Wu BF, Zhong R, Wang Y *et al.* VPI: Vehicle programming interface for vehicle computing. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 39(1): 22-44 Jan. 2024. DOI: 10.1007/s11390-024-4035-2

VPI: Vehicle Programming Interface for Vehicle Computing

Bao-Fu Wu[1, †] (吴宝福), Ren Zhong[2] (仲　任), Yuxin Wang[3] (王昱心), Jian Wan[1, *] (万　健)
Ji-Lin Zhang[1, *] (张纪林), and Weisong Shi[3] (施巍松), *Fellow, IEEE*

[1] *School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou 310018, China*

[2] *Department of Computer Science, Wayne State University, Detroit, MI 48202, U.S.A.*

[3] *Department of Computer and Information Sciences, University of Delaware, Newark, DE 19716, U.S.A.*

E-mail: baofu.wu@hdu.edu.cn; zhongren@wayne.edu; yuxw@udel.edu; wanjian@hdu.edu.cn; jilin.zhang@hdu.edu.cn
weisong@udel.edu

Received December 14, 2023; accepted January 24, 2024.

Abstract The emergence of software-defined vehicles (SDVs), combined with autonomous driving technologies, has enabled a new era of vehicle computing (VC), where vehicles serve as a mobile computing platform. However, the interdisciplinary complexities of automotive systems and diverse technological requirements make developing applications for autonomous vehicles challenging. To simplify the development of applications running on SDVs, we propose a comprehensive suite of vehicle programming interfaces (VPIs). In this study, we rigorously explore the nuanced requirements for application development within the realm of VC, centering our analysis on the architectural intricacies of the Open Vehicular Data Analytics Platform (OpenVDAP). We then detail our creation of a comprehensive suite of standardized VPIs, spanning five critical categories: Hardware, Data, Computation, Service, and Management, to address these evolving programming requirements. To validate the design of VPIs, we conduct experiments using the indoor autonomous vehicle, Zebra, and develop the OpenVDAP prototype system. By comparing it with the industry-influential AUTOSAR interface, our VPIs demonstrate significant enhancements in programming efficiency, marking an important advancement in the field of SDV application development. We also show a case study and evaluate its performance. Our work highlights that VPIs significantly enhance the efficiency of developing applications on VC. They meet both current and future technological demands and propel the software-defined automotive industry toward a more interconnected and intelligent future.

Keywords software-defined vehicle (SDV), vehicle computing (VC), vehicle programming interface (VPI), autonomous system

1 Introduction

The progression of autonomous vehicle technology is being significantly accelerated by advancements in algorithms and computational capabilities. Consequently, an increasing number of these vehicles are undergoing road tests, heralding a transformation in conventional modes of transportation. Predictions by Boston Consulting Group indicate that the emergence of software-defined vehicles (SDVs)[1] will create over \$650 billion in value for the auto industry by 2030, making up 15% to 20% of automotive value[①]. In this evolving landscape, autonomous vehicles emerge as sophisticated mobile platforms, endowed

Regular Paper

Bao-Fu Wu, Jian Wan, and Ji-Lin Zhang were supported by the National Natural Science Foundation of China under Grant No. 62072146, the Key Research and Development Program of Zhejiang Province of China under Grant Nos. 2023C03194, 2021C03187, and 2023C01044, and the National Natural Science Foundation of China (Youth Fund) under Grant No. 62302133.

[†]This work was done when Bao-Fu Wu was a visiting scholar in the Connected and Autonomous Research Laboratory (CAR Lab).

[*]Corresponding Author (Jian Wan guided the research work and paper writing. Ji-Lin Zhang provided financial support and guided the paper writing.)

[①]Alex K, Nikolaus L, Brian C, Alex X, Markus H, Sophie H, Johannes W, Maria A, Sebastian B, Maya B D. Rewriting the rules of software-defined vehicles. 2023. https://www.bcg.com/publications/2023/rewriting-rules-of-software-defined-vehicles, Jan. 2024.

with extensive computational, storage, communication, and energy resources. This development has catalyzed the emergence of vehicle computing (VC) as a key technological trend[2, 3]. With autonomous vehicles catering to their inherent software development requirements, there is a parallel emergence of an expansive software ecosystem, leveraging the vehicles' vast resource pool. Consequently, the development of applications centered around VC is becoming a pivotal area of research and technological innovation.

In the era of VC, autonomous vehicles will become a mobile computation platform, a mobile communication platform, a mobile energy consumption, storage, delivery platform, a mobile sensing platform, and a mobile data generation and storage platform. As illustrated in Fig.1, while addressing their own needs, these autonomous vehicles will also provide resources for surrounding devices[4]. At that time, ubiquitous high-performance computing, sensing, power, and communicating capabilities will be realized.

Intel estimates that autonomous vehicles of the future will produce 4 TB of data every day②. To effectively analyze this data on mobile computing platforms, OpenVDAP (Open Vehicular Data Analytics Platform)[5] provides a roadmap for on-board system data analysis. OpenVDAP is a complete stack edge-based platform comprising an on-board computing/communication unit, a security and privacy-preserving vehicle operation system supported by isolation, an edge-aware application library, and an optimal workload offloading and scheduling strategy.

A crucial aspect of OpenVDAP is its programming interface, which includes vehicle programming interfaces (VPIs). These VPIs are specifically designed to bridge the gap between the autonomous vehicle's computational capabilities and the requirements of advanced VC applications. VPIs enable developers to more effectively create and deploy applications that leverage the full potential of vehicular edge

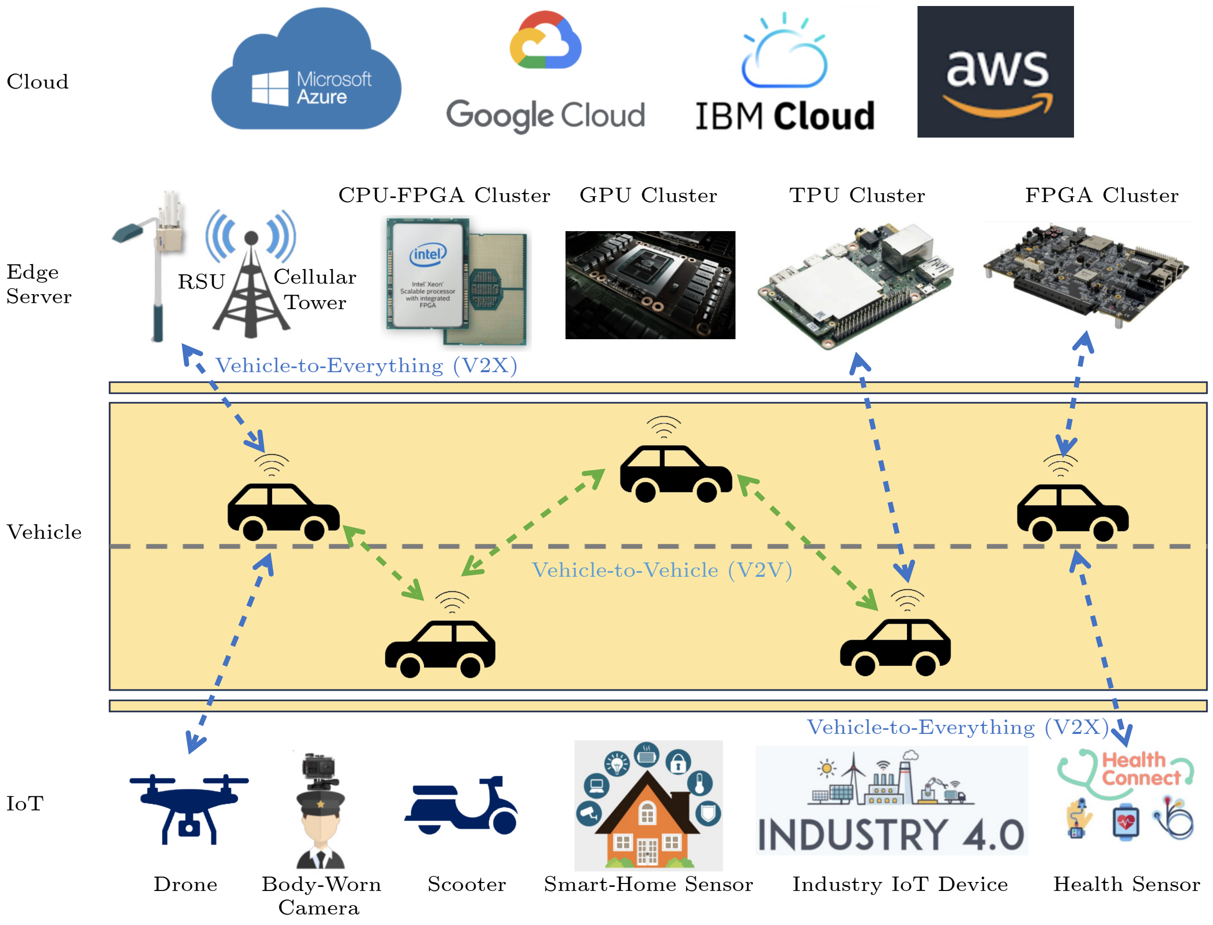


Fig.1. Paradigm of vehicle computing[2, 3].

②Yogesh M. 100 million lines of code, 4 TB data per day—Is that your next car? 2017. https://futuremonger.com/100-million-lines-of-code-4-tb-data-per-day-is-that-your-next-car-a2724e9bd3fa, Jan. 2024.

computing by providing a structured way to access and utilize the vehicle's resources.

Easily developing VC applications currently face high technical barriers, primarily stemming from four key challenges.

• First, the interdisciplinary nature of autonomous driving technology, which encompasses computer vision, machine learning, sensor fusion, control theory, and software engineering, contributes to the high technical threshold[6, 7]. This broad spectrum of disciplines sets the foundational challenge in the field, requiring a deep and integrated understanding across various areas of expertise.

• Second, the substantial variation in autonomous vehicles produced by different manufacturers[③] is another critical challenge. This diversity manifests in the types and numbers of sensors deployed, inconsistencies in data formats, and differences in the design of autonomous driving architectures. Such heterogeneity complicates the development of universal solutions that can be applied across various vehicle models.

• Third, the complexity of autonomous driving components presents a substantial challenge. Taking Autoware[④] as an example, a renowned open-source software for autonomous driving, it has developed hundreds of Robot Operating System (ROS)[8] nodes to support autonomous driving applications and illustrates the intricate nature of autonomous vehicle systems. Developing and integrating these complex components demands a high level of expertise.

• Finally, the need for data security protection and the focus on vehicle safety significantly impact the field[9, 10]. Currently, autonomous vehicle companies tend to focus on custom development for their specific vehicle models, with vehicle safety being a paramount consideration. However, this approach limits collaboration between companies and is a major impediment to the growth of an open-source community for VC.

Addressing these issues in this structured manner is essential to lower the barriers to autonomous driving software development and foster the growth of an open-source community for VC. To effectively promote this development, it is imperative to clearly define the requirements of VC. Based on these requirements, functional modules should be segmented, and a set of standardized programming interfaces specifically designed. Future autonomous driving software developers will then be able to focus on these designed interfaces, leveraging their respective areas of expertise to implement corresponding functionalities. This collaborative approach will not only facilitate the advancement of autonomous driving software development but also pave the way for a more integrated and innovative VC ecosystem.

