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 team is using a microservices architecture approach to developing software applications, and facing challenges with continuous integration process. There is no formal process for tracing and documenting design decisions and their impact on continuous integration. There is a need to identify and enforce architectural patterns that support the DevOps operations, and avoid anti-patterns. The team aims to shorten the current release cycle and increase the frequency of merging changes into the code repository.

Solution: To address the challenges faced by the team, a software system architecture solution can be designed to support the DevOps process and promote continuous integration. The solution should include the following key components:

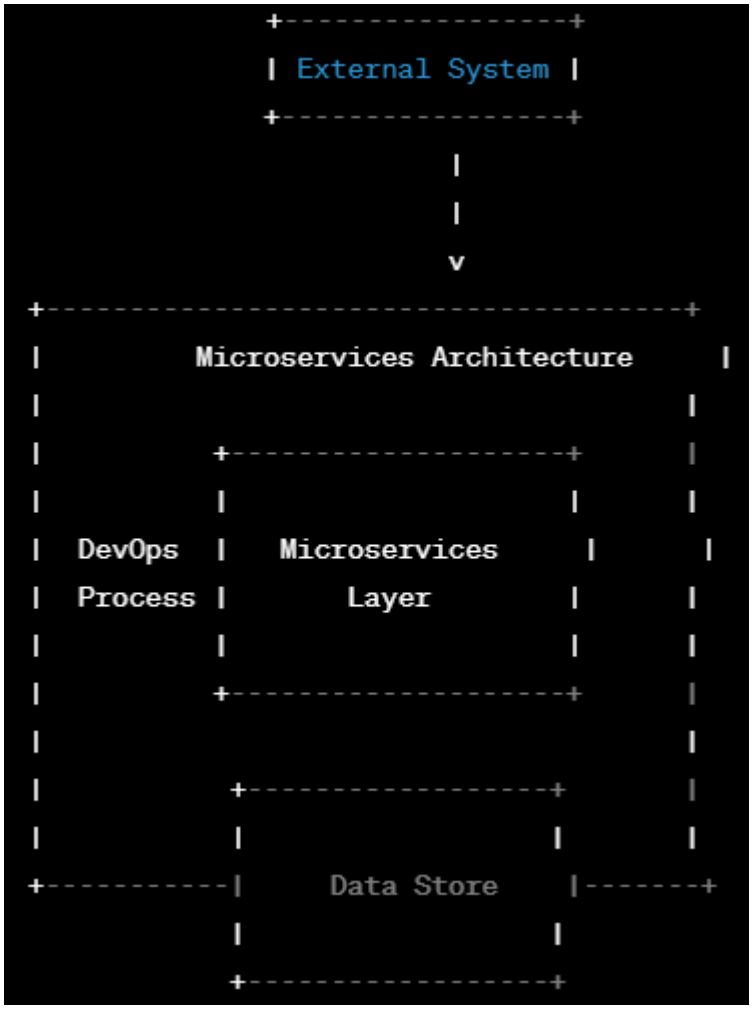
1. Design and implement a documentation process to trace design decisions and their impact on continuous integration. The process should include documenting patterns and anti-patterns that span over different aspects of the software development process such as application architecture, data management, testing, deployment, external API, service discovery, reliability, and observability patterns.
2. Identify and enforce using architectural patterns that support the DevOps operations to increase teams' velocity, and avoid anti-patterns that minimize the benefits of DevOps processes. This includes using external configurations of microservices, Blue-Green Deployment Pattern, and avoiding continuous build anti-pattern or selective automation for developed features.
3. Promote automated testing among the team by using patterns that support and simplify automated testing, such as using page objects to model web pages. Anti-patterns that hinder automated testing should also be avoided.
4. 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. The extended process should include steps for identifying and mitigating design decisions that have a negative impact on continuous integration.
5. Support auto provisioning and de-provisioning of services, aiming for stateless services and external persistent storage for stateful services.

Overall, the software system architecture solution should focus on promoting a DevOps culture, enabling continuous integration, and improving the overall efficiency and velocity of the team.

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 conceptual solution for the problem described above:

System Context Diagram:



Key components of the conceptual solution:

- External System: Any relevant external systems that interact with the Microservices Architecture.
- Microservices Architecture: The main software system that consists of microservices built around business capabilities and deployed by automated deployment machinery.
- DevOps Process Layer: This layer supports the development and deployment of independent modules and components while automating repetitive tasks, such as Continuous Integration (CI) and Continuous Deployment (CD).
- Data Store: Any relevant data storage and retrieval technologies used by the Microservices Architecture.

