# RobotKube: Orchestrating Large-Scale Cooperative Multi-Robot Systems with Kubernetes and ROS

Bastian Lampe<sup>®</sup>\*, Lennart Reiher<sup>®</sup>\*, Lukas Zanger<sup>®</sup>\*, Timo Woopen<sup>®</sup>\*, Raphael van Kempen<sup>®</sup>, and Lutz Eckstein

Abstract-Modern cyber-physical systems (CPS) such as Cooperative Intelligent Transport Systems (C-ITS) are increasingly defined by the software which operates these systems. In practice, microservice architectures can be employed, which may consist of containerized microservices running in a cluster comprised of robots and supporting infrastructure. These microservices need to be orchestrated dynamically according to ever changing requirements posed at the system. Additionally, these systems are embedded in DevOps processes aiming at continually updating and upgrading both the capabilities of CPS components and of the system as a whole. In this paper, we present RobotKube, an approach to orchestrating containerized microservices for large-scale cooperative multirobot CPS based on Kubernetes. We describe how to automate the orchestration of software across a CPS, and include the possibility to monitor and selectively store relevant accruing data. In this context, we present two main components of such a system: an event detector capable of, e.g., requesting the deployment of additional applications, and an application manager capable of automatically configuring the required changes in the Kubernetes cluster. By combining the widely adopted Kubernetes platform with the Robot Operating System (ROS), we enable the use of standard tools and practices for developing, deploying, scaling, and monitoring microservices in C-ITS. We demonstrate and evaluate RobotKube in an exemplary and reproducible use case that we make publicly available at github.com/ika-rwth-aachen/robotkube.

#### I. INTRODUCTION

Cyber-physical systems (CPS), such as Cooperative Intelligent Transport Systems (C-ITS), comprise a diverse range of interconnected entities that can vary in number and type. Next to automated vehicles, there may exist traffic control systems, road side units, control centers, and (edge-) clouds providing additional services. This leads to the question of how the software for all these different systems can work together efficiently and safely over time.

One popular approach to build such complex software systems are microservice architectures [1]. The complete system is broken up into many loosely coupled, fine-grained services. They communicate through predefined protocols. Services in this architecture are typically run in containers, a type of virtualization allowing services to be run in isolation of each other. A major challenge when using containers in microservice architectures is the orchestration of said containers. Orchestration involves the automated deployment, scaling, and management of containerized applications. Kubernetes is a popular open source system for this task used by many large software companies worldwide. It already comes with a lot of the capabilities that would also be needed in a C-ITS employing a microservice architecture. [2]

Kubernetes lacks methods for orchestration that are domain-specific, e.g., to C-ITS. Specific tasks that are needed only at certain times, like deploying required applications, or the recording of relevant data, must be defined either by C-ITS administrators, or programmatically by C-ITS developers. The required tasks may depend on the specific content of data exchanged in the Kubernetes cluster instead of on metadata, e.g., the load of a server. RobotKube describes new software components, which are themselves containerized and can be orchestrated by Kubernetes, that extend the regular capabilities of Kubernetes to those specific to robotic systems in general and to C-ITS in particular.

We present an *event detector*, a software component based on the Robot Operating System (ROS), that can take as input any data provided by services in the cluster, analyze the data based on analysis rules implemented by developers, and formulate tasks based on the result of these analyses. Possible tasks include deploying new applications, the reconfiguration of existing applications, or the recording of a set of data which was determined relevant for further analysis and storage as part of a DevOps process.

In addition, we introduce an *application manager* acting as the link between the event detector and the Kubernetes control plane. It translates requirements for specific applications or configurations communicated by the event detector into a specific workload for the Kubernetes cluster. Together with the event detector and its operator plugin, the application manager forms an *operator application* that automates the management of the cluster based on occurring events.

We apply RobotKube in an exemplary use case where a cloud-based operator application detects when two automated vehicles approach each other. It then deploys communication software allowing them to transmit additional sensor data to the cloud. There, a dynamically deployed *recording application* gathers corresponding sensor and location data and saves them to a database. This use-case represents the first step in a data-driven DevOps process allowing Collective Learning [3]. The location data of one vehicle can serve as a label in the sensor data set of the the other vehicle and vice versa. Training data sets for supervised learning can hereby be generated automatically without human labeling.

