Kuksa[*]: Self-Adaptive Microservices in Automotive Systems

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

[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. In pervasive dynamic environments, vehicles connect to other objects to send operational data and receive updates so that vehicular applications can provide services to users on demand. Automotive systems should be self-adaptive, thereby they can make real-time decisions based on changing operating conditions. Emerging modern solutions, such as microservices could improve self-adaptation capabilities and ensure higher levels of quality performance in many domains. We employed a real-world automotive platform called Eclipse Kuksa to propose a framework based on microservices architecture to enhance the self-adaptation capabilities of automotive systems for runtime data analysis. To evaluate the designed solution, we conducted an experiment in an automotive laboratory setting where our solution was implemented as a microservice-based adaptation engine and integrated with other Eclipse Kuksa components. The results of our study indicate the importance of design trade-offs for quality requirements' satisfaction levels of each microservices and the whole system for the optimal performance of an adaptive system at runtime.

Keywords: Microservices · Self-adaptive systems · Automotive · Cloud.

1 Introduction

The importance of software in the automotive domain has increased because most of the innovations in new cars originate from software, large computing power, and modern sensors. A vehicle is now a part of a network to collaborate with other vehicles, cloud, edge, and the surrounding infrastructure to deliver value-added services [19]. Cloud and edge platforms have enabled new services based on access to car data and functions from outside the car. Because of the large volume of these data, service management in the domain of connected vehicles is appealing for reliable and efficient solutions [23], allowing for scalable and autonomic processing.

In the highly dynamic environment that automotive systems operate, services should be able to automatically and quickly adapt to changes at runtime to preserve a level of quality under the complex operating conditions, such as scaling

workloads and faults [17]. To turn vehicles into self-adaptive systems, we have to combine self-managing capabilities with architectural design patterns that enable self-adaptive functioning under complex working conditions [2].

Microservice architectures are a potential solution for designing and deploying self-adaptive business capabilities [18]. Although there have been many attempts in the development of self-adaptive systems, a small number of microservices solutions, such as Kubernetes, load balancers, and circuit breakers could find their ways to the industry to facilitate autonomic management of microservice systems [17].

There is also an increasing interest in microservices in the automotive domain; however, the research on the microservice architectures that enable self-adaptive automotive systems is still in the early stage. A previous survey [4] highlighted this research gap for architecture-based self-adaptation in automotive systems. To address this need, we empirically investigated how microservices patterns, such as load balancing, resiliency mechanisms, and monitoring systems can improve the self-adaptive quality performance under the dynamic conditions of the automotive domain [3]. In this paper, we extend the study on the microservice architectures in the automotive domain *to suggest a framework called Kuksa* that aims at improving the self-adaptation capabilities of automotive systems by proposing a microservice-based autonomic controller.* We employed the Eclipse Kuksa architecture[3] in the automotive domain to design Kuksa* and evaluated it in an automotive experimental laboratory setting.

The results of this study are valuable for the system architects and academic researchers in the automotive domain. Industrial practitioners can benefit from experimental results on the performance and configurations of self-adaptive microservice systems in a real-world industrial case in the automotive domain. Academics can also gain insights into quality research gaps, service optimisation, and quality trade-off challenges in self-adaptive systems. The key contributions of the study are as follows: (1) the Kuksa* framework to enhance the self-adaptation capabilities of automotive systems by using microservice architectures and (2) the empirical evaluation of the framework in an experimental setting and showing the results of design and runtime trade-offs.

2   Background and Related Work

This section provides a brief overview of previous studies and related work.

2.1   Background

Microservices: Large monolithic systems hardly can scale when different modules have conflicting resource requirements, life cycles, and deployment frameworks [3]. Thus, there has been an increasing interest to migrate systems to more scalable and reliable architectures, such as microservices [5]. Breaking down applications into a set of smaller, interconnected services trades external complexity for microservice simplicity [18].

---

[3] https://www.eclipse.org/kuksa/

Microservices are a design alternative that is used to foster further research on self-adaptive systems [17]. They isolate business functions into small services to optimise the autonomy, modifiability, and replaceability of the services. Microservice architectures pertain to challenges that should be carefully addressed. For example, they increase the complexity due to only the fact that they are distributed. Therefore, it is necessary to have inter-process and autonomic communications among different microservices [5]. Although microservices improve the communication with the back-end databases, they apply partitioned database architecture, which means it is often necessary to update multiple databases that are owned by different services [21] and to have an eventual consistency-based approach. Also, a large number of microservices in an application may increase latency and performance issues [5]. Therefore, microservices should work as self-adaptive systems that dynamically learn and adapt their behaviours to preserve specific quality levels under dynamic operating conditions [17].

Cloud platforms in the automotive domain: Cloud platforms in the automotive domain are designed for high scalability and flexibility and offer the continuous delivery of new software applications, such as telematics, infotainment, fleet management, and remote diagnostics and maintenance [13]. The creation of scalable, extensible, and hardware-agnostic platforms has facilitated the deployment of collaborative vehicle applications [25]. With the vehicle movement, it is necessary not only to have access to an infinitely scalable architecture supporting data analytics and application development in the cloud [24] but also to adapt to changes in the operating conditions [26]. For instance, when there is an interruption in the communication network, services in vehicles or the cloud should adopt alternative offline scenarios [26].

