Kuksa: A Cloud-Native Architecture for Enabling Continuous Delivery in the Automotive Domain

Ahmad Banijamali$^{1[0000-0002-6283-142X]}$, Pooyan Jamshidi$^{2[0000-0002-9342-0703]}$, Pasi Kuvaja$^{1[[0000-0002-1488-6928]}$, and Markku Oivo$^{1[0000-0002-1698-2323]}$

[1] M3S Research Unit, ITEE Faculty, University of Oulu, Finland
{firstname.lastname}@oulu.fi

[2] Computer Science and Engineering Department, University of South Carolina, USA
pjamshid@cse.sc.edu

Abstract. Connecting vehicles to cloud platforms has enabled innovative business scenarios while raising new quality concerns, such as reliability and scalability, which must be addressed by research. Cloud-native architectures based on microservices are a recent approach to enable continuous delivery and to improve service reliability and scalability. We propose an approach for restructuring cloud platform architectures in the automotive domain into a microservices architecture. To this end, we adopted and implemented microservices patterns from literature to design the cloud-native automotive architecture and conducted a laboratory experiment to evaluate the reliability and scalability of microservices in the context of a real-world project in the automotive domain called Eclipse Kuksa. Findings indicated that the proposed architecture could handle the continuous software delivery over-the-air by sending automatic control messages to a vehicular setting. Different patterns enabled us to make changes or interrupt services without extending the impact to others. The results of this study provide evidences that microservices are a potential design solution when dealing with service failures and high payload on cloud-based services in the automotive domain.

Keywords: Microservices · Cloud-native architecture · Cloud computing · Automotive.

1 Introduction

In recent years, there has been an increased focus from industry and academia to investigate cloud platform architectures that enable continuous software delivery (CD) in vehicles [10]. Many industries have started to look for CD solutions as they need to release quality software more frequently, better respond to automotive market changes, avoid vehicle recalls, improve productivity, and increase customer satisfaction [28]. For this purpose, vehicular software and information resources are being virtualised and designed as services in the cloud [17]. Cloud platforms in the automotive domain (ACPs) provide the possibilities to exchange data beyond vehicles [19], connect vehicles to other objects in the environment,

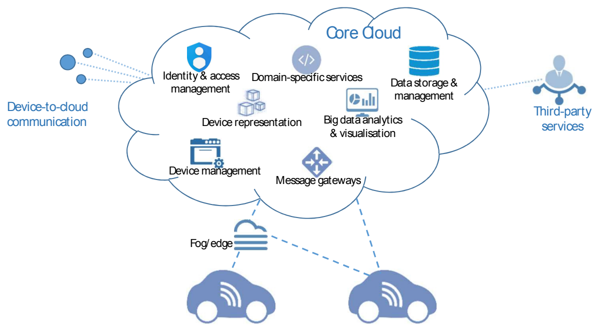

Core Cloud
Identity & access management
Domain-specific services
Data storage & management
Device representation
Big data analytics & visualisation
Device management
Message gateways
Device-to-cloud communication
Third-party services
Fog/edge

Fig. 1. Cloud platforms in the automotive domain

update automotive software using wireless communications systems (over-the-air) [33], and enable many more business services in the cloud (Figure 1).

Nevertheless, the migration of software delivery to ACPs has raised new research challenges. For example, vehicle-to-cloud (V2C) data transmission requires low latency and high reliability to satisfy the requirements of real-time systems [21]. Scalability is another challenge that demands the decomposition of functionalities and efficient data management [15]. Furthermore, the resiliency configuration explains runtime behaviour and faulty components [15], and security is a major requirement for protecting vehicles from malicious attacks [34].

In addition, as for the migration process towards distributed systems, such as cloud-native architectures, many architecture designs fail as long as their goal is to only replace the existing legacy architecture with a virtualised environment in the cloud [3]. The reasons may include but are not limited to a lack of solid business cases for cloud migration, neglecting adequate support teams, migrating at once to the cloud, and not considering applications' architecture refactoring [4, 5]. Consequently, the benefits from migration to the cloud platforms could be trivial, as the failure can happen anytime [3].

Despite the importance of CD and the mentioned quality challenges in ACPs, there has been insufficient focus from research that provides practical insights into designing software architectures that address those quality concerns [6]. Due to the impact of microservices on cloud-native architectures with respect to quality requirements, such as reliability, scalability, availability, and fault-tolerance [4, 5], microservices can be a potential solution for the existing challenges in ACPs. In relation to this, the ultimate objective of our paper is to investigate whether microservices *can enable over-the-air (OTA) continuous delivery in ACPs while improving reliability and scalability in this domain.* We have proposed a microservices architecture based on a real-world project in the automotive domain called Eclipse Kuksa and conducted a laboratory experiment to evaluate the architecture with respect to the mentioned quality attributes.

The results of this study can benefit industrial practitioners and academic researchers in the domains of automotive software engineering and cloud platform design. The study is aimed at researchers who would like to gain insight into the application of microservices in the domain of ACPs. From the practitioners' perspective, the findings provide experimental results for the reliability and scalability of microservices in a real-world industrial case in the automotive domain. The key contributions of the study are: (1) assessing the relative extent to which cloud-native architecture can enable continuous delivery in the automotive domain and (2) evaluating the role of microservices patterns in improving the reliability and scalability of services in this context.