This use case is of course only one of many which are possible with the approach described in this work.

© 2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

<sup>\*</sup>These authors contributed equally to this work.

All authors are with the Institute for Automotive Engineering (ika), RWTH Aachen University, 52074 Aachen, Germany. {firstname.lastname}@ika.rwth-aachen.de

In summary, our work makes these main contributions:

- Introduction of RobotKube, an approach to automatically orchestrating containerized microservices for large-scale cooperative multi-robot systems based on Kubernetes and ROS.
- Presentation of two main components of RobotKube: an event detector acting upon the occurrence of data patterns, e.g., requesting the deployment of additional applications; and an application manager capable of automatically configuring required changes in the Kubernetes cluster. Together, the two components act as an automated Kubernetes operator application.
- Examination of RobotKube in an exemplary use case involving the automated deployment of various software components to multiple connected C-ITS nodes, and the recording of data relevant to the use case.
- Publication of the experimental setup and involved Docker images to make the exemplary use case reproducible and allow other researchers to see the system in action.

## II. RELATED WORK

Our work heavily builds upon containerization, Kubernetes, and the Robot Operating System (ROS). For an introduction to these tools, see [2], [4], [5], [6]. Our approach combines advantages from each of these tools.

Containerization, the process of encapsulating applications and their dependencies into isolated units, offers a range of benefits for application management. It enables rapid deployment and over-the-air updates. Compatibility issues are reduced, and applications become portable across machines. Components can be easily reused and shared with others. Containers have a lightweight footprint and minimal overhead, optimizing resource utilization. Maintenance is simplified as applications and dependencies are encapsulated. This makes them suitable to container orchestration software [6]. Different tools for containerization exist. The authors in [7], [8], [9] propose *snaps* for production robots because they make available interfaces for accessing low-level hardware and come with a robust update system. In our approach, we choose Docker containers because of their popularity, their easy integration into Kubernetes, and their versatility in the development phase of our approach. It is conceivable to move to a different containerization framework later.

*Orchestration* is "the automated configuration, management, and coordination of computer systems, applications, and services" and "helps to more easily manage complex tasks and workflows" [10]. *Kubernetes* is one popular orchestration platform that brings several advantages to the deployment and management of containerized applications. It abstracts infrastructure complexities, ensuring portability across different environments. It enables automatically adjusting resources based on workload demands, and supports horizontal scaling. Kubernetes offers self-healing capabilities, automatically restarting failed containers, and supports rolling updates and rollbacks for seamless application upgrades. It provides service discovery and load balancing mechanisms for efficient traffic routing and distribution. Kubernetes' declarative configuration allows for reproducibility and version control, reducing configuration drift. [2], [4].

The described capabilities make Kubernetes suitable to large-scale robotic systems. Popular alternatives include docker compose [11] and docker swarm [12]. In the context of robotics, Kubernetes is employed for Industry 4.0 applications [13]. Examples in the context of C-ITS include [14], [15], [16], [17]. The current trend towards software-defined vehicles reinforces the importance of orchestration tools like Kubernetes in C-ITS. Their use can reduce the time period for the release of software updates [14], [18]. An important driver of the need for orchestration software can also be found in the increased connectedness between automated vehicles and supporting infrastructure. Increased research activity is found regarding the use of sensored infrastructure and edge clouds for C-ITS. Sensored road side units can play an important role in supporting automated vehicles in their operation [19], e.g., by providing additional perception data to mitigate challenges like occlusions and limited sensor range [20]. Edge clouds or clouds can be used for function offloading to make use of more powerful compute resources outside of vehicles [21]. An orchestration system managing C-ITS shall therefore encompass all interacting subsystems including automated vehicles and supporting infrastructure. While other research focuses on, e.g., real-time capabilities of orchestration [13], and requires human operators [22], our work describes an approach to automate cluster operations. In contrast to other automated systems in which operators are either only cloud-based [15], or vehicle-based [23], RobotKube is in principle decentralized and agnostic to whether operators are, e.g., based in the cloud, in sensored road side units, or in automated vehicles.