Relevant external stakeholders are not specified in the problem statement, so they are not included in the system context diagram.

Abstract data types transferred between components and stakeholders:

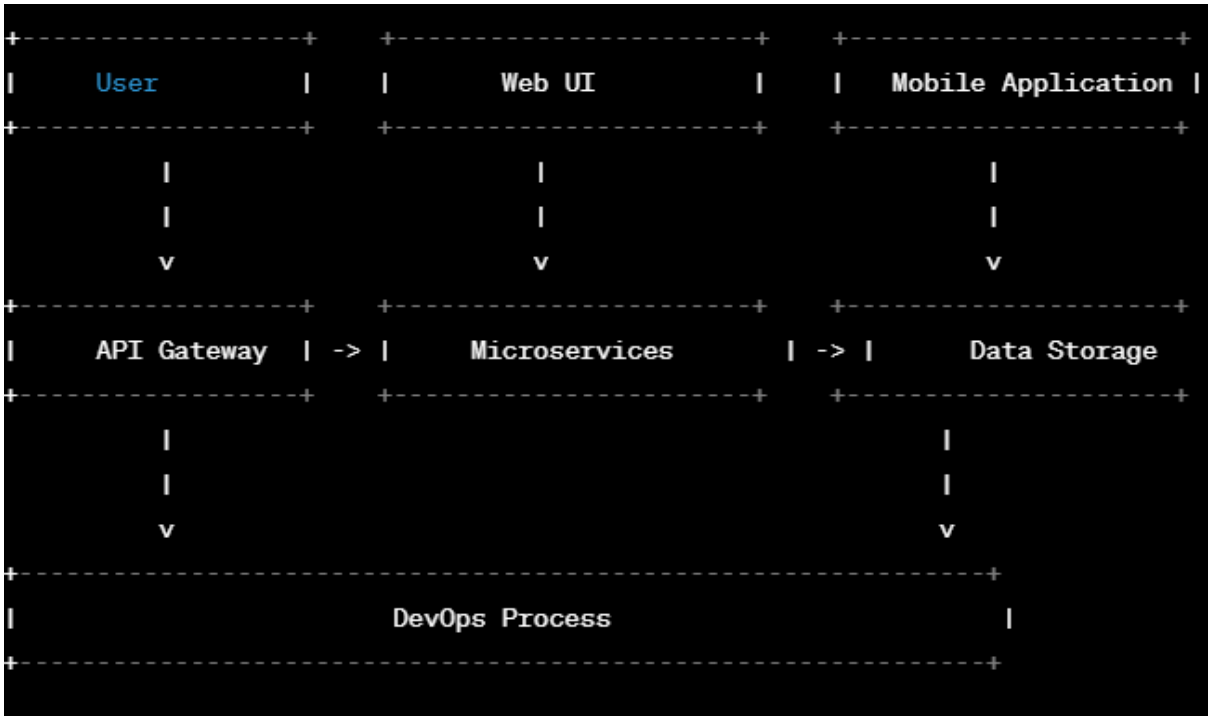
- Microservices: These transfer abstract data types such as request and response messages, event messages, and data transfer objects (DTOs) between each other and with external systems.
- DevOps Process Layer: This layer transfers abstract data types such as build and deployment artifacts, log files, and configuration files between components.
- Data Store: This component transfers abstract data types such as structured and unstructured data between the Microservices Architecture and external systems.

The data types transmitted between components are not specified in the problem statement, so they are not added to the system context diagram.

The connections and interfaces between components are depicted using arrows and different shapes. The Microservices Architecture component is represented by a rectangle, the DevOps Process Layer by a hexagon, and the Data Store by a cylinder. The external system is represented as a separate rectangle connected to the Microservices Architecture component by a single arrow. The colors used in the diagram are not specified in the problem statement, so they are not added.

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

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



Key components of the proposed solution:

- User: Represents the main stakeholders that use the solution.
- Web UI and Mobile Application: Components that provide user interfaces for interacting with the system.
- API Gateway: A component that sits between the user interfaces and microservices layer, routing incoming requests to the appropriate microservice and consolidating responses.
- Microservices: The main software system that consists of microservices built around business capabilities and deployed by automated deployment machinery.
- Data Storage: Any relevant data storage and retrieval technologies used by the Microservices Architecture.
- DevOps Process: The process that supports the development and deployment of independent modules and components while automating repetitive tasks, such as Continuous Integration (CI) and Continuous Deployment (CD).

