheus for monitoring and alerting.
- Grafana for visualization of metrics and logs.

For the components that need to be developed, we suggest using the following programming languages:

- The Continuous Delivery Web Application can be developed using Python and Flask.
- The Deployment Configuration component can be developed using YAML.
- The API Gateway and microservices can be developed using any programming language that supports RESTful API development.

C4 Component Solution Diagram with the toolchain:



The data types transferred between components include JSON payloads for requests and responses, YAML configuration files, and binary files for deploying the application.

The Continuous Delivery Web Application requires an interface to trigger the deployment process and receive deployment status updates. The Deployment Configuration component requires an interface to manage the pipeline configuration. The API Gateway requires an interface to receive requests from the Web Application and send them to the appropriate service. The Service A and Service B components require an interface to receive requests from the API Gateway and send responses.

The required interfaces are:

- Continuous Delivery Web Application requires an interface to trigger the deployment process and receive deployment status updates from the pipeline.
- Deployment Configuration component requires an interface to manage the pipeline configuration.
- API Gateway requires an interface to receive requests from the Web Application and send them to the appropriate service.

- Service A and Service B require an interface to receive requests from the API Gateway and send responses.

The provided interfaces are:

- Continuous Delivery Web Application provides an interface to trigger the deployment process and receive deployment status updates from the pipeline.
- Deployment Configuration component provides an interface to manage the pipeline configuration.
- API Gateway provides an interface to receive requests from the Web Application and send them to the appropriate service.
- Service A and Service B provide an interface to receive requests from the API Gateway and send responses.