Applications and Services for robotic systems are often developed using common software frameworks and middlewares. For RobotKube, we choose *ROS*, a widely adopted set of open source software libraries and tools for building robot applications. It provides advantages for development through its ecosystem of tools, libraries, and capabilities. Its global community continuously enhances the software ecosystem. ROS is widely used in research projects and production robots worldwide. It supports various platforms, including Linux, Windows, and microcontrollers. ROS is open source, offering customization and integration flexibility [5].

Alternatives to ROS include YARP [24], EB Assist ADTF [25], ASOA [26], and AUTOSAR Adaptive [27]. These come with different capabilities and objectives, but can partly also be made compatible with ROS. There also exist the products of Apex.AI [28] which build upon ROS.

Due to their extensive capabilities, a combination of our chosen tools is already used in various contexts of robotic applications in general and C-ITS in particular. Based on their characteristics, they are especially suited to be employed in microservice or service-oriented architectures (SOA). Automotive SOAs that make use of these tools have been proposed in various initiatives both in academia and in industry [23], [26], [29], [30], [31], [32], [33].

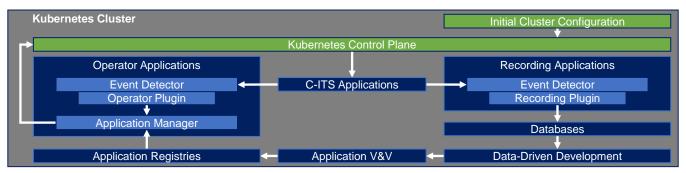


Fig. 1. The overall system architecture of the RobotKube approach contains its essential components and depicts the general orchestration and application development process in a C-ITS. An initial configuration is used to deploy C-ITS applications. These include two types of special applications. Operator applications can interact with the control plane to, e.g., deploy additional applications, which are made available in application registries. Recording applications can selectively store data in databases. The stored data can be used for data-driven development of new applications, or for application updates. All applications need to be verified and validated before they are made available in the application registries.

#### III. ROBOTKUBE

RobotKube is an approach to orchestrating containerized ROS-based robotic applications in microservice architectures using Kubernetes. We present and apply our approach in the context of C-ITS, but also point to its applicability to large-scale robotic systems in general. RobotKube describes new software components in a novel architecture that enables a highly automated operation of C-ITS, including dynamic software deployments, data-driven development, and verification and validation of applications.

Building on top of Kubernetes gives access to many features relevant for the operation of a C-ITS. These include, but are not limited to: fault tolerance through high availability nodes, sophisticated rollout and rollback processes, selfhealing mechanisms, load balancing mechanisms, and ondemand horizontal autoscaling. The incorporation of ROS similarly enables access to and usage of a vast existing ecosystem of open source software and tools for robotics applications, including automated vehicles and C-ITS.

## A. System Architecture

The overall system architecture of RobotKube is depicted in Fig. 1. A manually defined *initial cluster configuration* in the form of a set of *C-ITS applications* is deployed by the *Kubernetes control plane* to all agents that are part of the cluster and part of the C-ITS. Initially, the C-ITS administrators act as operators who define the cluster configuration and the set of initial deployments.

- An *application* within the scope of *RobotKube* is a set of one or several microservices with a particular purpose within the C-ITS.
- An *operator* within the scope of *RobotKube* is a person or software that manages application deployments, life cycles, cluster configurations and additional cluster management tasks within the C-ITS.

Applications are developed as part of a *data-driven development* process. Before being deployed, each application and its services must pass *application V&V*, i.e., the verification and validation of their desired functioning. Applications that have passed this stage are made available in *application registries*. A core goal of RobotKube is to automate the cluster operation through *operator applications* that can automatically deploy, configure, and manage applications in the cluster. The operator applications act on developer-defined events in the cluster. *Events* are associated with occurences of certain patterns in the data exchanged in the cluster, and are detected by dedicated *event detector* components. Based on the events, a second component, the *application manager*, issues new deployments or reconfigurations. As an example, an operator application may detect that a connected vehicle is approaching an intersection and then automatically deploys a supportive function onto a nearby sensored road-side unit.

The detection of developer-defined events is also relevant for *recording applications* that enable on-demand data recording to a database. Recorded data can in turn be used for data-driven development of C-ITS applications, including automated collective learning techniques [3].

Table I lists more key design principles of the RobotKube approach as a whole as well as design principles specific to operator applications in RobotKube. Note that the list of design principles is neither exhaustive nor set in stone, but acts a a guiding foundation for the presented approach and its individual components. We plan to continually revisit and refine the design principles in future work.

