

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 challenge is to improve the continuous integration process in a microservices architecture. The development team needs to shorten the release cycle, increase the frequency of merging changes, identify and enforce architectural patterns that support DevOps, avoid anti-patterns, promote automated testing, extend the DevOps process, and support auto-provisioning and de-provisioning of services.

**Solution:** To address these challenges, a software system architecture solution can be implemented using design patterns and anti-patterns. The system should support automated testing, external persistent storage for stateful services, and use architectural patterns that support DevOps operations. The DevOps process should be extended to include the knowledge of patterns and anti-patterns and enforce their use as part of the process. An automated build process should be implemented to detect integration errors during continuous integration. Services should be developed and deployed independently and communication should be done via lightweight mechanisms, such as an HTTP resource API.

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.

Unfortunately, as an AI language model, I cannot generate graphical content or diagrams. However, I can provide a textual representation of the conceptual solution architecture.

System Context Diagram:

- External System: Cloud-based microservices platform
- External Stakeholder: End users
- Components:
  - Microservices
  - DevOps pipeline
  - Automated testing tool
  - Configuration management tool
  - Code versioning repository
  - Monitoring and logging tool
  - External persistent storage
- Abstract Data Types:
  - Code changes and updates
  - Test results and reports

- Deployment configurations
- Monitoring data and logs
- External storage data
- Summary of Data Types Transmitted:
  - Microservices: Receive and transmit code changes and updates, deployment configurations, and external storage data.
  - DevOps pipeline: Receive and transmit code changes and updates, deployment configurations, and monitoring data and logs.
  - Automated testing tool: Receive code changes and updates, transmit test results and reports.
  - Configuration management tool: Receive deployment configurations and transmit them to microservices and DevOps pipeline.
  - Code versioning repository: Receive and transmit code changes and updates.
  - Monitoring and logging tool: Receive and transmit monitoring data and logs.
  - External persistent storage: Store and transmit external storage data.

The microservices architecture is deployed on a cloud-based platform, which serves as an external system. End users are external stakeholders who interact with the system. The microservices communicate with each other via lightweight mechanisms, such as an HTTP resource API, and receive and transmit code changes and updates, deployment configurations, and external storage data. The DevOps pipeline receives and transmits the same data types and also receives monitoring data and logs. An automated testing tool receives code changes and updates and transmits test results and reports. The configuration management tool receives deployment configurations and transmits them to microservices and the DevOps pipeline. The code versioning repository receives and transmits code changes and updates. The monitoring and logging tool receives and transmits monitoring data and logs. The external persistent storage stores and transmits external storage data.

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.

Unfortunately, as an AI language model, I cannot generate graphical content or diagrams. However, I can provide a textual representation of the solution architecture.

Container Diagram:

- Main Stakeholder: Development team
- Components:
  - User Interface
  - Microservices