2 Background

2.1 Microservices

Monolithic architectures are usually successful when the whole system is small and the number of functions is low [9]. Increasingly, the number of end users requires more deployment in the cloud [9], as every time that we apply a change to a small part of an application, we need to build and deploy the whole monolithic system again [3]. Furthermore, scalability means scaling the whole application rather than a part of the components that requires more resources [13]. As a consequence, many companies, such as Netflix, Amazon, and Atlassian, have migrated to more scalable and reliable architectures like microservices.

As for distributed systems, microservices are used to design fine-grained, modular services that have different life cycles but work together [23]. Each service deploys independently [2] using a potentially different deployment framework typically in the cloud [25], scales independently [31], is tested individually, and accomplishes responsibilities independently [31] while communicating through lightweight mechanisms, such as RESTful APIs [13]. The relevant architecture breaks down a system into services, each as a business capability [13].

Microservices promote a DevOps philosophy about separated small teams working together to meet the objectives of a large mission-critical system [5]. On the other hand, DevOps provides the framework for developing, deploying, and managing the microservices container ecosystem [11]. In this architecture, a microservice is developed and maintained by one small team while coordination among the teams is minimised [35]. It is noted that the largest size of the teams usually follows Amazon's notion of the "Two Pizza Team", meaning not a large group of people [13].

Despite all the advantages that microservices bring to the architecture designs, they have several challenges that should carefully be addressed. For example, replacing a monolithic architecture with a large number of inter-connected microservices can increase latency and other performance issues [5]. Having a system that is currently being used in production, it is necessary to make the migration incrementally [35] without data loss and interruption [5], during which we need adequate frameworks and experience in how to proceed [35]. Eventually, inconsistencies among microservices is another relevant challenge [13].

2.2 Software Architectures of Automoitve Cloud Platforms

Convergence of the internet of things and cloud computing has enabled innovative business use cases, ecosystems, and players in the automotive domain [17].

ACPs' application includes but is not limited to advanced vehicle connectivity, infotainment applications, voice and video data streaming, fleet management services, remote diagnostics and maintenance, and telematics services [14, 18].

Due to the increasing number of connected vehicles, the security, reliability, availability, robustness, and scalability of services are becoming new quality requirements in ACPs [16]. The extent of architectures in ACPs ranges from multilayered architectures [7] to service-oriented architectures (SOA) [22, 32]. Datta et al. [8] designed a framework for connected vehicles to offer consumer-centric services and a uniform mechanism for describing and collecting vehicular sensors' data. The designed architecture applies technologies such as road side units (RSUs) and machine-to-machine (M2M) gateways, including the fog computing platform [8]. The authors argued that using fog computing technologies can improve the fault tolerance, reliability, and scalability of the system [8]. Scalability and interoperability have been addressed in another study [26] in a modular architecture built upon DevOps practices to enable vehicle-to-everything (V2X) applications. The authors divided real-time applications for managing traffic into small modules to validate the functionality of the architecture [26].

A scalable and fault-tolerant data-processing design for real-time traffic-based routing was proposed by another study [27]. It argued that the designed architecture can serve a wide range of workloads and use cases with low-latency requirements [27]. Real-world scenarios of intelligent traffic system applications demonstrated the need for scalable big data analysis, service encapsulation, dynamic configuration, and optimisation strategies in this context [12]. Due to the technological variety in ACPs, architecture designs must assure stakeholders [5] that provisional services will meet the quality requirements at a specific level of cost and risk that is enforced by service level agreements (SLAs) [24].

3 Research Questions and Method

This section describes the study's objective, research questions, and research method.

3.1 Objective and Research Questions

The main objective of our study was to evaluate whether microservices can address CD in the context of ACPs and whether they can improve the reliability and scalability of services in this context. The research questions (RQs) for this study were as follows:

- RQ1: Can the microservices architecture design enable over-the-air continuous delivery from cloud platforms in the automotive domain?
- RQ2: How can the microservices architecture design improve the reliability and scalability of services in cloud platforms in the automotive domain?

3.2 Research Method

To design the target microservices architecture, we adopted a software architecture from a real-world project in the context of ACPs called Eclipse Kuksa (see

Section 4). It was important to initiate the migration process based on an existing project to review how the new architecture design could improve reliability and scalability in this domain. For the migration and refactoring process of the current architecture of Eclipse Kuksa, we applied microservices patterns from literature (e.g., [4]). Each refactoring represented a small and controlled change, so it was possible to identify how the quality attributes changed. The codes are available on GitHub[3].

Recent research [29] has explained that although it is critical to evaluate the requirements of a new software system to ensure system acceptance by users, real context evaluations are often complex. Before operating newly designed systems in real dynamic and complex environments, it is reasonable to assess them in laboratory setting experiments [29]. Thus, to evaluate the designed microservices architecture, we used laboratory experiments as the research method to answer the RQs of this study.

To date, there are several domain-specific services designed in Eclipse Kuksa. Among them, this study selected a service that is used for the purpose of motion control. Previous studies [4, 5, 20, 23, 30] have proposed frameworks and parameters in which architecture designers select microservices for migration, for example, according to their value to end users (e.g., improved user experience regarding the availability of services) or the project organisation (i.e., information exchange scalability and resiliency support) [5]. We selected the *motion control service* because of its value to end users and applicability in different scenarios. Furthermore, it demonstrates how end users can send control commands to vehicles from the cloud platform in Eclipse Kuksa using different user interfaces. It is a general service that can be part of many scenarios in this domain. The primary business driver for this service is to demonstrate OTA updates and messaging from the cloud to vehicles. This creates suitable grounds for future studies, e.g., on driver behaviour optimisation, natural language processing in vehicles, or OTA driver authentication.