## B. Event Detector

The event detector is an integral part both of operator applications and of recording applications. Its main purpose is to detect and act upon developer-defined events that are associated with patterns in the data exchanged in the cluster. An event detector is composed of three main subcomponents:

*Buffer* – All incoming data in the form of ROS messages is buffered for a configurable amount of time. It is infeasible and also undesirable to permanently store all accruing data, but the detection of events may require access to a history of data. The buffer is realized as a ring buffer covering a configurable duration of the immediate past. The buffer and the event detector in general are agnostic to specific data types and therefore compatible with all kinds of data exchanged in the cluster.

#### TABLE I

DESIGN PRINCIPLES FOR THE ROBOTKUBE APPROACH, INCLUDING SPECIFIC DESIGN PRINCIPLES REGARDING OPERATOR APPLICATIONS

Design Principle	Example	
Connected agents make at least some of their <i>compute units</i> available as nodes to a Kubernetes cluster.	A C-ITS cluster consists of nodes in the cloud, on edge servers, in control centers in sensored road side units, and in automated vehicles.	
An <i>application</i> is a set of one or several microservices with a particular purpose.	An environment perception application may be composed of two separate microservices for object detection and object tracking.	
An <i>operator</i> is a person or software that manages application deployments, life cycles, cluster configurations and more.	A cloud-based operator application detects a vehicle approaching an intersection and deploys a perception application to a road-side unit to support the vehicle.	
Different applications are managed by different <i>decentralized application-specific</i> operators.	In addition to the above cloud-based operator application for deploying supportive infrastructure functions, the deployment of a recording application is handled by another, separate vehicle-based operator application.	
Each microservice is packaged into its own container.	Two microservices of an environment perception application, object detection and object tracking, are separately packaged into their own containers.	
Applications are in principle node agnostic.	If one node is unavailable, an application may be deployed to a different node capable of running the application.	
Application <i>updates</i> are conducted by updating the individual container images of an application.	The object detection microservice of an environment perception application is updated independently of other containers of that application.	
Operator applications may deploy other operator applications forming <i>operator application chains</i> .	One cloud-based operator application deploys another vehicle-based operator applica- tion to vehicles approaching an intersection. The vehicle-based operator application detects uncertainty in order to eventually deploy an environment perception application on the connected intersection infrastructure.	
Operator applications <i>select</i> adequate applications and their components from <i>application registries</i> .	The application manager of an operator application for environment perception chooses adequate object detection and object tracking services based on requirements defined in the task description received from the event detector, and based on guarantees associated with applications in the <i>application registries</i> .	
Operator applications support automatic conflict resolution.	If there are no compute resources left in a vehicle, an operator application is able to cancel or postpone deployments, or resolve the conflict by other means such as offloading to a different connected agent.	

*Analysis* – The data available in the ring buffer is periodically analyzed in order to detect events. Events are associated with data patterns that are identified through an application-specific, developer-defined analysis of the buffered data. In order to support the detection of arbitrary data patterns, the event detector provides a generic and easily extensible framework for accessing the data buffer and for analyzing the data with regard to occurring events.

Action Plugin – Having detected an event, action plugins implement the resulting consequence. An operator plugin, e.g., is used for requesting the deployment or reconfiguration of applications. It forwards a corresponding task description to an application manager. An event detector in combination with an operator plugin and an application manager forms an operator application. Similarly, an event detector in combination with a recording plugin forms a *recording application*. Through its action plugin mechanism, the event detector is designed to also cover use cases beyond the presented operator and recording applications.

The event detector software component is implemented as a high-performance C++ ROS node.

## C. Application Manager

The application manager is an integral part of operator applications. It receives a task description from an event detector's operator plugin and translates it to a specific Kubernetes workload definition, which is then transmitted to the Kubernetes control plane.

The composition of requested applications from available microservices is handled by an application manager. The corresponding event detector only formulates highlevel requirements. The application manager then identifies suitable containerized services, configures and links them to form the requested applications, which are then deployed to appropriate nodes in the cluster. Within the context of RobotKube, application managers are the preferred way to interact with the Kubernetes control plane in the context of launching and managing applications.

