The AutoSPADA platform: User-friendly edge computing for distributed learning and data analytics in connected vehicles

Internet of Things 30 (2025) 101480

Contents lists available at ScienceDirect

Internet of Things

journal homepage: www.elsevier.com/locate/iot

The AutoSPADA platform: User-friendly edge computing for distributed learning and data analytics in connected vehicles

Adrian Nilsson [a], Simon Smith [a], Jonas Hagmar [a], Magnus Önnheim [a], Mats Jirstrand [a,b,*]

[a] Fraunhofer-Chalmers Centre, Gothenburg, SE-412 88, Sweden
[b] Department of Electrical Engineering, Chalmers University of Technology, Gothenburg, SE-412 96, Sweden

ARTICLE INFO

Keywords:
Edge computing
Big data
Distributed systems
Data analytics
Automotive
Connected vehicles
IoT

ABSTRACT

Contemporary connected vehicles host numerous applications, such as diagnostics and navigation, and new software is continuously being developed. However, the development process typically requires offline batch processing of large data volumes. In an edge computing approach, data analysts and developers can instead process sensor data directly on computational resources inside vehicles. This enables rapid prototyping to shorten development cycles and reduce the time to create new business values or insights. This paper presents the design, implementation, and operation of the AutoSPADA (Automotive Software Platform for Advanced Distributed Analytics) edge computing platform for distributed data analytics. The platform's design follows scalability, reliability, resource efficiency, privacy, and security principles promoted through mature and industrially proven technologies. In AutoSPADA, computational tasks are general Python scripts, and we provide a library to, for example, read signals from the vehicle and publish results to the cloud. Hence, users only need Python knowledge to use the platform. Moreover, the platform is designed to be extended to support additional programming languages. AutoSPADA has been demonstrated using live vehicles in workshop sessions where external users wrote payloads exploring use cases related to durability and energy consumption in electric vehicles. This places AutoSPADA at a Technology Readiness Level (TRL) of 7 as defined by the European Union.

1. Introduction

Today, network-connected devices outnumber the global population threefold [1]. Around half of these are Internet of Things (IoT) devices such as connected vehicles, home automation systems, and industrial smart sensors. Moreover, connected vehicles are forecasted to be the fastest growing IoT application with 30% yearly growth between 2018 and 2023 [1]. This growth further exacerbates the problems faced in big automotive data [2]. For example, the per year data collection need of Volvo Car Corporation and Volvo Group Trucks Technology is already estimated to be in the order of exabytes [3]. As the data volume and velocity continue to increase, offline batch processing becomes increasingly challenging, inefficient, or even infeasible. Therefore, new methods and workflows are needed to transform the rich edge-generated data into insights and business value.

Edge computing is a paradigm that emerged in response to the surge in the number of IoT devices and the massive amount of data they produce. In edge computing, data is processed close to the source of the data rather than sent to the cloud for processing [4],

* Corresponding author at: Department of Electrical Engineering, Chalmers University of Technology, Gothenburg, SE-412 96, Sweden.
E-mail address: jirstran@chalmers.se (M. Jirstrand).

https://doi.org/10.1016/j.iot.2024.101480
Received 1 April 2024; Received in revised form 19 September 2024; Accepted 17 December 2024
Available online 24 December 2024

making it naturally positioned to protect the privacy of data owners. This paradigm can reduce data transfer and storage costs, lower energy consumption, and improve latency [5,6] and has led to the development of novel privacy-preserving algorithms [7], federated analytics [8], as well as new applications of artificial intelligence [9,10]. However, distributed edge computing applications face several system-level challenges such as scalability, device and data heterogeneity, and device reliability. For example, applications must continue to function even if the network is unavailable, and the cloud must be able to orchestrate a possibly large number of devices. Therefore, edge computing applications usually rely on a framework or platform with components deployed to both the cloud and the edge.

This paper presents the AutoSPADA (Automotive Software Platform for Distributed Analytics) edge computing platform for distributed data analytics. The platform aims to seamlessly connect data analysts with the data and computational resources of edge devices such as those found in contemporary vehicles. Users do not have to be Kubernetes experts or proficient in low-level or specialized programming languages. Instead, AutoSPADA users write ordinary Python programs, significantly lowering the barrier of entry. A data scientist, researcher, or engineer may already have Python scripts for offline data analysis that could be transformed into a program ready to be deployed to AutoSPADA edge clients. We provide users with a Python library to, e.g., read signal values directly from the edge device and publish results to the cloud.

