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.

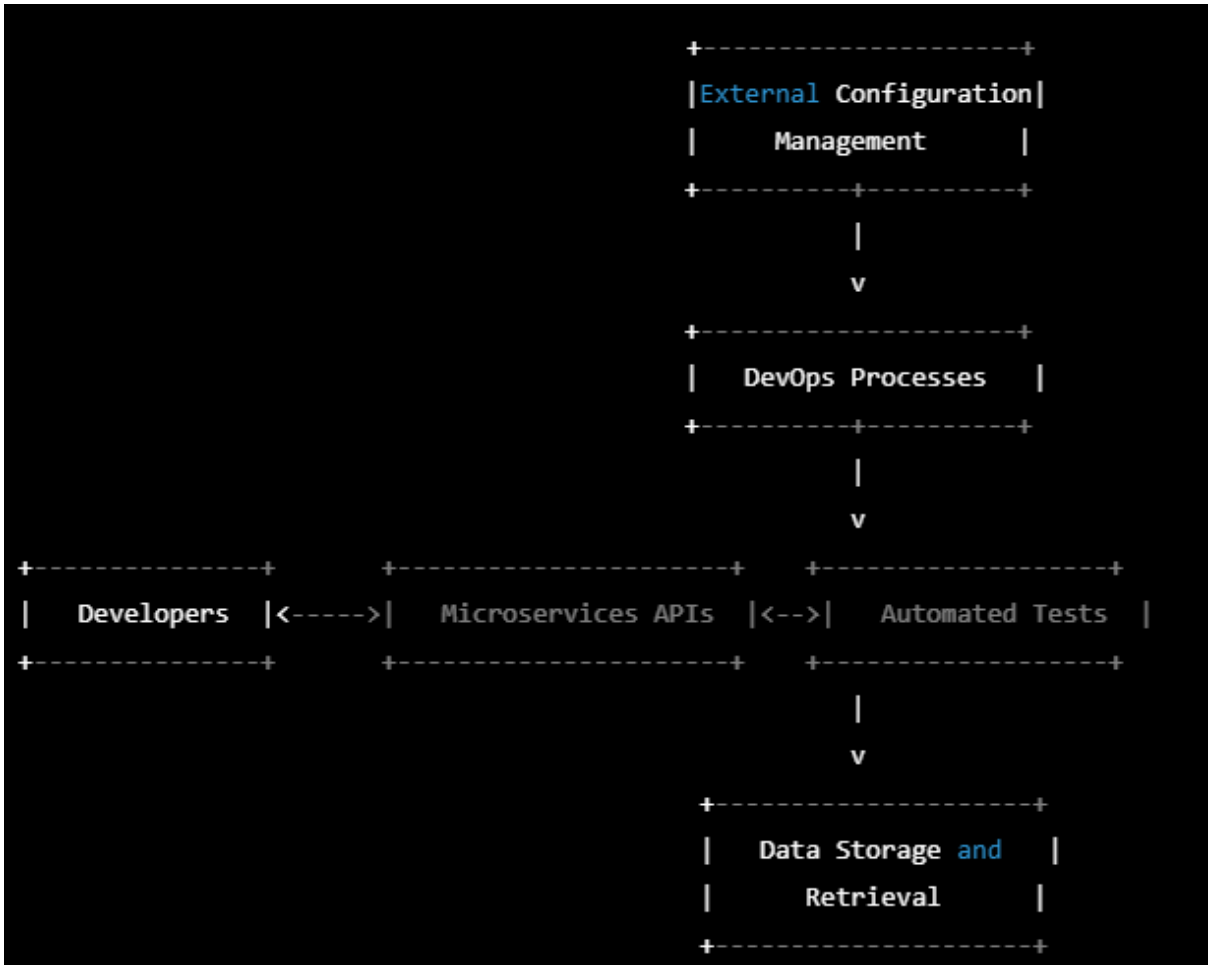
Problem: In the context of a software development team using a microservices architecture, there is a lack of formal processes for documenting the impact of design decisions on continuous integration, and for utilizing design patterns and avoiding anti-patterns to improve the DevOps process.