2.2 Related work

Architecture-based adaptation is addressed by (1) the adaptation capability of a system to change with minimal human intervention and (2) the control loop mechanisms in the system that separate core services from adaptation services [27]. Policy-based architecture adaptation is an approach to map a functional situation to relevant action, strategy, or reconfiguration [12]. We used this approach to decouple a system adaptation logic with knowledge about how to react when an adaptation is required. Architecture-based adaptation allows architectural reconfiguration based on predefined rules [27].

Architectural configurations impact the performance of self-adaptive software systems [20]. There exist many techniques, such as Bayesian optimisation [14] and multi-objective optimisation [10] that have been used for moving towards the optimal performance of self-adaptive systems. Furthermore, many approaches, such as planning and constraint-solving algorithms have been used to find a new target configuration that satisfies the given constraints of a system [29].

Control loops are commonly used in policy-based architecture adaptation approaches to collect data about external environments and make autonomic decisions [15]. In this regard, inducing microservices with the primitives of self-adaptivity is a strong candidate for addressing autonomic trade-offs [11]. Pereira

et al. [22] presented an adaptive system that designed the control loop components in the form of microservices that are easily deployed in a container-based system, such as Kubernetes. Another study [1] proposed an architecture-based self-adaptation service for cloud-native applications based on the Rainbow self-adaptation framework with support for Docker containers and Kubernetes. Current approaches mainly rely on the quantitative verification of system properties, including techniques to produce formal guarantees about the quantitative aspects of systems, such as performance [8].

There has been considerable work in the area of managing software systems by using service compositions and adaptations [7]. However, they hardly address the problem from the dynamic adaptation perspective by balancing microservices performance at runtime, specifically in the automotive domain. Our study has tackled this issue and investigated how self-adaptive microservices can optimise automotive systems' performance at runtime.

3   Research Method

This section describes the current study's objective, research questions, research methods, and the Eclipse Kuksa framework.

3.1   Objective and research questions

The study objective is to design a microservice architecture for enhancing the self-adaptation capabilities of automotive systems. To achieve this objective, we devised the following research questions (RQs):

> – RQ1: How can microservices be applied in the design of self-adaptive systems in the automotive domain?
> – RQ2: What are the quality trade-offs at runtime when using self-adaptive microservices in the automotive domain?

3.2   Research methods

To propose the Kuksa* framework, we applied the concept of control loops (MAPE-K) [15] to render a design solution that improves over the system life cycle through accumulating knowledge. In addition, we opted for an experimental method and a laboratory setting (see section 5 for the detailed description of the experimental scenario and setting) that allowed us to investigate performance trade-offs from the Kuksa* framework and the system under evaluation regarding self-adaptation capabilities. We investigated how self-adaptation microservices can adapt the system to changes and how they can register changes into the whole system. The laboratory setting helped us to demonstrate and control the impact of changes on the performance of the designed architecture because real context evaluations are often safety-critical, complex, and costly [28].

We designed multiple microservices to enable the self-adaptation in our designed setting and to evaluate the system performance based on an *"adaptive*

video streaming scenario". We selected this scenario because of its importance in other automotive scenarios, such as the driver assistance, collision-avoidance, safety-critical scenarios, vehicle-to-vehicle communication, autonomous driving, and comfort systems and its significance in security scenarios in the automotive domain. The video streaming presents an example scenario to show the performance of self-adaptive microservices under highly dynamic conditions of automotive systems, although Kuksa[*] has high flexibility to be adopted and customised for other scenarios in the automotive domain.

We used the results of previous studies ([8, 16]) to create a metric for measuring the performance of the self-adaptive system under evaluation and comparing it with the static system architecture. The performance of software systems denotes its capabilities in its execution [8], in which two important measures include the solution quality and the time taken to achieve the solution [16]. Hence, the total performance in our designed system composed of time performance and solution quality performance [8], as follows:

$$p(sys) = wt.tp(sys) + wq.qp(sys)$$

where $wt + wq = 1$. The parameters of wt and wq, respectively, indicated the weight factors (importance) for time performance $tp(sys)$ and quality performance $qp(sys)$. Normalisation of $tp(sys)$ and $qp(sys)$ in a range of 0 to 1 allowed us to sum them up in our metric.

Time and quality performances were measured based on the average performance from multiple runs. The $tp(sys)$ used the average time required for the reconfiguration of the video streaming microservice which translates into the time involved in finding a new adaptation strategy configuration in a separate run r. The $tp(sys)$ was calculated based on the ratio of the time needed for reconfiguration in a run r, as follows:

$$tp(r) = 1 - \frac{\sum Reconfiguration\ time(r)}{Duration(r)}$$

The quality performance was calculated based on the achieved video stream quality, using two factors of *frame rate* and *frame quality*, that the system was able to apply within a run r, as follows:

$$qp(r) = \frac{\sum Quality(r)}{Quality\ max}$$

The codomain of qp(r) was [0, 1] and a value close to 1 indicated a better-achieved quality as 1 would imply that the maximum quality performance was reached.