AutoSPADA is designed to support a wide range of paradigms, including data aggregation, monitoring, and machine learning. The server infrastructure collects results from user-defined tasks that run on clients, after which users can access results either interactively through streaming channels or on demand. An important concern has been to enforce privacy and security throughout the platform to prevent attacks by a third party, or even from malicious tasks. The architecture is also designed to be scalable since the platform is intended to be deployable to many clients.

Industries have utilized data for a long time, but the degree to which data is incorporated into their businesses varies. The authors of [11] propose a five-stage data maturity model within the embedded systems domain, including the automotive industry. At the first stage, data is used for quality assurance and operational diagnostics—some automotive companies have collected data for this purpose for decades [11]. However, customer data holds potential beyond simple diagnostics. At the higher stages, this data is used to create value for the individual customer, the whole customer base, or even an entirely different customer base. Under this model, AutoSPADA should be viewed as a platform for businesses working to advance their data practices and progress toward higher data maturity stages. In an automotive context, this means that AutoSPADA complements fleet management systems. Whereas fleet management systems excel at collecting data and monitoring the overall status of the fleet, AutoSPADA serves as a research platform enabling teams to explore, test, and develop new data-driven services. Examples of such services include dynamic route selection [12] and advanced driver support systems for, e.g., lane overtaking [13] and collision avoidance [14]. Hence, the AutoSPADA platform is mainly intended for rapid and iterative development of experimental software services rather than hosting production services.

This paper makes three primary contributions. First, we provide a thorough technical description of the AutoSPADA design and architecture in Sections 3 and 4. Second, we contribute a detailed pseudo-code implementation of our client node, which is the most challenging node to implement, in Section 4. Third, we present experimental results in Section 6 to assess the platform's inherent latencies, which are important to provide an interactive user experience.

2. Background

The AutoSPADA acronym is shared with the project in which it was developed, namely the Automotive Stream Processing and Distributed Analytics project [15], which is a continuation of the OODIDA (On-board/Off-board Distributed Data Analytics) project [3,16]. However, the meaning of the acronym has changed to reflect that the platform does not adhere to the stream processing paradigm of composing operations on data flows. Hereafter, *AutoSPADA* refers to the AutoSPADA platform — not the project — unless otherwise stated.

Although conceptually similar, the design of the AutoSPADA platform is in many ways different from that of OODIDA. For AutoSPADA, we wanted to create an edge computing platform characterized by a high degree of interactivity, flexibility, and ease of use. In particular, users should not be required to learn any new software tools, paradigms, or languages—the platform should be accessible to anyone comfortable writing Python code. Therefore, we have focused on enabling users to deploy general Python scripts directly to hardware at the edge. In contrast, OODIDA only had experimental support for deploying such scripts to edge clients [17], otherwise limiting users to a set of fixed-function aggregations.

A stated goal of AutoSPADA is to take the ideas and learnings from the OODIDA prototype to create a pre-production system designed and ready for deployment at scale. Specifically, we want to raise the Technology Readiness Level (TRL) [18] from an early prototype to a level where the platform should be tested in an operational environment, i.e., on a vehicle fleet. Taking this step places additional demands on supporting technologies and the overall design of the platform. The remainder of this section discusses the limitations of the design and technological choices made in OODIDA and argues that a revised design is required to reach the desired level of technological maturity.

2.1. Implementation language

OODIDA is a distributed application written in Erlang with an additional Python component on edge devices. In an Erlang application, processes can be executed seamlessly on any computational node. This facilitates the rapid development of distributed applications since the communication between processes is abstracted by the language. However, this choice of languages is not

ideal for realizing the design goals of the platform, which is also discussed to some degree in the OODIDA retrospective and future vision [16].