Solution: A software system architecture solution could include the following components:

- A documentation process for design decisions, patterns, and anti-patterns that impact continuous integration, categorized into different aspects of the software development process.
- Integration of these design decisions, patterns, and anti-patterns into an extended DevOps process, which enforces their use and improves the continuous integration process.
- Use of architectural patterns that support DevOps operations, such as external configurations of microservices or Blue-Green Deployment Pattern, to increase team velocity.
- Avoidance of anti-patterns that minimize the benefits of DevOps processes, such as continuous build anti-pattern or selective automation for developed features.
- Use of patterns that support and simplify automated testing, such as page objects to model web pages, and avoidance of anti-patterns that hinder automated testing.
- Auto-provisioning and de-provisioning of services, with an emphasis on stateless services and external persistent storage for stateful services. Overall, this solution aims to improve the efficiency and effectiveness of the software development process for a team using microservices architecture and DevOps processes.

2. Suggest architecture of a conceptual solution for this problem. We need a C4 system context diagram (https://c4model.com/#SystemContextDiagram). Please use text (arrows, dots, different shapes, and different colours) to visualise the solution. Consider the following: - Identify relevant external systems (if any) and add them to the diagram. - Identify relevant external stakeholders (if any) and add them to the diagram. - Identify high level abstract data types transferred between components and stakeholders. - Summarise data types transmitted between components. Add this information to the connections and interfaces between components.

Based on the problem statement and proposed solution, the following is a C4 system context diagram for the conceptual solution:



The diagram shows the system context for the conceptual solution, with three external components and two internal components. The external configuration management system is responsible for configuring and managing external dependencies. The DevOps processes component includes the continuous integration and delivery pipeline, which integrates code changes and deploys the microservices. The microservices APIs component provides interfaces for developers to access and modify the microservices. The automated tests component performs automated tests to verify changes to the code. Finally, the data storage and retrieval component is responsible for storing and retrieving data used by the microservices.

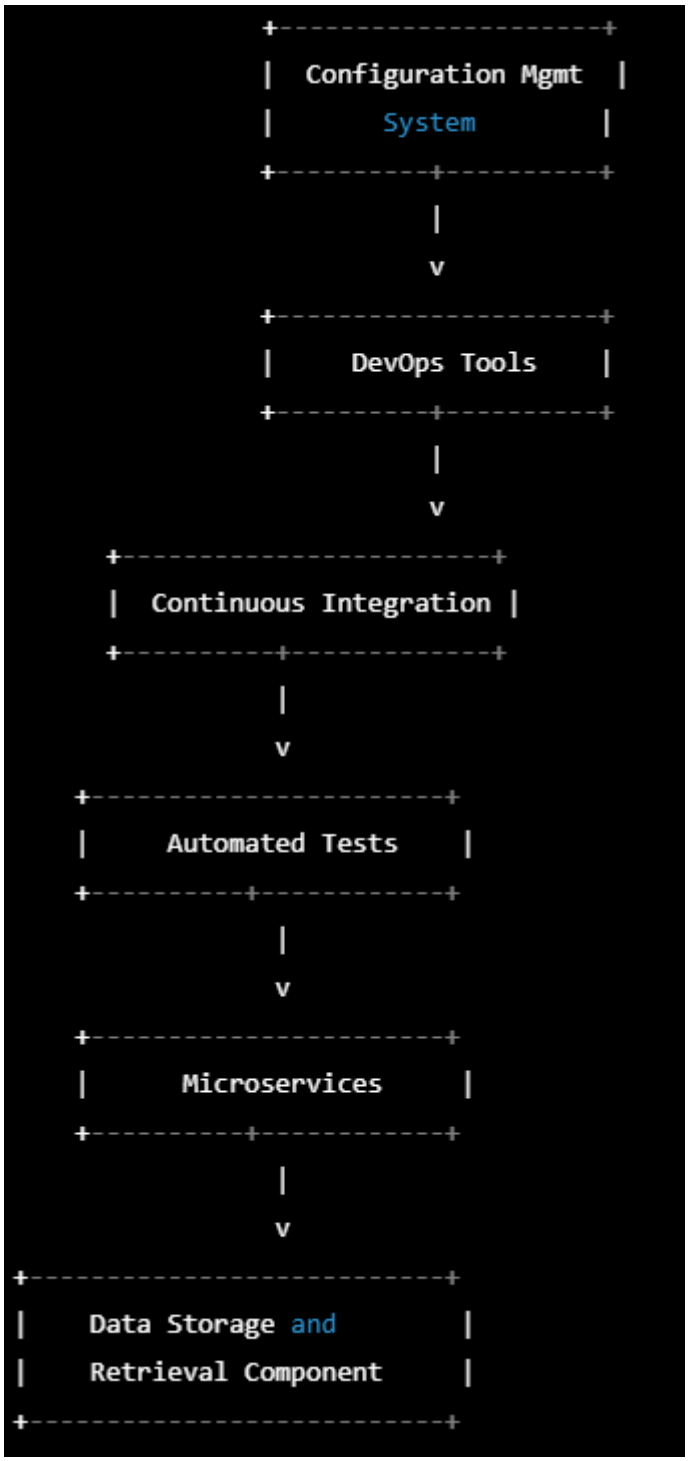
Developers interact with the microservices APIs to create and modify microservices, while the automated tests component interacts with the microservices APIs to verify changes to the code. Data is transmitted between the microservices APIs and the data storage and retrieval component to retrieve and store data used by the microservices.

