

Article

Demand-Driven Data Acquisition for Large Scale Fleets

Philip Matesanz ^{1,*} , Timo Graen ¹, Andrea Fiege ¹, Michael Nolting ¹ and Wolfgang Nejd1 ^{2,3}

¹ Volkswagen Group, 30163 Hannover, Germany; timo.graen@volkswagen.de (T.G.); andrea.fiege@volkswagen.de (A.F.); michael.nolting@volkswagen.de (M.N.)

² L3S Research Center, Leibniz University Hannover, 30167 Hannover, Germany; nejd1@l3s.de

³ Faculty of Electrical Engineering and Computer Science, Leibniz University Hannover, 30167 Hannover, Germany

* Correspondence: philip.matesanz@volkswagen.de

Abstract: Automakers manage vast fleets of connected vehicles and face an ever-increasing demand for their sensor readings. This demand originates from many stakeholders, each potentially requiring different sensors from different vehicles. Currently, this demand remains largely unfulfilled due to a lack of systems that can handle such diverse demands efficiently. Vehicles are usually passive participants in data acquisition, each continuously reading and transmitting the same static set of sensors. However, in a multi-tenant setup with diverse data demands, each vehicle potentially needs to provide different data instead. We present a system that performs such vehicle-specific minimization of data acquisition by mapping individual data demands to individual vehicles. We collect personal data only after prior consent and fulfill the requirements of the GDPR. Non-personal data can be collected by directly addressing individual vehicles. The system consists of a software component natively integrated with a major automaker's vehicle platform and a cloud platform brokering access to acquired data. Sensor readings are either provided via near real-time streaming or as recorded trip files that provide specific consistency guarantees. A performance evaluation with over 200,000 simulated vehicles has shown that our system can increase server capacity on-demand and process streaming data within 269 ms on average during peak load. The resulting architecture can be used by other automakers or operators of large sensor networks. Native vehicle integration is not mandatory; the architecture can also be used with retrofitted hardware such as OBD readers.

Keywords: sensor-data acquisition; connected vehicles; big data; cloud computing; floating car data; data streaming; fault-tolerant systems



Citation: Matesanz, P.; Graen, T.; Fiege, A.; Nolting, M.; Nejd1, W. Demand-Driven Data Acquisition for Large Scale Fleets. *Sensors* **2021**, *21*, 7190. <https://doi.org/10.3390/s21217190>

Academic Editor: Francesco Bellotti

Received: 29 September 2021

Accepted: 26 October 2021

Published: 29 October 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Modern cars have the potential to be utilized as a globally distributed sensor platform. The data generated, thus, provides insight into the vehicle, its driver, and the vehicle environment. However, these data streams are currently only accessible to the automakers and a limited set of suppliers. The associated potential remains untapped while there is great demand from the private and public sectors.

Insurance companies are part of the interested parties. They require fine-grained vehicle sensor data to provide Usage-Based Insurances (UBI), prevent fraud, and reconstruct the course of accidents. It is a growth market inhibited by the data collection capabilities of current connected car fleets. Most companies track UBI policies through retrofitted hardware solutions that are installed to the vehicles at high-cost [1]. Receiving data directly from the automaker would reduce operational cost significantly [2].

Besides the commercial demand identified in the insurance market, academia also has many active fields of research that depend on vehicle data. Among these fields are driver behavior identification [3], inference of lane change intentions [4], or drowsiness detection [5]. In addition, access to vehicle data is essential for municipalities to facilitate the transition to Smart Cities [6]. For example, cities may utilize vehicle data to reduce

fuel consumption of public transportation [7] and predict urban traffic flows from floating car data [8]. In-vehicle sensors also enable the detection of potholes [9], slippery road conditions [10], or emission hot-spots [11]. Each of these examples requires recent data from a large number of vehicles to unfold its full potential. Individual vehicles may not provide sufficient statistical significance and only provide limited geographic coverage.

It is up to the automakers to tap into this potential. They manage vast fleets with millions of connected vehicles. Opening the associated data can thus be accomplished by performing a software update remotely. When designing scalable data platforms, all car manufacturers face the same significant problems:

- Local data privacy laws cause a globally scattered regulatory landscape. The European General Data Protection Regulation (GDPR) is especially strict among the different data privacy regulations. Complying with it, therefore, supports the integration of weaker frameworks. The GDPR requires obtaining informed consent from the data subjects before their Personally Identifiable Information (PII) can be collected [12]. Additionally, GDPR key principles such as data minimization or storage limitation must be applied [13].
- System-design issues are arising from a large number of cars and their diverse sensing capabilities. The potential supply of data exceeds the demand many times over. Typical data consumers only access a subset of the whole fleet, and from these vehicles, only a limited number of sensors are required for their use cases. Consequently, each data consumer may access individual subsets of the total sensing capacity. The system must map this access pattern efficiently. Nevertheless, these subsets can be of substantial size and place high requirements on the scaling properties of the system.
- Even for a single automaker, hundreds of vehicle models are regularly customized with model derivatives for specific regions and local markets. Each can have different sensing capabilities on its own. Additionally, the customers can select optional features that might introduce additional sensors. The sensors themselves might be supplied from different producers with varying data-access channels. Therefore, in addition to the legal heterogeneity, the individual cars are highly heterogeneous.

This paper presents a system that allows individual data consumers to access sensor data from the globally distributed fleets of major automakers. For this purpose, we integrate software directly into the vehicle, and a cloud platform facilitates access to the consumers. This access is either provided via near real-time streaming or recorded trip files. The overall system was natively integrated with a current vehicle platform of the Volkswagen Group and thus provides insights into the capabilities of current connected cars. However, native integration is not mandatory; the software component can also run on hardware retrofitted in the vehicle.

Our manuscript presents a system architecture and its implementation capable of addressing all three connected car challenges described above. No vehicle transmits more than necessary to satisfy the demands of its relevant data consumers. We can enforce GDPR compliance by requiring prior consent to any transmission of personal data and address individual vehicles directly to acquire non-personal data. An abstraction layer enables us to operate in a highly heterogeneous environment that includes cars with different sensor availabilities and technologies. We compress transmitted data by combining a custom data preprocessing strategy with an existing generalized compression algorithm. The system is capable of handling more than 200,000 simultaneously active vehicles. During a performance evaluation, it had consistently low response times after self-adjusting the number of utilized servers.

Contributions

The main contributions of this paper are as follows:

- A method to minimize the scope of acquired sensor data for each vehicle individually: Each vehicle only transmits data required to fulfill the *demand* relevant to it.

A data consumer's *demand* is considered relevant if the data subject has consented to fulfill it (personal data) or the individual vehicle was assigned to the *demand* (non-personal data).

- Dynamic *demand* determination in the context of a fixed purpose and data scope. For this purpose, the data consumers manage tasks which may also include constraints evaluated on each vehicle locally and data processing options.
- An abstraction layer enables the integration of a heterogeneous fleet through a single interface. For this purpose, we continuously distribute instructions to the vehicles. Each instruction describes an individual sensor access pattern. The vehicle checks these instructions by applying a trial-and-error scheme to determine its sensor capabilities and accessing the corresponding data.
- A compression strategy combines custom data preprocessing with an existing algorithm and, in combination, produces better results than the preprocessing itself or other existing algorithms alone. The compression ratio was evaluated by using actual vehicle data.
- A cloud-based reference implementation that handles data processing as distributed and trip-related transactions. Overall system throughput is automatically adjusted by adding or removing servers.
- Validation of the scaling properties of the overall system through a realistic simulation of over 200,000 simultaneously active vehicles.

The remainder of this manuscript is organized as follows: We present an overview of existing architectures and systems in Section 2. Then, we introduce our proposed system in Section 3 and address implementation details of its main components in Section 4. The results of our performance evaluation are detailed in Section 5. We compare our contributions with the identified works in Section 6. The final Section 7 provides conclusions to the paper.

2. Related Work

This section differentiates between works from academia and commercial systems for collecting and distributing vehicle sensor data. We present an in-depth comparison with our contributions in Section 6. In addition, we compare closely related systems with ours in Table 1.

Table 1. Overview of closely related data acquisition systems regarding selected features, including commercial closed-source products, which are marked with an asterisk. Features that are not evident from their public documentations are marked with a dash. There are also architectural concepts that assume a native integration but have not integrated their architecture into a vehicle yet. Some systems include programmable hardware; in this case, we only consider software and cloud platforms provided by the manufacturer [14,15].

Work	Feature	Demand Driven Data Acquisition ¹	Multi Tenancy ²	Vehicle External Data Storage ³	Vehicle External Data Streaming ⁴	Conditional Data Acquisition ⁵	Individual Vehicle Addressing ⁶	Consistency Guarantees ⁷	Data Compression ⁸	Native Integration ⁹
[16]		No	Yes	Yes	Yes	No	No	No	No	No
[17]		No	No	No	No	Yes	No	No	No	No
[18]		No	No	Yes	Yes	No	No	No	Yes	No
[19]		No	No	Yes	No	No	No	No	No	No
[20]		No	No	Yes	No	No	No	No	No	No
[21]		No	No	Yes	No	No	No	No	No	No
[22]		No	Yes	Yes	No	No	No	No	Yes	No
[23]		No	Yes	Yes	Yes	No	No	No	Yes	Yes
[24]		No	Yes	Yes	No	No	No	No	No	No
[25]		No	Yes	Yes	No	No	No	No	No	No
[26]		No	Yes	Yes	No	No	No	No	No	Assumed
[27]		No	Yes	No	Yes	No	No	No	No	Assumed
[28]		No	Yes	Yes	No	No	Yes	No	No	Assumed
[29]		No	No	Yes	No	Yes	Yes	No	No	Assumed
[30] (*)		No	Yes	Yes	No	No	No	-	-	No
[31] (*)		No	Yes	Yes	No	No	No	-	-	No
[32] (*)		No	Yes	Yes	Yes	No	No	-	-	No
[33] (*)		No	Yes	Yes	Yes	No	-	-	-	Yes
[34] (*)		No	Yes	Yes	Yes	No	-	-	-	Yes
[35] (*)		No	Yes	Yes	Yes	No	-	-	-	Yes
[36] (*)		No	Yes	Yes	No	No	-	-	-	Yes
[37] (*)		No	Yes	Yes	Yes	No	-	-	-	Yes
[38] (*)		No	Yes	-	-	No	-	-	-	Yes
[39] (*)		No	Yes	Yes	Yes	No	-	-	-	Yes
[14]		No	Yes	Yes	No	-	Yes	No	-	No
[15]		No	No	No	No	No	Yes	No	No	No
Our Proposal		Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

¹ Vehicle-specific scope reduction of transmitted sensors based on data-requirements of data consumers (See Section 3.2). ² Functionality to let several independent parties work with the system, providing each party potentially different data. ³ Data can be transferred from the vehicle to an external system and persisted there. ⁴ The external system continuously provides the received sensor data to other systems via streaming. ⁵ Data transmission can be made dependent on a condition evaluated on each vehicle locally. ⁶ It is possible to collect data from a specific car within a fleet without collecting data from the other cars. ⁷ The system can collect consistent sensor time series and detect inconsistencies if necessary. ⁸ The vehicle compresses data before transmitting it to an external system. ⁹ The system is a native component of a car and does not require any additional hardware.

2.1. Academia

The development of systems to collect vehicle data is often a prerequisite for answering research questions based on the data obtained [16–21]. For example, the authors of [16] have developed driving behavior analysis methods based on continuous streams of vehicular data. For this purpose, they had to develop an infrastructure capable of providing the required data. Such a pattern is indicative of a lack of suitable public data-access channels provided by the automakers.

Customer feedback platforms in the context of the Industry 4.0 vision represent a subset of these specialized data acquisition systems. Such platforms enable manufacturers to receive usage data directly from their products to improve quality, flexibility and productivity [21]. The data obtained in this way usually does not leave the sphere of the automobile manufacturer.

Some works focus exclusively on building platforms to make vehicle data available to arbitrary data consumers [22,23]. In contrast to the previous works, they provide the functionality to handle multi-tenant setups in which distinct parties acquire data from a fleet. For this purpose, they have implemented an authorization layer that restricts data access for the individual tenants: The driver must grant permission for each tenant to access data collected from his vehicle. Permission differs from consent because it does not influence data collection; it takes place in any case.

We utilize consent instead and combine our multi-tenant capabilities with a dynamic procedure to reduce data acquisition for each vehicle individually. For this purpose, we natively integrate software with an existing vehicle platform of the Volkswagen Group. This software selects the sensors to be recorded and can evaluate conditions that influence data acquisition. The sensor selection depends on the sensors required by the data consumers and can be continuously adjusted even after the data subject gave consent. In other works, data acquisition takes place as a static routine that is decoupled from the requirements of individual data consumers [16,18–25,30–32].

Apart from [23], the presented systems are not integrated natively with the vehicle. Consequently, the authors had to utilize the On-Board Diagnostics (OBD) interface to acquire sensor data from the vehicle [16–18,20–22,25]. The OBD is a physical interface within the car that exposes a small fraction of the data available on the Controller Area Network (CAN) [40]. Most sensors are connected to the CAN and broadcast messages containing their latest measurements. The majority of these messages are encoded using a manufacturer's proprietary scheme, which can also vary depending on the vehicle model [26,40]. Thus, parties unaffiliated with the automaker must reverse engineer the CAN to process all measurements, which is a dedicated field of research [41,42].