3.3   Eclipse Kuksa

Eclipse Kuksa is a project that provides an open and secure cloud platform to connect a wide range of vehicles to the cloud via in-car and Internet connections. It aims at the mass differentiation of vehicles through application systems,
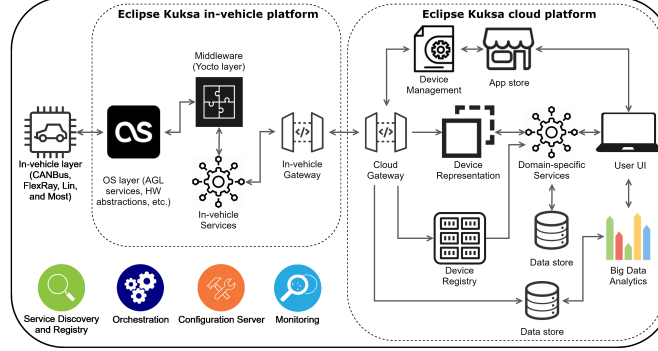
Fig. 1. Software architecture of Eclipse Kuksa.

software solutions, and services. The project comprises three open-source software platforms for the (1) in-vehicle, (2) cloud, and (3) application integrated development environment, shown in Figure 1 (adopted from [3]).

Gateways (in-vehicle/cloud) enable data communication from vehicles (i.e., car CAN-Bus) or control commands delivery to vehicles. The gateways provide remote service interfaces for connecting vehicles and devices to the cloud backend using protocols, such as MQTT. The *middleware layer* in the in-vehicle platform includes APIs to abstract the vehicle's electrical/electronic architecture and communication libraries to enable communication services and manage the network access. *OS layer* includes, for example, AGL core services, boot loader, hardware abstractions, and platform update manager.

Device representation provides a digital representation of devices to realise the distinct functionality of domain-specific services. *Device registry* grants access to a distinct functionality for only eligible devices and users. *Domain-specific services* are developed according to various use cases and scenarios in the cloud or vehicles to handle different functions. *Data storage* is necessary for reliable data management systems that can handle the data complexities related to the size, consistency, performance, scalability, and security of different microservices. *Device management* is responsible for tasks, such as the provisioning, configuration, monitoring, and diagnostics of connected devices. *Kuksa app store* is a digital repository for software applications developed by vehicle manufacturers and other third-party providers. *Visualisation and data analytics* efficiently identify, collect, clean, analyse, and visualise that data to enable new services.

The Kuksa architecture [3] relies on more microservices, including *Service discovery and registry*, which automatically registers and deregisters service instances. It provides a source to find out which of the service instances are currently available. *Configuration server* stores service configurations to isolate the configuration properties from codes and to enable autonomous service rebuilding or restarting. *Monitoring* presents the state of microservices-based systems using consolidated logs, reports, and infrastructure-level metrics, for example, for monitoring service usage and finding performance bottlenecks. *Orchestration* collectively provides the mechanisms that deploy, maintain, and scale services and applications based on configuration settings to meet different workloads.

4   Kuksa*: A Self-Adaptive Microservice-based Framework for Automotive Systems

Automotive systems, such as driver assistance or safety-critical systems should allow for modifications of services and systems by taking appropriate actions at runtime based on changes in the vehicle operating condition [7]. The actions can include, for example, replacing one microservice instance with another (e.g., upgrading power train and safety electronic services based on the new trailer attached to the vehicle), changing the number of microservice replicas (e.g., increasing load of vehicular data communication with surrounding objects in a crowded area), and dynamically changing the quality requirements of microservices (e.g., adding new safety requirements and constraints in bad weather conditions) [17]. The functions are automated at runtime in a control loop that collects the details from automotive systems and sensors and acts accordingly.

The autonomic controller, shown in Figure 2, applies the microservice architecture to improve the in-vehicle and cloud systems with self-adaptation capabilities. It continuously monitors and analyses vehicle's working conditions and executes appropriate actions to resolve issues or improve system quality performance in vehicles with minimal human intervention. Kuksa[*] was constructed on top of the Eclipse Kuksa platforms, in which container-based services are deployed in Kubernetes clusters to provide actuation and operation of vehicular systems (see section 3.3). The service discovery and registry continuously provides a list of available services and registers and deregisters service instances.

The autonomic controller is responsible for the execution of the MAPE-K control loop by using *Monitoring*, *Analysis*, *Planning*, and *Execution* services. These are container images that are registered to the device registry mechanism and deployed in orchestration mechanisms, such as Kubernetes to work as the control loop microservices. The monitoring services provide solutions for collecting, filtering, and reporting runtime executions and performance data from the managed resources in vehicles, such as domain-specific services, sensors, CAN-Bus, Lin, and network nodes. To define variables to be monitored at runtime, it is necessary to collect quality requirements for the healthy functioning of an automotive system (e.g., prioritised emergency messaging of a vehicle accident) and translate them to runtime variables (e.g., network speed $\geq$1MB/s and processing time <1s). The variables are associated with an automotive scenario (e.g., the emergency braking distance per speed) or the system's behaviour in that scenario (e.g., system reconfiguration time or response time). The monitoring results are affected by sensing issues, such as latency, inaccuracy, or reliability [9].