Section 5 provides more details of our evaluation setting and the technology stacks used in our experiment.

4 Eclipse Kuksa

The Eclipse Kuksa[4] utilises open, vehicle-independent protocols, ensuring lifetime value for vehicles through upgradable applications. It addresses application systems, software solutions, and services for the mass differentiation of vehicles. The ecosystem of Eclipse Kuksa is comprised of three main platforms, including the (1) in-vehicle platform, (2) cloud platform, and (3) an app IDE. The Eclipse Kuksa is supported by a wide range of integrated open source software technologies and development environments, such as automotive grade Linux (AGL) and Eclipse Paho for the in-vehicle platform and Eclipse-Hono, Eclipse Hawkbit, Eclipse MosQuitto, Keycloak, and InfluxDB in the cloud back-end.

[3] https://github.com/ahmadbanijamali/Rover-Control-Experiment.git
[4] https://projects.eclipse.org/projects/iot.kuksa

4.1 The Existing Architecture of Eclipse Kuksa

Figure 2 shows the components and services in the Eclipse Kuksa architecture. The architecture only provides information about the necessary components and services that we needed in our experiment in the scope of this paper. It neglects other parts of Eclipse Kuksa ecosystem, such as device management and representation, authentication and authorisation, and the app store.

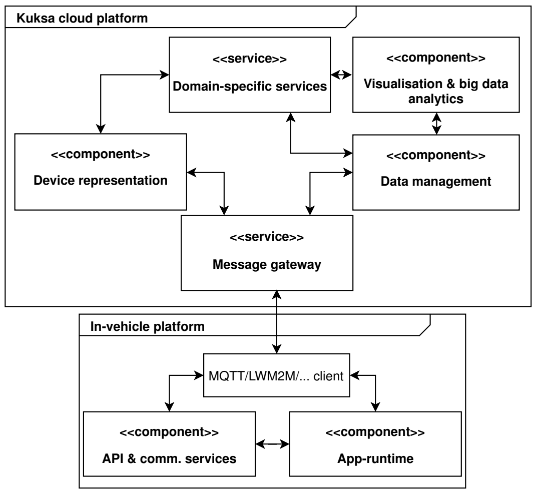


Fig. 2. Software architecture of Eclipse Kuksa

Message Gateway. The Eclipse Kuksa cloud platform (EKCP) sends and receives different types of messages from and to various sources, such as vehicles, devices, and third-party services. In general, messages include “telemetry messages” that depict data stemming from vehicles, devices, and sensors and “commands and controls messages” that are dedicated to the vehicles and device management components. The message gateway provides remote service interfaces for connecting vehicles and devices to the cloud back-end.

Data Storage and Management. An important part of the realisation of the EKCP is the storage and management of vehicles’ and IoT devices’ data in the appropriate database management system (DBMS). Although data management is a central aspect of every cloud platform architecture, due to the wide range of vehicles and devices connected to ACPs, it is necessary to establish a well-defined data management system that can handle complexities related to big data, consistency, performance, scalability, and security.

Visualisation and Big Data Analytics. The advances in the digitisation of the automotive domain have created a large amount of heterogeneous data coming from various sources. This has also yielded new requirements in terms of volume, variety, and velocity that are commonly called big data. The EKCP includes components and services to visualise and manage the big data in this domain.

Device Representation. To realise the distinct functionality of domain-specific services, a digital representation is important. Digital twin offers the possibility to access and alter the state of a vehicle’s functionality in a controlled manner.

Domain-specific Services. The domain-specific services are developed according to different use cases and business scenarios on top of the in-vehicle platform. They can handle different functions and tasks in vehicles and beyond them.

In-vehicle Platform. The communication protocols such as MQTT and LWM2M have enabled sending different messages from vehicles to the cloud and vice versa. The in-vehicle platform in Eclipse Kuksa includes an app runtime environment that is connected to an in-vehicle gateway, enabling software delivery and deployment in vehicles.

4.2 The Proposed Microservices Architecture for the Eclipse Kuksa Cloud Platform

Connected vehicles have high demands on the exchange of data between vehicles and a variety of services in the cloud. Due to the importance of the domain-specific services in ACPs, we selected a sample telemetry service that communicates with vehicles through sending command and control messages to vehicles (see Section 3.2). Figure 3 shows our proposal for the refactored architecture of EKCP that is described in greater detail in this section.

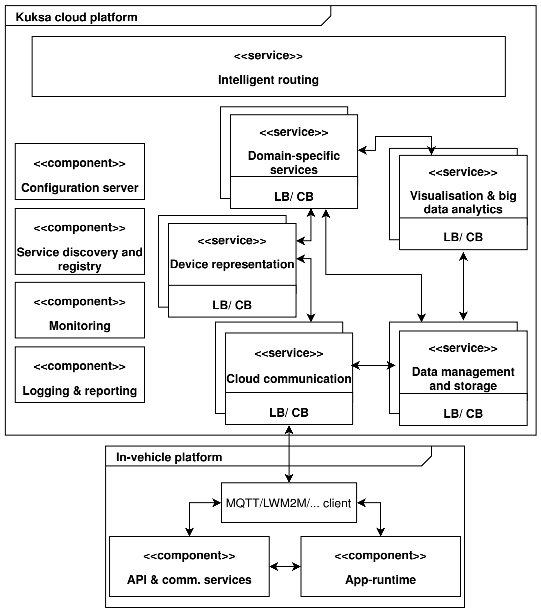


Fig. 3. The microservices architecture in Eclipse Kuksa

The migration to a microservices architecture in EKCP is a step-by-step process including new components and modules and modifying the existing components (Figure 4). We started the process by creating a better understanding of the existing architecture (Section 4.1) and introducing the CD pipeline.