It is imperative to clearly define the requirements to address the aforementioned challenges and promote the development of VC. Functional modules should be segmented on these requirements, and a set of standardized programming interfaces should be designed. The novelty of program interfaces lie in enabling future autonomous driving software developers to focus on these interfaces, leveraging their specific areas of expertise to implement corresponding functionalities. This collaborative method is not just about modularization, but about fostering an ecosystem where developers can contribute more efficiently and innovatively. The proposed approach contrasts with existing developments like AUTomotive Open System ARchitecture (AUTOSAR)[11], which has introduced Adaptive AUTOSAR to design a set of APIs (application programming interfaces). While these APIs offer a standardized method for automotive manufacturers and suppliers to develop and integrate vehicle software, making software development more efficient and interoperable, our approach extends this concept further. It aims to create a more comprehensive, open, and versatile framework for VC, surpassing the traditional boundaries of automotive software development. Similarly, the European Automobile Manufacturers Association (ACEA)[⑤] has been instrumental in promoting the standardization of vehicle data. This initiative has facilitated different manufacturers and service providers in exchanging and utilizing vehicle data more easily, reducing compatibility issues. However, our approach seeks to unify not just data standards, but also the broader spectrum of VC functionalities, including data processing, communication, and control systems. This unified API design, therefore, not only enables the transmission and parsing of data across different manufacturers but also paves the way for more intelligent and interconnected developments in the automotive industry. Our comprehensive solution is designed to fully meet the extensive and evolving requirements of VC, marking a significant advancement over existing systems.

[③]Goncharov I. Autonomous vehicle companies and their ML, 2013. https://wandb.ai/ivangoncharov/AVs-report/reports/Autonomous-Vehicle-Companies-And-Their-ML--VmlldzoyNTg1Mjc1, Jan. 2024.

[④]https://autoware.org/, Jan. 2024.

[⑤]https://www.acea.auto/, Jan. 2024.

To address the programming challenges in VC, centered around the design philosophy of OpenVDAP's APIs, we designed a set of vehicle programming interfaces (VPIs) for the development of VC applications. Proposed VPIs are designed to manage various aspects of vehicle hardware, data, computation, service, and system management for autonomous vehicles. These VPIs, being open-source, offer a universal solution for the open community. They have been meticulously developed, considering the future of connected vehicles as platforms for mobile computing, communication, energy consumption, storage and transfer, and mobile sensing. This suite of VPIs represents the first of its kind, specifically tailored to meet the application development needs of this new era, offering a holistic approach to developing applications for connected vehicles in the context of mobile computing and sensing.

Our main contributions can be summarized in three significant aspects.

• We have proposed the first standardized software programming development interfaces that comprehensively satisfy the requirements of VC application development, VPIs, which will facilitate the rapid development of VC software.

• We have developed the OpenVDAP prototype, a purpose-built framework to facilitate technological validation and support the implementation of these programming interfaces.

• We have validated the programming efficiency of VPIs in application development by conducting two sets of experiments, demonstrating their practicality and effectiveness in real-world scenarios involving VC platforms.

In this paper, we begin by reviewing related work in Section 2, then articulate the principles guiding our VPI design and provide a comprehensive explanation of the VPIs' design and considerations in Section 3. Section 4 introduces the OpenVDAP validation platform, designed to test and validate our VPI framework. Section 5 evaluates the efficiency performance of programming with proposed VPIs, combined with a case study and performance evaluation, and finally, in Section 6, we discuss our work. Section 7 summarizes our contributions and potential avenues for future research.

2 Related Work

In the era of vehicle computing (VC), connected vehicles emerge as formidable edge computing platforms, exhibiting proficiency in mobile computing, communication, energy storage and transfer, mobile sensing, and data storage. The analytical capabilities of VC extend to examining data streams originating from onboard sensors and surrounding connected devices, even during vehicle parking or charging periods. Confronted with myriad service needs within VC, researchers have delved into and deliberated upon resource allocation, task computing, and data scheduling[12–14]. Correspondingly, future connected vehicles should at least possess autonomous driving function modules with APIs for perception and localization, planning and decision, vehicle control, data services, communication, device, charging, user interface, monitoring, and safety.

Facing such demands, numerous automotive manufacturers, industry alliances, and large companies are striving toward this direction. Their research outcomes can partially meet the needs of connected vehicles in the VC era. The automotive industry's API design primarily focuses on autonomous driving applications and enhancing user experience. For instance, Ford Motor Company's AppLink technology[⑥], General Motors' Next Generation Infotainment System (NGI)[⑦], and Toyota Motor Corporation's introduction of the Mobile Service Platform (MSPF)[⑧] exemplify such developments. These technologies emphasize the importance of providing seamless connectivity between vehicles and mobile devices. Many associations and technological companies are also paying more attention to this part, like AUTOSAR, COVESA, Autoware, Baidu Apollo, NVIDIA Drive, SOAFEE, BlackBerry IVY, and ROS. These platforms vary in their API support, and the details are shown in Table 1.

AUTOSAR[15]. AUTOSAR is a widely recognized middleware solution for automotive software development, offering APIs for diagnostics, safety, network management, power management, and driver and service management. Despite its widespread adoption, AUTOSAR faces challenges in autonomous driving applications due to its complexity, high cost, and primary focus on in-vehicle communication. Developing

[⑥]https://developer.ford.com/infotainment/in-vehicle-apps, Jan. 2024.
[⑦]https://developer.gm.com/, Jan. 2024.
[⑧]https://toyotaconnected.co.jp/en/service/connectedplatform.html, Jan. 2024.

Table 1. API Comparison of Software Platforms and Automotive Companies for Vehicle Computing

Platform	Perception & Localization	Vehicle Control	Data Service	Communication	Charging API	User Interface	Monitoring API	Safety API
Ford Applink[9]						✓		
General Motors NGI[10]			✓	✓		✓		
Toyota MSPF[11]			✓					
AUTOSAR[15]		✓	✓	✓			✓	✓
COVESA[12]	-	-	-	-	-	-	-	-
Autoware[13]	✓	✓					✓	
Baidu Apollo[14]	✓	✓	✓	✓				
NVIDIA DRIVE[15]	✓	✓	✓					
SOAFEE[16]								✓
BlackBerry IVY[17]			✓	✓			✓	✓
ROS[16]	✓	✓		✓				
VPI	✓	✓	✓	✓	✓	✓	✓	✓

a custom operating system with AUTOSAR can increase the cost and complexity.

COVESA[12]. Connected Vehicle Systems Alliance (COVESA) is a community of automakers and suppliers collaborating on an open in-vehicle infotainment (IVI) and connectivity software platform. GENIVI[18] offers APIs for vehicle communication, data services, and hardware interfaces, catering to basic vehicle control and communication requirements. It focuses more on standardization and interoperability in IVI systems than advanced autonomous driving functionalities.

Autoware[13]. Developed by Tier IV in Japan, Autoware is an open-source platform based on the Robot Operating System (ROS). It offers extensive APIs for processing point cloud data, mapping, localization, perception, and control. While it is easy to deploy and maintain due to its open-source nature, further improvements are needed in Autoware's performance and stability.

Baidu Apollo[14]. Developed by Baidu, Apollo provides a comprehensive set of APIs for perception, localization, planning, and control. Its high level of autonomy enables the implementation of autonomous driving functions under various road conditions. Nevertheless, Apollo's steep learning curve and the need for high technical expertise may pose barriers to entry for some developers.

NVIDIA DRIVE[15]. NVIDIA DRIVE, developed by NVIDIA, is a comprehensive software platform for autonomous driving, encompassing everything from the vehicle to the data center. It includes hardware and software for AV development, such as NVIDIA DGX for training neural networks and DRIVE Sim for dataset generation and validation. NVIDIA DRIVE supports end-to-end development with rich APIs and tools, excelling in multi-sensor fusion. However, its high cost makes it more suitable for medium to large enterprises.

SOAFEE[16]. Arm's Scalable Open Architecture for Embedded Edge (SOAFEE) is designed to provide a cloud-native development environment that addresses the automotive industry's unique challenges and constraints. It offers standardized interfaces to avoid vendor lock-in and integrates container orchestration with automotive functional safety. However, the virtualized environment per ECU in SOAFEE may not align well with certain real-time SDV applications. The complexity and security concerns associated with its modular and scalable nature may pose integration

[9]Create utility here: In-vehicle apps. 2024. https://developer.ford.com/infotainment/in-vehicle-apps, Jan. 2024.
[10]https://developer.gm.com/in-vehicle-apps, Jan. 2024.
[11]https://toyotaconnected.co.jp/en/service/connectedplatform.html, Jan. 2024.
[12]https://covesa.global/about-covesa, Jan. 2024.
[13]https://autoware.org/, Jan. 2024.
[14]https://github.com/ApolloAuto/apollo, Jan. 2024.
[15]NVIDIA. NVIDIA DRIVE end-to-end solutions for autonomous vehicles. 2023. https://developer.nvidia.com/drive, Jan. 2024.
[16]SOAFEE Architecture. 2023. https://architecture.docs.soafee.io/en/latest/contents/introduction.html, Jan. 2024.
[17]https://www.blackberry.com/us/en/products/automotive/blackberry-ivy#features, Jan. 2024.
[18]https://github.com/genivi, Jan. 2024.

challenges, necessitating significant technical expertise and thorough validation for autonomous driving application development.

BlackBerry IVY[19]. Being a collaboration between BlackBerry and AWS, BlackBerry IVY is a scalable, cloud-connected software platform designed to enhance driver and passenger experiences in connected vehicles using the BlackBerry QNX and AWS technology. IVY provides scalable APIs and tools supporting various sensing devices and vehicle models. With its high security and stability, BlackBerry IVY is ideal for medium-to-large enterprises, though it requires enterprise authorization.

ROS[16]. The Robot Operating System (ROS) is a free, open-source software platform for robotic and autonomous systems development, supporting multiple programming languages and platforms. ROS offers an extensive suite of APIs and tools known for its ease of use and learning. Nonetheless, its performance and stability are areas that require further improvement.

In summary, although the methods mentioned above have unique features, they cannot meet the future development needs of VC. For this reason, we are the first to propose a comprehensive standard programming interface specifically designed to meet the requirements of VC. Simultaneously, in light of the development requirements for future VC applications, we have thoroughly considered the API design philosophy of the Open Vehicular Data Analytics Platform (OpenVDAP)[5]. We have meticulously analyzed and designed a set of programming development interfaces tailored to VC application development needs. This suite of VPIs represents the first attempt of its kind.

3 Vehicle Programming Interface Design

The design philosophy of VPI revolves around conceptualizing the vehicle as a platform for computation, storage, power management, and sensing. This concept is increasingly crucial in the realm of automotive technology. VPI is intended to harness modern vehicles' advanced computational capabilities and rich sensor resources. It aims to integrate vehicles' computational, storage, power management, and sensing capabilities, providing a unified programming interface. This interface supports the development of advanced applications in connected vehicles, such as autonomous driving, Vehicle-to-Everything (V2X) communications, and advanced infotainment systems.