Using Erlang and Python in the OODIDA implementation has several limitations. One is the need for separate runtimes, specifically, the BEAM virtual machine and the Python interpreter. These runtimes offer portability but consume additional disk space and memory, which is especially undesirable in a constrained edge environment. Another limitation is that both Erlang and Python are dynamically typed languages. Using statically typed languages eliminates large classes of type errors at compile time that dynamically typed languages fail to detect. For AutoSPADA, we prioritize the type safety and performance of a compiled and statically typed language over the convenience and portability of Erlang and Python.

Maintainability is a significant consideration in the design of AutoSPADA, and using a language with a low barrier of entry and easy access to a rich ecosystem of official third-party libraries is an important maintainability aspect. Erlang has excellent support for concurrency, but it is not a very popular language [19] and does not have the best ecosystem of third-party packages. For example, Erlang lacks official support for central libraries such as Docker Engine, MongoDB, and Protocol Buffers. Therefore, keeping Erlang as our implementation language would have forced the application to rely on possibly poorly maintained unofficial packages. Also, using languages with widespread use has many advantages, such as making it easier to hire developers in a market where software development skills are already in high demand.

2.2. Communication

OODIDA relies on the built-in network capabilities of the Erlang language. In contrast, AutoSPADA uses a proven IoT-oriented protocol with a minimal network footprint for notifications. A binary remote procedure call (RPC) framework is used for updates and data transfers, where server and client APIs are compiled from a shared interface specification to minimize protocol mismatches between communicating parties. For communication between components written in different languages, OODIDA instead relies on JSON messages. Being a text-based format, JSON interfaces are inefficient and have a high development and maintenance overhead.

2.3. Resiliency

OODIDA is built on direct communication with clients, assuming that these are available during the task lifetime. For example, results from a client are lost if it goes offline and fails to reconnect before a fixed deadline [16]. Moreover, the OODIDA application state is unreplicated and stored in local memory on the server node, thereby introducing a single point of failure. This design is not resilient against common failures and limits the scalability of deployments.

In the AutoSPADA redesign, we centralize the application state to avoid relying on any specific server node. Also, client state updates are cached to tolerate irregular availability and poor network conditions.

2.4. Privacy and security

AutoSPADA has a strong focus on privacy and security. All network communication is secured by state-of-the-art encryption, communicating parties are always mutually authenticated, and tasks are isolated from their hosts through containerization. OODIDA did not focus on these aspects to the same degree. For example, Erlang was originally designed to run on private networks [20]. Although modern-day Erlang supports communication over the Transport Layer Security (TLS) protocol [21], this heritage makes it difficult to use distributed Erlang securely [22]. Being a prototype system, OODIDA does not authenticate clients or users and lacks a user privilege system. Moreover, OODIDA does not containerize the tasks that run on the client, which increases the risk that tasks destabilize the client by, e.g., excessive consumption of host resources.

3. Platform design

AutoSPADA is built around three distributed nodes: users, servers, and clients. Client nodes are responsible for spawning tasks on demand and reporting the results of these tasks. The user nodes submit tasks to run on the clients and retrieve task results from the server, streaming or on demand. The server nodes bridge the gap between the client and user nodes by receiving and persisting task requests from users and results from clients. This organization of nodes mirrors the actor roles in the platform, illustrated in Fig. 1.

The server and client nodes are implemented using the Go programming language. A contributing factor to choosing Go for the core parts of AutoSPADA is its widespread use and simple structure, which lowers the threshold for new developers. The language also offers a large ecosystem of third-party libraries with official support for required APIs to reduce development time.

To reach a higher level of technological maturity, the AutoSPADA platform needs an underlying architecture designed with production-scale use in mind. This requires the design to be founded on solid principles of scalability, reliability, resource efficiency, privacy, and security. We will now discuss each of these aspects and motivate key technological choices made to address them.

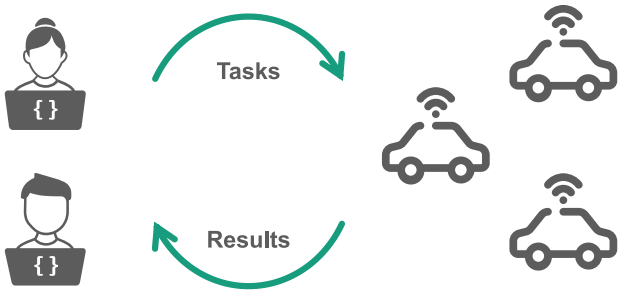

Fig. 1. The logical AutoSPADA actors are the users (left) and the clients (right). The server nodes implement the infrastructure for transmitting tasks from users to clients and results from clients back to users.