While some works use OBD as a sole vehicular data source [16,18,20,21], others also obtain data from their own sensors [17,22,25,43] or use them exclusively [19]. A combination of different data sources can also occur on a higher level by aggregating data from existing telematics systems and smartphones [24]. In some cases, the connection with the OBD is set up via self-designed hardware [44]. Our system is natively integrated with the vehicle via software and does not need additional hardware such as an OBD dongle. It only utilizes sensors that are part of the vehicle and acquires their readings via multiple software-based data-access channels provided by the vehicle platform.

The OBD interface essentially represents an abstraction layer, as it can provide a subset of the overall sensor readings of a vehicle through standardized message formats. Furthermore, it supports the discovery of available standardized sensor readings [21]. We have developed a higher-level abstraction layer that extends these features. It enables the data consumers to influence data acquisition in the absence of specific sensors. Instead of normalizing the sensor readings, they are supplemented with metadata such as resolution or sample rate. In addition, we can connect multiple data-access channels and handle vehicle-specific incompatibilities caused by diverging implementations of the data-access channels.

The acquired sensor data represent time series. There are compression algorithms optimized for time-series data that are not specific to the automotive domain [45,46]. Furthermore, there are generalized compression algorithms that can handle arbitrary binary data [47–53]. We have created a domain-specific and lossless data preprocessing strategy and combined it with an existing generalized compression algorithm. In [23], the authors utilize downsampling and histograms to reduce data transmission size, and a trajectory compression algorithm is used by [18]. In contrast, the authors of [54] propose to detect anomalies locally on the vehicle and restrict data transmission to them.

Different concepts are used for the processing and storage of data. Some systems provide a REST interface to access stored data requiring the data consumer to query for updates [22,23]. In [23,27], the authors use a message broker to push incoming data directly to data consumers. The works of [16,18] utilize multiple storage layers optimized for specific use cases. For example, the authors of [18] store the most frequently accessed data in a memory-based caching system, use a relational database for preprocessed data, and archive raw data in a NoSQL-based system. Our cloud-native system utilizes a single storage layer that enables near real-time streaming and continuous sensor data recording. We can integrate third-party systems from different clouds and deliver events to them, thus eliminating the need to query for updates.

There are also architectural concepts for acquiring vehicular data [26–29,55]. These concepts imply an immediate integration into the car without providing a reference implementation. Unlike the previously mentioned works, these architectures are not forced to adapt to the reality of the volatile and complex automotive domain. On the contrary, our work demonstrates the capabilities of modern mainstream vehicle platforms through its integration into a current platform of the Volkswagen Group.

2.2. Commercial Systems

There exists a commercial market for systems that enable data acquisition from vehicle sensors. We present a selection and distinguish between software-based solutions and those that require retrofitted hardware installed on the vehicles. These systems are mainly black boxes, and their implementation details are only known to the manufacturers. The descriptions are based on the manufacturers' public documentations.

2.2.1. Software Platforms

Otonomo [33], Caruso [34] and Smartcar [35] are neutral platforms that act as intermediaries between automakers and data consumers. The data consumer maintains a relationship with the neutral platform, which acquires data from the automaker and forwards it to the customer. The vehicle owner must grant consent by completing an OAuth flow with the automaker as specified by Extended Vehicle Standard [56]. Each of these platforms has a REST interface that provides sensor values of individual vehicles. In addition, they can stream incoming vehicle data to a data consumer supplied server by utilizing HTTP requests. The sensor availability differs across these platforms. Otonomo provides filters that limit the streaming of already collected data based on the location of the vehicles or the age of the data.

Additionally, various car manufacturers offer direct access to vehicle data. These include Mercedes Benz [36], BMW [37], or Ford [38]. They also require the vehicle owner to grant permission for data acquisition by performing an OAuth flow. Streaming is not supported; the customer must poll the data instead.

The Groupe PSA [39] has created a system for acquiring data from fleets belonging to a single owner. Customers can create *Monitors*. They consist of a condition validated on the backend and, if matched, results in a notification to customers. The condition refers either to a geographical area, the time, or an arbitrary vehicle state (e.g., AC active/inactive). Additionally, a record is created for each performed trip. It contains the distance and the start and end position.

2.2.2. Hardware/Software Platforms

Providers such as Zubie [30], Munic [32] or Vinli [31] offer OBD dongles that have built-in internet connectivity and send vehicle data to an associated cloud platform. All systems offer interfaces to poll sensor data from the cloud. Munic and Vinli store multiple updates of a sensor value, while Zubie provides only the last value. Sensor data can only be streamed to the customer via Munic. The others can send events like the start of a trip or its completion. Vinli can validate a condition and trigger an event once it is met.

There are also OBD devices designed to be used as software platforms [14,15,57,58]. As such, their users can deploy custom software to them in order to process and transmit the obtained data. For this purpose, the devices can provide cellular, WiFi, or Bluetooth connectivity. In addition to OBD, some devices also include their own sensors such as GPS or accelerometer [14,15,57]. Some providers also offer software to collect data from their hardware [15] or provide data acquisition as a service via a cloud platform [14,59]. These devices can also be utilized to run the software component we have natively integrated into the vehicle.

3. Proposed System

We have created a system that enables individual data consumers to access the sensing capabilities of vast heterogeneous fleets. It consists of an application that is natively executed by the associated vehicles and a cloud platform. The in-car application includes an abstraction layer that unifies sensor access across different sensor configurations and technologies. Each car continuously determines relevant sensors for recording by requesting the vehicle-specific data requirements from the cloud platform. The recorded sensor readings are then transferred to the cloud and processed there. As a result, the readings are either streamed to the data consumer or persisted as files that map individual journeys from start to finish (trip files). The data consumer's infrastructure can be connected directly to our cloud platform. The connection is established via a message-based interface that the consumer must provide. Our platform sends messages that include, for example, streaming data or notifications about newly created trip files. The use of our system is visualized in Figure 1.

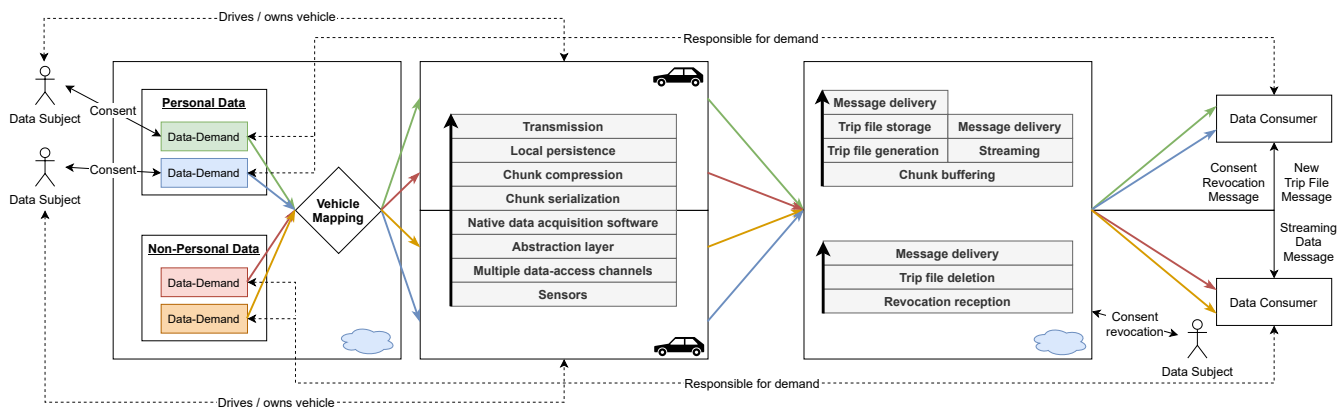


Figure 1. Use of our system to acquire personal and non-personal data.

3.1. GDPR Related Requirements

The GDPR distinguishes between personal data and non-personal data. Any processing of the former requires informed consent, which the data subject must give in advance [60]. Such consent includes a purpose limitation that also conclusively lists the parties involved in processing the data [61]. Additionally, the principle of data minimization must be applied. Therefore, only data indispensable for the stated purpose may be collected [13]. The GDPR does not regulate the processing of non-personal data. Consequently, a system that can legally process personal data is also allowed to process non-personal data.

The term personal data is to be understood very broadly. It refers not only to data that directly links a person, such as a name, but also to data with an indirect link to a person. Thus, a license plate is also considered personal data even though the processor might not be capable of looking up the owner. The mere possibility that such a lookup could be performed is sufficient for the classification as personal data [60]. The classification of a set of data into this binary scheme poses a considerable problem in practice [60]. Incorrect classifications can lead to severe penalties of up to €20 million or 4% of annual global turnover (whichever is higher) [62]. This is a strong incentive for companies to classify data as personal when in doubt.

The system we present is designed to meet the legal requirements for processing personal data (See Table 2). As such, we do not transmit any sensor data from a vehicle unless informed consent is given. The scope of data collection is determined individually for each vehicle by applying our demand-driven data acquisition strategy (See Section 3.2). In this way, we comply with the principle of data minimization and ensure that no data collection takes place without a purpose. Designing the system to handle personal data does not prevent its use for non-personal data, as explained in Section 3.8.

Table 2. Requirements for the processing of personal data derived from the GDPR.

Requirement	Realization	Section
Personal data may only be processed if prior consent is given.	The data collection software natively integrated with the vehicle does not collect and transmit any data by default. An individual vehicle only performs data collection if a consent object exists within our system associated with the vehicle.	Section 3.2
The collection of personal data may not exceed the scope of the associated consent.	The data subject consents to grant the data consumer access to an immutable set of sensors (project). Subsequent data acquisition by the data consumer can not exceed this scope.	Section 3.2
If consent was revoked, all associated data must be deleted from our system.	Personal data stored on our system is linked to the associated consent. If consent is revoked, we automatically delete the associated data.	Section 4.8
If consent was revoked, all associated data must be deleted from the infrastructure of the data consumer.	We notify the data consumer about the revocation via a message-based interface. The data provided to him always references the associated consent. Thus, he can carry out a targeted deletion on his infrastructure.	Section 3.5
If consent was revoked, associated data acquisition must stop; no further data may be collected.	Each vehicle communicates continuously with our cloud platform to stop individual data acquisitions if associated consent is revoked. The cloud platform will discard data the vehicle transferred before the revocation was propagated upon receipt.	Sections 4.3 and 4.8
No more data may be collected than necessary (data minimization).	Each vehicle only transmits sensors required to fulfill the data <i>demand</i> of the data consumers for which associated consent exists. Data consumers specify <i>demand</i> according to their needs and can adjust the scope of data acquisition even after consent is granted.	Section 3.2

3.2. Demand-Driven Data Acquisition

We have found that, in practice, an access pattern prevails in which individual data consumers want to access different sensors originating from different cars. There may be overlaps, i.e., multiple consumers can simultaneously access certain vehicles and sensors. The demand-driven data acquisition method we have developed can efficiently handle such patterns. It minimizes data collection for each vehicle individually. As a result, we only collect data required by the data consumers (*demand*) who obtained consent from an individual vehicle. An illustration of our applied method is presented in Figure 2.

Data consumers specify their *demands* within the context of so-called projects. Each project represents an independent data collection purpose within the meaning of the GDPR. It contains sensors that define the maximum scope of any data collection carried out through it. The *demand* is set separately from the project and can be continuously adjusted while being within the associated scope. The consent is directed at the project. Thus,

a *demand* adjustment affects existing and new consents equally. The projects are immutable and represent reliable constraints on the associated *demand*.

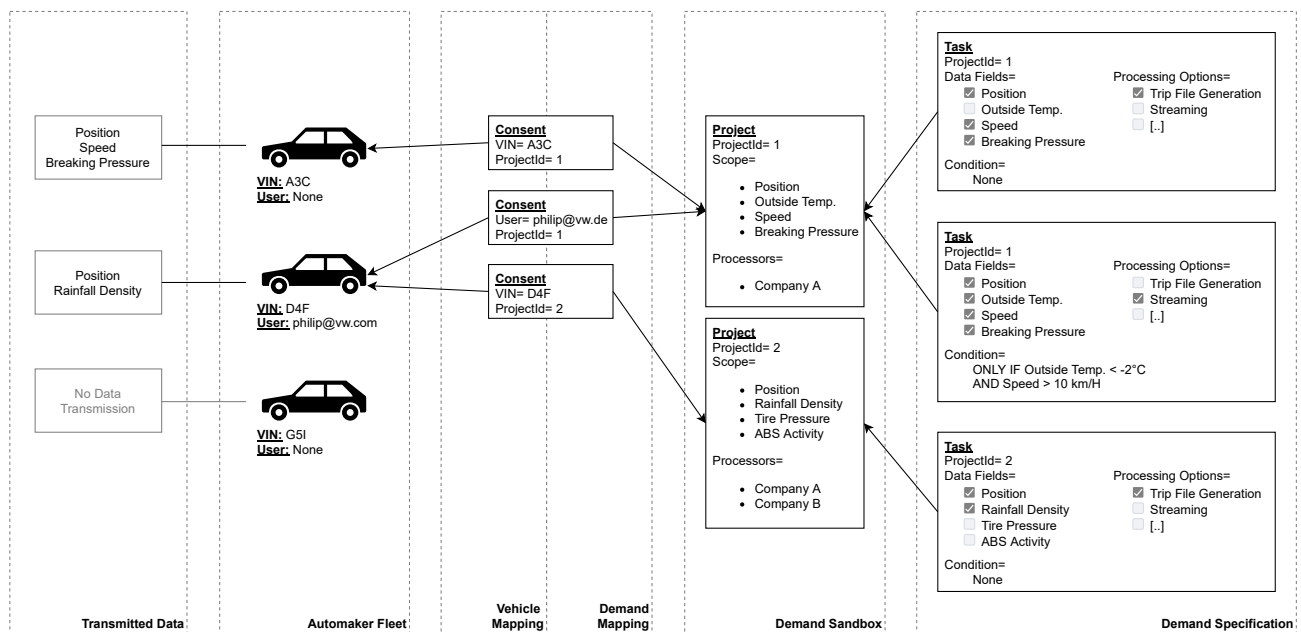


Figure 2. Application of the Demand-Driven Data Acquisition. The condition of one task does not apply. It is therefore not executed.