To optimally design VPIs, it is imperative to comprehend the data analysis architecture of future Connected and Autonomous Vehicles (CAVs)[17], along with the VPI design considerations predicated on this framework. OpenVDAP[5] provides a roadmap for it, which is an edge computing based platform. It integrates onboard heterogeneous computing units, a specialized operating system for vehicles, and an edge-aware application library. This platform supports a dual-tier architecture that enables dynamic service assessment and optimal offloading decisions for timely processing. Most importantly, unlike the proprietary platforms, the OpenVDAP design offers an open and free edge-aware library that contains how to access and deploy edge computing based vehicle applications and various common used AI models, which will enable the researchers and developers in the community to deploy, test, and validate their applications in the real environment.

In the increasingly complex automotive industry, VPI adheres to the key design principles for optimal functionality:

- *Layered*: assigns responsibilities across multiple layers, ranging from hardware abstraction to the user interface;
- *Decoupled*: minimizes dependencies between layers to enhance flexibility and maintainability;
- *Standardized*: ensures compatibility and integration with uniform interfaces and protocols;
- *Open*: promotes extensive integration and innovation by supporting an open design for third-party contributions.

3.1 Overview of Vehicle Programming Interface

The VPIs of connected vehicles in the era of VC can be conceptualized into five key categories: Hardware, Data, Computation, Service, and Management, which is shown in Fig.2. This layered approach signifies a progressive transition from the fundamental hardware to user-facing services, ensuring the scalability and flexibility of the system.

- *Hardware*. Hardware VPIs form the foundation of the connected vehicle system. They serve to shield

[19]https://www.blackberry.com/us/en/products/automotive/blackberry-ivy#features, Jan. 2024.

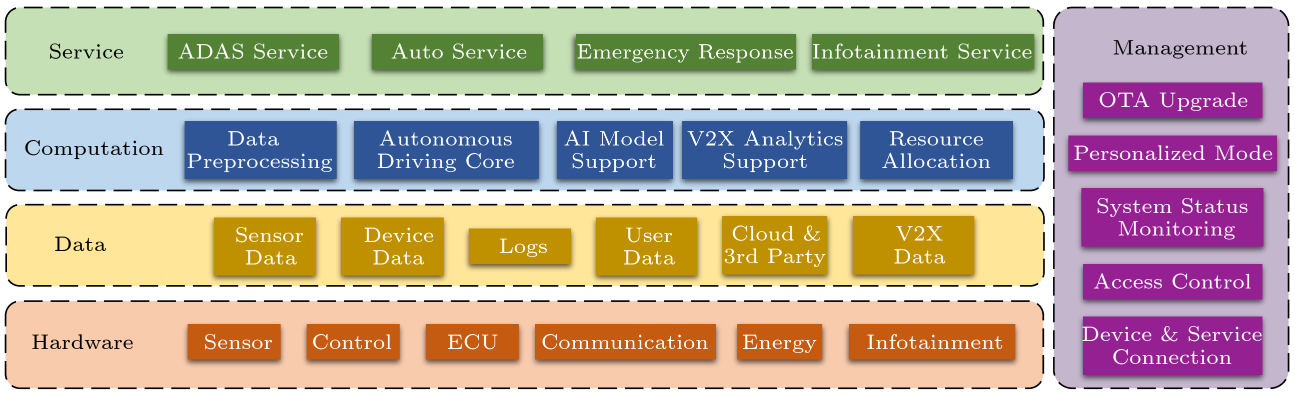


Fig.2. Structure of VPIs. VPIs are composed of five main categories: Hardware, Data, Computation, Service, and Management.

the hardware differences among autonomous vehicles from different manufacturers, providing a unified interface for direct access to sensors, communication systems, controllers, actuators, energy management, and intelligent cockpit hardware. This facilitates the rapid development and integration of advanced applications.

• *Data*. Data VPIs focus on acquiring and processing real-time and historical sensor data, which is crucial for the vehicle's dynamic adaptation and intelligent decision-making. They manage data from the vehicle itself, surrounding devices, other vehicles, road infrastructure, cloud, and driver data, ensuring a unified approach to data management.

• *Computation*. Computation VPIs are responsible for data processing and decision support. They support the computational capabilities of core autonomous driving modules like obstacle detection, end-to-end autonomous driving, and automatic detection. Computation VPIs also prioritize the coordinated allocation of computational resources from the vehicle, surrounding devices, road infrastructure, and cloud.

• *Service*. Service VPIs represent the functionally integrated module of VC for application developers, simplifying the invocation of high-level functionalities such as autonomous driving. Key services include Advanced Driver Assistance Systems (ADAS), autonomous driving, emergency response, and entertainment services.

• *Management*. Management VPIs act as a comprehensive control center for autonomous vehicles, integrating device and service connections, access control, system monitoring, and OTA (Over-the-Air technology) upgrade support. This ensures seamless, secure, and efficient management of the vehicle's overall performance and resources.

Through the collaborative efforts of these five categories of VPIs and the seamless flow of data, the connected vehicle system achieves efficient data processing, precise control decisions, and rich user interaction experiences, revolutionizing autonomous driving and intelligent transportation in the VC era.

3.2 Hardware VPIs

Due to the hardware variations among different vehicles, designing Hardware VPIs in VC is necessary to shield the underlying differences, supporting the development of advanced software. Centered around VC needs, the design of the Hardware VPIs should include the direct access and control of the vehicle's sensors, communication systems, controllers, various electronic control units (ECUs), and onboard entertainment devices. The specific Hardware VPIs are detailed in Appendix TableA1, which outlines the functionalities and interface designs of the VPI. Below, we provide a summarized explanation of these VPIs.

• *Sensors*. Sensors are crucial for gathering data from the vehicle's environment. VPIs such as `listSensors`, `configureSensor`, and `calibrateSensor` enable the enumeration, configuration, and calibration of various sensors. These functions are vital for ensuring the accuracy and reliability of the data collected by sensors, which include cameras, radars, and LiDARs, essential for functions like navigation, obstacle detection, and driver assistance.

• *Actuators*. Actuators play a key role in converting electronic signals into physical actions. VPIs like `controlActuator` and `configureActuator` manage the operation and configuration of actuators, which are responsible for actions such as steering, braking, and throttle control. These VPIs ensure precise and responsive actuation based on sensor inputs and control commands.

• *ECUs* (*Electronic Control Units*). ECUs are the brains behind the vehicle's electronic systems. With VPIs such as `listECUs`, `controlECU`, and `configureECU`, the vehicle can monitor and manage various ECUs, including those for engine control, transmission, and safety systems. Additionally, specialized VPIs like `controlLighting` manage specific functions like vehicle lighting, illustrating the versatility of ECU management.

- *Communication.* This category focuses on managing the vehicle's communication systems. VPIs like `listCommDevices`, `configureCommDevice`, and `toggleV2X` are essential for controlling various communication devices, including cellular, Wi-Fi, and V2X communication systems. These functions enable the vehicle to stay connected and communicate with external networks and devices, facilitating features like telematics and connected services.
- *Energy.* Managing the vehicle's energy systems, especially in electric vehicles, is critical. VPIs such as `controlCharging` and `controlPowerOutput` manage the battery charging process and power distribution to external systems. These functions are essential for optimizing battery life, ensuring energy efficiency, and even supporting vehicle-to-grid (V2G) capabilities.
- *Infotainment.* The infotainment system enhances the in-vehicle experience. VPIs like `configureDisplay` and `configureAudio` manage the settings of the vehicle's display and audio systems, ensuring an engaging and customizable entertainment experience for passengers. These functions cater to user preferences in media consumption, navigation, and connectivity.

These Hardware VPIs are unique and essential in the context of VC due to their focus on direct hardware control and management. They differ from existing interfaces by providing specialized control and access to vehicle hardware components, allowing for advanced software development tailored to the specific needs of autonomous vehicles. The benefits of these interfaces include enhanced flexibility, customization, and optimization of vehicle functionalities, which are crucial for the advancement of VC and the realization of advanced autonomous driving systems.

3.3 Data VPIs

In the vehicle system architecture, Data VPIs hold a critical position. They are principally responsible for gathering, managing, and preprocessing data from both the vehicle itself and external sources. This ensures efficient access and utilization of data across various system components. Data VPIs facilitate dynamic environmental adaptation and informed decision-making by offering unified management of data from the vehicle, surrounding devices, other vehicles, roadways, cloud sources, and the driver. The detailed design of Data VPIs is illustrated in Appendix TableA2. Data VPIs are categorized into following key areas based on their responsibilities.

- *Sensor Data.* Centered around acquiring and processing real-time and historical sensor data, these VPIs are crucial for dynamic environmental adaptation and informed decision-making. For instance, `getSensorData` captures immediate environmental data, while `getHistSensorData` allows for retrospective analysis of sensor readings. This category is vital for enhancing the vehicle's awareness and responsiveness to its surroundings, supporting functions like obstacle detection and navigation.
- *Device Data.* Addressing the increasing interconnectivity in vehicular ecosystems, these VPIs facilitate seamless data communication with external devices. Key functionalities include `getDeviceData` for ingesting data from devices such as smartphones and `storeDeviceData` for preserving such information. This category underscores the vehicle's role in the broader IoT landscape, enhancing user convenience and expanding the vehicle's operational capabilities beyond traditional boundaries.
- *Logs.* Focused on collecting and storing operational logs, these VPIs, such as `getOperationalLogs`, offer deep insights into the vehicle's performance and usage patterns. They are instrumental in predictive maintenance, troubleshooting, and long-term performance optimization, making them integral for maintaining vehicle health and efficiency.
- *User Data.* Tailoring the vehicle experience to individual preferences, VPIs in this category, like `getUserData`, cater to the personalization aspect of modern vehicles. They handle the storage and retrieval of user-specific settings and preferences, ensuring each journey is aligned with the user's comfort and convenience.
- *Infotainment Data.* Enhancing the in-cabin experience, VPIs such as `getMediaContent` manage diverse forms of entertainment content. They play a pivotal role in delivering a versatile and enjoyable in-vehicle infotainment experience, from streaming media to interactive navigation interfaces.
- *Cloud & 3rd Party Data.* These VPIs, exemplified by `syncDataFromCloud`, represent the vehicle's capability to integrate with cloud-based services and

third-party data sources. They ensure the vehicle remains at the forefront of technological integration, leveraging external computational power and expansive datasets for enhanced functionalities.

• *V2X Data.* Encompassing VPIs such as `getV2XData`, this category is essential for enabling interactive and cooperative functionalities with other vehicles, pedestrians, and road infrastructure. They foster a connected and cooperative road environment, enhancing safety, traffic management, and overall driving experience.

These Data VPIs are integral in the context of VC as they differ from existing interfaces by providing specialized data handling capabilities tailored to the unique requirements of autonomous vehicles and connected transportation systems. Their benefits encompass efficient data storage, data retrieval, and sharing, which are essential for enhancing the performance and functionality of VC systems. By providing these specialized interfaces, the architecture is optimized for VC, ensuring that data from various sources is effectively collected, processed, and shared, thus contributing to the advancement of autonomous and connected vehicles.