An application manager is not only capable of launching new applications, but also of managing existing applications it has launched. As part of the task description, application managers can also be requested to reconfigure existing applications, or to shut down running applications. An application manager is also responsible for deciding whether to issue requested Kubernetes workloads in the first place. Task descriptions transmitted to an application manager therefore only represent an intent or a request, not an obligation. If a requested deployment is not possible, it is the application manager's responsibility to resolve the conflict.

The application manager software component is implemented as a Python ROS node invoking the Kubernetes Python API to interact with the Kubernetes control plane.

#### **IV. EXPERIMENTAL SETUP**

We apply our approach in an exemplary use case with the goal of demonstrating its main abilities, namely employing an application-specific event detector for the detection of an event to trigger the deployment of an additional application in the cluster. In particular, this involves

- detecting an event in an event detector based on data that is exchanged in the cluster;
- transmitting a task description message containing highlevel requirements regarding desired applications from the event detector to an application manager via the operator plugin;
- configuring the application deployments in an application manager based on the received task description;
- deploying the requested applications in the cluster;
- and managing the running applications over time.

In our exemplary use case, we aim at automatically deploying a recording application in certain situations encountered by automated vehicles with the goal to selectively gather data in a cloud-based database. The individual vehicles have insufficient information for the decision when to send the desired data to the cloud for storage. A cloud-based operator application shall therefore automatically trigger the deployment of a cloud-based recording application plus the required communication components. The data collected that way would enable collective learning [3] methods. A detailed use case description follows.

- N vehicles  $V_0, \ldots, V_N$  follow their current routes and are part of a C-ITS.
- M ≤ N vehicles V<sub>0</sub>,..., V<sub>M</sub> are equipped with lidar sensors producing point clouds.
- All vehicles send their current location/pose at a frequency of  $f_p$  to a cloud server C, where an event detector with operator plugin receives the data. In order to save bandwidth, no lidar point clouds are sent initially.
- The cloud-based event detector with operator plugin continually analyzes the vehicles' poses in order to detect when any two of the lidar-equipped vehicles  $V_0, \ldots, V_M$  are within a distance of  $d_{start}$  to each other.
- Once any two lidar-equipped vehicles V<sub>i</sub>, V<sub>j</sub>, i ≠ j, i ≤ M, j ≤ M are close enough to each other, the event detector with operator plugin issues a request to launch a recording application for recording the two vehicles' poses and point clouds.
- A cloud-based application manager receives the request. To realize the requested application, it starts communication modules and a recording application in the cloud. The added communication modules transmit the point clouds at a frequency of  $f_{pc}$  from  $V_i$  and  $V_j$  to C.
- The event detector with recording plugin is configured to store poses and point clouds from  $V_i$  and  $V_j$  in a database without further analysis.
- Once vehicles  $V_i$  and  $V_j$  have veered away from each other by more than a distance of  $d_{stop}$ , the deployed recording application is shut down, including the point cloud transmission.

The concrete configuration of experimental parameters is found in Table II. The involved cluster components and data flows are illustrated in Fig. 2. Individual software components are deployed as Docker containers in Kubernetes pods, running on Kubernetes nodes. Most software components are ROS-based. We simulate live vehicle data by playing back ROS bags recorded in simulation. Within the scope of one Kubernetes node, data is exchanged in the form of ROS messages. For the data transmission from vehicles to cloud, a brokered communication model using MQTT is chosen. Dedicated mgtt\_client ROS nodes bridge ROS messages to MQTT and vice-versa [34]. Note that other communication models could be employed and that the communication latencies within the single-host KinD cluster do not realistically model real-world conditions. Recording applications store ROS message data in a MongoDB database.

In order to simulate and test the presented use case, we set up a Kubernetes cluster using *Kubernetes-in-Docker (KinD)* [35]. This setup allows to run a multi-node Kubernetes cluster in a controlled environment on a single machine, which also enhances reproducibility for other researchers.

## V. EVALUATION

The previously described experimental setup allows us to test and evaluate the applicability of our approach for complex C-ITS use cases such as the one at hand. The setup mainly serves the purpose of demonstrating RobotKube's capabilities and giving an idea of how the approach translates into practice.