The actual *demand* of a project is equal to the sum of its tasks. Each task contains a list of sensors to be acquired, an optional condition, and any processing-related options. Each vehicle is receiving all tasks for which project-related consent exists. The application deployed to the vehicles is executing them (See Section 4.4). Therefore, the condition is evaluated individually by each car and can include local sensor values, geographic restrictions, or time limits. A task can perform conditional data collection as it is only executed if the associated condition holds. Upon execution, the task-related sensors are recorded and transmitted with regard to the processing options: If streaming is activated, the recorded sensor readings are buffered locally for a shorter time, for example. Additionally, deduplication occurs; if several tasks contain a sensor, its readings are transmitted once.

Each consent links an individual car with a project. For this purpose, the vehicle can be addressed via its unique Vehicle Identification Number (VIN). Alternatively, indirect addressing can occur by utilizing the driver's account used to log on to the car. Thus, the data of an individual driver can be collected across multiple vehicles. There can be multiple consents for each project and car.

3.3. Sensor Abstraction

A highly heterogeneous car landscape characterizes the automotive sector. For example, the Volkswagen Group is currently producing 391 different car models globally. They are built on top of distinct platforms that utilize different technologies. Every model can be configured to contain optional features upon ordering. These features may result in additional sensors being added to the car. There are at least two suppliers for every sensor. Each supplier can introduce custom characteristics into their products, like different data resolutions or access channels. Additionally, software updates may be performed on subsets of the overall fleet. Such an update can potentially alter sensor-specific data-access channels.

We have created an abstraction layer that reduces the variety of sensors to the individual *data fields* that can be read from a car. Consequently, our system only reveals high-level *data fields* such as "speed". The underlying complex variety of different sensors is invisible to the data consumer. We map this variety via so-called instructions. For each *data field*, several instructions can exist. There is one for every distinct data-access channel

that may exist. The application deployed to the vehicles can interpret such instructions to acquire the associated *data fields* from its sensors. It can thus also determine local sensor capacity by trying out the instructions: If the application can successfully execute at least one instruction of a *data field*, it is considered available.

The data-access channels have a multi-layered architecture: A custom application logic is implemented based on existing protocols such as WebSocket or MQTT. An example for such an implementation is the VIWI interface [63]. The instructions we utilize are implemented at the protocol layer. For each protocol, there is a dedicated instruction processing logic in the abstraction layer. The protocol-level implementation enables us to handle diverging application logic exclusively by adding additional instructions.

We have developed a replication mechanism for this purpose. It continuously transfers the available instructions to all vehicles (See Section 4.2). In addition, each vehicle reports local changes in *data field* availability. An anomaly detection evaluates these changes. We can thus detect breaking modifications to the data-access channels indirectly and restore compatibility by adding new instructions. Additionally, we can monitor the overall health of the system (See Figure 3).

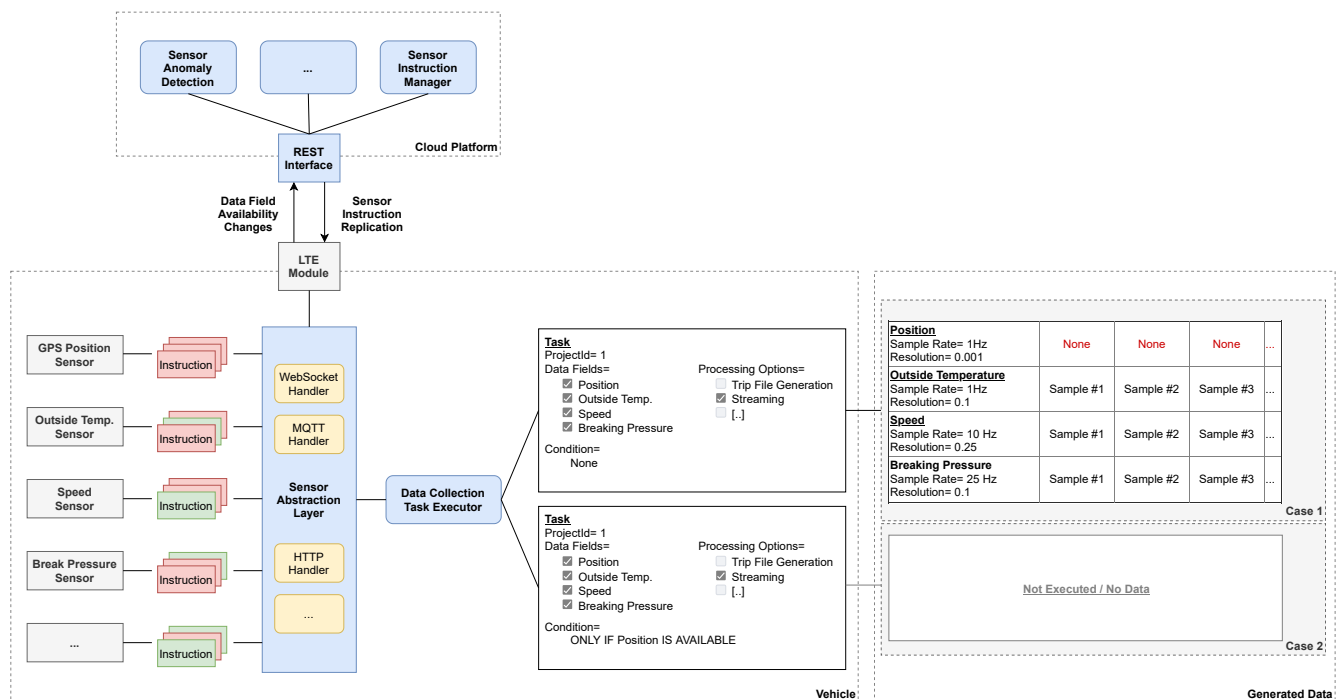


Figure 3. Usage of the sensor abstraction layer. Individual instructions are available for each sensor. Instructions that are compatible with the specific vehicle are marked in green, and incompatible instructions are marked in red. In the first case, as much data as possible is collected, resulting in a sparse dataset because a *data field* is not available. In the second case, no sensor readings are collected at all because the condition does not hold.

The abstraction layer is also integrated with the task concept of the demand-driven data acquisition strategy. By default, a task yields a sparse dataset, i.e., if a *data field* is not available on a vehicle, the remaining *data fields* are still transmitted and made available to the data consumer. However, it is possible to include the availability of individual *data fields* into the condition. As a result, a task can be configured to yield a dense dataset as it is only executed if the given *data fields* are available on the vehicle (See Figure 3).

The sensor abstraction layer does not explicitly normalize local sensor peculiarities such as different data resolutions, sampling rates, or units of measurement. Instead, the sensor readings are transmitted unchanged in combination with metadata containing these peculiarities. Consequently, a data consumer can receive sensor data in different resolutions and sampling rates via a single task.

3.4. Data Transmission and Processing

The vehicle transmits recorded *data fields* as chunk sequences to the cloud platform. It generates an independent sequence for each trip, whose components are numbered consecutively. Each chunk can include time series for multiple *data fields*, and it is structured so that there are no dependencies on other chunks. Thus, if chunks are lost, the data loss is limited to those chunks. The vehicle knows which *data fields* are retrieved by tasks that have streaming enabled. It transmits the corresponding data more frequently serialized as separate chunks. A custom chunk compression strategy is applied as described in Section 4.5.

A three-phase protocol is employed to transmit each sequence to the cloud platform: First, the vehicle performs a trip initialization by submitting a unique TripID. Afterward, it transfers the chunks upon their generation. Finally, after the vehicle has transmitted all chunks, it sends a *commit* message that indicates the end of a trip. The *commit* message includes the TripID and the number of the final chunk.

The cloud platform is storing all received chunks within a persistent buffer. Upon receiving the *commit* message, it performs an aggregation by merging all chunks of the trip. The resulting aggregate is split based on the contained tasks. A trip file is generated and stored for each task that has enabled persistence. Receiving a streaming chunk triggers its immediate processing: The contained data are transferred to the data consumers by utilizing endpoint messaging as described in the following subsection. The execution of a sequence transfer is shown in Figure 4.

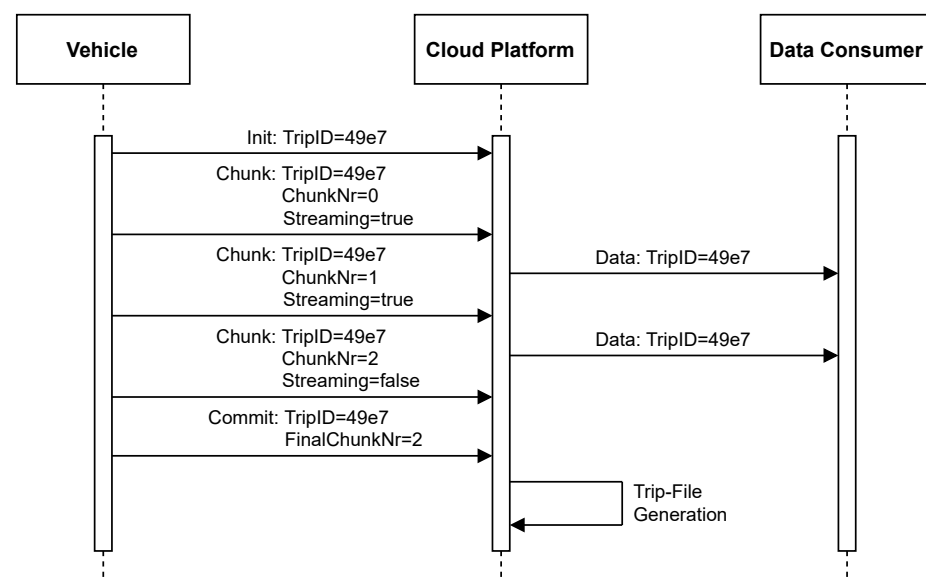


Figure 4. Transmission of a chunked trip sequence.

Multiple persistence layers within the car ensure no data loss occurs if the car is turned off or has lost its internet connectivity, as described in Section 4.7. However, such a condition can last for an extended period, e.g., if the vehicle is in an underground car park. Thus, there is no way to ensure that the vehicle sends the *commit* message on time. We enforce a timeout to keep the buffer requirements predictable by avoiding indefinite buffering. When the timeout occurs, the cloud platform performs the aggregation, which results in trip files that contain a flag indicating their possible incompleteness. Otherwise, the completeness can be determined reliably with the final chunk number from the *commit* message. Tasks can be configured to discard incomplete files.

Trip files include a reference to the associated consents. If there is a consent revocation, then the corresponding files are automatically deleted from our platform. The data consumer will be notified of the revocation via endpoint messaging and can perform a local deletion by utilizing the embedded references.

3.5. Endpoint Messaging

We assume that data consumers have automated systems for data processing. These systems can be connected to our cloud platform via a message-based interface (See Figure 5). Messages delivered through this interface can describe events such as new trip files or consent revocations. These messages only contain metadata that can be used, for example, to retrieve the actual trip file via our REST interface. In addition, we also use the message interface to deliver streaming data. Unlike trip files, it is limited in size because no aggregation is performed. Therefore, we embed streaming data directly within the message.

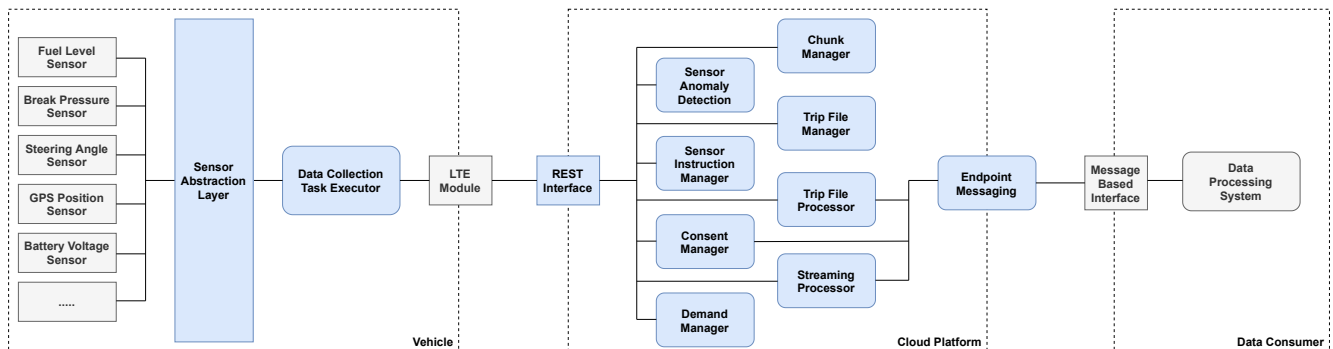


Figure 5. High-level architecture of the proposed system. Components that we have developed are coloured blue.

We use platform-independent Protocol Buffers to serialize the messages [64]. They enable us to support different transportation channels such as HTTPS, AWS SQS, or Google Pub/Sub. While the latter have their own authentication methods, the opposite is true for HTTPS. We utilize asymmetric cryptography to extend every message that is delivered via HTTPS with a signature. It can be verified by using our public keys.

The delivery of messages must be reliable, as the loss of individual messages can lead to a GDPR violation. If the data consumer is not notified of consent revocation, he may not fulfill his obligation to delete associated data. For this reason, we repeat delivery attempts until they finally succeed.