Configuration Server. According to previous research [4], we required two individual and separate repositories as source code storage and software configurations storage. The configuration server is a central place to support the externalised configuration and changes without rebuilding or restarting the services. The Spring cloud configuration server is a potential technology that stores

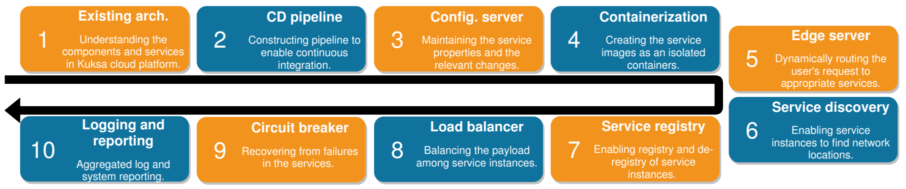

Existing arch.
1
Understanding the components and services in Kuksa cloud platform.
CD pipeline
2
Constructing pipeline to enable continuous integration.
Config. server
3
Maintaining the service properties and the relevant changes.
Containerization
4
Creating the service images as an isolated containers.
Edge server
5
Dynamically routing the user's request to appropriate services.
Service discovery
6
Enabling service instances to find network locations.
Service registry
7
Enabling registry and de-registry of service instances.
Load balancer
8
Balancing the payload among service instances.
Circuit breaker
9
Recovering from failures in the services.
Logging and reporting
10
Aggregated log and system reporting.

Fig. 4. The migration process to a microservices architecture

each microservice property based on the service-ID. The properties can be stored in the cloud or in other repositories, such as in GitHub.

Containerisation. The next step before establishing an intelligent routing (edge server) component was the containerisation of each service. This step is a part of the CD pipeline for building the container image for each service. The Docker and Docker Hub are the technology stacks used for this purpose.

Intelligent Routing (Edge Server). This is the layer right after the user interface (UI). Edge server dynamically routes requests to the appropriate microservices. Thus, it is possible here to monitor the service usage, as all requests pass this layer. As an instance of the technology stack, Netflix provides Zuul as the front door for all requests from devices and web sites to the back-end.

Service Discovery. Service instances dynamically find network locations of a service provider, which is critical for the service's auto-scaling and failures.

Service Registry. In addition to service discovery, service registry registers and de-registers service instances. It stores addresses of each service as the service initiates and removes the addresses once it does not receive the heartbeat or the service is terminated. Spring Eureka provides the technology stack for service discovery and registry.

Load Balancer. A purpose for migrating to a microservices architecture is to improve the scalability of each service based on the payload [5]. We used load balancers to distribute the payload among multiple instances of our services. Netflix Ribbon and Apache Zookeeper are examples of relevant technology stack.

Circuit Breaker. Once the number of consecutive failures in services crosses a specific threshold (open state), we call the circuit breaker to either invoke a response code or return the latest cached data from the service provider. Once the timeout expires, the circuit breaker allows a limited number of test requests to service providers, and, if they pass, it changes to a closed state. Hystrix and NGINX are relevant technology stacks here.

Logging and Reporting. To control what is happening in microservices, accessing the consolidated logs [5], implementing infrastructure-level metrics, and creating a holistic view of the system, we need to establish an efficient logging and reporting functionality. The system is used for a variety of purposes, such as monitoring the traffic and service usages, identifying the cause of errors, and finding performance bottlenecks. Due to the wide scope, different technologies (i.e., Hystrix, Grafana, Kibana, and fluentd) are used for specific purposes.

Continuous Delivery Pipeline. To establish a CD pipeline, we required continuous integration using following components. Jenkins was the solution used as the continuous integration server to build and deploy the applications.

Docker was the tool that we used for the containerisation of applications and to isolate them from each other. The Docker Hub, as the repository of Docker container images, pulls images from Docker's public registry instance. Figure 5 shows the CD pipeline in EKCP.

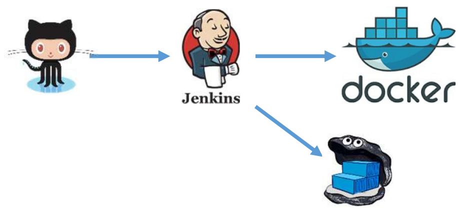


Fig. 5. The continuous delivery pipeline

5 Evaluation

5.1 Experimental Setting

To evaluate CD in the proposed architecture, we considered that our service sent automatically-generated updates as specific calls to forty vehicles in a specific region of the city. The calls were similar as they were demonstrating one released update. The software delivery cycle that the calls sent to the vehicles was one minute. In each call, we changed "next move direction" in the rover and the designed architecture should continue the message delivery without interruption. We ran the experiment for a duration of one hour to record how different microservices patterns behave in a CD environment in ACPs. We reviewed what percentages of calls is sent successfully to the rover and provide a statistics of successful and failed calls to show the CD performance in our design.

To review the scalability and reliability of the services in our designed architecture, we deployed three different scenarios. We aimed to measure metrics such as service downtime, recovery time, and load sharing behaviours. We registered four instances of Backserver service and one Client service (see Section 5) on a Spring Eureka server. The experimental scenarios were as follows.