The values of the monitored variables are sent for runtime analysis services to model the system's operating conditions and provide a perception of the current scenario surrounding the managed system of the vehicle. Many techniques and solutions (i.e., machine learning or fuzzy logic) can be used here to learn about the performing environment, such as traffic states in different urban areas and provide more efficient and reliable models that facilitate the prediction of future conditions (e.g., analysing data loads and driver's profiles to optimise services' data consumption). Using the microservice architecture assures that the analysis
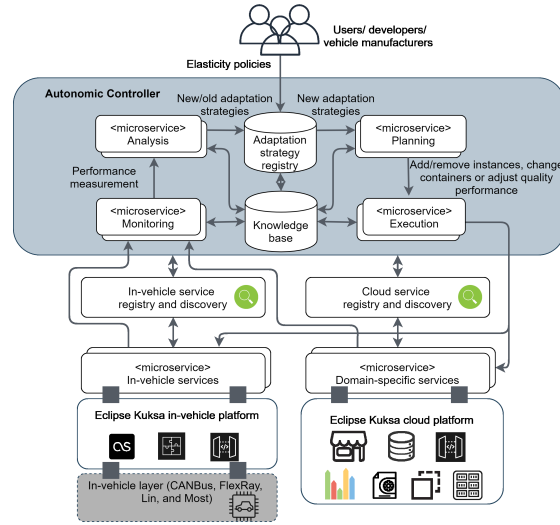
Fig. 2. The Kuksa* framework.

services are highly available to model operating conditions at complex runtime situations of vehicles, although the distribution of microservice may require extra monitoring data to decrease the possibility of wrong perceptions of the current environment scenario and the likelihood of triggering faulty adaptation decisions. For example, the adaptive cruise control can receive more data from the vehicle's radar to confirm that it provides reliable results (e.g. no distortion is present). The results of the analysis services are presented as the adaptation strategies, including creating new strategies or keeping existing ones based on the comparison of the optimum and actual working conditions. Adaptation strategies are collected and stored in databases called the adaptation strategy registry.

The adaptation planning services are responsible for making decisions about triggering a new adaptation strategy and sending a set of actions needed to reach the optimum state. The decisions can include, for example, adding or removing microservice replicas, triggering a new service while stopping a running one (e.g., disabling the cruise control service while alarming the user), replacing a container image with a newly downloaded service, optimising quality attributes in a list of multiple running automotive services, updating the service deployment processes (e.g., deploy new applications only when there is a high-speed network connection or the vehicle is parked), or a combination of these actions. The independency among different microservices makes them a secure alternative for runtime planning of new updates and reconfigurations in vehicular systems. The solution space could include inputs from, for example, drivers, application providers, or vehicle manufacturers or be created at runtime by using, for example, artificial intelligence models. The analysis and planning services may confront challenges concerning latency or model uncertainty [9].

The adaptation execution services control the implementation of a strategy and are able to deploy runtime updates. Comparing to monolithic architectures, microservices allow simplifying the solution space as we need to test and execute

a smaller set of services. The strategy could be directly updated in services at runtime, e.g., infotainment systems, or it may require extra privileges from vehicle manufacturers before deployment, e.g., cruise control systems. The knowledge from this phase is stored in the knowledge base to be used in more informed decision making over the lifetime of the system. Uncertainties related to execution latency or executor reliability are expected at this stage [9].

The final part of the Kuksa* framework is the knowledge base that contains multiple repositories, which provide access to the knowledge and data from microservices according to the interfaces prescribed by the architecture. The knowledge base maintains data of the managed automotive system (i.e., configuration data, adaptation models, service data consumption, service configurations) and the environment (i.e., traffic modelling data, network data, connected objects profiles), adaptation goals, and the states shared by other four services in the autonomic controller. The challenge with using historical data in future scenarios is the effort to capture the delta in knowledge from prior and posterior states, quality requirements, and quality satisfaction levels.

5   Experimental Evaluation

5.1   Experimental scenario

To show how we can implement the Kuksa* framework, we used a laboratory setting and deployed a container-based control loop on the in-vehicle platform of the Eclipse Kuksa to demonstrate self-adaptation microservices in the automotive domain. The experiment was implemented on a mobile robotic device called rover. The control loop enabled adaptive services for video streaming from a camera on the rover to the cloud back-end. The video received in the cloud was encoded and forwarded to a cloud-native streaming service. The streaming service was attached to a storage service running on the cloud back-end that was responsible for archiving the received video packages. The evaluation has been done by comparing the system performance under evaluation when we had static and adaptive configurations (see section 5.2 for more detail on the configuration settings). The null hypothesis of this experiment indicates that there is no significant difference between the performance of the static and adaptive systems under evaluation.