	<ul style="list-style-type: none"> <li>• DevOps pipeline</li> <li>• Automated testing tool</li> <li>• Configuration management tool</li> <li>• Code versioning repository</li> <li>• Monitoring and logging tool</li> <li>• External persistent storage</li> </ul>
• Quality Properties:	<ul style="list-style-type: none"> <li>• Scalability: Achieved through the use of microservices architecture, which allows for horizontal scaling of individual services.</li> <li>• Agility: Achieved through the use of DevOps process, which supports parallel development and reduces operational overhead.</li> <li>• Reliability: Achieved through the use of monitoring and logging tools, which provide real-time feedback and enable quick resolution of issues.</li> <li>• Maintainability: Achieved through the use of configuration management tools and code versioning repository, which allow for easy tracking of changes and updates.</li> </ul>
• Data Types:	<ul style="list-style-type: none"> <li>• Code changes and updates</li> <li>• Test results and reports</li> <li>• Deployment configurations</li> <li>• Monitoring data and logs</li> <li>• External storage data</li> </ul>
• Summary of Data Types Transmitted:	<ul style="list-style-type: none"> <li>• User Interface: Sends and receives deployment configurations and monitoring data and logs.</li> <li>• Microservices: Receive and transmit code changes and updates, deployment configurations, and external storage data.</li> <li>• DevOps pipeline: Receive and transmit code changes and updates, deployment configurations, and monitoring data and logs.</li> <li>• Automated testing tool: Receive code changes and updates, transmit test results and reports.</li> <li>• Configuration management tool: Receive deployment configurations and transmit them to microservices and DevOps pipeline.</li> <li>• Code versioning repository: Receive and transmit code changes and updates.</li> <li>• Monitoring and logging tool: Receive and transmit monitoring data and logs.</li> <li>• External persistent storage: Store and transmit external storage data.</li> </ul>
• Interfaces:	<ul style="list-style-type: none"> <li>• User Interface: Requires deployment configurations and monitoring data and logs, provides feedback to end users.</li> <li>• Microservices: Requires code changes and updates, deployment configurations, and external storage data. Provides functionality to end users.</li> <li>• DevOps pipeline: Requires code changes and updates, deployment configurations, and monitoring data and logs. Provides deployment and continuous integration functionality.</li> <li>• Automated testing tool: Requires code changes and updates. Provides test results and reports.</li> <li>• Configuration management tool: Requires deployment configurations. Provides deployment configurations to microservices and DevOps pipeline.</li> <li>• Code versioning repository: Requires and provides code changes and updates.</li> </ul>

- Monitoring and logging tool: Requires and provides monitoring data and logs.
- External persistent storage: Requires and provides external storage data.
- Anti-patterns to avoid:
  - Continuous Build Anti-pattern: Build automation should be selective, not continuous, to avoid excessive build time and resource consumption.
  - Selective Automation Anti-pattern: All developed features should be subject to automated testing to ensure consistent and reliable functionality.

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.

Unfortunately, as an AI language model, I cannot generate graphical content or diagrams. However, I can provide a textual representation of the solution architecture.

Component Diagram:

- Main Stakeholder: Development team
- Components:
  - User Interface
  - Microservice A
  - Microservice B
  - Microservice C
  - DevOps Pipeline
  - Build Automation
  - Automated Testing Tool
  - Configuration Management Tool
  - Code Versioning Repository
  - Monitoring and Logging Tool
  - External Persistent Storage
- Quality Properties:
  - Scalability: Achieved through the use of microservices architecture, which allows for horizontal scaling of individual services.
  - Agility: Achieved through the use of DevOps process, which supports parallel development and reduces operational overhead.
  - Reliability: Achieved through the use of monitoring and logging tools, which provide real-time feedback and enable quick resolution of issues.
  - Maintainability: Achieved through the use of configuration management tools and code versioning repository, which allow for easy tracking of changes and updates.
- Data Types:

- Code changes and updates
- Test results and reports
- Deployment configurations
- Monitoring data and logs
- External storage data

- Summary of Data Types Transmitted:

- User Interface: Sends and receives deployment configurations and monitoring data and logs.
- Microservice A: Receives and transmits code changes and updates, deployment configurations, and external storage data.
- Microservice B: Receives and transmits code changes and updates, deployment configurations, and external storage data.
- Microservice C: Receives and transmits code changes and updates, deployment configurations, and external storage data.
- DevOps Pipeline: Receives and transmits code changes and updates, deployment configurations, and monitoring data and logs.
- Build Automation: Receives and transmits code changes and updates, builds and verifies code changes, and transmits successful builds to DevOps Pipeline.
- Automated Testing Tool: Receives code changes and updates, runs automated tests, and transmits test results and reports to DevOps Pipeline.
- Configuration Management Tool: Receives deployment configurations, provides deployment configurations to microservices and DevOps Pipeline.
- Code Versioning Repository: Receives and transmits code changes and updates.
- Monitoring and Logging Tool: Receives and transmits monitoring data and logs.
- External Persistent Storage: Stores and transmits external storage data.

