

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: The development team is using a microservices architecture that requires an efficient DevOps process for continuous integration. However, the team is facing challenges in integrating changes into the code base, identifying problematic design decisions, and reducing the time needed to complete the sprints.

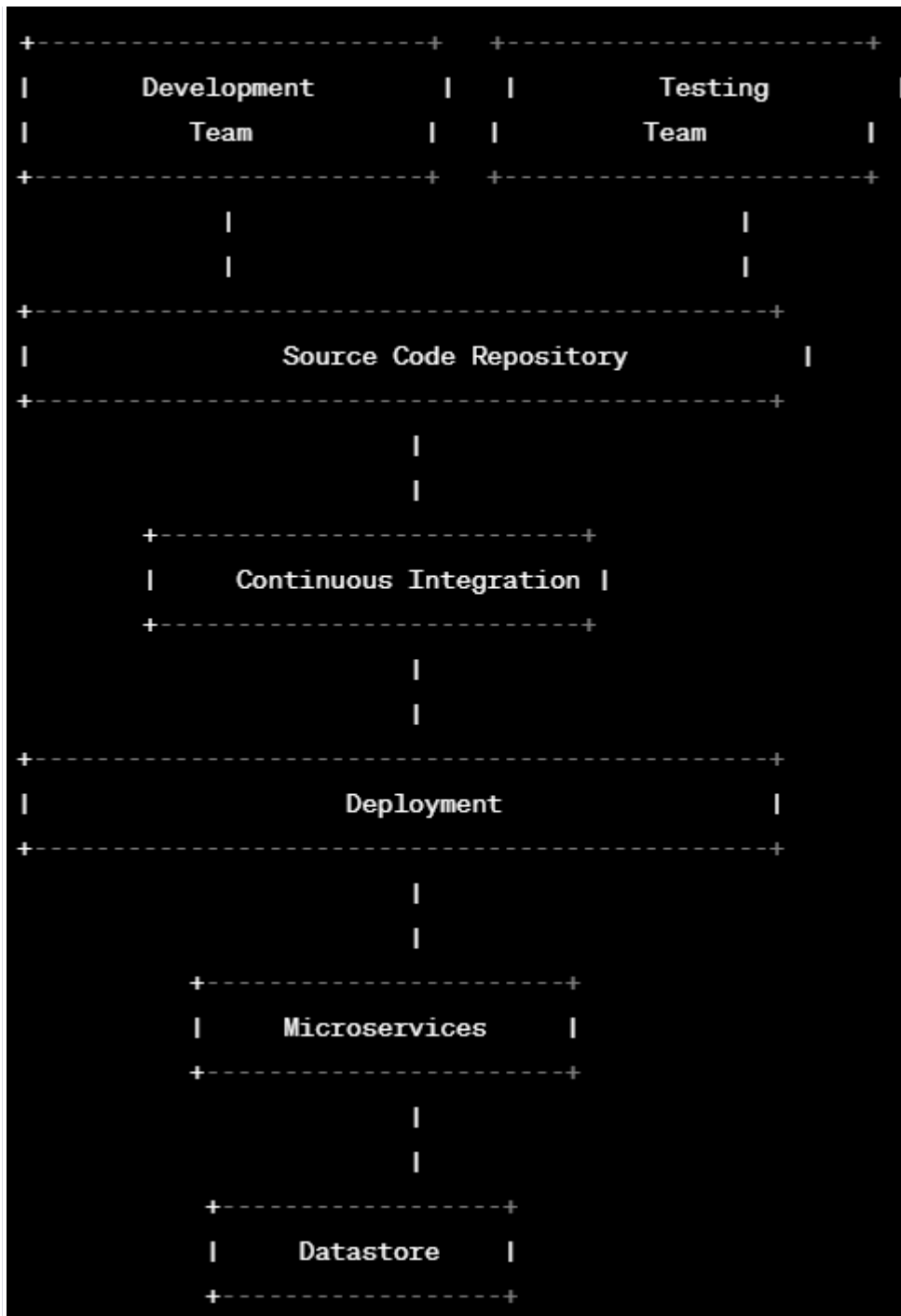
Solution: A software system architecture solution can be proposed to address the requirements mentioned below:

1. To shorten the release cycle, a solution can be proposed to automate the deployment process and create a pipeline for continuous delivery.
2. To increase the frequency of merging changes, a solution can be proposed to enforce using a code versioning repository and perform code integrations frequently.
3. To identify and enforce using architectural patterns that support DevOps, a solution can be proposed to create a catalog of design decisions (patterns and anti-patterns) that impact continuous integration and enforce using them as part of the DevOps process.
4. To identify and avoid anti-patterns, a solution can be proposed to perform code reviews sessions and enforce not using anti-patterns that hinder DevOps.
5. To promote automated testing, a solution can be proposed to use patterns that support and simplify automated testing and avoid anti-patterns that hinder automated testing.
6. To extend the DevOps process, a solution can be proposed to utilize the acquired knowledge of patterns and anti-patterns and enforce using them as part of the process.
7. To support auto-provisioning and de-provisioning of services, a solution can be proposed to design services as stateless and maintain any state in external persistent storage.

Overall, the proposed software system architecture solution should focus on automating the DevOps process, enforcing the use of design decisions that support DevOps, promoting automated testing, and designing services that can be provisioned and de-provisioned easily.

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 suggested C4 system context diagram for the proposed solution:



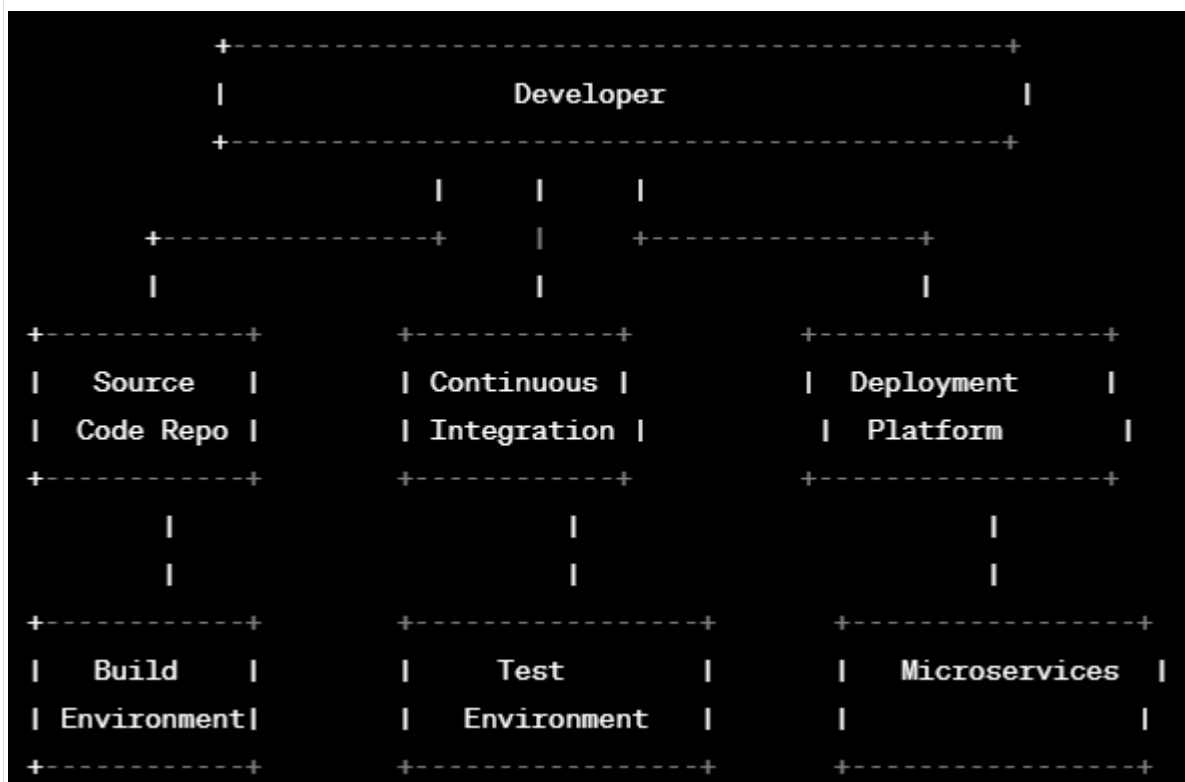
The diagram includes the following:

- External stakeholders: Development Team, Testing Team
- External systems: Source Code Repository
- High level abstract data types: Code changes, configuration data, test results
- Summarised data types: Code changes are pulled from the Source Code Repository into the Continuous Integration system. Configuration data is used to deploy the Microservices onto the Deployment platform. Test results are generated and communicated between the Continuous Integration and Testing Teams.