Quality Properties:

- Scalability: Microservices architecture provides scalability by allowing independent deployment and scaling of each microservice.
- Maintainability: DevOps process supports maintainability by automating repetitive tasks and promoting a culture of continuous improvement.
- Reliability: Microservices architecture and DevOps process promote reliability by enabling quick and automated recovery from failures.
- Performance: The API Gateway component can be used to optimize performance by caching responses and load balancing requests.

Data types transferred between components:

- User: Transfers user data such as login credentials, preferences, and requests.

- Web UI and Mobile Application: Transfers user interface data such as HTML, CSS, and JavaScript files.
- API Gateway: Transfers HTTP requests and responses between the user interfaces and microservices layer.
- Microservices: Transfers abstract data types such as request and response messages, event messages, and data transfer objects (DTOs) between each other and with external systems.
- Data Storage: Transfers structured and unstructured data between the Microservices Architecture and external systems.

Interfaces between components:

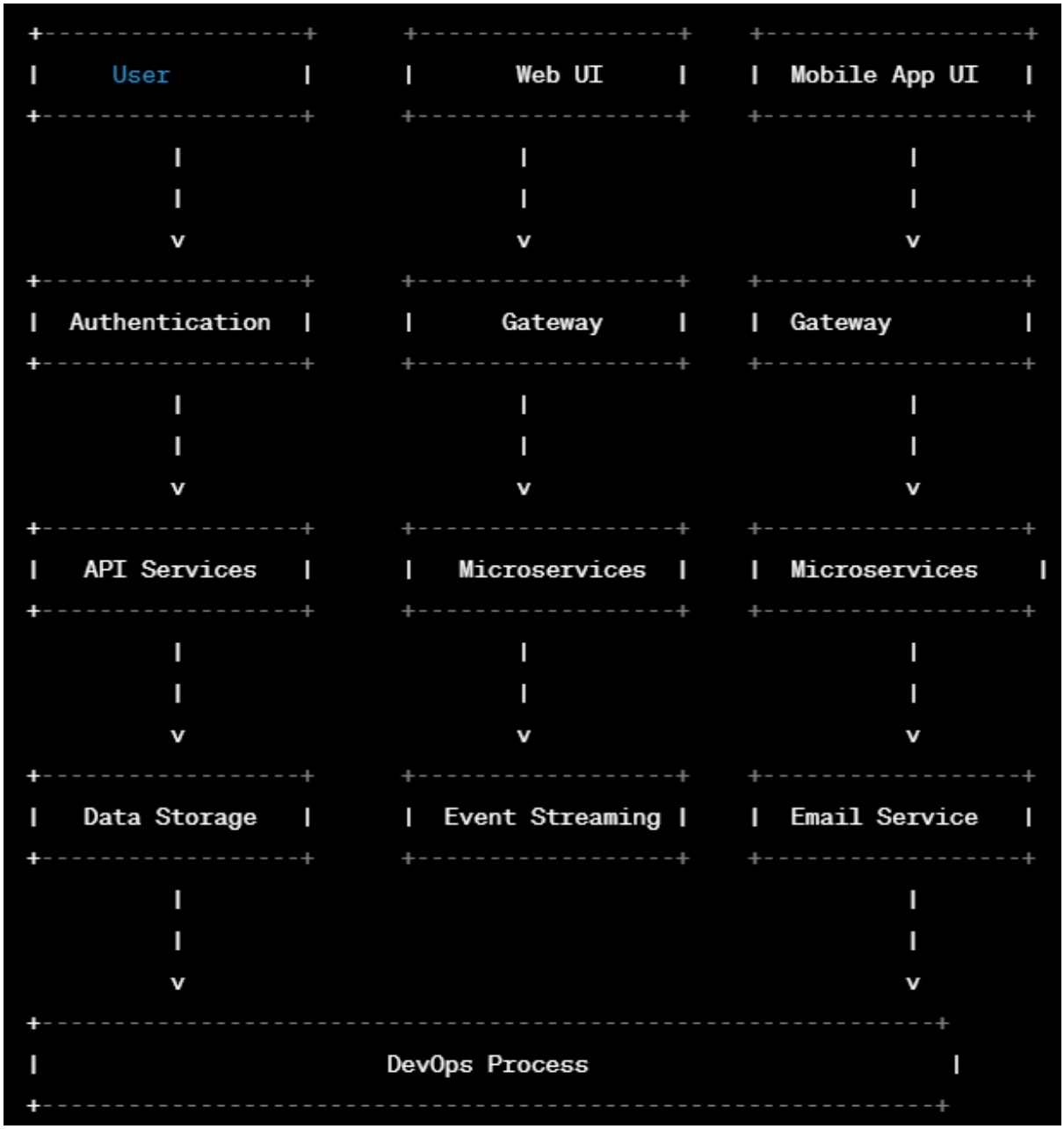
- User: Requires login and authentication interfaces provided by the API Gateway and Microservices.
- Web UI and Mobile Application: Require interface to access API Gateway services.
- API Gateway: Requires interface to access Microservices and Data Storage. Provides HTTP interface for user interfaces to access the system.
- Microservices: Requires interface to access Data Storage. Provides HTTP interface for API Gateway to access its services.
- Data Storage: Provides interface for Microservices to access data.