3.6. Vehicle Authentication

Requests originating from a vehicle are authenticated via signed tokens that contain the VIN and optionally the user id from the driver. A central Identity Provider (IDP) issues them. The vehicles have embedded certificates linked to their individual VIN. They utilize these certificates to obtain tokens from the IDP continuously. In addition, the driver can log in to the IDP via the vehicle user interface. Then, the IDP extends the token with the user id. Our cloud platform verifies these tokens and can thus provide the origin of individual data. Additionally, all communication between the vehicle and our cloud platform is encrypted by utilizing HTTPS.

3.7. Vehicle Simulators

The system provides functionality to simulate active vehicles. It realizes such simulations by using recordings from real cars. The data contained therein were captured at the protocol layer and represents a complete snapshot of the data-access channels during a trip. These snapshots are replayed to create a virtualization of the data-access channels in the car. The same software components that are usually executed by the car are connected to them. The result is a realistic simulation covering all system features and offers the same data as the actual vehicle (See Figure 6).

Simulations of different vehicle models are provided by creating separate recordings for them. Model-specific characteristics, such as differences in the data-access channels, do not have to be explicitly mapped since the recordings are made at the protocol layer. We provide the recordings from different car models as a service.

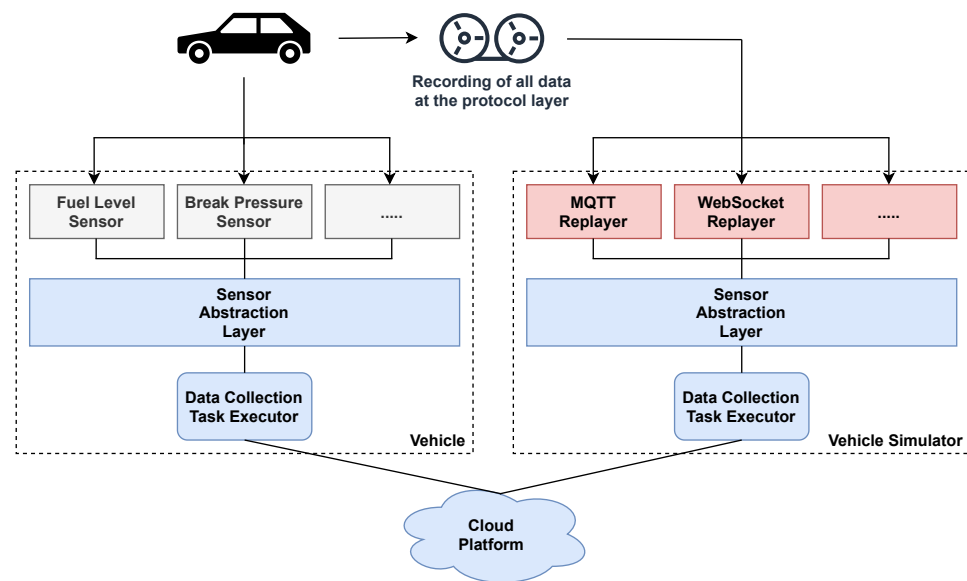


Figure 6. Implementation of a vehicle simulator. It executes the same software as the car. We provide simulated data-access channels that can replay arbitrary recordings from actual vehicles.

The simulators are made available to the data consumers. They can thus develop their own data processing systems without needing any consent or own vehicles. We also utilize them to perform load tests (See Section 5) and functional tests.

3.8. Acquisition of Non-Personal Data

The presented system can also collect and process non-personal data for its data consumers. For this purpose, we utilize the mapping resulting from the consent: As soon as a consent object is present, an individual vehicle begins collecting and transmitting the associated *demand*. When dealing with non-personal data, we can create these objects without actual consent from the data subjects and utilize them to address individual vehicles for data collection. There are no additional legal requirements (See Section 3.1).

For practical use in the context of individual data consumers accessing subsets of the overall fleet, such addressing would have to be automated and intelligent: The intended fleet includes millions of vehicles spread across the globe. Therefore, we must select the vehicles that provide the most significant benefit for the goals of the data consumer. For example, if a data consumer wants to determine the most popular radio station in New York, vehicles in Brazil would be unsuitable. Economic efficiency correlates with selection accuracy since an incorrect selection generates costs that are not offset by benefits. Our system does not yet include such intelligent addressing. The selection must precede the task distribution because task conditions are not designed for this purpose. Using task conditions would require all non-personal data tasks to be executed on every vehicle. Consequently, we cannot provide access to non-personal data for arbitrary data consumers efficiently.

However, the system can still efficiently handle use cases that do not depend on such car selection optimization. Automakers have an interest in collecting data to track wear and tear on their vehicles. They are also interested in statistics on average consumption and emissions [21]. It is possible to acquire such data for each model individually: The automakers assign the VINs, which indicate the associated model. Thus, they can either create a consent object for all of them or perform sampling by only utilizing a subset of the models' VINs.

4. Implementation

Our system is split into multiple components, which are distributed over different infrastructures (See Figure 5). Most of them are critical for the platform's reliability and

must be capable of handling high-frequency access patterns. We describe implementation details of selected components within this section.

4.1. Utilized Technology

The system components running on the vehicle were developed in Node.js without using an additional framework. They are being executed as a background application without having a user interface. Our cloud platform was developed in Python and uses the FastAPI framework [65]. The connection between the two systems is made via client libraries automatically generated from the Swagger definitions provided by FastAPI. In addition, we utilize Protocol Buffers for message and sensor data serialization [64]. Furthermore, the following services provided by Google Cloud Platform and their associated libraries are used:

- App Engine: A managed service to host web applications. It deploys a user-supplied application to proprietary virtual machines that spawn within seconds. Thus, it allows handling sudden spikes of traffic by adjusting the underlying servers just in time [66].
- Datastore: Managed NoSQL database that stores items with unique keys. Its underlying servers manage continuous subsets of the keyspace. Every item is limited to one update per second. It supports consistent queries and transactions spanning multiple operations [67].
- Cloud Tasks: Provides the capability to schedule asynchronous HTTP requests. It supports delayed executions and retries failed requests until they finally succeed. The requests are stored within queues [68].
- Pub/Sub: Asynchronous messaging service. Messages are published to topics that can have multiple subscriptions. Published messages are replicated to every subscription and will be delivered at least once [69].
- Compute Engine: A service that provides virtual servers and auto-scaling. The scaling can be tied to the in-flight messages of a Pub/Sub subscription [70].
- Cloud Storage: An object store that can persist unstructured blobs/objects within buckets. Events can be pushed to a Pub/Sub Topic [71].

4.2. Instruction Distribution

The instructions of the sensor abstraction layer must be eventually replicated to the connected cars (See Section 3.3). No inconsistencies may occur during this process, as this could cause *data fields* to become locally unavailable. This process is read-heavy and requires minimal writing: There are potentially millions of cars regularly querying for updates. On the contrary, we are adding new instructions very infrequently.

We utilize a replication strategy that relies on multiple Last-Writer-Wins Registers (LWW Registers), a known conflict-free replicated data type [72]. There is an individual register for each *data field* that contains all associated instructions. We eventually apply the final state of each register to the vehicles upon performing changes to them. For this purpose, we use an ordered sequence whose elements are the registers. The sorting order corresponds to the modification date of the registers, which is unique and consistent with causality. Thus, each element represents a savepoint from which the vehicle queries the following elements. If we make multiple changes to the same register, only the final state remains in the sequence. The vehicle can thus skip intermediate states within the replication process.

A distributed cache layer accelerates the sequence replication of the vehicles. They regularly query for elements to continue with the local sequence. In return, they receive the subsequent elements as a sorted list. The sorting property enables distributed caching that does not require synchronization. No inconsistency occurs when the last item of an ordered sequence is missing. A subsequent query will yield it as long as the cache eventually expires.

We have implemented the instruction replication by utilizing Datastore and App Engine. The former acts as a reliable persistence layer. There is an item for every register,

respectively, *data field* that contains all associated instructions. Each item has a timestamp property, and updates are performed in-place. The sequence is generated via a strongly consistent Datastore query [73] that performs sorting by utilizing the timestamp. The resulting sequences are cached on the individual App Engine instances and can thus be accessed without additional network calls (See Figure 7). The used timeout has a random component to avoid a cache stampede.

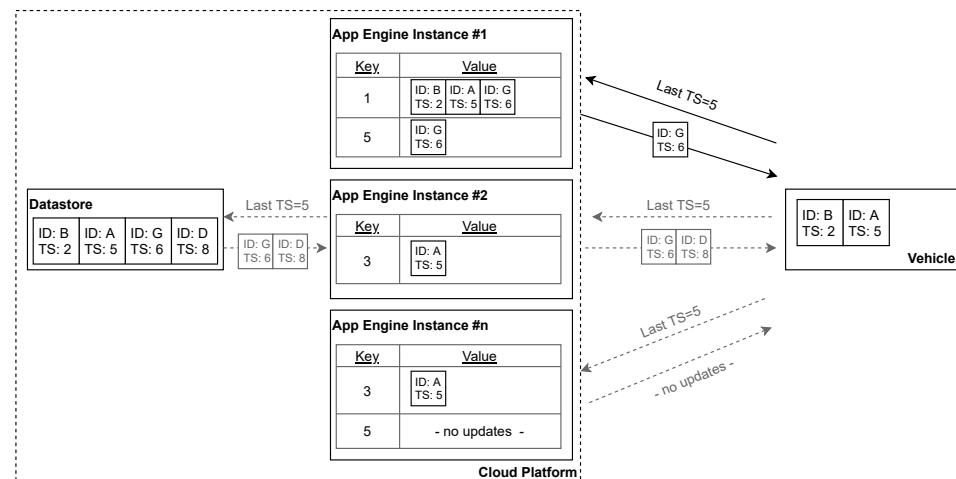


Figure 7. Sharding during instruction distribution. Datastore acts as single source of truth. App Engine instances cache queries locally. The timestamp submitted by the car acts as cache key and query parameter. The cache items can be out of sync as long as they represent an ordered subsequence and eventually expire (See Section 4.2).

We use a monotonically increasing value as a timestamp. The system time is not suitable because we are not able to synchronize the clocks appropriately. Even if this were the case, a small risk of duplicate timestamps would remain. For this, we maintain a dedicated Datastore entry that acts as a counter. Each register change is a transaction that includes the counter and increments it to determine a new timestamp.

As a result, we have increased the read throughput by exploiting the auto-scaling capabilities of App Engine. However, the write throughput is limited to one *data field* update and creation per second due to the counter including transaction. As this guarantees consistency, we consider the restriction to be appropriate due to the rarity of instruction updates.

4.3. Consent Lookup

Each vehicle performs regular queries to the cloud platform to determine the relevant projects. They are determined by evaluating the active consents. For this purpose, each vehicle transmits its VIN and optionally a user id (identifiers). This operation is primarily read-heavy, but there cannot be any drawbacks on write performance. It must be possible to add and revoke the associated consents at a high frequency.

We store each consent as individual Datastore objects and maintain a reverse index for each identifier to enable querying by them. The access intensity to the individual objects tends to be equally distributed; consequently, we do not use a cache. Instead, we utilize the data distribution strategy of Datastore to balance the load evenly across the underlying server capacity. The assignment of an object to individual servers results from its key [67].

For each consent, we generate a random key with high entropy. The creation is done within a transaction to ensure that the key is unique. In addition, we maintain the reverse index within the transaction. For each identifier, there is an object storing all keys and project references of the associated consents. We derive the key of the reverse index object deterministically from the corresponding identifier. An index lookup is thus possible via a single get-operation.

The identifiers may represent monotonically increasing values or have a common prefix. For example, the first three characters of a VIN are identical for many vehicles because they contain a manufacturer's abbreviation [74]. Such keys represent a Datastore anti-pattern causing a concentration of the associated objects on a small subset of the server capacity [75]. Therefore, we use a cryptographic hash function to create uniformly distributed prefixes for the identifiers. This additional prefix causes an even distribution of the associated objects.

We found that our implementation has few limitations. Each identifier generates approximately the same load, and we distribute the underlying objects evenly across the Datastore servers. We are thus not limited in the number of vehicles that query our platform simultaneously. There are also no limitations in terms of concurrent consent creations and revocations. However, there cannot be more than one consent revocation or creation per second for each distinct identifier. This limitation results from the transaction through which we maintain the reverse index. We consider the limitation to be appropriate.

4.4. Task Distribution

Each vehicle must determine the active tasks for the project references obtained through consent lookup (See Section 4.3). There is a continuous synchronization with the cloud platform so that task-related changes can be detected promptly. The process is predominantly reading-heavy. There is a substantial potential for contention if many vehicles are linked to the same project and thus query the same data.

A vehicle can be assigned to multiple projects, and each can contain multiple tasks. We have designed our process to avoid the synchronization of individual tasks. For this purpose, we have made the tasks immutable and manage a list of active tasks for each project. Thus, a task is modified by creating an updated task and deactivating the previous task. The vehicle synchronizes these references continuously and requests the associated tasks separately. Due to their immutability, tasks can be cached indefinitely by the vehicle and cloud platform.

We store the tasks as individual Datastore objects. In addition, there is a reverse index for each project that contains references to its active tasks. We store the reverse index as a separate object whose key is composed of the project key. This assignment enables an index-lookup via a single get-operation. When the data consumer creates or deactivates a task, the reverse index is updated via a transaction.

The reverse index includes a version property turning it to an LWW Register [72]. We can therefore cache the indexes on the individual App Engine instances without synchronizing them. The vehicle uses the version to ignore stale data. A short expiry interval ensures the timely propagation of updates.