The Development Team works on the source code repository and pushes code changes, which are automatically pulled into the Continuous Integration system. The Continuous Integration system performs builds and tests on the code changes and communicates the results to the Development Team. The Deployment platform takes the configuration data and deploys the Microservices based on the version specified by the Development Team. The Testing Team performs further tests on the deployed Microservices and communicates the results to the Development Team. The Microservices read and write data from the Datastore as necessary.

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 suggested C4 container diagram for the proposed solution:



The diagram includes the following:

- Stakeholder: Developer
- Quality properties: Scalability, Agility, Reliability, Maintainability, Testability

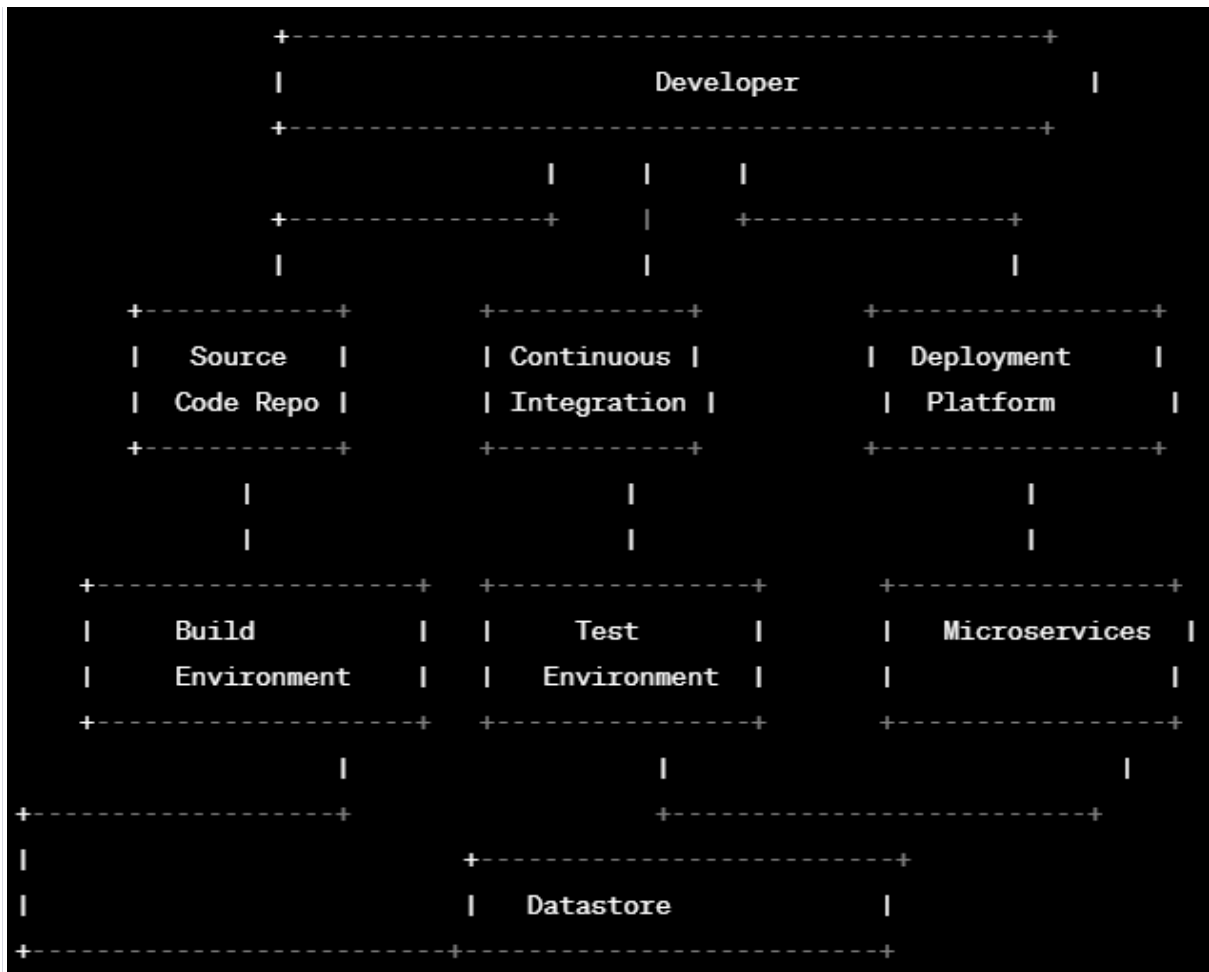
- Components: Developer, Source Code Repository, Continuous Integration, Deployment Platform, Build Environment, Test Environment, Microservices
- Data types: Code changes, configuration data, test results
- Summarized data types: Code changes are pushed from Developer to Source Code Repository and pulled into Continuous Integration. Configuration data is used to deploy Microservices onto Deployment Platform. Test results are generated and communicated between Continuous Integration and Test Environment. Microservices read and write data from/to datastore.
- Interfaces: The Developer pushes code changes to the Source Code Repository. The Continuous Integration system pulls code changes from the Source Code Repository and builds them in the Build Environment. The Build Environment generates the binary artifact and sends it to the Deployment Platform, which deploys the artifact. The Test Environment communicates with the deployed Microservices and returns test results to the Continuous Integration system. The Microservices read and write data from/to the datastore.
- Anti-patterns to avoid: Monolithic architecture, non-scalable components, non-modular code, manual deployments.