We planned that the self-adaptive video streaming scenario uses three microservices in the control loop to monitor, analyse, plan, and execute new configurations based on the changes in the performing conditions and adjust its performance by adapting the video frame rate and scale according to network speed conditions. During the experiment, the rover was constantly moving in a circular path in the lab with a speed of 3 km/h. The *connection speed test service* was responsible for the monitoring of the network connection speed in the rover. The *user's config. service* enabled the registration of new user's configurations over-the-air using a Wi-Fi network. The *video adaptation service* took the responsibility of the planning and generating new adaptation strategies according to the network connection speed and the configuration setup received from

users. The list of adaptation strategies was defined as the microservice's input in the adaptation space (see Table 2). The *video-streaming services* dynamically executed new adaptation strategies from the adaptation strategy registry repository in case of changing configurations or network load conditions. We designed multiple data stores (representing the knowledge-base in MAPE-K loop) to be connected to the microservices, in which it allowed us to minimise failures and the overhead load in our system because microservices could access the historical data in case there were service interruptions, service failures, or no new configurations. All services were container-based and were running in a Kubernetes cluster that was responsible for creating managing service replicas. In the designed scenario, the frame rate and scale of the video were automatically and dynamically reduced if the load of the network increased and passed a specific threshold. The threshold was calculated based on the average upload speed in a period of three hours before the experiment.

5.2 Experimental setting

The in-vehicle platform: We used an open-source mobile robotic car, which has a RoverSense layer designed for in-vehicle communication demonstrations and a motor driver layer (Arduino) on top of a Raspberry Pi 3 Model B (RPi3). The rover hardware was simulating an electronic control unit (ECU) in vehicles. It runs based on the in-vehicle Kuksa platform, which included AGL as the operating system and an API for handling rover communications. The in-vehicle Kuksa platform also runs customised software called roverapp[4] designed for a Linux-based, embedded single board computer (i.e., RPi3). The RoverSense layer enables control of the rover and sends telemetry data from rover sensors and camera data, such as video streaming, infrared proximity, and ultrasound. In our experiment, the rover sent the video streams using the real-time messaging protocol (RTMP), which was enabled by the FFMPEG project. The list of microservices that were running on the Kuksa in-vehicle platform is as Table 1.

**Table 1.** Microservices configuration

Microservice	Technology	Configuration
Service discovery	Hashicorp	consul discovery
Video adaptation	python	–
Speed test	speedtest-cli	dynamic IP (DHCP)
Video streaming	FFMPEG	maxrate:3M, bufsize:6M, t:30s

Also, we had three configuration settings of 'low rate' (LR), 'high rate' (HR), and 'adaptive' in the adaptation space, as shown in Table 2.

Table 2. Adaptation space

Configuration setting		Frame rate	video scale
Static	LR	30	320:240
	HR	60	720:480
Adaptive		30	320:240
		60	720:480

---

[4] https://app4mc-rover.github.io/rover-app/
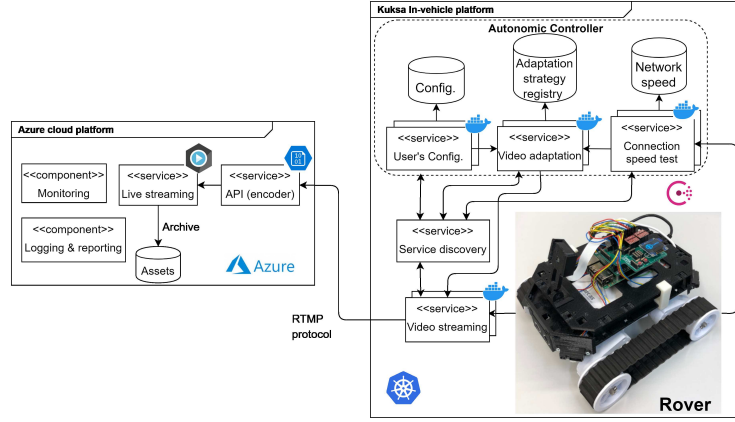
Fig. 3. The experimental setting.

The cloud back-end: In the designed experiment, the cloud back-end was running on the Microsoft Azure. We had a message encoder that was connected to a live streaming service on the back-end. The live streaming service was created in "Media service" on Microsoft Azure and allowed transforming, analysing, and streaming media on a single platform. The videos received in the cloud were archived in a database. All communications between the rover and Azure cloud platform were done through an open Wi-Fi connection on the RPi3. Figure 3 shows the automotive setting in our experiment.

5.3   Results

There were three rounds of video streaming from the rover to the cloud back-end. The first two rounds were static with fixed parameters for 'frame rate' and 'video scale', and the third round used the adaptive configuration based on the network load. Each experiment round included 100 runs of video streaming with a length of 30 seconds each, meaning 3000 seconds of video streaming in each round.

The adaptive system dynamically changed the video streaming configuration according to the latest adaptation strategies. The video adaptation service continuously provided new adaptation strategies based on the results from the monitoring and user config. services. In the adaptive configuration setting, the video adaptation service generated the LR configuration in 31% of the total runs, and the rest was done with the HR configuration. Figure 4 shows the outcome of our experiment using different configuration settings. It shows how the video streaming configuration was changing according to network connection loads. It indicates that the adaptive system configuration attempted to reach an optimum for the video frame rate and quality (low values in video frame quality graph means better image quality).