By not synchronizing individual tasks, we reduce replication overhead. We can apply different caching strategies due to the separation of tasks and their activation properties. Immediate caching on App Engine instances automatically adjusts the read capacity to the current load. There is no limitation on overall projects and tasks. However, for each project, there can be only one task addition and deactivation per second. We consider this limitation to be appropriate.

4.5. Chunk Compression

Each vehicle continuously generates and transmits chunks. Under certain circumstances, persistence to a local storage medium occurs (See Sections 3.4 and 4.7). Reducing the chunk size is thus a necessity. Local storage is limited and composed of eMMC memory. Writing large amounts of data may cause a memory failure [76]. In addition, multiple applications are consuming the limited internet uplink, and it may be degraded in rural areas [77]. Compressed chunks have less impact on other consumers, and their transmission is more likely to succeed on degraded networks. Furthermore, they improve economic efficiency as uploads from a vehicle are expensive.

We have developed a data preprocessing method that we combine with an existing compression method. As a result, we achieve a significantly higher compression ratio than either method can provide on its own.

The preprocessing procedure is lossless and targets the time series contained in a chunk. Its elements are composed of a limited set of primitives, and each has an associated timestamp. We convert all values to the smallest possible integers and apply variable-length integer encoding (VLI Encoding). There is a different conversion strategy for each primitive, as explained below.

1. **Timestamp:** The system does not have real-time capabilities and cannot perform measurements at exact intervals. In addition, most data-access channels only report measurements if they differ from the previous ones. Consequently, we cannot drop the timestamps in favor of storing the sequence interval length. However, many timestamps still represent recurring intervals with slight variations. They originate from sensors whose values changed with almost every measurement. For example, the engine speed most likely varies continuously, given a resolution of 1 RPM. In [46], the authors have found that Delta-Of-Delta (DOD) encoding is a good fit for timestamps with such characteristics. It enhances Delta encoding, which is a procedure that only keeps the first value of a sequence. The subsequent values are the delta to the predecessor ($a[n] = a[n] - a[n - 1]$). In DOD encoding, the first value is unchanged, and the second is the delta to the first. Subsequent values are computed as follows: $a[n] = (a[n] - a[n - 1]) - (a[n - 1] - a[n - 2])$. Thus, they represent the delta of two deltas.

Applying DOD encoding to the timestamps results in an average compression ratio of 3.66 (See Table 3).

2. **Integer:** We apply Delta encoding to integer values. The average compression ratio is 3.76. DOD encoding has not led to further improvements (See Table 3).
3. **Float:** We found that no sensor exploits the full precision of a float. Thus, we can perform a reversible conversion into integers (F2I) without information loss. The conversion is performed by shifting the decimal sign n places to the right ($float * 10^n$) and cutting the remaining decimals. n is equivalent to the decimals of the sensor resolution. A resolution of 0.25 corresponds to an n of 2.

Applying F2I with Delta- and VLI encoding results in an average compression ratio of 3.16 (See Table 3).

We have dismissed two other approaches: The authors of [46] are utilizing XOR-based compression. Their method is optimized for repeating values and can encode them with a single bit. The reference data consist of 59% repetitions. Deducting them from their results gives a compression ratio of 2.09. Furthermore, we have validated the strategy presented by Lindstrom and Isenburg [45]. It relies on a predictor that benefits from repeated values as well. Applying it to our dataset resulted in an average compression ratio of 1.24.

4. **String:** The majority of string-producing sensors yield low cardinality ENUMs. We apply a dictionary compression scheme to them. The strings are stored once within the dictionary, and the time series only contains dictionary indexes. An average compression ratio of 3.53 was achieved (See Table 3).

We have applied multiple compression algorithms to the preprocessed data and found that preprocessing improves the compression ratio of every algorithm (See Table 4). We chose to use Gzip [47] due to its low resource requirements and wide availability. Our preprocessing achieves an average compression ratio of 2.99 (See Table 4). Applying Gzip afterward improves the ratio to 4.82 while using Gzip alone only results in a ratio of 1.84 (See Table 4).

Table 3. Performance of the preprocessing methods.

Primitive	Preprocessing	Average Size	Ratio
Timestamp ¹	None	5563 Byte	-
	Delta + VLI	1748 Byte	3.18
	DOD + VLI	1518 Byte	3.66
Integer ²	None	1069 Byte	-
	VLI	445 Byte	2.40
	Delta + VLI	284 Byte	3.76
	DOD + VLI	284 Byte	3.76
Float ³	None	7050 Byte	-
	F2I + VLI	3239 Byte	2.18
	F2I + Delta + VLI	2228 Byte	3.16
	F2I + DOD + VLI	2243 Byte	3.14
String ⁴	None	120 Byte	-
	Dictionary + VLI	34 Byte	3.53

The results have been computed from datasets acquired from a real vehicle that was driven on a public street. ¹ 291 sequences; 39 sensors; 269,678 samples. ² 109 sequences; 12 sensors; 29,143 samples. ³ 137 sequences; 11 sensors; 241,231 samples. ⁴ 127 sequences; 17 sensors; 2330 samples.

Table 4. Combining our preprocessing strategy with generalized compression algorithms.

Algorithm	Preprocessing Applied		Time (ms)
	No (Ratio)	Yes (Ratio)	
None	1.00	2.99	-
Gzip [47]	1.84	4.82	38.08
bzip2 [48]	1.87	5.05	21.71
LZMA [49]	2.68	5.36	121.32
LZ4 [50]	1.43	4.05	82.00
Brotli [51]	2.60	5.59	762.41
zstd [52]	1.97	5.01	377.04
zlib [53]	1.84	4.83	38.92

The compression ratios are related to the average size of a chunk (182 kB). The underlying dataset was acquired from a real vehicle. It contains 589,904 samples from 39 different sensors.

4.6. Chunk Processing

Our system is processing chunks to generate trip files and perform near real-time streaming. Both operations have different service level requirements. Trip file generation is not a time-critical operation, and thus large fluctuations in the execution time are tolerable. For streaming, processing must be as fast as possible, and prolonged outages are not acceptable. Therefore, we operate separated infrastructures for each operation, aligned to the appropriate service levels. This separation allows us to make data processing more cost-effective.

Both infrastructures use the same message-based architecture (See Figure 8). In this architecture, each message represents an executable job that is self-contained and idempotent. The messages are consumed and processed by a server pool whose capacity is automatically adjusted. There is a persistent message broker that handles job scheduling and thus decouples scheduling from execution. The decoupling enables us to prevent the overloading of individual servers. Each server will only execute a limited number of jobs in parallel. If job creation frequency exceeds the pool's processing capacity, the broker will queue the excess. Job scheduling is possible even if there are no servers in the pool.

We adjust the processing pool capacity based on the number of messages scheduled for execution and currently being executed. For this purpose, we determine how many servers are required to process these messages in parallel. In addition, we grant each server a backlog that affects scaling aggressiveness (See Equation (1)). A larger backlog tends to increase average processing time but suppresses erratic upscaling.

$$required_servers = messages / per_server_limit * backlog_modifier \quad (1)$$

The trip file creation infrastructure utilizes preemptible Compute Engine instances for message processing. These instances provide a cost advantage but have a limited lifetime, and the cloud provider can terminate them at any time. In addition, there may be periods when preemptible instances are not available due to lack of supply [78]. Furthermore, we use a higher backlog modifier and thus accept longer delays in execution. For the streaming infrastructure, we use regular Compute Engine instances and a reduced backlog modifier.

The vehicle performs a chunk transmission in two phases. Initially, it sends the chunk associated metadata and receives a signed Cloud Storage URL in response. The chunk is then uploaded via the received URL. We create a Datastore object for each trip to store the associated chunk metadata and Cloud Storage references. Its key is composed of the trip identifier, a random value that ensures even distribution across the Datastore keyspace.

Our cloud platform schedules the trip file creation upon receiving a *commit* message from the vehicle. For this purpose, we load the trip-associated Datastore object to access the Cloud Storage references and metadata. These fields are used to create a message that we send to the message broker of the corresponding infrastructure. During processing, a server from the pool loads all trip chunks and merges them. It creates a separate file for each contained task and forwards it to another system for storage (See Figure 8).

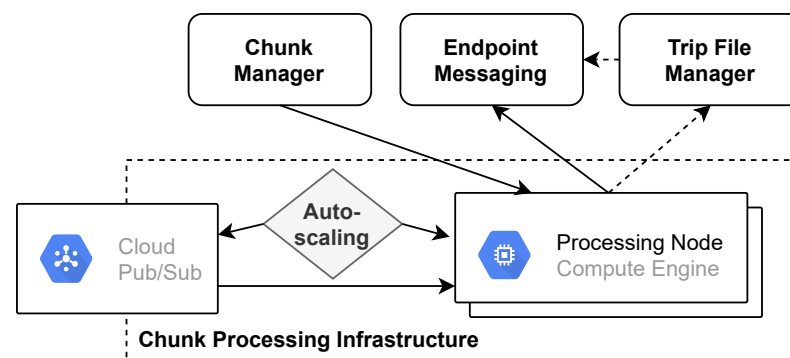


Figure 8. A message-based chunk processing infrastructure architecture: There is a pool of stateless processing nodes to handle the messages. Each message represents an executable job. We use this architecture to perform streaming or to generate trip files.

The upload of a streaming chunk immediately triggers its processing. The Cloud Storage bucket sends a message upon receiving the chunk. A random server from the processing pool loads the chunk and forwards the data using endpoint messaging.

Chunk receive capacity is limited to one per trip and second. This limitation originates from the single Datastore item utilized to store trip-related chunk metadata and Cloud Storage references. There is no limitation on concurrent trips due to the randomized Datastore keys and the fact that the throughput of Cloud Storage and Pub/Sub is not limited.

4.7. Chunk Processing Fault Tolerance

Chunk processing is a distributed transaction spanning multiple system boundaries. Individual failures may not result in data loss or a corrupted state. We have developed a transaction procedure that relies on passing multiple savepoints sequentially. Each savepoint represents a self-contained and consistent state. Thus, reaching a new savepoint enables the deletion of the previous state. Operations to reach a new savepoint are designed to be idempotent and repeated until they eventually succeed.

The vehicles transmit their data via a three-phase protocol. In this protocol, the sequence of operations is essential. For example, a *commit* may only occur when all chunks are transmitted (See Section 3.4). The vehicle manages the corresponding operations in a queue and executes them sequentially using the first-in-first-out strategy. An operation is only removed from the queue after its successful execution. If an error occurs, the vehicle delays the next iteration by applying an exponential backoff. The vehicle persists the queue

to a local storage device upon shutdown. It restores the queue and resumes execution when the car is started again.

The cloud platform receives the vehicle data idempotently and performs deduplication for this purpose. We use deterministic key generation to ensure that repeated invocations of the same operation will affect the same objects within our system. The initialization of a trip results in creating a cloud task, which eventually executes the timeout. We name the task by combining the trip identifier and the VIN. Cloud Tasks performs deduplication based on the task name, so reinitialization does not yield another timeout task [68].

Repeated chunk transmissions are also directed to the same Cloud Storage object. Additionally, we only store distinct references within the associated reverse index. Thus, we can avoid a continuous object expansion caused by a repeated transmission error. The vehicle only considers a *commit* to be successful if the cloud platform has created and persisted a message to initiate the trip file processing. Otherwise, it repeats the *commit* as already described.

By persisting the chunk processing message, we can guarantee its eventual processing. We obtain this guarantee by utilizing the Pub/Sub message redelivery strategy. Pub/Sub requires an explicit ACK for each message. Otherwise, it applies an exponential backoff and schedules the message for redelivery [69]. We ACK a message only in case of successful execution. The execution itself is designed to be idempotent. Thus, it does not alter or delete the associated chunks. Chunk deletion is carried out by expiry instead, for which we use a multiple of the Pub/Sub message retention time. Chunk processing represents the final savepoint. It encapsulates external calls, such as endpoint messaging. If these calls fail, the whole processing operation is eventually re-executed.

The trip file generation infrastructure performs deduplication as part of the processing. Immediately after a server has received a job, it checks whether a Datastore object exists whose key is composed of the trip identifier and the VIN. If this object exists, it skips the job. Otherwise, it executes the job and creates the object after successful execution. There is no mutex to prevent this logic from running in parallel so that race conditions can occur. In reality, most duplicate calls are filtered. The applied logic does not limit processing capacity, as the trip identifier is a random value with high entropy. Thus, the used objects are evenly distributed within Datastore.

As a result, the savepoints ensure that a wide variety of system components can fail without causing any data loss. However, chunk and message expiry require us to act timely on processing related outages. We must address them before expiration to avoid data loss. Thus, we continuously monitor related Pub/Sub metrics such as backlog size and oldest message age.

4.8. Trip-File Management

The Trip File Manager is responsible for persisting trip files and providing access to them. It stores them on Cloud Storage and manages their metadata as Datastore objects. There can be parallel and duplicate calls to the provided functionality (See Figure 9). Thus, there is a high risk of inconsistencies. Any inconsistency may constitute a data protection incident in the context of the GDPR and can have serious consequences. To prevent inconsistencies, we have adjusted the order of our operations and use Cloud Tasks to ensure their eventual execution. The following causes for inconsistency were identified and addressed by us:

1. The Trip File Manager may not accept files associated with revoked consents.
2. The revocation of consent must eventually result in the removal of all associated files without any leftovers.
3. A previously deleted file must not become available after its repeated submission.