1. During the first 10 minutes, all services were up and running. Half of the Backserver instances (two instances) shutdown automatically at 00:10 and re-started simultaneously at 00:15.
2. All service instances from the Backserver shut down automatically at 00:20 and started gradually (one by one) every five minutes until they all came up at 00:40.
3. All service instances from the Backserver went off at 00:45 and re-started simultaneously at 00:50.

Figure 6 presents the experimental setting in this study, including the different services, components, and technology stacks.

The cloud Back-end. We developed the Backserver and Client services using Spring Boot. All microservices were running on a computer with an Intel Core i7-6600U CPU @2.6 GHz and 20 GB installed RAM. Eclipse Hono version 7.0 was

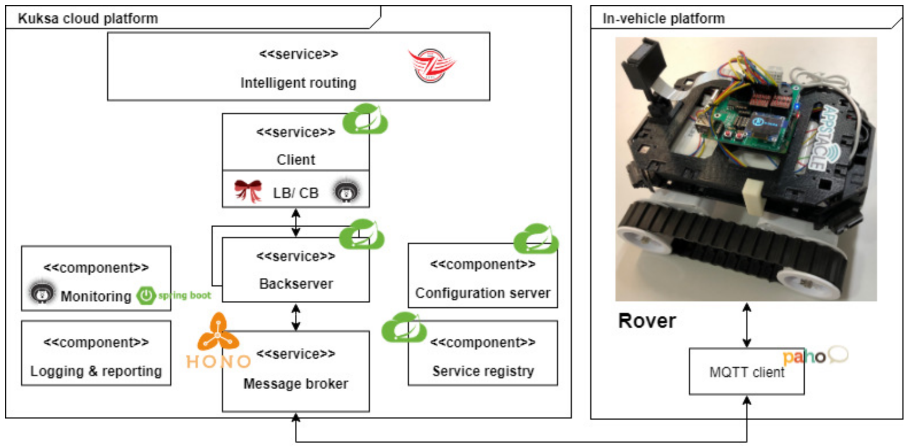


Fig. 6. The experimental setting

used as the message broker to connect the Backserver to the in-vehicle platform via MQTT using a 4G connection. The Hono instance was placed on an Azure Kubernetes service (AKS) cluster.

The Client service automatically triggered the delivery to the Backserver instances. Each message delivered to the rover contained the "rover id", "speed control", and "next move direction". The Backserver was responsible for sending the messages to the Hono instance and from there to the rover. The microservice patterns and technologies used are shown in Table 1.

Table 1. Microservice patterns and technologies used in this experiment

Pattern	Technology	Customised configuration
Intelligent routing	Netflix Zuul	serviceId: backserver, serviceId: Client
Load Balancing	Netflix Ribbon	Server list refresh interval: 2s
Circuit breaker	Hystrix	Sleep window: 5s, Request volume threshold: 20, Error threshold: 50%
Configuration server	Eureka	–
Service registry	Eureka	eureka.client.register-with-eureka=false eureka.client.fetch-registry=false
Monitoring	Hystrix dashboard	–

The in-vehicle Platform. To demonstrate the outcomes of the experiment, we used a rover, which is an open source mobile robot. The rover includes a Raspberry Pi 3 Model B (RPi3), a motor driver layer (Arduino), and a RoverSense layer designed for in-vehicle communication demonstrations. A customised software (called roverapp[5]) was designed that runs on a Linux-based embedded single board computer (i.e., RPi3). The roverapp includes an API to handle various functions in the rover, such as motion control.

In addition to the commands sent to the rover, the RoverSense layer sends telemetry data from different sensors, such as infrared proximity sensors, ultrasonic sensors, temperature and humidity sensors, and an accelerometer to the

[5] https://app4mc-rover.github.io/rover-app/

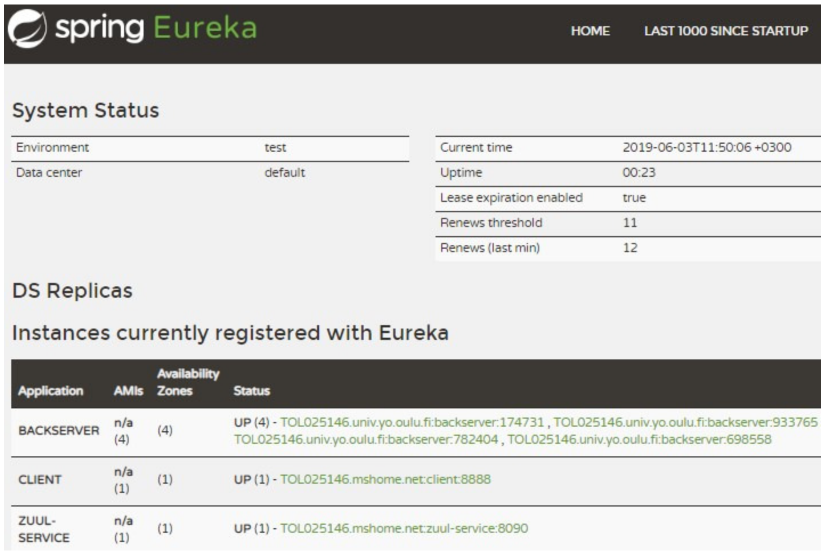
cloud. The roverapp creates the possibility of real-time video streaming to the cloud platforms, such as Azure or AWS. It also allows the marker detection used in platooning or autonomous driving scenarios.