Overall, this conceptual solution aims to support DevOps processes and continuous integration while utilizing design patterns and avoiding anti-patterns.

3. Suggest architectural components that we need to solve this problem. We need a C4 container solution diagram (https://c4model.com/#ContainerDiagram). Please use text (arrows, dots, different shapes, and different colours) to visualise the solution. Consider the following: - Define how the main stakeholder will use the solution. Add it to the diagram.. - Which quality properties are important? Suggest adequate architectural solutions to address them (architectural style,

design patterns, design tactics, components and their relationships). Structure the components according to it. - Define data types transferred between components. - Summarise data types transmitted between components. Add this information to the connections and interfaces between components. - Define interfaces between components. Add information about required and provided interfaces to the components, along with the data types. - Document which anti-patterns we need to avoid when implementing this solution.

Based on the problem statement and proposed solution, the following is a C4 container diagram for the proposed architecture:



The main stakeholder using the solution is the development team, who use the DevOps tools and continuous integration to build, test, and deploy microservices.

Quality properties important for this solution include scalability, agility, reliability, and maintainability. To address these properties, the proposed architecture utilizes a microservices architectural style, with design patterns such as external configurations, Blue-Green Deployment Pattern, and page objects for automated testing.

Data types transmitted between components include code changes, configuration data, test data, and data used by the microservices.

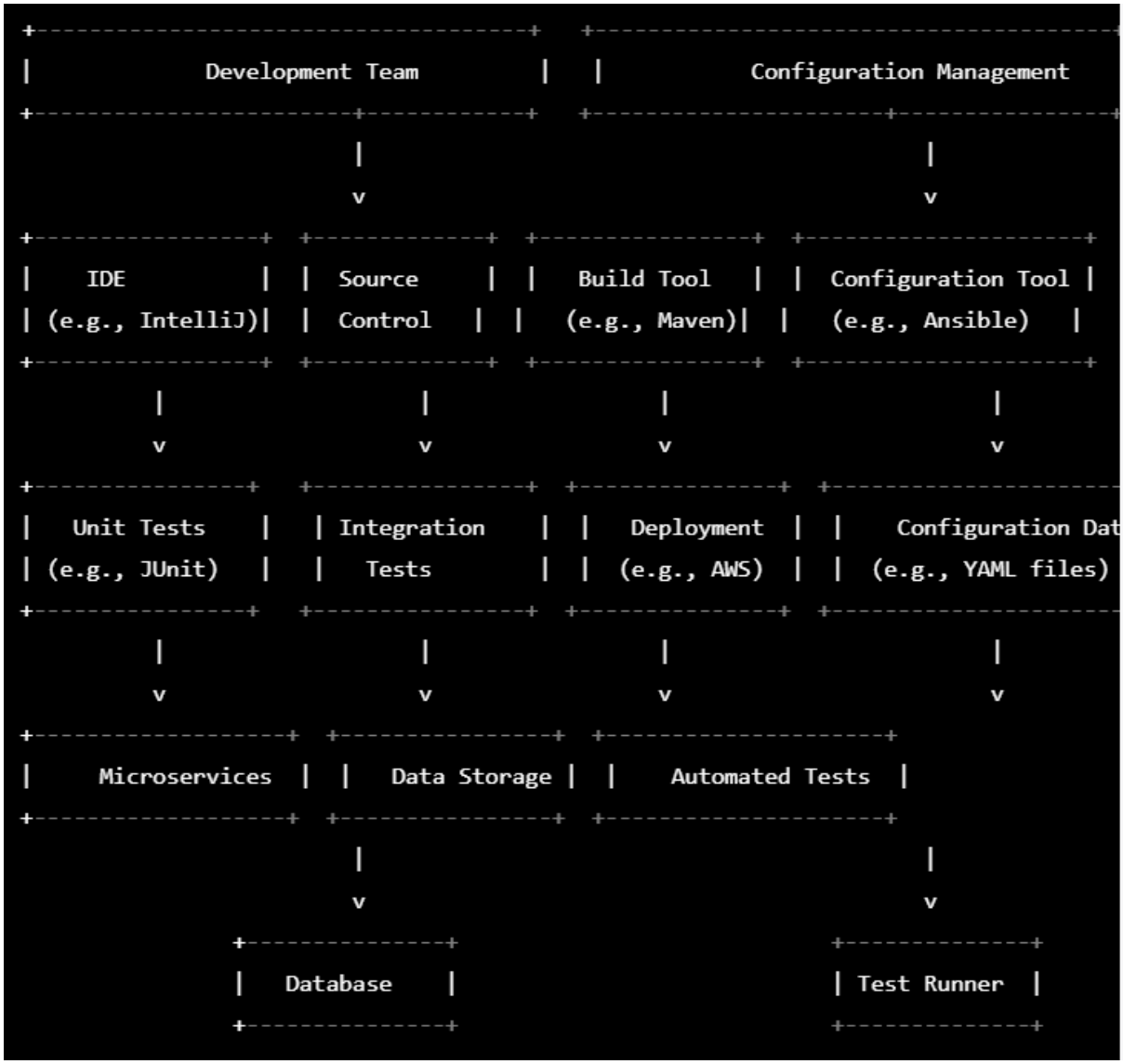
Interfaces between components include those required by the DevOps tools to deploy microservices and run automated tests, as well as those provided by the microservices APIs to allow developers to create and modify microservices. Data types transmitted between components are specified in the interfaces, along with required and provided interfaces.