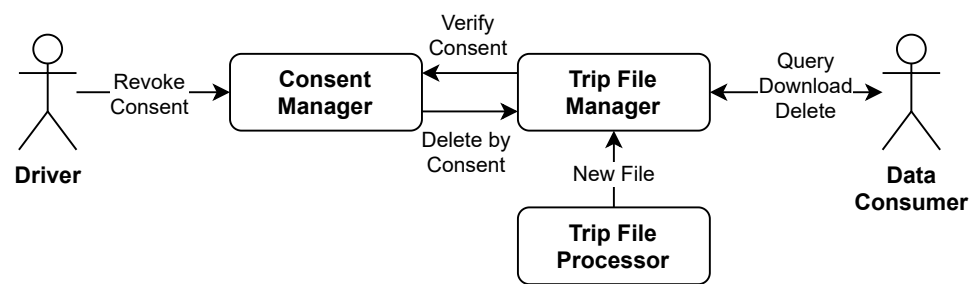


Figure 9. Interactions of the Trip File Manager with other system components or stakeholders. They may be performed concurrently and have the potential to cause inconsistencies.

When a data subject revokes consent, we first block the creation of additional files before initiating the deletion of existing files. This order enables a one-time sequential deletion. Upon receiving a revocation request, we create a cloud task that encapsulates both operations and eventually executes them in the given order. We perform the blocking by adding a revocation flag to the Datastore object storing the consent. The deletion is initiated by creating another task, which then performs the same.

The deletion task executes a Datastore query, which targets all files associated with the consent. It then fetches only a subset of the result and creates another task containing the query's cursor. Afterward, it sequentially deletes the individual files contained in the subset. The additional task does the same but utilizes the cursor to skip previous files. We thus distribute the deletion across multiple servers, each handling a small fraction of the associated files.

We create a cloud task for each trip file to be deleted. First, it turns the associated Datastore object into a tombstone by adding a flag. The object thus becomes invisible to user-facing queries, and the associated file can no longer be accessed. In addition, the tombstone blocks the re-creation of the associated file. The task deletes the files afterward from Cloud Storage.

The trip file manager rejects creating a trip file if an associated consent was revoked or a tombstone is present. It initially writes the file to Cloud Storage. Afterward, it performs a Datastore transaction fetching the consent objects and a potentially existing metadata object. If either a consent was revoked or a tombstoned metadata object exists, it cancels the transaction and deletes the associated Cloud Storage item. Otherwise, we create the metadata object within the same transaction. We ensure idempotency by creating deterministic Datastore keys and Cloud Storage names. The operation is only used by the trip file processor and thus retried until it eventually succeeds (See Section 4.7).

Preventing the identified inconsistencies has resulted in a limited trip file creation frequency for each consent. This limitation originates from the transaction that includes the metadata object and all associated consents. We have deemed this limitation acceptable: A consent usually relates to a single car, resulting in a limited creation frequency on its own.

5. Performance Evaluation

We have conducted a performance evaluation to assess the scaling capabilities of our system. For this purpose, we have simulated more than 200,000 vehicles simultaneously transmitting data to our system. As cars usually idle 95% of the time [79], this simulation is representative of a much larger fleet in reality. The scope of data collection thus corresponds to what commercially available products in the field of connected vehicles also have to achieve.

The performance evaluation is carried out by using real vehicle data: We have performed a 15-min drive with a Volkswagen T6.1 and recorded all data provided by the vehicle's data-access channels. The resulting recording was used to generate virtual vehicles using our simulators, which realistically repeat the drive (See Section 3.7).

We aim to answer the following questions via the performance evaluation:

1. Can the system react to elevated load with an automated increase in server capacity?
2. Is the increased server capacity able to keep processing time constant?
3. Can the system respond to reduced load by automatically reducing server capacity?

5.1. Setup

We have configured our system to make use of all its components. Therefore, we have created a project containing a single task that has enabled streaming and trip file generation. Since there are separate infrastructures for each of these kinds of data processing (See Figure 10), we can thus utilize both (See Section 4.6). In addition, we make use of endpoint messaging by configuring the project and task to use a Google Pub/Sub endpoint to receive the streaming data and trip file notifications (See Section 3.5). Furthermore, the simulators do not have any instructions pre-installed and must perform a complete instruction replication on startup. The other components, such as vehicle authentication or chunk compression, are integral to data processing and do not require separate configuration.

The cloud system is distributed across three infrastructures (See Figure 10): There are separate message based infrastructures to handle streaming and trip file processing. Furthermore, there is a REST backend providing all remaining functionality which is served via App Engine.

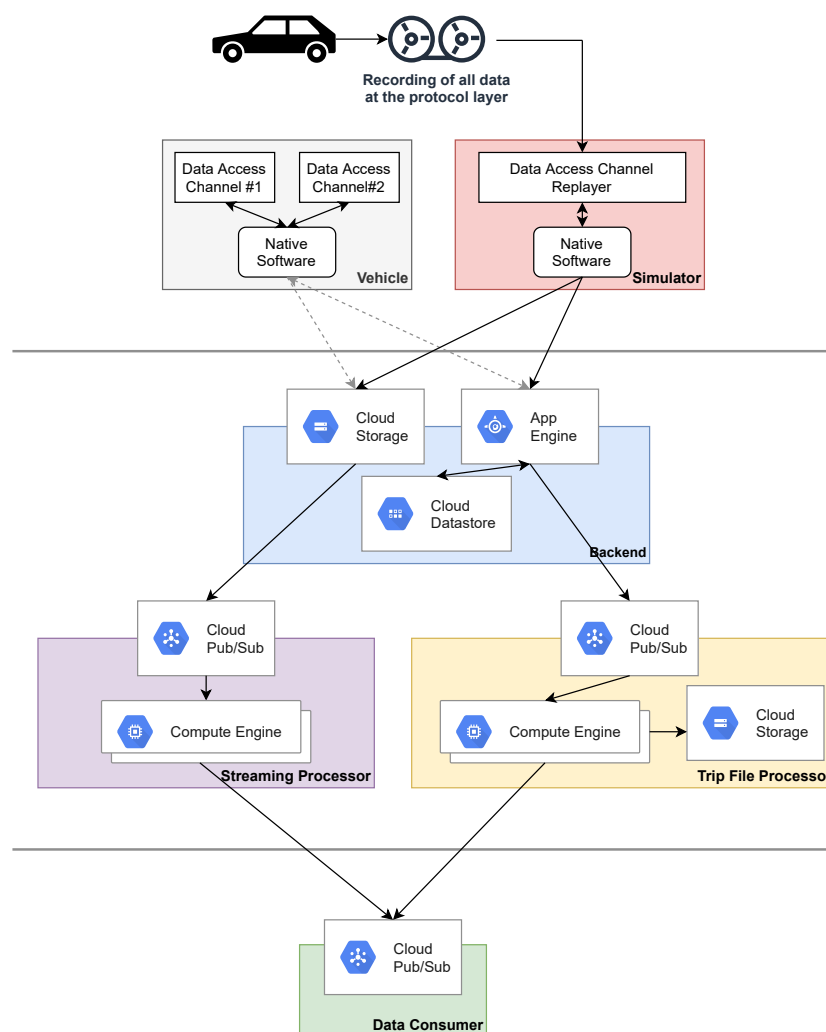


Figure 10. Setup of the performance evaluation: Using our simulators, we generate the same load as regular vehicles. The load is processed by three different infrastructures, for which we perform separate monitoring.

Streaming and trip file infrastructures are configured to use e2-highcpu-2 Compute Engine instances. Each instance may process up to 20 streaming chunks or 17 trip aggregations in parallel. We have set the backlog modifiers to 30 and 368 for streaming and trip processing, respectively (See Section 4.6). The backend is configured to use F1 instances with a target CPU utilization of 70% and a limit of 15 concurrent requests.

Performance monitoring is conducted by using Cloud Monitoring [80]. We record the number of active vehicles and the server count of each infrastructure. For the backend, we record the average request latency and request volume. Furthermore, we capture the average processing time and the number of unprocessed messages for the message-based infrastructures.

5.2. Results

We have found that the first and third questions can be answered positively. All monitored infrastructures were able to increase server capacity in response to the increased load. In addition, server capacity was reduced in response to the drop in load (See Figure 11).

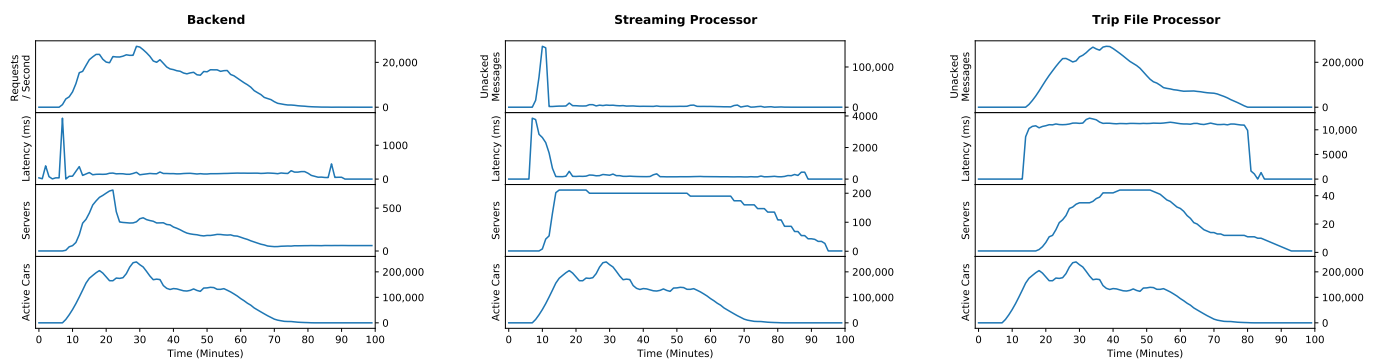


Figure 11. Results of a performance evaluation that was conducted by simulating over 200,000 concurrent cars. Performance metrics have been recorded for each of our infrastructures. It can be seen that the applied load causes an initial latency spike on the App Engine and Streaming infrastructure. This results from the rapidly increasing load for which insufficient capacity is available at this moment. The latency returns to normal after the capacity was automatically increased. It remains continuously low and thus enables time critical processing operations even at peak load. The latency of the Trip File Processor is elevated but stays on the same level. Its operations are not considered to be time critical. Thus, we have applied a less aggressive scaling strategy.

The second question can also be answered positively. There were initial spikes in latency on the backend and streaming infrastructure. They were caused by insufficient server capacity to handle the rapidly increasing load. After an automated adjustment, the latency remained continuously low even at peak load. At that time, the average streaming processing latency was 269 ms, while the backend had an average request latency of 190 ms. Trip file processing latency was consistently higher due to a less aggressive scaling strategy. This result is anticipated, as we consider the creation of trip files a time-insensitive operation (See Section 4.6).

A summary of the measured latencies can be found in Table 5.

Table 5. The latency of the individual infrastructures during the experiment.

Infrastructure	Mean	Latency (ms)		STD. Dev.
		99th Percentile	95th Percentile	
Backend	221	471	230	370
Streaming	192	1220	350	263
Trip Files	11,329	14,020	12,997	880

6. Discussion

6.1. Demand-Driven Data Acquisition

The presented demand-driven data acquisition makes large fleets accessible to individual data consumers more efficiently than previous methods. Instead of acquiring a static set of sensors from each vehicle, such as [16,18–25,30–32], we minimize the scope of data acquisition for each vehicle individually. This optimization facilitates access to subsets of the total fleet and sensor capacity, as the remaining data do not need to be transmitted and processed. Thus, operational costs no longer depend primarily on fleet size but rather on the actual data consumption of the data consumers. This is not the case with any of the other works.

Minimizing the scope of data acquisition is based on the *demand* specified by the data consumers. For this purpose, they manage so-called tasks. These tasks can include conditions that are evaluated locally on the vehicles, enabling conditional data collection. In addition, the options for data processing are part of each task. By coupling these options to the condition, dynamic choice of data processing is possible. Some of the presented systems can also evaluate conditions. However, they evaluate it on their backends and utilize the condition to yield events [31,81]. None of the other systems can make data collection and data processing dependent on a condition evaluated on the individual vehicles.

We can process personal data in a GDPR compliant manner. The vehicles do not transmit any personal data without the prior consent of the data subject. Many of the present works do not consider consent handling, e.g., [16,19–21,26,28,29]. The authors of [17] utilize a smartphone app that asks for permission before processing data. Some works acquire data without prior consent and request consent to access the data already stored on their platforms [22,23]. By separating consent and *demand*, we enable the data consumers to continuously adjust their data collection within the boundaries of the consent—even after it was given. In addition, we can address individual vehicles for data acquisition by utilizing the identity signed on to the car. No other system enables data consumers to adjust their collection of personal data for existing consents or tracking data subjects across different vehicles.

Furthermore, the presented system is a suitable low-level architecture to acquire and process non-personal data. The consent objects can be utilized to address individual vehicles to start data acquisition on them. All benefits of the system also apply to collecting non-personal data, e.g., the demand-driven data acquisition, scalability, abstraction layer, or data compression.

6.2. Abstraction Layer

The presented system has an abstraction layer to integrate a heterogeneous fleet through a single interface. There is an instruction for each sensor particularity existing in the fleet. The vehicles utilize them to self-determine their sensor capabilities and retrieve the associated data. In addition, conditional task execution can be performed depending on sensor availability. Many of the presented works do not consider the need for an abstraction layer [16–20,25–28,55]. The authors of [29] would like to establish new technical standards to provide interoperability across different vehicles. In [23], the authors propose a standard based on a common data format: Automakers should adhere to the format and use it to submit data to a data marketplace. Another work is utilizing a mapping table to map varying OBD data IDs with a common natural language identifier such as “SpeedKmHr” [22]. The authors of [21] make use of OBD discovery in order to determine sensor availability and report it to their platform.