The rover's features' applications and tooling use AGL as the operating system, which runs on RPi3. The in-vehicle Kuksa layers, including a middleware layer (containing Kuksa APIs and Eclipse Paho) and an application layer (containing a runtime and sandbox environment), run on top of AGL. These two layers enable functions such as communication to the cloud via MQTT and third party applications' implementation.

5.2 Results

This section is structured to address the research questions and includes the aggregated results of our experiment.

RQ1. Can the microservices architecture design enable over-the-air continuous delivery from cloud platforms in the automotive domain? CD helps teams to produce applications in short cycles and ensures that the software can be reliably released at any time. Figure 7 shows the service registry dashboard in a Spring Eureka server. It shows that four instances of the Backserver and one Client service were up and running at the time of the experiment.



System Status

Environment	test
Data center	default

Current time	2019-06-03T11:50:06 +0300
Uptime	00:23
Lease expiration enabled	true
Renews threshold	11
Renews (last min)	12

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
BACKSERVER	n/a (4)	(4)	UP (4) - TOL025146.univ.yo.oulu.fi:backserver:174731 , TOL025146.univ.yo.oulu.fi:backserver:933765 TOL025146.univ.yo.oulu.fi:backserver:782404 , TOL025146.univ.yo.oulu.fi:backserver:698558
CLIENT	n/a (1)	(1)	UP (1) - TOL025146.mshome.net:client:8888
ZUUL-SERVICE	n/a (1)	(1)	UP (1) - TOL025146.mshome.net:zuul-service:8090

Fig. 7. Registered services for the designed architecture

Table 2 shows the aggregated results of the duration that each service instance of the Backserver was up during our designed scenarios. In addition, it shows statistical information on the service resiliency in our setting.

During our experiment, we could make changes (shutdown, re-start, and update the code) in a service without affecting other services. According to our experimental scenarios, Backserver instances set up and down multiple times, even though it did not impact other available services. It was easy to make

Table 2. Experiment results using the designed microservices architecture

		No. of running instances in three scenarios	**Time**
		zero (shutdown all instances)	10 min.
Total duration of experiment	60 min.	one	5 min.
		two	10 min.
Cycle time	1 min.	three	5 min.
		four	30 min.
Circuit breaker status	**No.**	**Circuit breaker status**	**No.**
Success (execution completed with no errors)	22	Failure (execution threw an Exception)	24
Timeout (execution started, but did not complete in the allowed time)	4	Short-Circuited	0
Quickest time to call Backserver when came up		1 min.	

changes on a service, e.g., updating the listening port or rover direction, without interruption to other services.

Our findings indicated that although we had a number of failed calls and timeout errors due to the following reason, the circuit breaker could prevent cascading failures to other services. The Client service talked with the service registry to receive the IP addresses of available Backserver instances and used its load balancer to choose one of them. The Client service could not know directly that a Backserver instance was no longer available. This is the job of the service registry to continuously discover which Backserver instances are dead or alive via heartbeat mechanisms. During our experiment, the Backserver instances shutdown several times while the Client service could not get the list of the remaining instances from the service registry in real-time. In this approach, the service discovery logic tightly coupled with clients, in which it could improve through other approaches, such as server-side service discovery.

Summary. The designed architecture preserved continuous software delivery by automatic registering and de-registering service instances and continuing OTA software delivery after each change.

RQ2. How can the microservices architecture design improve the reliability and scalability of services in cloud platforms in the automotive domain? Table 3 shows a summary of the results of the total calls on each Backserver instance. The Client service sent more calls on the Backserver instances that were up for a longer time in our scenarios. In total, we had 990 successful calls distributed among four Backserver instances to control the rover speed and movement direction.

Table 3. The number of calls on the Backserver instances

		Service #1	455
Total calls sent to rover	990	Service #2	276
		Service #3	153
		Service #4	106

Figure 8 presents how the load balancing mechanism distributed the load among the different instances. In addition, it shows the circuit breaker behaviour regarding different errors to improve the reliability of the system.

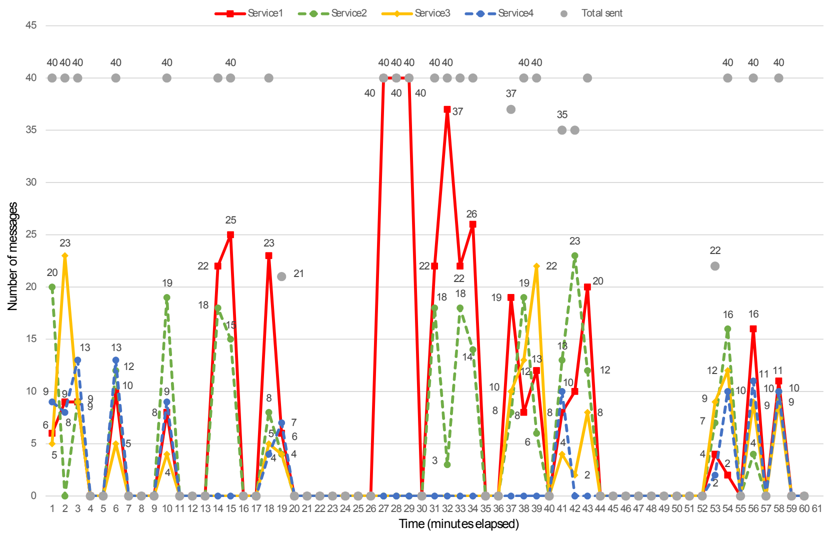


Fig. 8. The number of calls on each service in three scenarios

The client-side strategy load balancing automatically distributed concurrent calls to the available Backserver instances. The Netflix Ribbon load balancer continuously rotated a list of Backserver instances that were attached to it (the Round Robin method). In addition, to manage failures that happened in a service (e.g., timeout), Hystrix prevented cascading failures to other services, which improved the fault tolerance of our system. Broken service instances automatically recovered and registered themselves into the Eureka service registry, which made the designed microservices recoverable.