3.4 Computation VPIs

In the vehicular system, Computation VPIs act as the "brain", orchestrating advanced data processing and decision-making support. They enable essential autonomous functions, such as route planning and obstacle detection, and manage the allocation of computational resources across the vehicle, surrounding devices, and cloud services. These VPIs are crucial for integrating AI support into the vehicle's operational framework, enhancing its autonomous capabilities and environmental interaction. As shown in Appendix TableA3, the followings are the main Computation VPIs components.

• *Data Preprocessing.* This category lays the groundwork for all subsequent data-driven operations within the vehicle system. VPIs like `cleanData` and `formatData` are vital in refining raw data, removing inaccuracies, and transforming them into formats suitable for advanced analysis. These functions ensure data integrity and consistency, which are critical for accurate sensor data interpretation and reliable vehicle operations.

• *Autonomous Driving Core.* Central to the vehicle's self-driving capabilities, this category involves complex data fusion and processing. VPIs such as `earlyFusion`, `intermFusion`, and `lateFusion` deal with integrating various data streams, including camera, Radar, and LiDAR inputs at different stages for nuanced perception[18]. VPIs provide enhanced situational awareness like `getFusedBEVResult`[19] and `get360View`, which generate comprehensive visual representations of the vehicle's surroundings, crucial for safe autonomous navigation and obstacle avoidance.

• *AI Model Support.* Reflecting the vehicle's advanced intelligence, this category encompasses VPIs that facilitate sophisticated AI tasks. Functions such as `processAIInference` and `runAIModel` showcase the vehicle's prowess in handling complex AI algorithms, ranging from real-time image processing to predictive analytics. These VPIs empower the vehicle with capabilities like object recognition, behavior prediction, and even personalized user interaction, enhancing the overall autonomy and user experience.

• *V2X Analytics Support.* Emphasizing collaborative computation, this category includes VPIs like `reqCloudCompute` and `reqDeviceCompute`, which enable the vehicle to extend its computing capabilities beyond its physical confines. By utilizing external computational resources such as cloud services, nearby devices, or other vehicles, these VPIs allow for more extensive and complex data processing tasks. This collaborative approach enhances the vehicle's ability to make more informed decisions, adapt to dynamic environments, and offer enriched services like traffic management and environmental monitoring.

• *Resource Allocation.* This category focuses on the efficient management of the vehicle's onboard computational resources. VPIs such as `allocateResources` dynamically balance CPU, GPU, and memory usage, optimizing the performance for varying computational loads. This ensures that the vehicle's computing system operates efficiently, conserving energy while maintaining high performance, which is particularly critical in resource-intensive scenarios like high-definition mapping and real-time sensor data processing.

Computation VPIs are unique in the context of VC as they differ from existing interfaces by providing specialized computational support tailored to the complex demands of autonomous driving and connected vehicles. Their benefits encompass advanced data processing, enhanced decision-making, and efficient resource management, all of which are crucial for realizing the potential of VC and enabling advanced self-driving capabilities. By offering these specialized interfaces, the architecture is optimized to handle the specific computational requirements of autonomous vehicles, contributing to their safe and efficient operation.

3.5 Service VPIs

Various components and services are required in vehicle system applications to ensure effective interaction between the system and the end users. Service VPIs encompass the user interface design and a range of advanced applications and services. By offering these components and services, Service ensures that vehicle systems can meet the diverse needs of modern driving and provide users with a highly personalized driving experience. Application developers can utilize these services for secondary development without designing underlying implementations from scratch, thereby enhancing development efficiency. As shown in Appendix TableA4, Service VPIs should include the following aspects.

• *Advanced Driver Assistance Systems* (*ADAS*) *Service.* This category is pivotal in enhancing road safety and driver assistance capabilities. The ADAS Service VPIs, including `startADAS`, `stopADAS`, and `configADAS`, are designed to activate, manage, and customize various driver-assistance functionalities like automatic braking, lane keeping, and adaptive cruise control. Each VPI in this category is tailored to respond to real-time driving conditions, ensuring heightened safety and situational awareness.

• *Auto Service.* Central to the autonomous driving experience, this category encompasses VPIs crucial for managing the vehicle's self-driving features. VPIs such as `startAutoMode` and `configAutoMode` facilitate the seamless transition between the manual and autonomous driving modes. With the integration of V2X communication in `startAutoModewithV2X`, these services empower vehicles to interact intelligently with their surroundings, enhancing navigation, traffic management, and overall driving efficiency.

• *Emergency Response.* This category addresses urgent safety and emergency scenarios, underscoring the vehicle's responsiveness in critical situations. VPIs like `initEmergencyCall` and `sendEmergencyAlert` are designed to ensure rapid and effective communication during emergencies, automatically contacting emergency services and alerting predefined contacts, thereby offering an essential lifeline in times of need.

• *Infotainment Service.* Tailored to enrich the in-vehicle entertainment experience, this category includes VPIs that manage various forms of media content. Functions such as `playMedia` and `pauseMedia` reflect the vehicle's role as an entertainment hub, providing passengers with access to a wide range of multimedia content and interactive infotainment options, thereby transforming the vehicle into a space of relaxation and enjoyment.

Service VPIs play a crucial role in VC by offering interfaces and services that cater to the unique requirements of modern driving. These interfaces differ from existing ones by providing advanced driver assistance, autonomous driving capabilities, and personalized services, which are essential for enhancing the driving experience and safety. By incorporating these specialized interfaces, the vehicle system can offer features that go beyond traditional vehicle functionalities, improving the overall driving experience and aligning with the expectations of modern users.

3.6 Management VPIs

Management VPIs play a crucial role in overseeing and enhancing the overall functionality of the connected vehicle system. As detailed in Appendix TableA5, management VPIs are responsible for integrating device and service connections, access control, system monitoring, and OTA upgrades, thereby ensuring seamless, secure, and efficient management of the vehicle's overall performance and resources. By providing precise control and monitoring capabilities, these VPIs ensure the reliability and safety of the vehicle system in all aspects, offering users a highly dependable and satisfying driving experience.

• *Device & Service Connection.* This category is fundamental to the vehicle's ability to integrate and interact with external devices and services. Critical VPIs such as `pairWithDevice` and `connectCloud` enable seamless conline-height-add:0.4ptnectivity and authentication with a variety of devices and cloud services. These functions are essential for maintaining a connected ecosystem, allowing the vehicle to leverage external computing power, access a broader range of services, and enhance the in-vehicle experience.

• *Access Control.* Centered around the security of the vehicle's systems, this category incorporates sophisticated VPIs for robust user authentication and data protection. Key functions like `authenticate-User` and `setAccessControl` ensure that the access to the vehicle's systems and data is securely regulated. These VPIs are critical in safeguarding against unauthorized access and potential security breaches, thereby maintaining the integrity and confidentiality of sensitive information.

- *System Status Monitoring.* Focused on continuous monitoring and maintenance of the vehicle's health, this category includes VPIs that provide real-time insights into the status of various hardware components and resource usage. VPIs like `monitorHWStatus` and `monitorCompResourceUse` are instrumental in preemptive maintenance and resource optimization, ensuring the vehicle operates at peak efficiency and reliability.
- *Personalized Mode.* Reflecting the increasing demand for personalized experiences in vehicles, this category includes VPIs that enable users to tailor the vehicle's settings to their preferences. Functions such as `startPersonalMode` and `configPersonalMode` allow for customization of user profiles and modes, enhancing comfort and convenience for each user. This personalization extends from driving preferences to infotainment settings, offering a bespoke user experience.
- *OTA Upgrade.* Ensuring the vehicle's software remains up-to-date and secure, this category comprises VPIs dedicated to the management of over-the-air software updates. VPIs such as `scheduleOTAUpdate` and `verifyOTAUpdate` streamline the update process, from scheduling and downloading to installation and verification. This continuous updating process is crucial for enhancing features, fixing bugs, and improving the vehicle's overall security.

Management VPIs are essential components of VC as they differ from existing interfaces by providing centralized control over access, monitoring system status, and managing OTA updates. They ensure that every aspect of vehicle management, from device connectivity to system monitoring and user personalization, is executed with precision and user-centric focus, thereby playing a pivotal role in the evolution of smart and connected vehicles.

4 VPI Implementation

This section introduces the experimental scenario design for deploying VPI to conduct subsequent experimental evaluations. It consists of two main parts: the hardware deployment, which involves the indoor autonomous driving vehicle Zebra, and the software deployment, which includes the implementation of the VPI-driven system OpenVDAP.

4.1 Hardware: Zebra

In this study, we used the Zebra hardware platform which is implemented to emulate real autonomous vehicles to evaluate the performance of VPIs (as shown in Fig.3). A general autonomous driving system includes the computing unit, perception sensors, the drive-by-wire (DBW) system, and the battery management system (BMS). On Zebra, the computing unit is the NVIDIA Jetson AGX Xavier Developer Kit. Jetson AGX Xavier[20] is a compact, high-performance computing device designed for autonomous machines, offering up to 32 TOPS of AI performance with its 512-core NVIDIA Volta GPU, 64 Tensor cores, and an 8-core ARM v8.2 64-bit CPU. Besides, we deployed two sensors on Zebra. The first is the Intel RealSense Depth Camera D435i, which features an RGB sensor, providing a maximum RGB resolution of 1 920 × 1 080. The second is Velodyne VLP-16, a compact and high-value 3D LiDAR sensor offering a 100 m range and 16-channel high-definition environmental mapping with an accuracy of ±3 cm. The chassis of Zebra is the hunter robot of AgileX, which integrates an Ackermann control based DBW and a BMS for reading energy-related information.

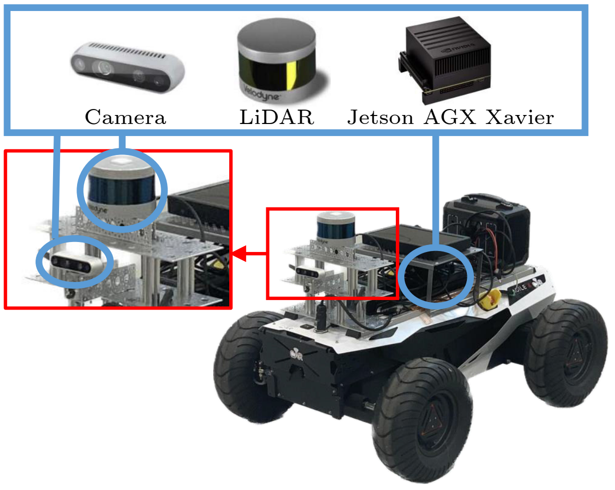


Fig.3. Zebra: general usage indoor robot vehicle. The sensor camera is realsense D435i and LiDAR is Velodyne VLP-16. The computation unit is NVIDIA Jetson AGX Xavier.

[20]NVIDIA. NVIDIA Jetson Xavier: A breakthrough in embedded applications. 2023. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-series/, Jan. 2024.

4.2 Software: OpenVDAP

In this subsection, we developed a VPI verification software prototype platform[21], OpenVDAP[5], which is designed to optimize vehicle applications through a multi-layered architecture approach. Fig.4 details the architecture and the components involved.