- Interfaces:

- User Interface: Requires deployment configurations and monitoring data and logs, provides feedback to end users.
- Microservice A: Requires and provides functionality to end users, requires deployment configurations and external storage data.
- Microservice B: Requires and provides functionality to end users, requires deployment configurations and external storage data.
- Microservice C: Requires and provides functionality to end users, requires deployment configurations and external storage data.
- DevOps Pipeline: Requires and provides deployment configurations and monitoring data and logs, orchestrates the deployment of microservices.
- Build Automation: Requires and provides code changes and updates, provides successful builds to DevOps Pipeline.
- Automated Testing Tool: Requires code changes and updates, provides test results and reports to DevOps Pipeline.
- Configuration Management Tool: Requires and provides deployment configurations.
- Code Versioning Repository: Requires and provides code changes and updates.
- Monitoring and Logging Tool: Requires and provides monitoring data and logs.
- External Persistent Storage: Requires and provides external storage data.

- Anti-patterns to avoid:

- Continuous Build Anti-pattern: Build automation should be selective, not continuous, to avoid excessive build time and resource consumption.

- Selective Automation Anti-pattern: All developed features should be subject to automated testing to

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.

#### Toolchain:

- Code Versioning: Git
- Build Automation: Jenkins
- Configuration Management: Ansible
- Automated Testing: Selenium
- Monitoring and Logging: ELK Stack (Elasticsearch, Logstash, Kibana)
- External Persistent Storage: Amazon S3

#### Programming Languages:

- Microservice A: Java
- Microservice B: Python
- Microservice C: Node.js

#### Data Types:

- Code changes and updates: Text files
- Test results and reports: JSON files
- Deployment configurations: YAML files
- Monitoring data and logs: JSON files
- External storage data: Binary files (images, videos, etc.)

#### Summary of Data Types Transmitted:

- User Interface: Sends and receives deployment configurations and monitoring data and logs in YAML and JSON formats.
- Microservice A: Receives and transmits code changes and updates, deployment configurations, and external storage data in text and binary formats.
- Microservice B: Receives and transmits code changes and updates, deployment configurations, and external storage data in text and binary formats.
- Microservice C: Receives and transmits code changes and updates, deployment configurations, and external storage data in text and binary formats.
- DevOps Pipeline: Receives and transmits code changes and updates, deployment configurations, and monitoring data and logs in text and JSON formats.
- Build Automation: Receives and transmits code changes and updates, builds and verifies code changes, and transmits successful builds to DevOps Pipeline in text format.
- Automated Testing Tool: Receives code changes and updates, runs automated tests, and transmits test results and reports to DevOps Pipeline in JSON format.



- Configuration Management Tool: Receives deployment configurations, provides deployment configurations to microservices and DevOps Pipeline in YAML format.
- Code Versioning Repository: Receives and transmits code changes and updates in text format.
- Monitoring and Logging Tool: Receives and transmits monitoring data and logs in JSON format.
- External Persistent Storage: Stores and transmits external storage data in binary format.

#### Interfaces:

- User Interface: Requires deployment configurations and monitoring data and logs in YAML and JSON formats, provides feedback to end users.
- Microservice A: Requires and provides functionality to end users, requires deployment configurations and external storage data in text and binary formats.
- Microservice B: Requires and provides functionality to end users, requires deployment configurations and external storage data in text and binary formats.
- Microservice C: Requires and provides functionality to end users, requires deployment configurations and external storage data in text and binary formats.
- DevOps Pipeline: Requires and provides deployment configurations and monitoring data and logs in YAML and JSON formats, orchestrates the deployment of microservices.
- Build Automation: Requires and provides code changes and updates in text format, provides successful builds to DevOps Pipeline in text format.
- Automated Testing Tool: Requires code changes and updates in text format, provides test results and reports to DevOps Pipeline in JSON format.
- Configuration Management Tool: Requires and provides deployment configurations in YAML format.
- Code Versioning Repository: Requires and provides code changes and updates in text format.
- Monitoring and Logging Tool: Requires and provides monitoring data and logs in JSON format.
- External Persistent Storage: Requires and provides external storage data in binary format.