Overall, the solution components are designed to address the quality properties of scalability, agility, reliability, maintainability, and testability. The Microservices architecture style is used to achieve scalability and agility, while the use of design patterns such as Continuous Integration and Blue-Green Deployment addresses reliability and maintainability. The use of automated testing in the Test Environment ensures testability. The interfaces between components are defined to ensure smooth flow of data between them. The Developer, as the main stakeholder, interacts with the system by pushing code changes to the Source Code Repository.

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 suggested C4 component diagram for the proposed solution:



The diagram includes the following:

- Stakeholder: Developer
- Quality properties: Scalability, Agility, Reliability, Maintainability, Testability
- Components: Developer, Source Code Repository, Continuous Integration, Deployment Platform, Build Environment, Test Environment, Microservices, Datastore
- Data types: Code changes, configuration data, test results, data
- Summarized data types: Code changes are pushed from Developer to Source Code Repository and pulled into Continuous Integration. Configuration data is used to deploy Microservices onto Deployment Platform. Test results are generated and communicated between Continuous Integration and Test Environment. Microservices read and write data from/to the datastore.
- Interfaces: The Developer pushes code changes to the Source Code Repository. The Continuous Integration system pulls code changes from the Source Code Repository and builds them in the Build Environment. The Build Environment generates the binary artifact and sends it to the Deployment Platform, which deploys the artifact. The Test Environment communicates with the deployed Microservices and returns test results to the Continuous Integration system. The Microservices read and write data from/to the datastore.
- Anti-patterns to avoid: Monolithic architecture, non-scalable components, non-modular code, manual deployments.

The solution components are designed to address the quality properties of scalability, agility, reliability, maintainability, and testability. The Microservices architectural style is used to achieve scalability and agility, while the use of design patterns such as Continuous Integration and Blue-Green Deployment addresses reliability and maintainability. The use of automated testing in the Test Environment ensures testability. The Datastore component stores and retrieves data used by the Microservices. The interfaces between components are defined to ensure smooth flow of data between them. The Developer, as the main stakeholder, interacts with the system by pushing code changes to the Source Code Repository.

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.

Here is a suggested toolchain to implement the solution:

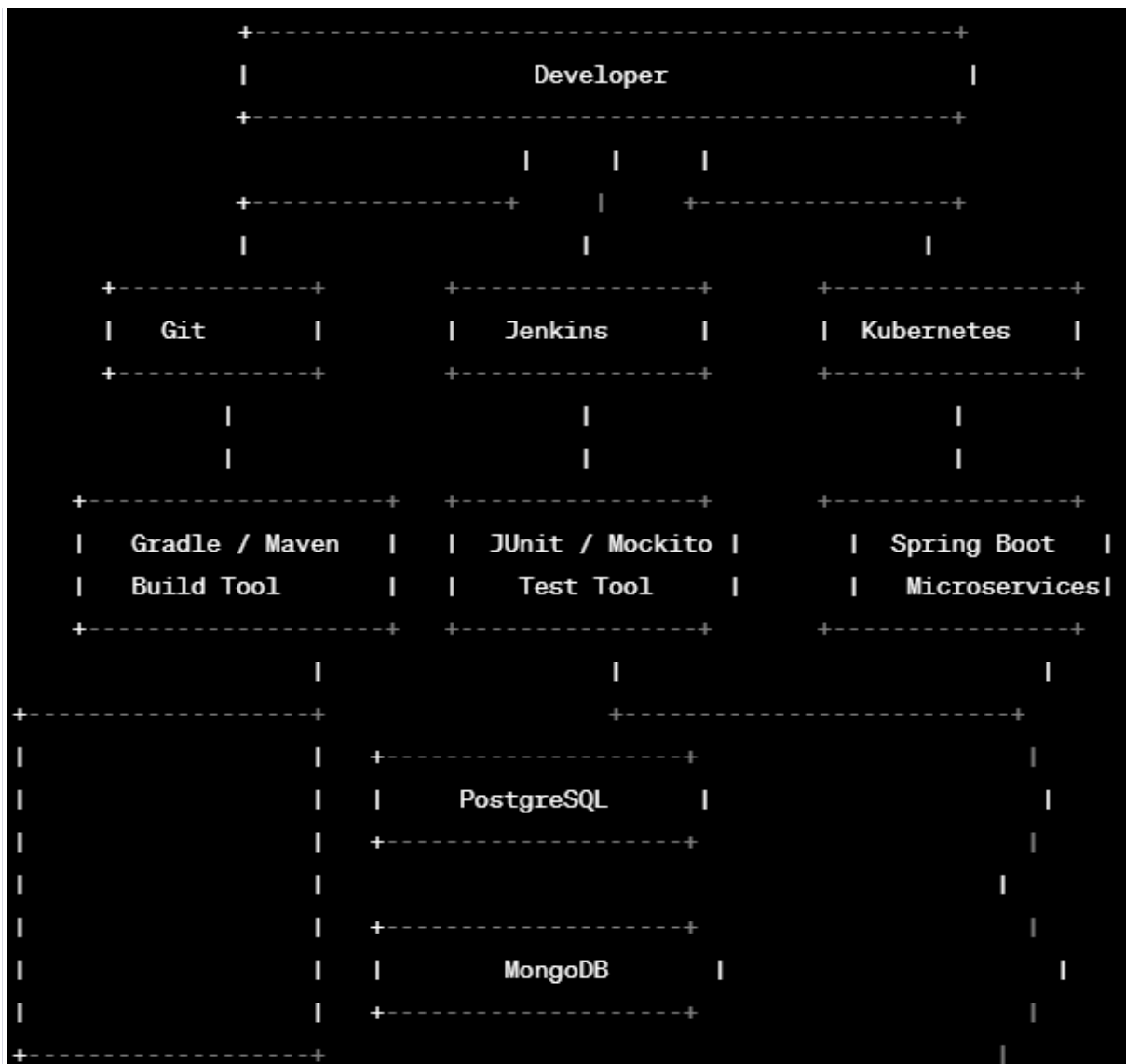
- Source Code Repository: Git
- Continuous Integration: Jenkins or Travis CI
- Deployment Platform: Kubernetes or Docker Swarm
- Build Environment: Gradle or Maven
- Test Environment: JUnit or Mockito
- Microservices: Spring Boot
- Datastore: PostgreSQL or MongoDB

The toolchain includes a mix of widely used and popular tools that are free and open-source. Git is used for version control, while Jenkins or Travis CI can be used for continuous integration. Kubernetes or Docker Swarm can be used for container orchestration and deployment. Gradle or Maven can be used for build automation, and JUnit or Mockito for testing. Spring Boot can be used for building and deploying Microservices. PostgreSQL or MongoDB can be used as the datastore.

The data types transferred between components include code changes, configuration data, test results, and data. Code changes are transmitted between the Developer and the Source Code Repository, and between the Source Code Repository and the Continuous Integration component. Configuration data is transmitted between the Continuous Integration component and the Deployment Platform. Test results are transmitted between the Continuous Integration component and the Test Environment. Data is transmitted between the Microservices and the Datastore.

Here is a revised C4 component diagram that includes the programming language for each component and the data types transmitted between them:





The data types transmitted between components include:

- Code changes (pushed from Developer to Git)
- Binary artifacts (generated by Gradle/Maven and deployed by Kubernetes)
- Configuration data (used by Kubernetes to deploy Microservices)
- Test results (generated by JUnit/Mockito and communicated to Jenkins)
- Data (read and written by Microservices from/to PostgreSQL or MongoDB)

The use of open-source tools makes it easy to integrate and reuse different components of the toolchain. For instance, Spring Boot integrates easily with Kubernetes for container orchestration, while Jenkins can be used for continuous integration and delivery with Git and Gradle/Maven.