The framework of the VPI supporting system, designed for connected autonomous vehicles (CAVs), centers around the OpenVDAP on CAVs, efficiently structured into distinct containers for specific functions: a server container as the central hub, an ROS2 container for managing sensor data, an ROS1 container for vehicle control, and a database container for storing essential configurations. Key components include VPIs for vehicle function access, hardware control for hardware interaction, data management for integrity and storage, algorithm processing for real-time data computation, and application deployment for seamless integration.

In terms of implementation, OpenVDAP utilizes C++ and Python for effective integration with the ROS1 and ROS2 systems, employing Docker containers for modularity. Data flows through the system via sensor nodes in ROS2 for processing, algorithm nodes for decision-making, and control nodes in ROS1 for executing vehicle commands. The system's configuration and management are streamlined through the AD manager, which provides a comprehensive interface for system updates and maintenance.

OpenVDAP is a prototype platform and still needs to complete the development of all VPIs. The first version includes libraries for data-related VPIs (`getCameraData`, `getLiDARdata`, `getHistCameraData`, and `getHistLiDARdata`), vehicle computation related VPIs (`controlVehicle` and `runAIModel`), services-related VPIs (`startADAS` and `stopADAS`), and access control related VPIs (`checkAccess` and `validateToken`). We have developed applications based on VPI, such as lane-keeping, remote control, and remote lane inspection. Subsequently, we will conduct performance evaluations for some of these developed applications.

5 Evaluation

This evaluation section systematically examines the capabilities and performance of the vehicle programming interfaces (VPIs) in real-world autonomous driving applications.

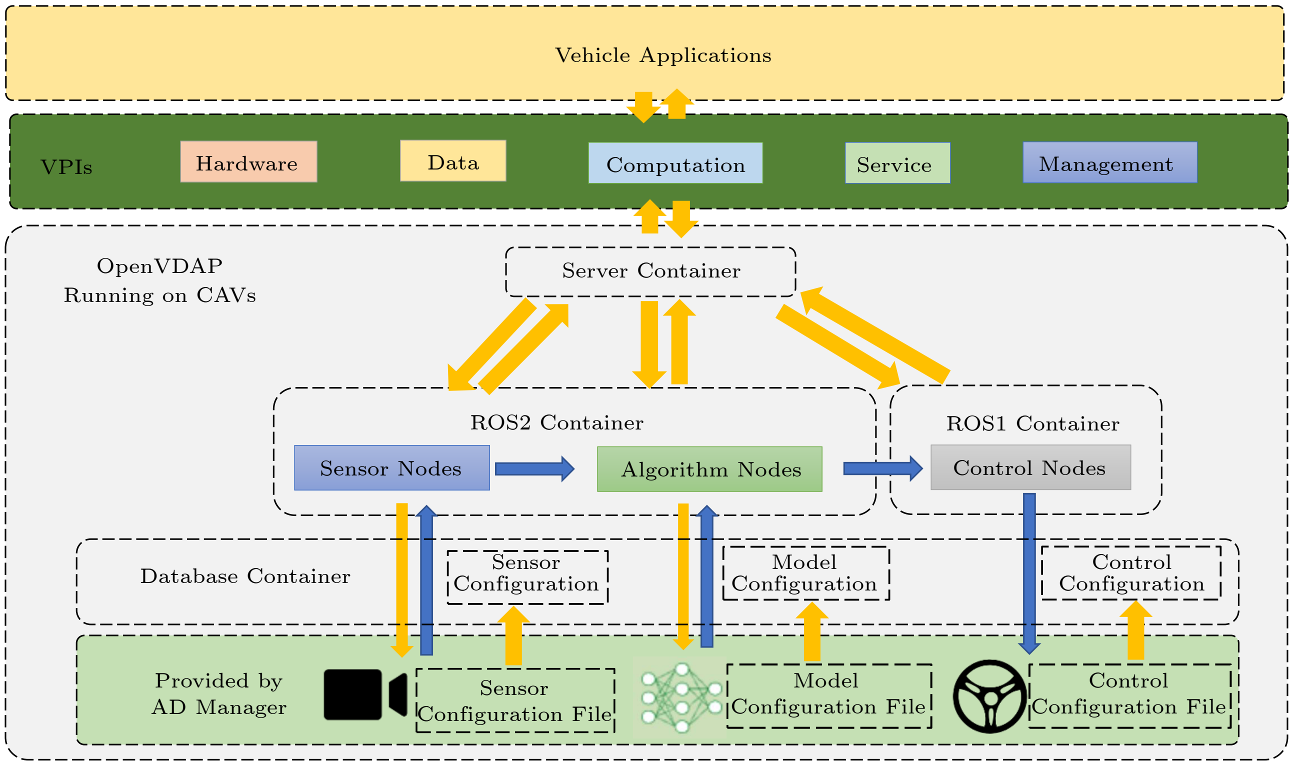


Fig.4. Framework of VPI supporting system.

[21]https://github.com/thecarlab/vpi, Jan. 2024.

5.1 Simpler Coding: VPI vs AUTOSAR

Lane keeping, a critical feature in ADAS for autonomous driving, particularly in high-speed scenarios, has been implemented using VPIs. The listing 1 code segment demonstrates the VPI approach.

Listing 1. Lane Keeping with VPI

```python
import vpi

modelID = "E2E_Lane_Keeping"

# Get front camera data
front_camera_data = vpi.getCameraData("front")

# Run AI model for lane keeping
ai_model_output = vpi.runAIModel(modelID, front_camera_data, params = {})

# Control vehicle using the output twist
vehicle_control_status = vpi.controlVehicle(ai_model_output["twist"])

return vehicle_control_status
```

Comparatively, we provide a foundational framework demonstrating how AUTOSAR API concepts can be utilized in C++. It is important to note that this is a conceptual example. Listing 2 code segment is a rudimentary example illustrating the definition of the lane keeping service interface and a basic service implementation.

When comparing VPI with AUTOSAR API for implementing lane-keeping functionality, using VPI, the task can be accomplished with only 14 lines of code, while AUTOSAR requires at least 35 lines. This difference is not merely due to language syntax but stems from the need in AUTOSAR to implement the service interface for functionality. In contrast, VPI focuses on interface functionalities and the basic workflow logic without delving into the underlying implementation.

The primary advantage of VPI lies in its simplicity and ease of use. Written in Python, VPI offers an intuitive and user-friendly interface, facilitating rapid development and prototyping. This approach suits scenarios that demand quick iterations and simplified system integration. In contrast, while AUTOSAR API provides more advanced customization capabilities and safety features suitable for complex and safety-critical applications, it comes with a steeper learning curve and greater complexity. Therefore, for

Listing 2. Lane Keeping with AutoSAR

```cpp
#include <ara/com/com.h>
#include <ara/core/future.h>
#include <iostream>

// Service interface for lane keeping
namespace ara::com::sample {
  class LaneKeepingServiceInterface {
  public:
    virtual ~LaneKeepingServiceInterface() = default;

    // Method to activate lane keeping
    virtual ara::core::Future<void> KeepLane() = 0;
  };
}
// Service implementation
class LaneKeepingServiceImpl : public ara::com::sample::LaneKeepingServiceInterface {
public:
  // Lane keeping logic implementation
  ara::core::Future <void> KeepLane() override {
    std::cout << "Keeping the lane..." << std::endl;
    // Placeholder for actual lane keeping logic
    return ara::core::Promise<void>().get_future();
  }
};

int main() {
  // Service instance creation
  LaneKeepingServiceImpl laneKeepingService;

  // Activating lane keeping service
  auto future = laneKeepingService.KeepLane();
  future.wait(); // Waiting for service execution
  return 0;
}
```

projects prioritizing development efficiency and a streamlined programming experience, VPI stands out as a more efficient and accessible option.

5.2 Case Study: Remote Control and Latency Evaluation

This case will demonstrate the remote transmission and remote control. The L4-level autonomous vehicles currently in trial operation cannot still cope with all scenarios, and manual takeover is necessary. It is an economical and effective option to centralize the driver to drive the vehicles that need to be taken over remotely. We use the OpenVDAP platform to implement the remote control demo. The remote control center application is deployed in Intel Fog Reference[22], which acts as a center station.

Fig.5 outlines the workflow within a remote control application. Initially, the remote control application in the center station submits a query to the access manager to obtain visual data from the vehicle's front camera (`getCameraData(vehicleID, 'front')`) (step 1). Upon gaining authorization (step 2), the `web_stream_node` is launched, which, in turn, activates the `front_camera_node` responsible for streaming the `front_camera_msg` back to the remote application, as depicted in step 5. Concurrently, the application requests control access (`controlVehicle (vehicleID, 'twist')`) shown in step 6. Once allowed (step 7), the `control_service_node` comes online to manage the `steer_control_msg`, channeling the control signals to the `control_node`. This node then communicates the commands through the CAN bus to execute the vehicle's control actions, completing the process at step 8.

Communication latency is a pivotal metric for the operation of indoor remote-controlled vehicles, mainly when it involves transmitting high-definition video data. The experiment demonstrates that when using Wi-Fi communication, our system can maintain an end-to-end latency with a median value of 1.54 seconds for commands to retrieve front camera data. This latency persists across the transmission of a high-definition video stream at a resolution of $1\,920 \times 1\,080$ and a frame rate of 30 Hz from the vehicle to the backend.

The cumulative distribution function (CDF) depicted in Fig.6 indicates that 90% of the commands are executed with a latency of up to 1.77 seconds, establishing a significant reliability benchmark for the system. This latency profile is considered acceptable within the context of indoor environments, where the

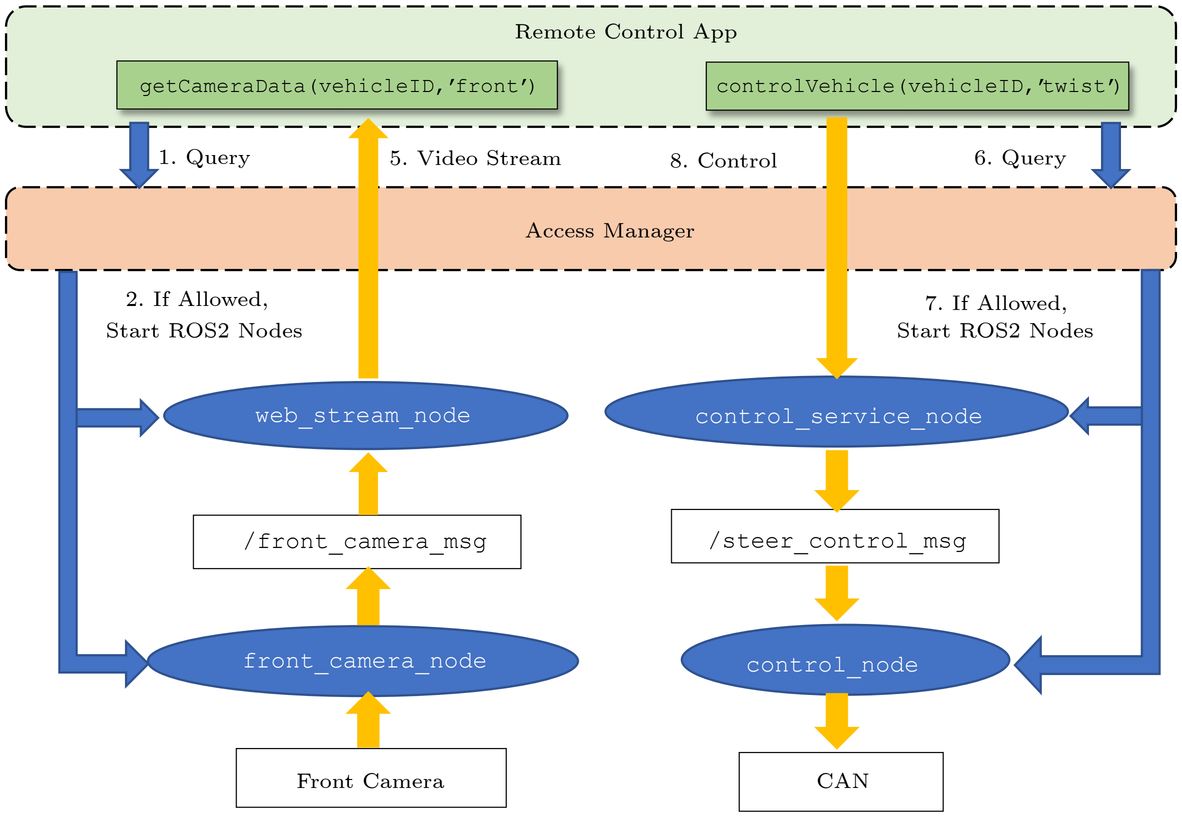


Fig.5. Workflow of remote control.