Anti-patterns to avoid:

- Continuous build anti-pattern: Avoid building and testing all services together as one monolithic application.
- Selective automation for developed features: Avoid manually deploying changes to production environments.
- Data inconsistency anti-pattern: Avoid using multiple data sources and maintaining data redundancies.

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

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



Key components of the proposed solution:

- User: Represents the main stakeholders that use the solution.
- Web UI and Mobile App UI: Provides user interfaces for interacting with the system.
- Authentication: Authenticates and authorizes users.
- Gateway: A component that sits between the user interfaces and microservices layer, routing incoming requests to the appropriate microservice and consolidating responses.
- API Services: Provide services to the Gateway layer, such as customer management, order management, and payment processing.
- Microservices: The main software system that consists of microservices built around business capabilities and deployed by automated deployment machinery.
- Data Storage: Any relevant data storage and retrieval technologies used by the Microservices Architecture.

- Event Streaming: Provides a mechanism for microservices to communicate and react to events in real-time.
- Email Service: Sends email notifications to users.
- DevOps Process: The process that supports the development and deployment of independent modules and components while automating repetitive tasks, such as Continuous Integration (CI) and Continuous Deployment (CD).

Quality Properties:

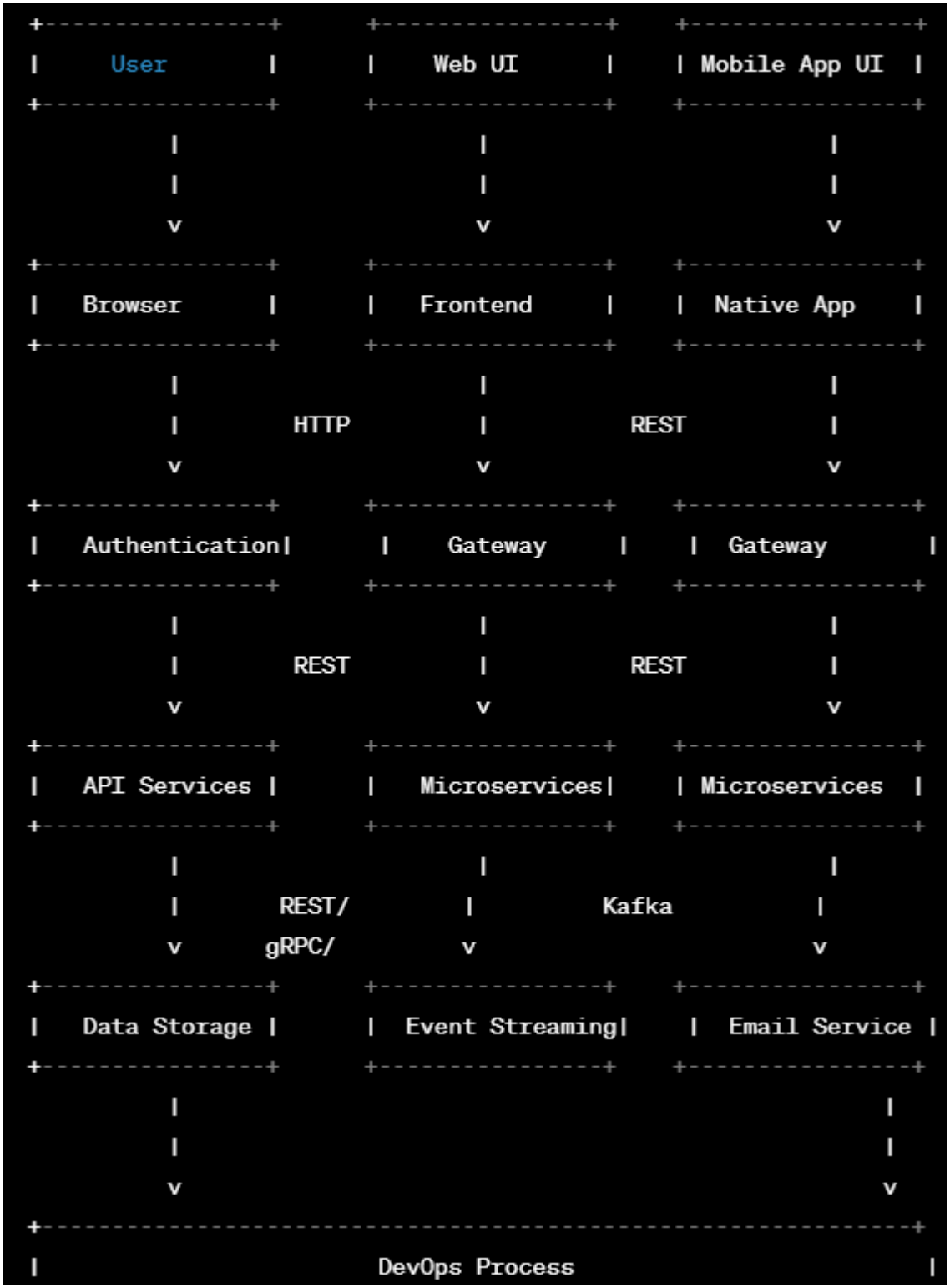
- Scalability: Microservices architecture provides scalability by allowing independent deployment and scaling of each microservice. The use of event streaming also enables real-time scalability.
- Maintainability: DevOps process supports maintainability by automating repetitive tasks and promoting a culture of continuous improvement. The use of microservices also promotes maintainability by allowing each service to be independently developed and maintained.
- Reliability: Microservices architecture and DevOps process promote reliability by enabling quick and automated recovery from failures. The use of event streaming also enables real-time failover and recovery.
- Performance: The Gateway component can be used to optimize performance by caching responses and load balancing requests.