Fig. 2 illustrates the quickly growing complexity of seemingly simple use cases for C-ITS and large-scale CPS in general. In a distributed microservice architecture, the number of software components quickly outgrows the number of connected hardware components. Combined with the dynamic nature of C-ITS, an efficient and manageable orchestration approach becomes a key enabler.

RobotKube's core orchestration component, i.e., operator applications, allow a Kubernetes-orchestrated cluster to dynamically act upon data that is currently being exchanged in the cluster. Using event detectors and application managers as described by RobotKube, we can successfully demonstrate a fully-automated event-based distributed data collection use case in a C-ITS. Since the experimental setup is made open source as part of this work, the exemplary use case and the accompanying cluster behavior can also be reproduced and studied by other researchers.

 TABLE II

 Configuration of experimental parameters

Parameter	Description	Value
N	Number of vehicles	15
M	Number of lidar-equipped vehicles	2
$f_p$	Pose frequency	100 Hz
$f_{pc}$	Point cloud frequency	10 Hz
$d_{start}$	Trigger distance between $V_i$ , $V_j$	400 m
$d_{stop}$	Stopping distance between $V_i$ , $V_j$	500 m

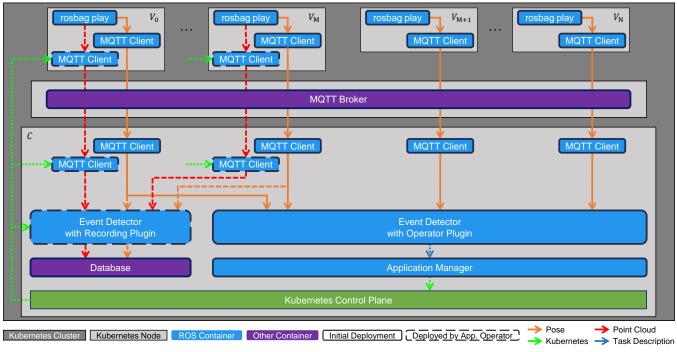


Fig. 2. Software components involved in experimental setup: vehicle poses (orange lines) are sent from vehicles  $V_0, \ldots, V_N$  to the cloud C using MQTT; an event detector with operator plugin triggers a task description (blue line), when two lidar-equipped vehicles are close to each other; the task description, asking to record pose and point cloud data of the two involved vehicles  $V_0, V_M$ , is received and processed by an application manager; the application manager transmits a workload request to the Kubernetes Control Plane in order to launch new communication modules and a recording application (dashed lines and blocks); poses and point clouds are stored in a database.

As a first quantitative evaluation of the presented approach, we describe different involved latencies. It is important to note that the results depend on the individual hardware setup and use case. Therefore, their main purpose is to give a general idea of the involved latencies, but they cannot be generalized easily. Thus, only approximate values are given here. Nonetheless, considering the involved latencies remains crucial in the development of safe C-ITS.

- Communication: Communication latencies play a large role in all distributed systems, especially if wireless communication is involved as required in C-ITS. Concrete latencies depend on various factors such as the used communication technology, e.g., 5G or ITS-G5. Note that communication latencies are largely neglected in our experimental use case, as the distributed cluster of nodes is only virtualized.
- Event detection: The developer-defined data analysis for detecting events naturally depends on the complexity of the data analysis. Here, detecting that two vehicles are close is a matter of a few milliseconds.
- 3) Translation to Kubernetes workload: The application manager translates the event detector's task description into a matching Kubernetes workload definition. This step involves the composition of a requested application from a list of available microservices, the configuration of to-be-launched components, validity checks, and possibly additional information requests via the control plane. Overall, latencies in the range of one hundred milliseconds can be expected.

- 4) Cluster reconciliation: Having received a workload definition from the application manager, the Kubernetes control plane induces the desired cluster state through a reconciliation process. At this point, new applications in the form of Kubernetes pods and potentially other Kubernetes components are launched, reconfigured, or shut down. In our experimental use case, the reconciliation phase is responsible for the majority of the total latency: the four new MQTT clients and the event detector with recording plugin take approximately five seconds to assume operation.
- 5) Data storage: A recording application stores data to a database in our use case, without further analysis or event detection. The data storage latency naturally scales with the amount of data to store. Given sufficiently large data buffers, it can also run asynchronously from the other processes. In our use case, ten seconds worth of pose and point cloud data from the two involved vehicles are written to the database in approximately half a second.