[22]Intel. Intel's Fog Reference Design overview. 2018. https://www.reflexces.com/wp-content/uploads/2018/11/fog-reference-design-overview-guide.pdf, Jan. 2024.

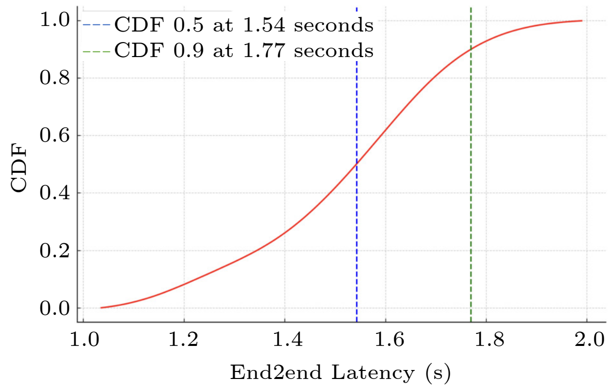


Fig.6. CDF of remote control end2end latency.

need for real-time control is not so critical as in outdoor or high-speed scenarios[20]. It enables a balance between high-quality video transmission and the responsiveness of the control system.

In conclusion, while the system exhibits higher latency than might be desired for real-time applications, it remains within a tolerable range for its intended use case. The reliable transmission of high-definition video is achieved, suggesting that the current setup is suited for applications where a slight delay is permissible. This reliability and performance balance underscores the system's potential for indoor remote-controlled operations where high-resolution visual feedback is required, and a marginal delay in lab environment is operationally acceptable[20].

6 Discussion

In this paper, our primary focus has been on designing and developing programming interfaces for vehicle computing (VC). We have presented a comprehensive framework of vehicle programming interfaces (VPIs) that cover various aspects of connected vehicle systems, from hardware management to service interactions. Our work has aimed to provide a structured and standardized approach to enable efficient application development for autonomous vehicles.

It is important to note that while performance optimization is a critical aspect of VC, it was not the primary objective of this paper. Our emphasis has been on defining clear and standardized interfaces to facilitate seamless communication and interaction between different components of the vehicle system. These interfaces aim to address the challenges associated with the diversity of autonomous vehicles and the need for collaborative development.

Performance improvement remains an important consideration in VC, and we acknowledge that there is room for enhancing the performance aspects of the proposed interfaces. This includes optimizing computational tasks, reducing latency, and ensuring efficient resource utilization. However, we view performance enhancement as part of future work, and we believe that the foundation provided by our standardized programming interfaces will enable researchers and developers to build upon it and further improve the performance of autonomous vehicle systems.

After all, this paper has laid the groundwork for a structured approach to VC through the development of programming interfaces. While performance optimization is a critical aspect of connected vehicle systems, our main contribution lies in providing a clear and standardized framework for application development. We look forward to future research endeavors aimed at enhancing the performance aspects of VC while building upon the foundations established in this work.

7 Conclusions

In response to the urgent need for rapid application development centered around vehicle computing (VC), we delineated and proposed a comprehensive set of standardized vehicle programming interfaces (VPIs) with five main sets: Hardware, Data, Computation, Service, and Management. We developed an OpenVDAP prototype, in which we experimentally validated the efficiency of programming in VC application development using the proposed VPIs. Additionally, through a remote control example, we demonstrated the underlying workflow in the OpenVDAP platform after invoking VPI and analyzed the end-to-end latency characteristics of this example. Our work represents an essential contribution to the development of VC applications and highlights the importance of interdisciplinary collaboration. In the future, we plan to further enrich and refine the design of VPIs according to emerging requirements, expand the functionalities of OpenVDAP, and optimize its performance.

Conflict of Interest Weisong Shi is an editor for Journal of Computer Science and Technology and was not involved in the editorial review of this article. All authors declare that there are no other competing interests.

References

[1] Liu Z W, Zhang W, Zhao F Q. Impact, challenges and prospect of software-defined vehicles. *Automotive Innovation*, 2022, 5(2): 180–194. DOI: 10.1007/s42154-022-00179-z.

[2] Lu S D, Shi W S. Vehicle as a mobile computing platform: Opportunities and challenges. *IEEE Network*, 2023. DOI: 10.1109/MNET.2023.3319454.

[3] Lu S D, Shi W S. The emergence of vehicle computing. *IEEE Internet Computing*, 2021, 25(3): 18–22. DOI: 10.1109/MIC.2021.3066076.

[4] Dong Z, Shi W S. Vehicle computing. *IEEE Internet Computing*, 2023, 27(5): 5–6. DOI: 10.1109/MIC.2023.3310367.

[5] Zhang Q Y, Wang Y F, Liu L K, Wu X P, Shi W S, Zhong H. OpenVDAP: An open vehicular data analytics platform for CAVs. In *Proc. the 38th IEEE International Conference on Distributed Computing Systems*, Jul. 2018, pp.1310–1320. DOI: 10.1109/ICDCS.2018.00131.

[6] Liu L K, Lu S D, Zhong R, Wu B F, Yao Y T, Zhang Q Y, Shi W S. Computing systems for autonomous driving: State of the art and challenges. *IEEE Internet of Things Journal*, 2021, 8(8): 6469–6486. DOI: 10.1109/JIOT.2020.3043716.

[7] Padmaja B, Moorthy C V K N S N, Venkateswarulu N, Bala M M. Exploration of issues, challenges and latest developments in autonomous cars. *Journal of Big Data*, 2023, 10(1): Article No. 61. DOI: 10.1186/s40537-023-00701-y.

[8] Macenski S, Foote T, Gerkey B, Lalancette C, Woodall W. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 2022, 7(66): eabm6074. DOI: 10.1126/scirobotics.abm6074.

[9] Pham M, Xiong K Q. A survey on security attacks and defense techniques for connected and autonomous vehicles. *Computers & Security*, 2021, 109: 102269. DOI: 10.1016/j.cose.2021.102269.

[10] Sun X Q, Yu F R, Zhang P. A survey on cyber-security of connected and autonomous vehicles (CAVs). *IEEE Trans. Intelligent Transportation Systems*, 2022, 23(7): 6240–6259. DOI: 10.1109/TITS.2021.3085297.

[11] Fürst S, Bechter M. AUTOSAR for connected and autonomous vehicles: The AUTOSAR adaptive platform. In *Proc. the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop*, Jul. 2016, pp.215–217. DOI: 10.1109/DSN-W.2016.24.

[12] Liu L, Zhao M, Yu M, Jan M A, Lan D P, Taherkordi A. Mobility-aware multi-hop task offloading for autonomous driving in vehicular edge computing and networks. *IEEE Trans. Intelligent Transportation Systems*, 2023, 24(2): 2169–2182. DOI: 10.1109/TITS.2022.3142566.

[13] Luo Q Y, Li C L, Luan T H, Shi W S. Collaborative data scheduling for vehicular edge computing via deep reinforcement learning. *IEEE Internet of Things Journal*, 2020, 7(10): 9637–9650. DOI: 10.1109/JIOT.2020.2983660.

[14] Liu L, Feng J, Mu X Y, Pei Q Q, Lan D P, Xiao M. Asynchronous deep reinforcement learning for collaborative task computing and on-demand resource allocation in vehicular edge computing. *IEEE Trans. Intelligent Transportation Systems*, 2023, 24(12): 15513–15526. DOI: 10.1109/TITS.2023.3249745.

[15] Martínez-Fernández S, Ayala C P, Franch X, Nakagawa E Y. A survey on the benefits and drawbacks of AUTOSAR. In *Proc. the 1st International Workshop on Automotive Software Architecture*, May 2015, pp.19–26. DOI: 10.1145/2752489.2752493.

[16] Quigley M, Conley K, Gerkey B, Faust J, Foote T, Leibs J, Leibs J, Wheeler R, Ng A. ROS: An open-source robot operating system. In *Proc. the 2009 ICRA Workshop on Open Source Software*, Jan. 2009.

[17] Rana M M, Hossain K. Connected and autonomous vehicles and infrastructures: A literature review. *International Journal of Pavement Research and Technology*, 2023, 16(2): 264–284. DOI: 10.1007/s42947-021-00130-1.

[18] Tang Q, Liang J, Zhu F Q. A comparative review on multi-modal sensors fusion based on deep learning. *Signal Processing*, 2023, 213: 109165. DOI: 10.1016/j.sigpro.2023.109165.

[19] Chang C, Zhang J W, Zhang K P, Zhong W Q, Peng X Y, Li S, Li L. BEV-V2X: Cooperative birds-eye-view fusion and grid occupancy prediction via V2X-based data sharing. *IEEE Trans. Intelligent Vehicles*, 2023, 8(11): 4498–4514. DOI: 10.1109/TIV.2023.3293954.

[20] Liu L K, Wu B F, Shi W S. A comparison of communication mechanisms in vehicular edge computing. In *Proc. the 3rd USENIX Workshop on Hot Topics in Edge Computing*, Jan. 2020.

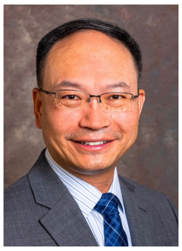
Bao-Fu Wu is pursuing his Ph.D. degree in computer science at the School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou. He visited Professor Weisong Shi's CAR Lab from 2020 to 2023, where he engaged in research on cooperative vehicle infrastructure systems (CVIS). He specializes in edge computing, parallel computing, and CVIS. His work is particularly centered on advancing the technology behind CVIS, aiming to improve vehicular communication systems for a smarter and more efficient transportation infrastructure.