Table 3 provides the aggregated results of our experiment. In this table, $XrYq$ shows quality trade-offs for the video streaming sent to the cloud. For example, $5r5q$ declares equal weight for the video frame rate per second (fps) and the video frame quality. $9r1q$ shows that the importance of having double fps is nine times more than the frame quality, where $1r9q$ shows the opposite.
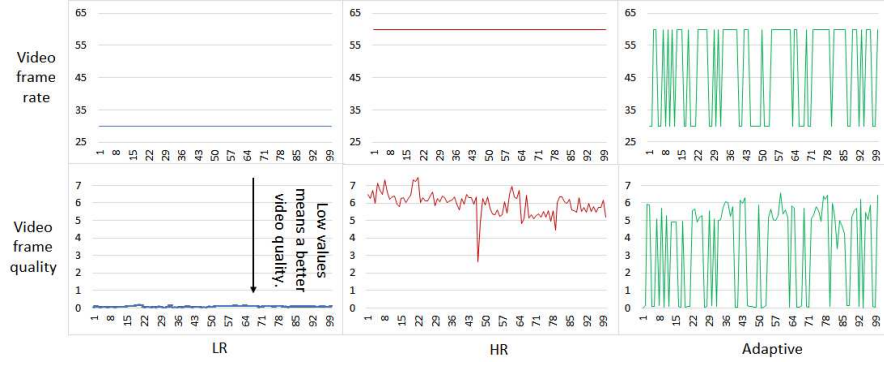
Fig. 4. The system behaviour in three configuration settings.

On the other side, $p1(sys)$ shows equal weight factors for the time and quality performances ($wt = wq = 0.5$), and $p2(sys)$ shows more importance given to the time performance rather than quality performance ($wt = 9 * wq$), where $p3(sys)$ shows the opposite weight factors and more stress on the quality performance ($wq = 9 * wt$).

Table 3. Evaluation results for two architecture designs

Metric	Static system						Adaptive system		
	LR			HR					
	5r5q	9r1q	1r9q	5r5q	9r1q	1r9q	5r5q	9r1q	1r9q
tp		1			1			0.91	
qp	0.74	0.55	0.94	0.6	0.92	0.28	0.68	0.78	0.58
$p1(sys)$	0.87	0.78	0.97	0.8	0.96	0.64	0.8	0.85	0.75
$p2(sys)$	0.97	0.96	0.99	0.96	0.99	0.93	0.89	0.9	0.88
$p3(sys)$	0.77	0.6	0.95	0.64	0.93	0.35	0.7	0.79	0.61

Remark: tp: time perf., qp: quality perf., $pN(sys)$: three weight settings for time and quality perf., $XrYq$: three weight settings for video frame rate and video frame quality.

It was necessary to provide multiple sets of trade-off in the evaluation setting because various scenarios in vehicles require customised, adaptive settings according to the system quality requirements; for example, safety-critical scenarios may need higher frame rates or lower configuration times comparing to the vehicular comfort systems.

The results show a better time performance when we had static system, $p2(sys)$, configurations. The reason for this is that the configuration time was shorter because the video streaming microservice did not require checking external configuration strategies. However, in an adaptive configuration scenario, the video streaming service had to continuously check the new adaptation strategies from video adaptation strategies, in which it reduced the time performance.

The quality performance of the adaptive system stood between the quality performances from the HR and LR configuration settings. This means that depending on the importance of the fps or frame quality in our requirements, the adaptive system configuration can be adjusted to deliver the required outcome. We could see similar results in the total system performance, $p(sys)$, measure-

ments. For example, when we required a higher frame rate with an emphasis on qp, the adaptive system showed better results than the static LR configuration setting, but when we needed higher frame quality and higher qp, we had better results from the adaptive system compared with the static HR setting.

> **Summary.** To have optimal performance from a self-adaptive automotive system, design trade-offs are necessary at runtime for balancing the expected quality requirements' satisfaction levels of each microservice and the whole system.

6   Discussion

6.1   Overview of findings and their implications

RQ1. How can microservices be applied in the design of self-adaptive systems in the automotive domain? The vehicular systems should be able to adjust their behaviours according to uncertainties in operating conditions, such as changes in the list of running services, quality requirements satisfaction levels, accessibility to data, adaptability of resources, and working contexts. Importing several microservices in the control loop of a self-adaptive system architecture can increase the complexity of the whole system, for example, because of the long reconfiguration time of multiple service replicas or a wrong perception of the vehicle's environment as the reason of conflicting data from different services. Although presenting control components in a form of separate microservices to optimise the behaviour of other services increased the modifiability, configurability, and availability of the system when there was a need for reconfigurations. System adaptations were isolated to limited services, while other services were not affected. This provides higher flexibility in proposing new complex adaptation strategies and models into safety-critical systems of a vehicle.