The *data fields* exposed by our abstraction layer are similar to *virtual sensors*, which are software components that process and make data available from physical sensors or other software components [82]. There are implementations of *virtual sensors* that abstract sensor access via a unified interface. However, the software components of these implementations are strongly coupled to specific sensors and communicate directly with them [83,84]. Our abstraction layer is more decoupled from actual sensors and requires little to no modifi-

cations when integrating a new vehicle model that has new sensors. Compatibility can be retrofitted via the replicated instructions since the underlying protocols usually remain static. Furthermore, we enhance the provided data with metadata from the originating sensor as already suggested by [82].

6.3. Data Compression

We developed a method for data preprocessing and combined it with an existing compression algorithm. This combination resulted in a significant reduction in the size of transmitted data without any loss of information. Many other works do not consider compression [16,19–21,24–29,55]. In [18], the authors report using a lossless trajectory compression method. Another work is utilizing downsampling and histograms to reduce transmission size [23]. The authors of [54] also propose a lossy compression that restricts data transmission to anomalies detected locally on the vehicle. Finally, the authors of [22] mention using a proprietary message format to minimize the volume of transmitted data but do not elaborate on this issue. We have also compared two stand-alone methods for the compression of time series data and found that they are not efficient within our domain (See Section 4.5).

6.4. Data Processing

Our system can record all data generated during a trip and provide it to the data consumer. By using sequence numbers, we can validate the completeness of trip-related data. Some systems provide the latest value for each sensor and therefore require the data consumer to poll, which can cause value changes to be missed [36–38]. Other systems have a streaming interface to push value changes directly to the data consumer. However, the consumer cannot determine whether all data of a trip were received [33–35]. In addition, some platforms store every received sensor value but cannot determine the end of a trip. Therefore, they cannot reliably generate trip aggregates [18,19,21–23]. The authors of [23] have also mentioned the use of sequence numbers to determine the completeness and correctness of data. However, they do not describe the consequence of missing sequence numbers. Finally, Refs. [18,22] have identified "data loss" as a common issue within this area.

We have implemented the processing of each trip's data as independent and fault-tolerant transactions. Multiple savepoints enable operations to be retried upon errors until eventual success. This implementation prevents data loss during server failures, connection problems, or a car shutdown. The authors of [22,27] utilize a vehicle-side buffer to repeat transmissions in case of errors. Only [22] uses a persistent buffer that can survive a shutdown. The other works do not describe strategies to avoid data loss in case of transmission errors [16,18–21,23–26,28,29,55]. Our implementation moves beyond preventing transmission errors and extends to the subsequent data processing.

Our implementation achieves scalability through sharding. For this purpose, we exploit the internal data distribution of the utilized cloud technologies like Datastore. Our servers are stateless, except for caching. We can thus distribute requests evenly and randomly among them and adjust server count on demand. The statelessness also enables us to use cost-effective preemptible instances for trip file processing. In [26], the author has identified the acquisition and processing of vehicle data as a scaling issue. He is suggesting the use of cloud technology. The authors of [22] utilize a SQL Database to store the acquired data and perform partitioning by creating a separate table for every car. The database is hosted on a cloud server. Two other works also state to use cloud servers [19,24]. The authors of [16,18] make use of different storage systems. The most requested data are stored in memory, while pre-processed data are stored in a relational database, and archival takes place in a NoSQL database. Most works do not consider scalability [20,23,25,27–29,55]. Furthermore, none of the systems can perform automatic scaling by adjusting its server count. In addition, no system does utilize preemptible instances.

6.5. Data Authenticity

We can verify the authenticity of the received data because the vehicles authenticate themselves via an embedded certificate. In addition, the driver can log in to his car and thus effectively authenticates against our system. Many of the presented works do not include procedures to verify the authenticity of received data ([16,18–23,26–28,55]). The authors of [29] also identified the need to verify the vehicle identity but do not specify a method for it. Another work also proposes to issue a certificate to each car for authentication [85].

6.6. Performance Evaluation

We performed a simulation by virtualizing the data-access channels with a recording from an actual vehicle. By running our car application combined with the virtualized access channels, we could simulate over 200,000 concurrent vehicles. The average processing time of streaming data is 269 ms at peak load. In [18], the authors injected a historical dataset into their system. It can handle over 30,000 concurrent vehicles and has an average processing time of fewer than 2 s. The authors of [59] evaluated the processing performance of the commercial AutoPi Cloud and measured an average processing and transmission time of 3 s. Another system handles data from 5500 vehicles during production [24]. The other presented systems do not perform a performance evaluation [19,21–23,25].

7. Conclusions

In this work, we present a system that enables the opening of large fleets of heterogeneous vehicles for individual data consumers. We have found that, in practice, different data consumers have diverging data requirements in terms of required sensors. In order to handle such access patterns efficiently, we have developed an original demand-driven data acquisition strategy. It reduces data acquisition for each vehicle individually to only transmit data required by the data consumers (*demand*). Furthermore, we map individual *demands* to individual vehicles; thus, each car potentially transmits different sensors. Our design enables *demand* adjustments even after the data subjects consent by directing the consents to an immutable maximum collection scope (project). The data consumers express their *demand* as a set of tasks that include locally evaluated conditions and processing options.

Our implementation consists of a cloud platform and a software component we have integrated natively with a current vehicle platform of the Volkswagen Group. Automakers usually operate highly heterogeneous fleets, for which we can provide access via a single interface. For this purpose, we have developed an abstraction layer based on the distinct *data fields* vehicles can provide. Data transmitted by the vehicles is compressed by combining custom data preprocessing with an existing generalized compression algorithm. A validation with sensor data from an actual vehicle has shown this strategy to be significantly more efficient than existing generalized compression algorithms alone. The cloud platform provides data streaming or performs a recording of individual trips. We have explicitly partitioned the data flows within the platform to distribute them evenly across multiple systems. This partitioning scheme enables us to apply horizontal scaling, which we have implemented as an automated operation: Servers are added or removed based on the current system load. The processing of vehicle data is implemented as distributed transactions. We persist our data on redundant public cloud services; our servers are stateless and can fail without data loss. This statelessness allows us to make use of preemptible instances with low availability for selected data processing jobs. The overall system performance was evaluated by simulating data acquisition from over 200,000 cars simultaneously.

By integrating our system into a current vehicle platform of the Volkswagen Group, we have demonstrated the capabilities of modern mainstream cars. We have thus shown that it is possible to transform an automaker's fleet into a globally distributed sensor network by deploying software. However, the resulting architecture is not limited to Volkswagen vehicles. It is essentially a system to acquire data from large distributed and heterogeneous

multi-sensor networks. As such, other automakers or operators of large sensor networks may also utilize it.

We believe there is great potential in extending this work with an intelligent vehicle selection to improve efficiency for collecting non-personal data (See Section 3.8). Personal data also contain information that is not related to the identity of the data subject. Integration of differential privacy [86,87] or local differential privacy [88] into the proposed system would make it available while preserving user privacy. Furthermore, integrating blockchain into consent handling could enhance trackability and verifiability of personal data processing [89].

Author Contributions: Conceptualization, methodology, software, writing—original draft preparation, visualization P.M.; writing—review and editing T.G. and A.F.; funding acquisition T.G. and M.N.; supervision W.N. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by the German Federal Ministry for Economic Affairs and Energy (Grant No. 01 MD 19007A).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Conflicts of Interest: The Volkswagen Group employs all authors except W.N. The Group can gain a financial advantage by selling the sensor data of its vehicle fleet.

References

1. Automotive Usage Based Insurance Market Forecast to 2027—COVID-19 Impact and Global Analysis by Technology Fitted (Smartphones, Black Box, and Dongles); and Policy Type (Pay-As-You-Drive (PAYD) and Pay-How-You-Drive (PHYD)); and Geography. Available online: <https://www.theinsightpartners.com/reports/automotive-usage-based-insurance-market/> (accessed on 8 December 2020).
2. Mai, A.; Schlesinger, D. A Business Case for Connecting Vehicles. Available online: https://www.cisco.com/c/dam/en_us/about/ac79/docs/mfg/Connected-Vehicles_Exec_Summary.pdf (accessed on 8 December 2020).
3. Ullah, S.; Kim, D.H. Lightweight driver behavior identification model with sparse learning on in-vehicle can-bus sensor data. *Sensors* **2020**, *20*, 5030. [CrossRef]
4. Díaz-álvarez, A.; Clavijo, M.; Jiménez, F.; Serradilla, F. Inferring the driver's lane change intention through lidar-based environment analysis using convolutional neural networks. *Sensors* **2021**, *21*, 475. [CrossRef] [PubMed]
5. Jeon, Y.; Kim, B.; Baek, Y. Ensemble CNN to Detect Drowsy Driving with In-Vehicle Sensor Data. *Sensors* **2021**, *21*, 2372. [CrossRef] [PubMed]
6. Young, R.; Fallon, S.; Jacob, P.; O'Dwyer, D. Vehicle Telematics and Its Role as a Key Enabler in the Development of Smart Cities. *IEEE Sens. J.* **2020**, *20*, 11713–11724. [CrossRef]
7. Delussu, F.; Imran, F.; Mattia, C.; Meo, R. Fuel Prediction and Reduction in Public Transportation by Sensor Monitoring and Bayesian Networks. *Sensors* **2021**, *21*, 4733. [CrossRef]
8. Zahid, M.; Chen, Y.; Jamal, A.; Memon, M.Q. Short term traffic state prediction via hyperparameter optimization based classifiers. *Sensors* **2020**, *20*, 685. [CrossRef]
9. Fox, A.; Kumar, B.V.; Chen, J.; Bai, F. Multi-Lane Pothole Detection from Crowdsourced Undersampled Vehicle Sensor Data. *IEEE Trans. Mob. Comput.* **2017**, *16*, 3417–3430. [CrossRef]
10. Enriquez, D.; Bautista, A.; Field, P.; Kim, S.i.; Jensen, S.; Ali, M.; Miller, J. CANOPNR: CAN-OBd programmable-expandable network-enabled reader for real-time tracking of slippery road conditions using vehicular parameters. In Proceedings of the 15th International IEEE Conference on Intelligent Transportation Systems (ITSC), Anchorage, AK, USA, 16–19 September 2012; pp. 260–264. [CrossRef]
11. Bishop, J.D.; Stettler, M.E.; Molden, N.; Boies, A.M. Engine maps of fuel use and emissions from transient driving cycles. *Appl. Energy* **2016**, *183*, 202–217. [CrossRef]
12. Lee, G.Y.; Cha, K.J.; Kim, H.J. Designing the GDPR Compliant Consent Procedure for Personal Information Collection in the IoT Environment. In Proceedings of the 2019 IEEE International Congress on Internet of Things (ICIOT), Milan, Italy, 8–13 July 2019; pp. 79–81. [CrossRef]
13. Vallet, F. The GDPR and Its Application in Connected Vehicles—Compliance and Good Practices. In *Electronic Components and Systems for Automotive Applications*; Springer: Cham, Switzerland, 2019; pp. 245–254. [CrossRef]
14. AutoPi Documentation. Available online: <https://docs.autopi.io/> (accessed on 13 October 2021).
15. Freematics Homepage. Available online: <https://freematics.com/> (accessed on 13 October 2021).
16. Peppes, N.; Alexakis, T.; Adamopoulou, E.; Demestichas, K. Driving Behaviour Analysis Using Machine and Deep Learning Methods for Continuous Streams of Vehicular Data. *Sensors* **2021**, *21*, 4704. [CrossRef]