3.1. Scalability

The AutoSPADA platform should be able to have many client nodes, which makes scalability an important factor in the architecture's design. The scalability of distributed systems is often understood as sustaining quality of service when faced with increased workloads by adding more system resources [23]. Two ways of increasing a system's resources are commonly referred to as *vertical* and *horizontal* scaling. If the resources are servers in a cluster, vertical scaling means upgrading the existing servers with more powerful hardware, while horizontal scaling means adding more servers.

Horizontal scalability is usually preferred over vertical scalability. One reason is that there is a limit to how much vertical scaling one can achieve—a motherboard can only fit so much RAM and have so many processors. In contrast, horizontal scaling is facilitated by cloud providers that give access to practically unlimited numbers of virtual machine instances. Moreover, horizontal scaling does not require downtime and increases redundancy for better fault tolerance. However, horizontal scaling introduces complexity since the incoming workload must be load-balanced among the available resources to be effective. In practice, load-balancing is achieved through the use of an orchestrator such as Kubernetes [24], which by itself is a non-trivial tool to master.

The server nodes must adhere to the stateless paradigm to achieve the scalability that the platform requires. The stateless property of the server instances means that they do not retain any application state between requests. Instead, the necessary state is read from a common database for each request, and any modified state is written to the database before the end of the request. The benefit of stateless servers is that they become ephemeral, i.e., they only need to live for the duration of a single request. This means that servers are horizontally scalable and that malfunctioning or superfluous instances can be taken down with little prior notice. The pool of server instances can therefore be quickly scaled up or down to meet surges in request volume while holding the spare capacity, and thereby the cost, to a minimum.

3.1.1. Scalable databases

Scaling the number of server nodes horizontally is only effective if the database can support them. Therefore, it is equally important that the selected database can scale to handle the increased traffic. Specifically, a scalable database should at least have a shared-nothing architecture with automatic sharding and shard replication [25]. Traditional relational database management systems such as MySQL and PostgreSQL typically share primary memory and disk storage, which makes them unsuitable for the AutoSPADA platform.

Although scalable relational databases exist [26], e.g. MySQL Cluster, Citrix, and the Citus PostgreSQL extension [27], so-called NoSQL databases are often designed to address specific challenges of scaling relational databases. For example, Dynamo [28] was designed to achieve higher availability than previously possible in strongly consistent shared-memory relational databases. At the time of writing, the most popular database in the NoSQL family is MongoDB [29]. Because MongoDB has a shared-nothing architecture, support for automatic sharding with configurable sharding policies as well as automated replication, failover, and recovery [30], it can be used as the foundation for scalable systems [25].

For AutoSPADA, we have chosen MongoDB as our database since it is a proven choice in demanding applications [31] and is one of the few NoSQL databases to offer per-document, multi-document, and distributed transactions [32]. In particular, distributed transactions are essential to the integrity of the platform. Moreover, MongoDB has an official Go library, and because it is widely used and open source, we limit the risk of being subject to vendor lock-in.

3.2. Reliability

Scalability also ties into reliability since enough healthy server instances always need to be available to handle a massive number of requests from clients and users. The ability to quickly scale the pool of stateless server instances up or down translates to a reliable backend service since a specified level of spare capacity can be easily met. The user, server, and client nodes also need to implement robust error handling. Moreover, we make use of proven third-party components where possible. Using tested and established components helps to ensure that our platform operates reliably while also making it easier to maintain.

Some of our previous technological choices also cover aspects of reliability. With the MongoDB database, it is possible to have replicas of the partitioned dataset, allowing for redundancy and, therefore, reliable operation. One of the central features of Go, the language chosen for the server and client nodes, is the very explicit error handling, unlike languages such as C++ where unexpected exceptions can be thrown from third-party code. This facilitates code robustness and, consequently, reliable operation of the services.

3.2.1. Reliable operation of the client node

While clients are not directly callable by the server, they need to quickly respond to changes in their assigned set of tasks. Clients achieve this by subscribing to a per-client topic on a message broker. When the application state affecting a client has changed, the server notifies the client through the message broker that it needs to dial in to retrieve the updated state.