Ren Zhong has been pursuing his Ph.D. degree in computer science at Wayne State University, Detroit, under the supervision of Dr. Weisong Shi since 2019. His research focuses on autonomous driving, especially high-definition map building and maintenance. He received his M.S. degree in software engineering from University of Science and Technology of China, Suzhou, in 2016, and his B.S. degree in electrical engineering from Chongqing University, Chongqing, in 2013.

Yuxin Wang is a first-year Ph.D. student at the University of Delaware in The CAR Lab (Connected and Autonomous Research Laboratory), under the guidance of Professor Weisong Shi. She earned her Master's degree in electrical engineering from the University of Pennsylvania, Philadelphia. Her academic journey has primarily focused on embedded intelligence, particularly on microcontrollers and machine learning algorithms. As she transitions into the Ph.D., her research delves into the realms of vehicle computing systems to autonomous vehicles.

Jian Wan received his Ph.D. degree in computer application technology from Zhejiang University, Hangzhou, in 1989. His research interests include grid computing, service computing, and cloud computing. He is currently a professor in software engineering with the School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou.

Ji-Lin Zhang received his Ph.D. degree in computer application technology from University of Science Technology Beijing, Beijing, in 2009. His research interests include high-performance computing and cloud computing. He is currently a professor in School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou.


Weisong Shi is a professor and the Chair of the Department of Computer and Information Sciences, the University of Delaware (UD), Newark, where he leads the Connected and Autonomous Research Laboratory (CAR Lab). Before he joined UD, he was a professor at Wayne State University (2002–2022) and served in multiple administrative roles, including an associate dean for Research and Graduate Studies at the College of Engineering and the interim chair of the Computer Science Department. He is an internationally renowned expert in edge computing, autonomous driving, and connected health. His pioneer article titled "Edge Computing: Vision and Challenges" has been cited more than 7 000 times.

Appendix

Table A1. Hardware VPIs Specifications

VPI Type	VPI Method	Parameter	Return Value	Functionality
Sensor	`listSensors(vehicleId)`	vehicleId: String (default: self)	List	Enumerates vehicle sensors
	`configureSensor(vehicleId, sensorId, config)`	vehicleId: String (default: self), sensorId: String, config: Object	Boolean	Configures a sensor
	`calibrateSensor(vehicleId, sensorId)`	vehicleId: String (default: self), sensorId: String	Boolean	Calibrates a sensor
Actuator	`controlActuator(vehicleId, actuatorId, command)`	vehicleId: String (default: self), actuatorId: String, command: Object	Boolean	Controls an actuator
	`configureActuator(vehicleId, actuatorId, config)`	vehicleId: String (default: self), actuatorId: String, config: Object	Boolean	Configures an actuator
ECU	`listECUs(vehicleId)`	vehicleId: String (default: self)	List	Lists vehicle ECUs
	`controlECU(vehicleId, ecuId, command)`	vehicleId: String (default: self), ecuId: String, command: Object	Boolean	Sends command to an ECU
	`configureECU(vehicleId, ecuId, config)`	vehicleId: String (default: self), ecuId: String, config: Object	Boolean	Configures an ECU
	`controlLighting(vehicleId, ecuId, lightingSettings)`	vehicleId: String (default: self), ecuId: String, lightingSettings: Object	Boolean	Controls the lighting ECU with specified settings. `controlLighting` is an example of a VPI for managing common devices, similar to VPIs for controlling air conditioning and windows
Communication	`listCommDevices(vehicleId)`	vehicleId: String (default: self)	List	Lists communication devices
	`configureCommDevice(vehicleId, deviceId, config)`	vehicleId: String (default: self), deviceId: String, config: Object	Boolean	Configures a communication device
	`toggleV2X(vehicleId, enable)`	vehicleId: String (default: self), enable: Boolean	Boolean	Toggles V2X communication
Energy	`controlCharging(vehicleId, action)`	vehicleId: String (default: self), action: String	Boolean	Controls battery charging process
	`controlPowerOutput(vehicleId, action, params)`	vehicleId: String (default: self), action: String, params: Object	Boolean	Controls power output to external systems
Infotainment	`configureDisplay(vehicleId, settings)`	vehicleId: String (default: self), settings: Object	Boolean	Configures the display settings
	`configureAudio(vehicleId, settings)`	vehicleId: String (default: self), settings: Object	Boolean	Configures the audio system

Table A2. Data VPIs Specifications

VPI Type	VPI Method	Parameter	Return Value	Functionality
Sensor Data	`getSensorData(vehicleId, sensorType, sensorId, params)`	vehicleId: String (default: self), sensorType: String, sensorId: String, params: Object	Sensor data	Retrieves real-time data from a specified sensor based on type and ID
	`getCameraData(vehicleId, cameraID, params)`	vehicleId: String (default: self), cameraID: String, params: Object	Camera data	Retrieves real-time data from a specified camera, similar to `getLiDARData`, `getRadarData`, and `getGPSData`
	`storeSensorData(vehicleId, sensorType, sensorId, data)`	vehicleId: String (default: self), sensorType: String, sensorId: String, data: Object	Boolean	Stores real-time data from a specified sensor based on type and ID
	`storeCameraData(vehicleId, cameraID, data)`	vehicleId: String (default: self), cameraID: String, data: Object	Boolean	Stores real-time data from a specified camera, similar to `storeLiDARData`, `storeRadarData`, and `storeGPSData`

(to be continued)

Table A2. Data VPIs Specifications (Continued)

VPI Type	VPI Method	Parameter	Return Value	Functionality
	`getHistSensorData(vehicleId, sensorType, timeRange, params)`	vehicleId: String (default: self), sensorType: String, timeRange: TimeRange, params: Object	Sensor data	Retrieves historical data from a specified sensor within a defined time range
	`getHistCameraData(vehicleId, cameraID, timeRange, params)`	vehicleId: String (default: self), cameraID: String, timeRange: TimeRange, params: Object	Camera data	Retrieves historical data from a specified camera, similar to `getHistLiDARData`, `getHistRadarData`, and `getHistGPSData`
Device Data	`getDeviceData(vehicleId, deviceId)`	vehicleId: String (default: self), deviceId: String	Object	Receives data from a connected external device, for example, a body-worn camera or a smartphone
	`storeDeviceData(vehicleId, deviceId, data)`	vehicleId: String (default: self), deviceId: String, data: Object	Boolean	Stores data from specified nearby devices
	`shareDataToDevice(vehicleId, deviceId, data)`	vehicleId: String (default: self), deviceId: String, data: Object	Boolean	Sends data to a connected external device
Logs	`getOperationalLogs(vehicleId, params)`	vehicleId: String (default: self), params: Object	Object	Retrieves operational logs
	`storeOperationalLogs (vehicleId, data)`	vehicleId: String (default: self), data: Object	Boolean	Stores vehicle operational data like speed, fuel consumption
User Data	`getUserData(vehicleId, userId, params)`	vehicleId: String (default: self), userId: String, params: Object	User data	Accesses user-related information within the vehicle, including settings and preferences
	`storeUserData(vehicleId, userId, data)`	vehicleId: String (default: self), userId: String, data: Object	Boolean	Stores user-related information within the vehicle, including settings and preferences
Infotainment Data	`getMediaContent(vehicleId, mediaId)`	vehicleId: String (default: self), mediaId: String	Object	Retrieves stored media content
	`storeMediaContent(vehicleId, mediaData)`	vehicleId: String (default: self), mediaData: Object	String	Stores media content like music and videos
Cloud & 3rd Party Data	`syncDataFromCloud(vehicleId, sourceId, params)`	vehicleId: String (default: self), sourceId: String, params: Object	Shared data	Retrieves data shared from cloud sources or other vehicles
	`syncDataWithCloud(vehicleId, destinationId, data)`	vehicleId: String (default: self), destinationId: String, data: Object	Boolean	Shares data from the vehicle to a specified destination, such as cloud service
	`getHDMap(vehicleId, location, detailLevel)`	vehicleId: String (default: self), location: Location, detailLevel: String	MapData	Fetches high-definition map data for a specified location with desired level of detail
	`getWeatherData(vehicleId, location)`	vehicleId: String (default: self), location: Location	WeatherData	Provides current weather information for the vehicle's location, including forecasts
V2X Data	`getV2XData(vehicleId)`	vehicleId: String (default: self)	V2XData	Receives and processes incoming V2X data via C-V2X, DSRC
	`sendV2XData(vehicleId, type, content)`	vehicleId: String (default: self), type: String, content: Object	Boolean	Sends V2X messages with specified content for communication
	`storeV2XData(vehicleId, data)`	vehicleId: String (default: self), data: Object	Boolean	Stores road and vehicle information received through V2X communications
	`getNearbyVehicles(vehicleId)`	vehicleId: String (default: self)	VehicleInfo	Retrieves information of nearby vehicles from V2X messages
	`getSafetyAlert(vehicleId)`	vehicleId: String (default: self)	SafetyAlert	Receives and acts on safety alerts from surrounding sources
	`getTrafficSignals(vehicleId)`	vehicleId: String (default: self)	TrafficSignalInfo	Obtains traffic signal status from V2X infrastructure
	`getEnviroInfo(vehicleId)`	vehicleId: String (default: self)	EnvironmentalInfo	Collects environmental data via V2X sensors
	`getParkingInfo(vehicleId)`	vehicleId: String (default: self)	ParkingInfo	Retrieves available parking spot information and recommendations from V2X messages

(to be continued)

Table A2. Data VPIs Specifications (Continued)

VPI Type	VPI Method	Parameter	Return Value	Functionality
	`getMapData(vehicleId, area)`	vehicleId: String (default: self), area: Area	MapData	Obtains detailed map data for a specified area, including roads, landmarks, and traffic conditions from V2X messages
	`getRoadConditions(vehicleId)`	vehicleId: String (default: self)	RoadConditions	Gathers information about current road conditions, such as construction, closures, or hazards from V2X messages