Anti-patterns to avoid when implementing this solution include continuous build anti-pattern, selective automation for developed features, and other anti-patterns that hinder automated testing or the DevOps process.

Overall, this solution aims to improve efficiency and effectiveness in the software development process by utilizing DevOps tools, continuous integration, and microservices architecture, while avoiding anti-patterns that hinder the process.

4. Suggest architectural components that we need to solve this problem. We need a C4 component solution diagram (https://c4model.com/#ComponentDiagram). Please use text (arrows, dots, different shapes, and different colours) to visualise the solution. Consider following: - Define how the main stakeholder will use the solution. Add it to the diagram. - Which quality properties are important? Suggest adequate architectural solutions to address them (architectural style, design patterns, design tactics, components and their relationships). Structure the components according to it. - Define data types transferred between components. - Summarise data types transmitted between components. Add this information to the connections and interfaces between components. - Define interfaces between components. Add information about required and provided interfaces to the components, along with the data types. - Document which anti-patterns we need to avoid when implementing this solution.

Based on the problem statement and proposed solution, the following is a C4 component diagram for the proposed architecture:



The development team uses the IDE to write code and the source control system to manage changes. The build tool is used to build the code and create artifacts, while the configuration management tool is used to manage and configure the infrastructure. The unit and integration tests are used to verify the code changes and the deployment tool is used to deploy the microservices to the cloud. Data storage is used to store data used by the microservices, while automated tests are used to verify changes made to the codebase.

Quality properties important for this solution include scalability, agility, reliability, and maintainability. To address these properties, the proposed architecture utilizes microservices architecture style, with design patterns such as external configurations and Blue-Green Deployment Pattern. Additionally, unit and integration tests are used to ensure code quality and automated testing is utilized to speed up testing and deployment.

Data types transmitted between components include code changes, configuration data, test data, and data used by the microservices.

Interfaces between components include those required by the build tool to compile and package the code, as well as those required by the deployment tool to deploy the microservices. Data

types transmitted between components are specified in the interfaces, along with required and provided interfaces.