Summary. Although failures often happen in services, load balancing mechanisms were able to skip unhealthy instances.

6 Discussions

The objective of this research was to review whether the recent architectural design styles, such as microservices, could address CD and DevOps in the automotive domain. In an experimental setting, we evaluated how quality attributes such as the scalability and reliability of services could be improved by microservices patterns.

RQ1. Can the microservices architecture design enable over-the-air continuous delivery from cloud platforms in the automotive domain? A previous study [6] noted that to maintain continuous software delivery, it is necessary to address architectural challenges, such as the deployability and modifiability. Our findings showed that the proposed architecture could improve the deployability of the system as there was no need to resolve the conflicts between changes afterwards. Furthermore, we could deploy changes in different services independently and quickly without any interruption in other services.

We noticed that microservices created the possibility to make the changes localised to one service while other services were not affected. We had lightweight services that made any update in the codes easier. In safety-critical systems, such as ACPs, it is vital that changes in a service or technology do not interrupt other running services. Our findings showed that microservices could improve the modifiability of the architecture. Although the designed architecture could enable the CD in this domain by sending OTA messgaes to the rover, there were several failed and timeout calls that should be optimised with respect to different service level agreements.

RQ2. How can the microservices architecture design improve the reliability and scalability of services in cloud platforms in the automotive domain? Scalability is the property of a system that handles a growing amount of requests by adding resources to the system. The Backserver instances allowed us to support a good number of concurrent calls coming from the Client to the rover. The Backserver was also stateless, which did not retain consumer states. It enabled us to have autoscaling of the services when the load required. The load balancing mechanism in our system could also distribute the load automatically among available service instances.

Our findings in this study showed how the fault-tolerant mechanisms, such as the circuit breaker, could handle the resiliency and reliability in our proposed architecture. We defined different thresholds such as the error threshold percentage and request volume threshold to force the circuit breaker to open and prevent slow or failed calls from interrupting other services in our architecture, which improved reliability of the architecture.

6.1 Threats to validity

Construct validity, in our research, is concerned with using the right measures in our experiment. To assess the reliability and scalability, we used the common metrics that are widely applied in the literature (see [5, 23]). Internal validity concerns the relationship between the constructs and the proposed explanation. Our implementation was run in three scenarios in a laboratory experimental setting with specific and defined objectives. Although we established a controlled environment, aspects related to the performance of Azure cloud platform or 4G network connection could not be customised or controlled. In addition, the implementation and results were discussed and reviewed among the authors of this study. In our experiment, we selected the technology stacks that are commonly used by companies and the performance analysis of those technologies are out of scope of this research.

External validity is related to the generalisability of the study. A previous study [1] noted that it is not essential to satisfy all requirements by a given benchmark candidate to be considered useful for empirical research. We applied microservices patterns from scientific literature, established a controlled experiment with three defined scenarios, and used a real-world project to evaluate the behaviour of one single microservice in the designed architecture. Future studies can replicate the experiment with multiple services in real continuous software delivery environments in the automotive domain to evaluate generalisability of the results. Reliability concerns the repeatability of the research procedure and conclusions. We explained in detail the experimental setting and all publicly available materials, which can be applied by future studies.

7 Conclusion

Automotive cloud platforms have received increasing attention from research and industrial communities. To increase the reliability and scalability in ACPs and enable continuous software delivery in the automotive domain, we proposed a microservices architecture for a real-world project called Eclipse Kuksa and ran an experiment to evaluate the designed architecture.

Our findings showed that the proposed architecture could handle CD through improving the deployability, modifiability, and availability of the architecture. Our designed architecture could address quality issues, such as payload distribution among different instances and the resiliency of services. The research findings showed that microservices are an interesting design alternative to address quality concerns of future cloud platforms in the automotive domain.