17. Khandakar, A.; Chowdhury, M.E.; Ahmed, R.; Dhib, A.; Mohammed, M.; Al-Emadi, N.A.M.A.; Michelson, D. Portable System for Monitoring and Controlling Driver Behavior and the Use of a Mobile Phone While Driving. *Sensors* **2019**, *19*, 1563. [CrossRef]
18. Zhang, M.; Wo, T.; Xie, T.; Lin, X.; Liu, Y. CarStream: An industrial system of big data processing for Internet-of-Vehicles. *Proc. VLDB Endow.* **2017**, *10*, 1766–1777. [CrossRef]
19. Hussain, S.; Mahmud, U.; Yang, S. Car e-Talk: An IoT-enabled Cloud-Assisted Smart Fleet Maintenance System. *IEEE Internet Things J.* **2021**, *8*, 9484–9494. [CrossRef]
20. Silva, M.; Signoretti, G.; Andrade, P.; Silva, I.; Ferrari, P. Towards a customized vehicular maintenance based on 2-layers data-stream application. In Proceedings of the 2021 IEEE International Workshop on Metrology for Automotive (MetroAutomotive), Bologna, Italy, 1–2 July 2021; pp. 193–198. [CrossRef]
21. Silva, M.; Vieira, E.; Signoretti, G.; Silva, I.; Silva, D.; Ferrari, P. A Customer Feedback Platform for Vehicle Manufacturing Compliant with Industry 4.0 Vision. *Sensors* **2018**, *18*, 3298. [CrossRef]
22. Wilhelm, E.; Siegel, J.; Mayer, S.; Sadamori, L.; Dsouza, S.; Chau, C.K.; Sarma, S. Cloudthink: A scalable secure platform for mirroring transportation systems in the cloud. *Transport* **2015**, *30*, 320–329. [CrossRef]
23. Pillmann, J.; Wietfeld, C.; Zarcu, A.; Raugust, T.; Alonso, D.C. Novel common vehicle information model (CVIM) for future automotive vehicle big data marketplaces. In Proceedings of the 2017 IEEE Intelligent Vehicles Symposium (IV), Los Angeles, CA, USA, 11–14 June 2017; pp. 1910–1915. [CrossRef]
24. Rehrl, K.; Henneberger, S.; Leitinger, S.; Wagner, A.; Wimmer, M. Towards a National Floating Car Data Platform for Austria. In Proceedings of the 25th World Congress on Intelligent Transportation Systems (ITS), Copenhagen, Denmark, 17–21 September 2018; pp. 1–10.
25. Xiao, Z.; Li, F.; Wu, R.; Jiang, H.; Hu, Y.; Ren, J.; Cai, C.; Iyengar, A. TrajData: On Vehicle Trajectory Collection with Commodity Plug-and-Play OBU Devices. *IEEE Internet Things J.* **2020**, *7*, 9066–9079. [CrossRef]
26. Liu, N. Internet of Vehicles: Your Next Connection. Available online: https://www.huawei.com/mediafiles/CORPORATE/PDF/Magazine/WinWin/HW_110848.pdf (accessed on 20 October 2021)
27. Miche, M.; Bohnert, T.M. The Internet of Vehicles or the Second Generation of Telematic Services. *ERCIM News* **2009**, *77*, 43–45.
28. Bonomi, F. The Smart and Connected Vehicle and the Internet of Things. In Proceedings of the Workshop on Synchronization in Telecommunication Systems, San Jose, CA, USA, 16–18 April 2013.
29. Contreras-Castillo, J.; Zeadally, S.; Guerrero-Ibanez, J.A. Internet of Vehicles: Architecture, Protocols, and Security. *IEEE Internet Things J.* **2018**, *5*, 3701–3709. [CrossRef]
30. Zubie Platform Documentation. Available online: <https://zubie.com/developer/> (accessed on 17 December 2020).
31. Vinli Services Documentation. Available online: <http://docs.vin.li/en/latest/> (accessed on 17 December 2020).
32. Munic Documentation. Available online: https://store.munic.io/documentations/get_started (accessed on 17 December 2020).
33. Otonomo Platform. Available online: <https://otonomo.io/> (accessed on 17 December 2020).
34. Caruso Platform. Available online: <https://www.caruso-dataplace.com/> (accessed on 17 December 2020).
35. Smartcar Platform. Available online: <https://smartcar.com/> (accessed on 17 December 2020).
36. Mercedes Benz API Platform. Available online: <https://developer.mercedes-benz.com/products> (accessed on 17 January 2020).
37. BMW CarData. Available online: <https://bmw-cardata.bmwgroup.com/thirdparty/public/car-data/overview> (accessed on 17 December 2020).
38. Ford Connected Vehicle API. Available online: <https://developer.ford.com/fordconnect> (accessed on 17 December 2020).
39. PSA B2B Web API. Available online: <https://developer.groupe-psa.io/webapi/b2b/overview/about/> (accessed on 17 December 2020).
40. Foster, I.; Koscher, K. Exploring Controller Area Networks. *Login Usenix Mag.* **2015**, *40*, 6–10.
41. Marchetti, M.; Stabili, D. READ: Reverse engineering of automotive data frames. *IEEE Trans. Inf. Forensics Secur.* **2019**, *14*, 1083–1097. [CrossRef]
42. Young, C.; Svoboda, J.; Zambreno, J. Towards Reverse Engineering Controller Area Network Messages Using Machine Learning. In Proceedings of the 2020 IEEE 6th World Forum on Internet of Things (WF-IoT), New Orleans, LA, USA, 2–16 June 2020; pp. 1–6. [CrossRef]
43. Shaily, S.; Krishnan, S.; Natarajan, S.; Periyasamy, S. Smart driver monitoring system. *Multimed. Tools Appl.* **2021**, *80*, 25633–25648. [CrossRef]
44. Palomino, J.; Cuty, E.; Huanachin, A. Development of a CAN Bus datalogger for recording sensor data from an internal combustion ECU. In Proceedings of the 2021 IEEE International Workshop of Electronics, Control, Measurement, Signals and Their Application to Mechatronics (ECMSM), Liberec, Czech Republic, 21–22 June 2021; pp. 1–4. [CrossRef]
45. Lindstrom, P.; Isenburg, M. Fast and efficient compression of floating-point data. *IEEE Trans. Vis. Comput. Graph.* **2006**, *12*, 1245–1250. [CrossRef]
46. Pelkonen, T.; Franklin, S.; Teller, J.; Cavallaro, P.; Huang, Q.; Meza, J.; Veeraraghavan, K. Gorilla: A Fast, Scalable, in-Memory Time Series Database. *Proc. VLDB Endow.* **2015**, *8*, 1816–1827. [CrossRef]
47. Deutsch, L.P. GZIP File Format Specification Version 4.3. RFC 1952, 1996. Available online: <https://datatracker.ietf.org/doc/html/rfc1952> (accessed on 20 October 2021).
48. Seward, J. bzip2 Homepage. Available online: <https://sourceware.org/bzip2/> (accessed on 30 August 2021).
49. Pavlov, I. LZMA Software Development Kit (SDK). Available online: <https://www.7-zip.org/sdk.html> (accessed on 30 August 2021).

50. Collet, Y. LZ4 Frame Format Description. Available online: https://github.com/lz4/lz4/blob/dev/doc/lz4_Frame_format.md (accessed on 30 August 2021).
51. Alakuijala, J.; Farruggia, A.; Ferragina, P.; Kliuchnikov, E.; Obryk, R.; Szabadka, Z.; Vandevenne, L. Brotli: A General-Purpose Data Compressor. *ACM Trans. Inf. Syst.* **2018**, *37*. [CrossRef]
52. Cottet, Y.; Kucherawy, M. Zstandard Compression and the 'Application/zstd' Media Type. Available online: <https://www.rfc-editor.org/rfc/rfc8878.txt> (accessed on 20 October 2021).
53. Deutsch, P.; Gailly, J.-L. ZLIB Compressed Data Format Specification Version 3.3. Available online: <https://www.rfc-editor.org/rfc/rfc1950.txt> (accessed on 20 October 2021).
54. Signoretti, G.; Silva, M.; Andrade, P.; Silva, I.; Sisinni, E.; Ferrari, P. An Evolving TinyML Compression Algorithm for IoT Environments Based on Data Eccentricity. *Sensors* **2021**, *21*, 4153. [CrossRef]
55. Golestan, K.; Soua, R.; Karray, F.; Kamel, M.S. Situation awareness within the context of connected cars: A comprehensive review and recent trends. *Inf. Fusion* **2016**, *29*, 68–83. [CrossRef]
56. Road Vehicles—Extended Vehicle (ExVe) Web Services. Standard, International Organization for Standardization, Geneva, CH. 2019. Available online: <https://www.iso.org/standard/66978.html> (accessed on 28 September 2021).
57. Carloop Documentation. Available online: <https://carloop.readme.io/docs> (accessed on 13 October 2021).
58. Macchina Documentation. Available online: <https://docs.macchina.cc/> (accessed on 13 October 2021).
59. Ferrari, P.; Sisinni, E.; Bellagente, P.; Depari, A.; Flammini, A.; Pasetti, M.; Rinaldi, S. Experimental characterization of an IoV framework leveraging mobile wireless technologies. In Proceedings of the 2021 IEEE International Instrumentation and Measurement Technology Conference (I2MTC), Glasgow, UK, 17–20 May 2021; pp. 1–6. [CrossRef]
60. Finck, M.; Pallas, F. They who must not be identified—distinguishing personal from non-personal data under the GDPR. *Int. Data Priv. Law* **2020**, *10*, 11–36. [CrossRef]
61. Forgó, N.; Händold, S.; Schütze, B., The principle of purpose limitation and big data. In *New Technology, Big Data and the Law*; Springer: Singapore, 2017; pp. 17–42. [CrossRef]
62. Gruschka, N.; Mavroeidis, V.; Vishi, K.; Jensen, M. Privacy Issues and Data Protection in Big Data: A Case Study Analysis under GDPR. In Proceedings of the 2018 IEEE International Conference on Big Data (Big Data), Seattle, WA, USA, 10–13 December 2018; pp. 5027–5033. [CrossRef]
63. Patrick, L.; Martin, W. Volkswagen Infotainment Web Interface Protocol Specification (Viwi Protocol). W3c Member Submission, W3C. 2019. Available online: <https://www.w3.org/Submission/viwi-protocol/> (accessed on 25 January 2021).
64. Protocol Buffers Homepage. Available online: <https://developers.google.com/protocol-buffers/> (accessed on 19 October 2021).
65. FastAPI Homepage. Available online: <https://fastapi.tiangolo.com/> (accessed on 18 October 2021).
66. Google App Engine Documentation. Available online: <https://cloud.google.com/appengine> (accessed on 29 December 2020).
67. Google Datastore Documentation. Available online: <https://cloud.google.com/datastore> (accessed on 29 December 2020).
68. Google Tasks Documentation. Available online: <https://cloud.google.com/tasks> (accessed on 29 December 2020).
69. Google Pub/Sub Documentation. Available online: <https://cloud.google.com/pubsub> (accessed on 29 December 2020).
70. Google Compute Engine Documentation. Available online: <https://cloud.google.com/compute> (accessed on 25 January 2021).
71. Google Cloud Storage Documentation. Available online: <https://cloud.google.com/storage> (accessed on 29 December 2020).
72. Shapiro, M.; Pregoça, N.; Baquero, C.; Zawirski, M. Conflict-Free Replicated Data Types. In Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2011), Grenoble, France, 10–12 October 2011; pp. 386–400. [CrossRef]
73. Google Datastore—Features Documentation. Available online: <https://cloud.google.com/datastore/docs/firestore-or-datastore> (accessed on 25 January 2021).
74. Road Vehicles—Vehicle Identification Number (VIN)—Content and Structure. Standard, International Organization for Standardization, Geneva, CH, USA. 2009. Available online: <https://www.iso.org/standard/52200.html> (accessed on 28 September 2021).
75. Google Datastore—Best Practices Documentation. Available online: <https://cloud.google.com/datastore/docs/best-practices> (accessed on 18 January 2021).
76. Zhang, T.; Zuck, A.; Porter, D.E.; Tsafir, D. Apps Can Quickly Destroy Your Mobile's Flash: Why They Don't, and How to Keep It That Way. In Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys), Seoul, Korea, 17–21 June 2019; pp. 207–221. [CrossRef]
77. Rizzato, F. Germany's Rural 4G Users Still Spend One-Fourth of Their Time on 3G and 2G Networks. Available online: <https://www.opensignal.com/blog/2019/06/13/germanys-rural-4g-users-still-spend-one-fourth-of-their-time-on-3g-and-2g-networks> (accessed on 19 January 2021).
78. Costa, B.G.; Reis, M.A.S.; Araújo, A.P.; Solis, P. Performance and cost analysis between on-demand and preemptive virtual machines. In Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER), Funchal, Portugal, 19–21 March 2018; pp. 169–178. [CrossRef]
79. Frenken, K.; Juliet, S. Putting the sharing economy into perspective. *Environ. Innov. Soc. Transit.* **2017**, *23*, 3–10. 2017.01.003 [CrossRef]
80. Cloud Monitoring Homepage. Available online: <https://cloud.google.com/monitoring> (accessed on 19 October 2021).
81. PSA Monitors. Available online: <https://developer.groupe-psa.io/webapi/b2b/monitor/about/> (accessed on 17 December 2020).
82. Martin, D.; Kühl, N.; Satzger, G. Virtual Sensors. *Bus. Inf. Syst. Eng.* **2021**, *63*, 315–323. [CrossRef]

-
83. Ko, J.; Lee, B.B.; Lee, K.; Hong, S.G.; Kim, N.; Paek, J. Sensor Virtualization Module: Virtualizing IoT Devices on Mobile Smartphones for Effective Sensor Data Management. *Int. J. Distrib. Sens. Netw.* **2015**, *11*, 730762. [[CrossRef](#)]
 84. Madria, S.; Kumar, V.; Dalvi, R. Sensor Cloud: A Cloud of Virtual Sensors. *IEEE Softw.* **2014**, *31*, 70–77. [[CrossRef](#)]
 85. Guo, L.; Dong, M.; Ota, K.; Li, Q.; Ye, T.; Wu, J.; Li, J. A Secure Mechanism for Big Data Collection in Large Scale Internet of Vehicle. *IEEE Internet Things J.* **2017**, *4*, 601–610. [[CrossRef](#)]
 86. Nelson, B.; Olovsson, T. Introducing Differential Privacy to the Automotive Domain: Opportunities and Challenges. In Proceedings of the 2017 IEEE 86th Vehicular Technology Conference (VTC-Fall), Toronto, ON, Canada, 24–27 September 2017; pp. 1–7. [[CrossRef](#)]
 87. Wallace, B.; Goubran, R.; Knoefel, F.; Marshall, S.; Porter, M.; Harlow, M.; Puli, A. Automation of the Validation, Anonymization, and Augmentation of Big Data from a Multi-year Driving Study. In Proceedings of the 2015 IEEE International Congress on Big Data, New York, NY, USA, 27 June–2 July 2015; pp. 608–614. [[CrossRef](#)]
 88. Zhao, P.; Zhang, G.; Wan, S.; Liu, G.; Umer, T. A survey of local differential privacy for securing internet of vehicles. *J. Supercomput.* **2020**, *76*, 8391–8412. [[CrossRef](#)]
 89. Barati, M.; Rana, O. Tracking GDPR Compliance in Cloud-based Service Delivery. *IEEE Trans. Serv. Comput.* **2020**. 2020.2999559. [[CrossRef](#)]