Table A3. Computation VPIs Specifications

VPI Type	VPI Method	Parameter	Return Value	Functionality
Data Preprocessing	`cleanData(dataType, rawData, params)`	dataType: String, rawData: Object, params: Object	Cleaned data	Performs data cleansing operations on raw data to remove inaccuracies and inconsistencies
	`formatData(inputFormat, outputFormat, data, params)`	inputFormat: String, outputFormat: String, data: Object, params: Object	Formatted data	Converts data from one format to another, ensuring data consistency across systems
Autonomous Driving Core	`earlyFusion(vehicleId, sensorTypes, params)`	vehicleId: String (default: self), sensorTypes: Array, params: Object	Early fused data	Integrates raw data from diverse sensors like cameras, radars, and LiDARs for initial insights
	`intermFusion(vehicleId, sources, params)`	vehicleId: String (default: self), sources: Array, params: Object	Intermediate fused data	Fuses partially processed data from sources like navigation and environmental sensors for enhanced analysis
	`lateFusion(vehicleId, data, params)`	vehicleId: String (default: self), data: Array, params: Object	Late fused data	Merges fully processed data from systems including V2X and diagnostics for comprehensive decisions
	`getFusedBEVResult(vehicleId)`	vehicleId: String (default: self)	Fused bird view result	Provides a fused overhead view of the vehicle's surroundings
	`get360View(vehicleId)`	vehicleId: String (default: self)	Fused 360 view result	Provides a comprehensive 360-degree fused view of the vehicle's surroundings, integrating data from all around-view cameras
	`getFusedLocation(vehicleId)`	vehicleId: String (default: self)	Fused location result	Retrieves the fused location data of the vehicle, combining GPS, IMU, map data, and sensor data
	`getDrivableAreas(vehicleId)`	vehicleId: String (default: self)	Drivable areas result	Identifies drivable areas around the vehicle by fusing data from cameras, radars, LIDARs, and maps
	`predictPaths(vehicleId, trafficData, params)`	vehicleId: String (default: self), trafficData: Object, params: Object	Path predictions result	Forecasts the driving paths of surrounding vehicles based on current traffic data and specified parameters, enhancing proactive driving strategies
	`planRoute(vehicleId, destination, params)`	vehicleId: String (default: self), destination: Location, params: Object	Route plan result	Generates an optimized route plan considering traffic, safety, and efficiency
	`decideMotion(vehicleId, situationData, params)`	vehicleId: String (default: self), situationData: Object, params: Object	Decision result	Processes real-time data to make driving motion decisions in response to road conditions

(to be continued)

Table A3. Computation VPIs Specifications (Continued)

VPI Type	VPI Method	Parameter	Return Value	Functionality
	`controlVehicle(vehicleId, twist)`	vehicleId: String (default: self), twist: Object (containing linear and angular velocity components)	Boolean	Sends control commands to the vehicle based on the twist data for precise movement and navigation control
AI Model Support	`processAIInference(vehicleId, taskType, data)`	vehicleId: String (default: self), taskType: String, data: Object	Inference result	Performs AI inference tasks, such as image recognition or natural language processing
	`runAIModel(vehicleId, modelID, inputData, params)`	vehicleId: String (default: self), modelID: String, inputData: Object, params: Object	AI output	Runs a specified AI model on input data, providing intelligent analysis
	`runE2EDriving(vehicleId, params, environmentData)`	vehicleId: String (default: self), params: Object, environmentData: Object	Boolean	Executes comprehensive end-to-end driving tasks, integrating perception, planning, and control based on real-time environmental data
	`runDiagnostics(vehicleId)`	vehicleId: String (default: self)	Diagnostic report	Runs a full diagnostic check of the vehicle
	`processVoiceCommand (vehicleId, audioData)`	vehicleId: String (default: self), audioData: Data	Boolean	Processes voice commands
V2X Analytics Support	`reqCloudCompute(vehicleId, task, data)`	vehicleId: String (default: self), task: String, data: Object	Task result	Sends data to the cloud for processing or analysis and retrieves the results
	`reqDeviceCompute(vehicleId, deviceId, task, data)`	vehicleId: String (default: self), deviceId: String, task: String, data: Object	Compute result	Utilizes computation resources of a nearby device like a smartphone for specified tasks
	`reqVehicleCompute(vehicleId, targetVehicleID, task, data)`	vehicleId: String (default: self), targetVehicleID: String, task: String, data: Object	Compute result	Accesses computation resources of a nearby vehicle for data processing or analysis
	`reqInfraCompute(vehicleId, infraId, task, data)`	vehicleId: String (default: self), infraId: String, task: String, data: Object	Compute result	Leverages computational power of road infrastructure for complex data tasks
Resource Allocation	`allocateResources(vehicleId, resourceType, params)`	vehicleId: String (default: self), resourceType: String, params: Object	Allocation status	Dynamically allocates computational resources based on current needs and priorities

Table A4. Service VPIs Specifications

VPI Type	VPI Method	Parameter	Return Value	Functionality
ADAS Service	`startADAS(vehicleId, functId, params)`	vehicleId: String (default: self), functId: String, params: Object	Boolean	Activates a specific ADAS function such as automatic braking or lane keeping
	`stopADAS(vehicleId, functId)`	vehicleId: String (default: self), functId: String	Boolean	Deactivates a specific ADAS function
	`configADAS(vehicleId, functId, settings)`	vehicleId: String (default: self), functId: String, settings: Object	Boolean	Configures settings for an ADAS function, allowing customization based on user preferences
	`startACC(vehicleId, params)`	vehicleId: String (default: self), params: Object	Boolean	Activates a specific ADAS function Adaptive Cruise Control (ACC), similar to `startLKA` VPI for Lane Keeping Assist (LKA), `startNVA` VPI for Night Vision Assist (NVA)
	`stopACC(vehicleId)`	vehicleId: String (default: self)	Boolean	Deactivates a specific ADAS function ACC, similar to `stopLKA`, `stopNVA`
	`configACC(vehicleId, settings)`	vehicleId: String (default: self), settings: Object	Boolean	Configures settings for an ADAS function ACC, allowing customization based on user preferences, similar to `configLKA`, `configNVA`

(to be continued)

Table A4. Service VPIs Specifications (Continued)

VPI Type	VPI Method	Parameter	Return Value	Functionality
Auto Service	`startAutoMode(vehicleId, settings)`	vehicleId: String (default: self), destinationParams: Object	Boolean	Activates the vehicle's autonomous driving mode with minimal driver intervention
	`stopAutoMode(vehicleId)`	vehicleId: String (default: self)	Boolean	Deactivates the autonomous driving mode, returning control to the driver
	`startAutoModewithV2X (vehicleId, destinationParams)`	vehicleId: String (default: self), params: Object	Boolean	Activates the vehicle's autonomous driving mode with V2X capabilities
	`stopAutoModewithV2X (vehicleId)`	vehicleId: String (default: self)	Boolean	Deactivates the autonomous driving mode, returning control to the driver
	`configAutoMode(vehicleId, settings)`	vehicleId: String (default: self), settings: Object	Boolean	Configures settings for the autonomous driving mode
Emergency Response	`initEmergencyCall(vehicleId, emergencyType)`	vehicleId: String (default: self), emergencyType: String	Boolean	Automatically contacts emergency services with vehicle details and location
	`sendEmergencyAlert(vehicleId, alertData)`	vehicleId: String (default: self), alertData: Object	Boolean	Sends an emergency alert to predefined contacts or systems
Infotainment Service	`playMedia(vehicleId, mediaId)`	vehicleId: String (default: self), mediaId: String	Boolean	Plays a media file
	`pauseMedia(vehicleId)`	vehicleId: String (default: self)	Boolean	Pauses the currently playing media

Table A5. Management VPIs Specifications

VPI Type	VPI Method	Parameter	Return Value	Functionality
Device & Service Connection	`pairWithDevice(vehicleId, deviceId, authCredentials)`	vehicleId: String (default: self), deviceId: String, authCredentials: Object	Boolean	Pairs the vehicle with an external device with necessary authentication credentials
	`unpairDevice(vehicleId, deviceId)`	vehicleId: String (default: self), deviceId: String	Boolean	Unpairs the vehicle from the external device
	`checkDevConnStatus(vehicleId, deviceId)`	vehicleId: String (default: self), deviceId: String	[DevConnStatus]	Checks the current connection status of the specified external device
	`checkV2XConnStatus(vehicleId)`	vehicleId: String (default: self)	[V2XConnStatus]	Retrieves the status of nearby V2X-enabled devices, including type, connection quality, and sharing capability
	`connectCloud(vehicleId, serviceId, credentials)`	vehicleId: String (default: self), serviceId: String, credentials: Object	Boolean	Connects the vehicle to a specified cloud service with authentication
	`disconnectCloud(vehicleId, serviceId)`	vehicleId: String (default: self), serviceId: String	Boolean	Disconnects the vehicle from a specified cloud service
Access Control	`authenticateUser(userId, credentials)`	userId: String, credentials: Object	Boolean	Validates user identity to ensure authorized access
	`setAccessControl(policyDetails)`	policyDetails: Object	Boolean	Defines access control policies for resources
	`checkAccess(userId, resource)`	userId: String, resource: String	Access permission	Determines if a user has access to a specific resource
	`logAccess(userId, resource, accessDetails)`	userId: String, resource: String, accessDetails: Object	Boolean	Records access attempts and details for auditing
	`encryptData(data), decryptData(encryptedData)`	data: Object, encryptedData: Object	Processed data	Encrypts or decrypts data for security purposes
	`generateToken(credentials), validateToken(token)`	credentials: Object, token: String	Token/ validation	Generates/validates tokens for access control

(to be continued)

Table A5. Management VPIs Specifications (Continued)

VPI Type	VPI Method	Parameter	Return Value	Functionality
System Status Monitoring	`monitorHWStatus(vehicleId, componentId, params)`	vehicleId: String (default: self), componentId: String, params: Object	Hardware status	Monitors and reports the health, performance, and operational status of specific hardware components in the vehicle
	`monitorSensorStatus(vehicleId, sensorId, params)`	vehicleId: String (default: self), sensorId: String, params: Object	Sensor status	Retrieves the current status of a specified hardware device, for example, `monitorSensorStatus`, `monitorActuatorStatus`, `monitorECUStatus`, `monitorChargingStatus`
	`monitorEnergyUse(vehicleID, functID, params)`	vehicleId: String (default: self), functID: String, params: Object	Energy usage	Monitors the energy when running a function
	`monitorCompResourceUse(vehicleId, functID, params)`	vehicleId: String (default: self), functID: String, params: Object	Resource usage	Monitors the usage of CPU, GPU, and memory when running a function
Personalized Mode	`startPersonalMode(vehicleId, userId, modeID)`	vehicleId: String (default: self), userId: String, modeID: String	Boolean	Activates a user-specific mode in the vehicle with a single click, providing a tailored experience based on individual preferences
	`stopPersonalMode(vehicleId, userId, modeID)`	vehicleId: String (default: self), userId: String, modeID: String	Boolean	Deactivates the currently active mode for a specific user, reverting to standard or predefined settings
	`switchMode(vehicleId, userId, currentModeID, newModeID)`	vehicleId: String (default: self), userId: String, currentModeID: String, newModeID: String	Boolean	Switches between modes for a specific user, facilitating seamless transitions tailored to individual preferences
	`configPersonalMode(vehicleId, userId, modeID, settings)`	vehicleId: String (default: self), userId: String, modeID: String, settings: Object	Boolean	Configures personalized modes for a specific user, such as comfort mode, gaming mode, and so on, according to their preferences
OTA Upgrade	`scheduleOTAUpdate(vehicleId, updateContent, params)`	vehicleId: String (default: self), updateContent: String, params: Object	Boolean	Schedules OTA updates for hardware drivers, software, and algorithms
	`downloadOTAUpdate(vehicleId, updateContent, params)`	vehicleId: String (default: self), updateContent: String, params: Object	Download status	Manages the downloading process for OTA update packages
	`installOTAUpdate(vehicleId, updateContent, params)`	vehicleId: String (default: self), updateContent: String, params: Object	Installation status	Executes the installation process for OTA updates
	`verifyOTAUpdate(vehicleId, updateContent, params)`	vehicleId: String (default: self), updateContent: String, params: Object	Verification status	Verifies the results of OTA updates and provides system-level feedback