References

1. Aderaldo C. M., Mendonça N.C., Pahl C., Jamshidi P.: Benchmark requirements for microservices architecture research. In: 1st Int. Workshop on Establishing the Community-Wide Infrastructure for Arch.-Based Soft Eng., pp. 8-13. IEEE (2017).
2. Balalaie, A., Heydarnoori, A., Jamshidi, P.: Microservices architecture enables devops: Migration to a cloud-native architecture. IEEE Software **33**, 42-52 (2016).
3. Balalaie, A., Heydarnoori, A., Jamshidi, P.: Migrating to cloud-native architectures using microservices: An experience report. In: Adv. in Service-Oriented and Cloud Comp., pp. 201-215. Springer, Switzerland (2015).
4. Balalaie, A., Heydarnoori, A., Jamshidi, P., Tamburri, D.A., Lynn, T.: Microservices migration patterns. Software: Practice and Experience. J. Software: Prac. and Expe. **48**, pp. 2019-2042 (2018).
5. Bass, L., Weber, I., Zhu, L.: DevOps: A software architect's perspective. Addison-Wesley Professional (2015).
6. Chen, L.: Microservices: Architecting for continuous delivery and DevOps. In: Int. Conf. on Software Arch. (ICSA), pp. 39-397. IEEE (2018).
7. Contreras-Castillo, J., Zeadally, S., Guerrero-Ibanez, J.A.: Internet of vehicles: Architecture, protocols, and security. Internet of Things J. **5**, pp. 3701-3709 (2018).
8. Datta, S. K., Gyrard, A., Bonnet, C., Boudaoud, K.: oneM2M architecture based user centric IoT application development. In: 3rd Int. Conf. on Future Internet of Things and Cloud, pp. 100-107. IEEE (2015).
9. Dragoni, N., Dustdar, S., Larsen, S.T., Mazzara, M.: Microservices: Migration of a mission critical system. arXiv preprint arXiv:1704.04173 (2017).
10. Ebert, C., Favaro, J.: Automotive software. IEEE Software **34**, pp. 33-39 (2017).
11. Ebert, C., Gallardo, G., Hernantes, J., Serrano, N.: Devops. IEEE Software **33**, 94-100 (2016).
12. Fiosina, J., Fiosins, M., Müller, J.P.: Big data processing and mining for next generation intelligent transportation systems. J. Teknologi **63**, pp. 21-38 (2013).
13. Fowler, M., Lewis, J.: Microservices. https://martinfowler.com/articles/microservices.html.

14. Google Cloud: Designing a Connected Vehicle Platform on Cloud IoT Core - 2019-05-07. https://cloud.google.com/solutions/designing-connected-vehicle-platform.
15. Häberle, T., Charissis, L., Fehling, C., Nahm, J., Leymann, F.: The connected car in the cloud: A platform for prototyping telematics services. IEEE Software **32**, 11-17 (2015).
16. Haghighatkhah A., Banijamali A., Pakanen O., Oivo M., Kuvaja P.: Automotive software engineering: A systematic mapping study. J. Syst. Soft. 128, 25-55 (2017).
17. He, W., Yan, G., Da, Xu L.: Developing vehicular data cloud services in the IoT environment. IEEE Trans. on Ind. Info. **10**, pp. 1587-1595 (2014).
18. Jain, P.: Automotive Cloud Technology to Drive Industrys New Business Models - 2019-05-07. http://shiftmobility.com/2017/06/automotive-cloud-technology-drive-automotive-industrys-new-business-models.
19. Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, A., Zaharia, M.: A view of cloud computing. Commun of the ACM **53** (2010).
20. Levcovitz, A., Terra, R., Valente, M.T.: Towards a technique for extracting microservices from monolithic enterprise systems. arXiv:1605.03175, (2016).
21. Lu, N., Cheng, N., Zhang, N., Shen, X., Mark, J.W.: Connected vehicles: Solutions and challenges. Internet of Things J. **1**, pp. 289-299. IEEE (2014).
22. Mietzner, R., Leymann, F., Unger, T.: Horizontal and vertical combination of multi-tenancy patterns in service-oriented applications. Enterprise Info. Syst. **5**, pp. 59-77 (2011).
23. Newman, S.: Building microservices: designing fine-grained systems. O'Reilly Media, Inc. (2015).
24. O'Brien, L., Merson, P., Bass, L.: Quality attributes for service-oriented architectures. In: Proc. of the Int. Workshop on Syst. Dev. in SOA Env., pp. 3 (2007).
25. Pahl, C., Jamshidi, P.: Microservices: A systematic mapping study. In: Proc. of the 6th Int. Conf. on Cloud Computing and Services Science, pp. 137-146 (2016).
26. Rufino, J., Alam, M., Ferreira, J.: Monitoring V2X applications using DevOps and docker. In: Int. Smart Cities Conf., pp. 1-5 (2017).
27. Serrano, D., Baldassarre, T., Stroulia, E.: Real-time traffic-based routing, based on open data and open-source software. In: 3rd World Forum on Internet of Things, pp. 661-665 (2016).
28. Shavit, M., Gryc, A., Miucic, R.: Firmware update over the air (FOTA) for automotive industry. SAE Tech. (2007).
29. Stol, K., Fitzgerald, B.: The ABC of software engineering research. ACM Trans. on Software Eng. and Meth. **27**, 11 (2018).
30. Taibi, D., Lenarduzzi, V., Pahl, C.: Architectural patterns for microservices: A systematic mapping study. In: Proc. of the 8th Int. Conf. on Cloud Computing and Services Science, pp. 221-232 (2018).
31. Thönes, J.: Microservices. IEEE Software **32**, pp. 116-116 (2015).
32. Yang, M., Mahmood, M., Zhou, X., Shafaq, S., Zahid, L.: Design and implementation of cloud platform for intelligent logistics in the trend of intellectualization. China Commu. **14**, pp. 180-191 (2017).
33. Zeller, M., Prehofer, C., Krefft, D., Weiss, G.: Towards runtime adaptation in AUTOSAR. In: 5th Workshop on Adaptive and Reconfigurable Embedded Syst. **10**, pp. 17-20 (2013).
34. Zhang, T., Antunes, H., Aggarwal, S.: Defending connected vehicles against malware: Challenges and a solution framework. Internet of Things J. **1**, pp. 10-21.(2014).
35. Zhu, L., Bass, L., Champlin-Scharff, G.: DevOps and its practices. IEEE Software **33**, 32-34 (2016).