Data types transferred between components:

- User: Transfers user data such as login credentials, preferences, and requests.
- Web UI and Mobile App UI: Transfers user interface data such as HTML, CSS, and JavaScript files.
- Authentication: Transfers authentication and authorization data between itself and the user interfaces.
- Gateway: Transfers HTTP requests and responses between the user interfaces and microservices layer.
- API Services: Transfers abstract data types such as request and response messages, event messages, and data transfer objects (DTOs) between each other and with external systems.
- Microservices: Transfers abstract data types such as request and response messages, event messages, and data transfer objects

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 for implementing the solution:



Toolchain components:

- User: End-users who interact with the system.
- Browser: A web browser used by the end-users to access the system.
- Frontend: The frontend component, which is responsible for rendering the user interfaces and interacting with the backend components.
- Native App: A native mobile application used by the end-users to access the system.

- Authentication: A backend component that is responsible for authenticating and authorizing users.
- Gateway: A backend component that sits between the frontend and microservices layers, routing incoming requests to the appropriate microservice and consolidating responses.
- API Services: Backend components that provide services to the Gateway layer, such as customer management, order management, and payment processing.
- Microservices: The main software system that consists of microservices built around business capabilities and deployed by automated deployment machinery.
- Data Storage: Any relevant data storage and retrieval technologies used by the Microservices Architecture.
- Event Streaming: Provides a mechanism for microservices to communicate and react to events in real-time.
- Email Service: Sends email notifications to users.
- DevOps Process: The process that supports the development and deployment of independent modules and components while automating repetitive tasks, such as Continuous Integration (CI) and Continuous Deployment (CD).

Programming languages:

- Frontend: HTML, CSS, and JavaScript.
- Backend: Java for the Authentication, Gateway, and API Services components. Each microservice in the Microservices layer can use a different programming language, depending on the requirements of the service.
- Data Storage: SQL for relational databases or NoSQL for non-relational databases.
- Event Streaming: Apache Kafka, which supports multiple programming languages, including Java, Python, and Go.
- Email Service: Any language that supports email protocols, such as SMTP.

Data types transferred between components:

- User: Transfers user data such as login credentials, preferences, and requests.
- Browser and Native App: Transfers user interface data such as HTML, CSS, and JavaScript files.
- Authentication: Transfers authentication and authorization data between itself and the