The largest share of latency observed between event and data storage is attributed to the cluster reconciliation. Launching containers in the cluster comes with an overhead in the range of several seconds. While this latency is still acceptable for many C-ITS use cases, it poses constraints on the kinds of applications that can be realized in a C-ITS operated in this way. More advanced techniques like pre-launching idle containers are expected to open up the approach to more use cases over time.

## VI. CONCLUSION

The presented approach aims to automate the orchestration of microservices in multi-robot CPS such as C-ITS built upon Kubernetes and ROS. For this purpose, we describe design principles, provide necessary new software components, and place them in an overall C-ITS architecture connecting dynamically deployed applications, automated data-driven development, and the verification and validation of applications running in a C-ITS. Two essential types of applications are explained in this paper: *operator applications* and *recording applications*.

To enable these applications, the two software components *event detector* and *application manager* are developed. With the event detector, developer-defined events in a Kubernetes cluster can be detected, and high-level requirements regarding new applications are defined. The application manager may then deploy or reconfigure specific applications in the cluster based on these requirements and available resources. The overall approach is demonstrated in an exemplary use case. Important latencies like the startup time of an automatically deployed application are examined. The results underline that deployments of new applications or reconfiguration need to take orchestration latencies into account.

In general, our approach benefits from the vast capabilities of Kubernetes and ROS that reach far beyond those tackled in this paper. Here, we want to lay the foundation to apply the presented approach to a wide range of use cases in robotics in general and C-ITS in particular. For this purpose, and to make our research reproducible, we publish the software, data, and Kubernetes configurations used in our experiments.

#### VII. ACKNOWLEDGEMENTS

This research is accomplished within the research projects "AUTOtech.*agil*" (FKZ 1IS22088A), "UNICAR*agil*" (FKZ 16EMO0284K), and "6GEM" (FKZ 16KISK036K). We acknowledge the financial support by the Federal Ministry of Education and Research of Germany (BMBF).

#### REFERENCES

- I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microser-vice architecture: Aligning principles, practices, and culture.* O'Reilly Media, 2016.
- [2] Kubernetes. [Online]. Available: kubernetes.io
- [3] B. Lampe, L. Reiher, T. Woopen, and L. Eckstein, "Cloud intelligence and collective learning for automated and connected driving," *ATZelectronics worldwide*, 2022.
- [4] B. Burns, J. Beda, and K. Hightower, *Kubernetes: Up and running*. O'Reilly Media, 2019.
- [5] Robot Operating System (ROS). [Online]. Available: ros.org
- [6] S. P. Kane and K. Matthias, *Docker up & running*. O'Reilly Media, 2023.
- [7] R. Davies. (2020) Keep Enterprise ROS Robots Up-to-Date with Snaps. [Online]. Available: ubuntu.com/blog/keep-enterpriseros-robots-up-to-date-with-snaps
- [8] G. A. Noury. (2023) ROS Docker; 6 reasons why they are not a good fit. [Online]. Available: ubuntu.com/blog/ros-docker
- [9] G. Barbieri. (2023) Snapping out of Docker: a robotics guide for migrating Docker to Snap. [Online]. Available: ubuntu.com/blog/ keep-enterprise-ros-robots-up-to-date-with-snaps
- [10] Red Hat, Inc. What is orchestration? [Online]. Available: redhat.com/ en/topics/automation/what-is-orchestration
- [11] Docker, Inc. Docker compose overview. [Online]. Available: docs.docker.com/compose