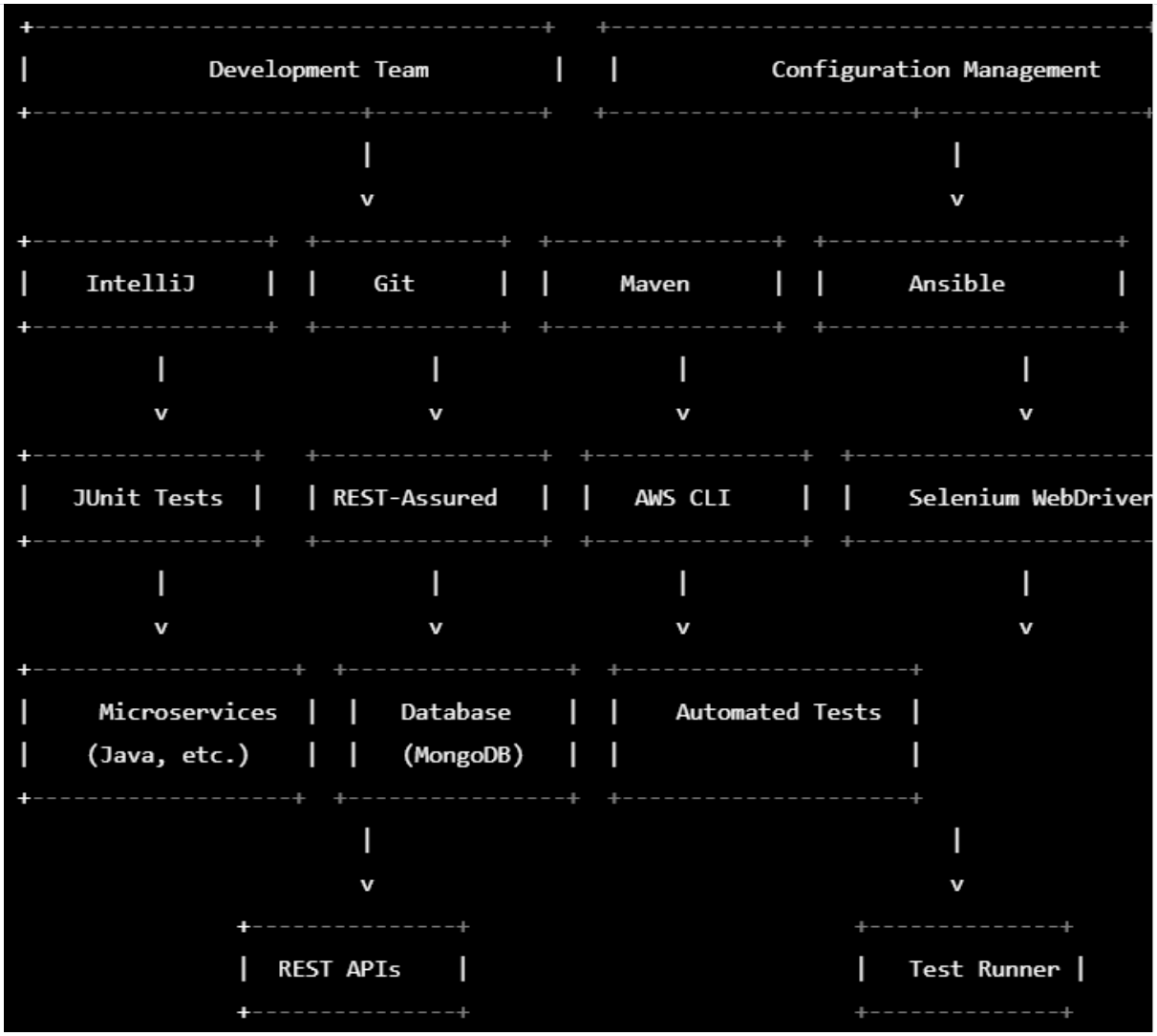
Anti-patterns to avoid when implementing this solution include continuous build anti-pattern, selective automation for developed features, and other anti-patterns that hinder automated testing or the DevOps process.

Overall, this solution aims to improve efficiency and effectiveness in the software development process by utilizing DevOps tools, continuous integration, and microservices architecture, while avoiding anti-patterns that hinder the process.

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.

Based on the proposed architecture, the following toolchain can be used to implement the solution:

- IDE: IntelliJ IDEA (reused)
- Source Control: Git (reused)
- Build Tool: Maven (reused)
- Configuration Management: Ansible (reused)
- Unit Tests: JUnit (reused)
- Integration Tests: REST-Assured (reused)
- Deployment: Amazon Web Services (AWS)
- Database: MongoDB
- Automated Testing: Selenium WebDriver



Data types transmitted between components include code changes, configuration data, test data, and data used by the microservices. The REST APIs are used to communicate between the microservices and with external systems. The connections and interfaces between components specify the data types transmitted between components, along with the required and provided interfaces.

Overall, this toolchain aims to provide an integrated and efficient development and deployment process, while utilizing established tools and technologies to minimize development time and cost.