In vehicular systems, it is necessary to prioritise safety-critical quality requirements over other quality requirements when we make quality trade-off decisions. Microservices make it easier to design the safety-critical trade-offs when making adaptation decisions as they minimise the posterior conflicts among services. Microservice design allowed us to keep all services lightweight, which reduced the time and effort necessary for reconfiguration.

RQ2. What are the quality trade-offs at runtime when using self-adaptive microservices in the automotive domain? We evaluated the runtime performance of a self-adaptive architecture based on microservices in a laboratory setting. Our results indicate that the total performance of the system widely depends on the performance of the individual service instances. Increasing communication among the multiple service instances increased the total time of reaching the final optimum state after registering a new adaptation strategy, resulting in reduced configuration time performance in the system.

The designed architecture resulted in better fault-tolerance of the system because different service instances could fetch the historical data from the knowledge base in case there was no updated information available. This kept the whole

system running if the control services could not register adaptation strategies; although multiple data stores may raise challenges, such as data inconsistency. Using service management mechanisms, such as Kubernetes improved resiliency and healthy function of all services.

To guarantee performance levels, we designed multiple sets of configuration settings (see Table 3) that may be considered in various operating situations of vehicles. The preliminary set of quality attributes and their satisfaction levels for each service and the whole system can be negotiated and designed in vehicular systems; although new configurations may be fetched at runtime from analysis services that use machine learning.

6.2   Threats to validity

There are threats to the validity [28] of our results. Improving the construct validity required using the right measures in the experiment. The measures in this study were composed based on the relevant approaches and metrics for measuring software system performance in peer-reviewed studies. Internal validity is addressed through the relationship between the constructs and the proposed explanation. We applied an experimental approach in a laboratory setting with defined objectives. Also, the study planning and implementation, the experimental setting, and study results were reviewed and discussed among the authors.

External validity is related to the generalisability of the study. We decided to use a real-world platform and a mobile robotic device (simulating the vehicular ECUs) to address the adaptive performance of a video streaming service based on time and quality measures. Even though the system under evaluation does not represent all scenarios in the automotive domain, such as safety-critical situations, the findings show how self-adaption microservices can be used to adapt to dynamic real-time requirements of vehicular systems. In addition, Kuksa* was designed to be flexibly applied in other automotive scenarios. However, the findings of this study should not be generalised beyond its original scope. We have provided all the details about the experimental setting and all publicly available materials on GitHub[5].

6.3   Recommendations for future research

The measures used in this study were extracted from the prior software engineering studies. We propose future studies to investigate the performance of self-adaptive automotive microservices using domain-specific measures from automotive software engineering. In addition, future studies could strengthen our results by evaluating our services and experimental setting in a real automotive environment where there exist safety-critical scenarios, such as the service unreliability due to communication network interruptions. To improve the understanding of the self-adaptive microservices in automotive systems, we propose more studies on the learning configuration algorithms, domain-specific testing approaches, and quality measurements of self-adaptive microservices.

---

[5] https://github.com/ahmadbanijamali/Adaptive-video-streaming

Adaptive video-streaming services are used as an example to introduce the self-adaption microservice into automotive systems. We propose future studies to investigate the application of self-adaptation microservices in other automotive scenarios and review the set decisions about how decentralised each of the MAPE-K services must be made.

7   Conclusions

Microservices have gained increasing popularity in the automotive domain. Considering the dynamic working conditions in vehicles and beyond them, software systems should adapt their behaviours based on changes in the operating and surrounding environments. We aimed to investigate self-adaptation microservices in the automotive domain. To achieve that aim, we proposed the Kuksa* framework by updating the architecture of a real-world automotive platform and investigated the framework applicability in an automotive laboratory setting.

Our proposed design could adapt the system behaviour according to the changes in the performing conditions. Our study showed that system performance requires design trade-offs for balancing the expected quality requirements' satisfaction levels. Our findings show the potential of using microservices for self-adaptive automotive systems at runtime.

References

1. Aderaldo C. M., Mendonça N. C., Schmerl B., Garlan D., Kubow: An architecture-based self-adaptation service for cloud-native applications. In: Proc. of the 13th Eur. Conf. on Soft. Arch., pp. 42-45 (2019).
2. Banijamali A., Heisig P., Kristan J., Kuvaja P., Oivo M.: Software architecture design of cloud platforms in automotive domain: An online survey. In: 12th IEEE Int.l Conf. on Service-Oriented Computing and App., pp. 168-175. IEEE, (2019).
3. Banijamali A., Jamshidi P., Kuvaja P., Oivo M.: Kuksa: A cloud-native architecture for enabling continuous delivery in the automotive domain. In: Int. Conf. on Product-Focused Soft. Process Improv. (PROFES), pp. 455-472, LNCS, (2019).
4. Banijamali A., Pakanen O., Kuvaja P., Oivo M.: Software architectures of the convergence of cloud computing and the internet of things: A systematic literature review. Info. and Soft. Tech. **122**, 106271 (2020).
5. Bass, L., Weber, I., Zhu, L.: DevOps: A software architect's perspective. Addison-Wesley Professional (2015).
6. Baylov K., Dimov A.: Reference architecture for self-adaptive microservice systems. In: Int. Symp. on Intell. and Distri. Computing, pp. 297-303. Springer, (2017).
7. De Lemos R., Garlan D., Ghezzi C., Giese H., Andersson J., Litoiu M., Schmerl B., Weyns D., Baresi L., Bencomo N.: Software engineering for self-adaptive systems: Research challenges in the provision of assurances. In: Soft. eng. for self-adaptive sys. III. assurances, pp. 3-30. Springer (2017).
8. Eberhardinger B., Ponsar H., Klumpp D., Reif W.: Measuring and evaluating the performance of self-organization mechanisms within collective adaptive systems. In: Int. Symp. on Leveraging App. of Formal Methods, pp. 202-220. Springer, (2018).
9. Esfahani N., Malek S.: Uncertainty in self-adaptive software systems. In: Software engineering for self-adaptive systems II, pp. 214-238. Springer (2013).