- [12] —. Docker swarm mode overview. [Online]. Available: docs.docker.com/engine/swarm
- [13] M. Barletta, M. Cinque, L. D. Simone, and R. D. Corte, "Introducing k4.0s: a model for mixed-criticality container orchestration in industry 4.0," in *IEEE Intl Conf on Dependable, Autonomic and Secure Computing*, 2022.
- [14] Microsoft, "Mercedes-Benz R&D creates 'container-driven cars' powered by Microsoft Azure," 2020. [Online]. Available: customers.microsoft.com/en-us/story/784791-mercedes-benz-rand-d-creates-container-driven-cars-powered-by-microsoft-azure
- [15] N. Slamnik-Kriještorac, G. M. Yilma, F. Zarrar Yousaf, M. Liebsch, and J. M. Marquez-Barja, "Multi-domain mec orchestration platform for enhanced back situation awareness," in *IEEE Conference on Computer Communications Workshops*, 2021.
- [16] S. Aldegheri, N. Bombieri, F. Fummi, S. Girardi, R. Muradore, and N. Piccinelli, "Late Breaking Results: Enabling Containerized Computing and Orchestration of ROS-based Robotic SW Applications on Cloud-Server-Edge Architectures," ACM/IEEE Design Automation Conference, 2020.
- [17] J. Kunkel et al., "DECICE: Device-Edge-Cloud Intelligent Collaboration Framework," 2023. [Online]. Available: arxiv.org/ abs/2305.02697
- [18] A. W. Malik, A. U. Rahman, A. Ahmad, and M. M. D. Santos, "Overthe-air software-defined vehicle updates using federated fog environment," *IEEE Transactions on Network and Service Management*, 2022.
- [19] L. Klöker, A. Klöker, F. Thomsen, A. Erraji, and L. Eckstein, "Traffic Detection Using Modular Infrastructure Sensors as a Data Basis for Highly Automated and Connected Driving," in *Aachen Colloquium Sustainable Mobility*, 2020.
- [20] X. Duan, H. Jiang, D. Tian, T. Zou, J. Zhou, and Y. Cao, "V2I based environment perception for autonomous vehicles at intersections," *China Communications*, 2021.
- [21] P. Arthurs, L. Gillam, P. Krause, N. Wang, K. Halder, and A. Mouzakitis, "A Taxonomy and Survey of Edge Cloud Computing for Intelligent Transportation Systems and Connected Vehicles," *IEEE Transactions* on Intelligent Transportation Systems, 2022.
- [22] J. Ichnowski *et al.*, "Fogros2: An adaptive platform for cloud and fog robotics using ros 2," 2023.
- [23] A. Kampmann et al., "A dynamic service-oriented software architecture for highly automated vehicles," in *IEEE Intelligent Transportation* Systems Conference (ITSC), 2019.
- [24] Robotology. Yarp. [Online]. Available: github.com/robotology/yarp
- [25] Elektrobit. Eb assist adtf. [Online]. Available: elektrobit.com/products/ automated-driving/eb-assist/adtf
- [26] A. Kampmann, A. Mokhtarian, S. Kowalewski, and B. Alrifaee, "ASOA - A Dynamic Software Architecture for Software-defined Vehicles," in *Aachen Colloquium Sustainable Mobility*, 2022.
- [27] AUTOSAR. Adaptive platform. [Online]. Available: autosar.org/ standards/adaptive-platform
- [28] J. Becker, "Betriebssystem für softwaredefinierte fahrzeuge," ATZelektronik, vol. 17, no. 5, pp. 40–45, May 2022. [Online]. Available: doi.org/10.1007/s35658-022-0755-7
- [29] S. Furst and M. Bechter, "AUTOSAR for Connected and Autonomous Vehicles: The AUTOSAR Adaptive Platform," in *IEEE/IFIP International Conference on Dependable Systems and Networks Workshop*, 2016.
- [30] T. Woopen et al., "UNICARagil Disruptive Modular Architectures for Agile, Automated Vehicle Concepts," Aachen Colloquium Automobile and Engine Technology, 2018.
- [31] M. Rumez, D. Grimm, R. Kriesten, and E. Sax, "An Overview of Automotive Service-Oriented Architectures and Implications for Security Countermeasures," *IEEE Access*, 2020.
- [32] M. Pöhnl, A. Tamisier, and T. Blass, "A middleware journey from microcontrollers to microprocessors," 2022.
- [33] Scalable Open Architecture for Embedded Edge (SOAFEE). [Online]. Available: soafee.io
- [34] L. Reiher, B. Lampe, T. Woopen, R. Van Kempen, T. Beemelmanns, and L. Eckstein, "Enabling Connectivity for Automated Mobility: A Novel MQTT-based Interface Evaluated in a 5G Case Study on Edge-Cloud Lidar Object Detection," in *International Conference on Electrical, Computer, Communications and Mechatronics Engineering* (ICECCME), 2022.
- [35] Kubernetes in Docker (KinD). [Online]. Available: kind.sigs.k8s.io