10. Filieri A., Hoffmann H., Maggio M.: Automated multi-objective control for self-adaptive software design. In: Proc. of the 2015 10th Joint Meeting on Foundations of Soft. Eng., pp. 13-24 (2015).
11. Hassan S., Bahsoon R., Microservices and their design trade-offs: A self-adaptive roadmap. In: Int. Conf. on Services Computing (SCC), pp. 813-818. IEEE, 2016.
12. Ho H. N., Lee E.: Model-based reinforcement learning approach for planning in self-adaptive software system. In: Proc. of the 9th Int. Conf. on Ubiquitous Info. Management and Commun., pp. 1-8 (2015).
13. Jain, P.: Automotive Cloud Technology to Drive Industry's New Business Models - 2019-05-07. http://shiftmobility.com/2017/06/automotive-cloud-technology-drive-automotive-industrys-new-business-models.
14. Jamshidi P., Casale G.: An uncertainty-aware approach to optimal configuration of stream processing systems. In: 2016 IEEE 24th Int. Symp. on Modeling, Analysis and Simul. of Computer and Telecommu. Sys., pp. 39-48. IEEE, (2016).
15. Kephart J. O., Chess D.M.: The vision of autonomic computing. Computer **36(1)**, pp. 41-50 (2003).
16. McGeoch C.: A guide to experimental algorithmics. Cambridge Univ. Press (2012).
17. Mendonça N. C., Jamshidi P., Garlan D., Pahl C.: Developing self-adaptive microservice systems: Challenges and directions. IEEE Software (2019).
18. Newman S.: Building microservices: designing fine-grained systems. "O'Reilly Media, Inc." (2015).
19. Nobre J. C., de Souza A.M., Rosário D., Both C., Villas L.A., Cerqueira E., Braun T., Gerla M.: Vehicular software-defined networking and fog computing: Integration and design principles. Ad Hoc Netw. **82**, pp. 172-181 (2019).
20. Pahl C., Jamshidi P., Weyns D.: Cloud architecture continuity: Change models and change rules for sustainable cloud software architectures. J. of Soft. Evolution and Process **29 (2)**, e1849 (2017).
21. Pahl C., Jamshidi P., Zimmermann O.: Architectural principles for cloud software. ACM Trans. on Internet Tech. (TOIT) **18(2)**, pp. 1-23 (2018).
22. Pereira J. A., Silva R., Antunes N., Silva J., de França B., Moraes R., Vieira M.: A Platform to Enable Self-Adaptive Cloud Applications Using Trustworthiness Properties. In: 15th Int. Symp. on Soft. Eng. for Adaptive and Self-Managing Sys. (SEAMS), IEEE, (2020).
23. Shukla R. M., Sengupta S., Chatterjee M.: Software-defined network and cloud-edge collaboration for smart and connected vehicles. In: Proc. of the Workshop Prog. of the 19th Int. Conf. on Distri. Computing and Netw., pp. 1-6 (2018).
24. Siegel J. E., Erb D.C., Sarma S.E.: A survey of the connected vehicle landscape—architectures, enabling technologies, applications, and development areas. IEEE Trans. on Intell. Transp. Sys. **19**, pp. 2391-2406 (2017).
25. Staron M.: Automotive software architectures. Springer, (2017).
26. Sun J., Yang C., Tanjo T., Sage K., Aida K.: Implementation of self-adaptive middleware for mobile vehicle tracking applications on edge computing. In: Int. Conf. on Internet and Distri. Computing Sys., pp. 1-15. Springer, (2018).
27. Weyns D., Iftikhar M.U., Hughes D., Matthys N.: Applying architecture-based adaptation to automate the management of internet-of-things. In: European Conf. on Soft. Arch., pp. 49-67. Springer (2018).
28. Wohlin C., Runeson P., Höst M., Ohlsson M.C., Regnell B., Wesslén A.: Experimentation in software engineering. Springer Science & Business Media. (2012).
29. Zeller M., Prehofer C.: Timing constraints for runtime adaptation in real-time, networked embedded systems. In: 7th Int. Symp. on Soft. Eng. for Adaptive and Self-Managing Sys. (SEAMS), pp. 73-82. IEEE, (2012).