Client devices cannot be expected to always be online since the mobile connections used for communicating with the server are inherently unreliable. Therefore, designing the task lifetime around remote procedure calls is not suitable for the platform. Consequently, a state-based approach was chosen, where the state of a task, including its results, is persisted in the centralized database. The client also persists results locally until they are confirmed to be recorded in the database. This makes the application resilient against any disruptions in communication.

3.3. Resource constraints

Clients that connect to the AutoSPADA platform are expected to do so over slow and unreliable network connections. Furthermore, with a large number of clients, only a slight increase in the per-client network requirements leads to a large increase in the total network capacity requirements, potentially risking network budget issues. The design of the application must, therefore, take the strict network constraints into account to give control over the amount of network traffic.

While client nodes are expected to be powerful enough to run an embedded Linux OS, the processing power and storage space available in the node are nevertheless limited. These resources should primarily be used for running tasks, which means that the client program that manages tasks has to be efficient. Therefore, the client code should preferably be compiled, native code rather than interpreted to avoid both low performance and the bloat of a runtime system. Also, the amount of CPU and RAM that a task can allocate needs to be controllable by the application.

The Go programming language has been shown to be an energy-efficient language, at the very least on an x86 processor [33]. This makes Go a fitting language for writing programs that target the resource-constrained hardware typically found in edge devices.

3.3.1. Communication protocols

To address the resource constraints imposed by mobile connections and a possibly restricted network budget, MQTT [34] — a communication protocol designed specifically for IoT applications — was chosen for server-to-client communication. The MQTT protocol has low message size overhead and is widely adopted [35]. The protocol also has configurable quality of service (QoS) levels, providing message delivery guarantees at the expense of performance.

The MQTT protocol requires a message broker to route messages between MQTT clients. Many message brokers support MQTT, including ActiveMQ [36], EMQX [37], Mosquitto [38], and RabbitMQ [39]. However, for reliability and scalability, we require a message broker that can be configured to run as a distributed cluster. Moreover, because MQTT sends data as plain text, the chosen broker must support TLS. A benchmark of popular distributed MQTT brokers found only small differences between the cluster-based brokers, including EMQX and RabbitMQ [40]. This suggests that any of the most widely used MQTT brokers are suitable alternatives.

In AutoSPADA, we chose RabbitMQ as our message broker. Although originally designed for the AMQP protocol, RabbitMQ also has an MQTT plugin with TLS support. With RabbitMQ, the more feature-rich AMQP protocol can simultaneously be used between user and server nodes. This is facilitated by official AMQP client libraries for both Go and Python. RabbitMQ has a maintainability advantage since it is open-source software and combines the features of MQTT and AMQP into one technology. The main downside to RabbitMQ's MQTT plugin is that the highest QoS level is unsupported.

The information passed to the client by the broker is kept to a minimum, only consisting of the current version number of the client state. The database structure was designed with the immutability of, e.g., task payloads in mind. This makes it possible for clients to cache these entries locally. Hence, network traffic is further reduced since the client can avoid repeatedly querying the database for the same information.

Many distributed applications rely on JSON for message encoding, which is less space-efficient than binary encodings and adds significant network overhead. A case study showed that using Protocol Buffers (protobuf) to encode over 50 000 messages from a vehicle tracking system reduced the total data amount by five times compared to BSON (Binary JSON) and nearly six times compared to JSON [41]. Other binary protocols, such as Apache Thrift, Apache Arvo, and Microsoft Bond, are also space efficient compared to JSON [42]. However, because of Protocol Buffers' maturity, widespread use in industry [42], and official open-source libraries for both Go and Python, we chose to use protobuf in the AutoSPADA network.

3.4. Privacy

Data transmitted in the AutoSPADA network is inherently sensitive. Therefore, it is of utmost importance that communication is protected end-to-end with strong and proven encryption. To avoid malicious actors in the network, authentication mechanisms must be in place to prove the identity of clients, users, and servers. Ideally, the system should offer mechanisms for the anonymization of client data. This is likely very difficult to achieve. As a compromise, the application should offer a privilege system where only authorized users can request client results.

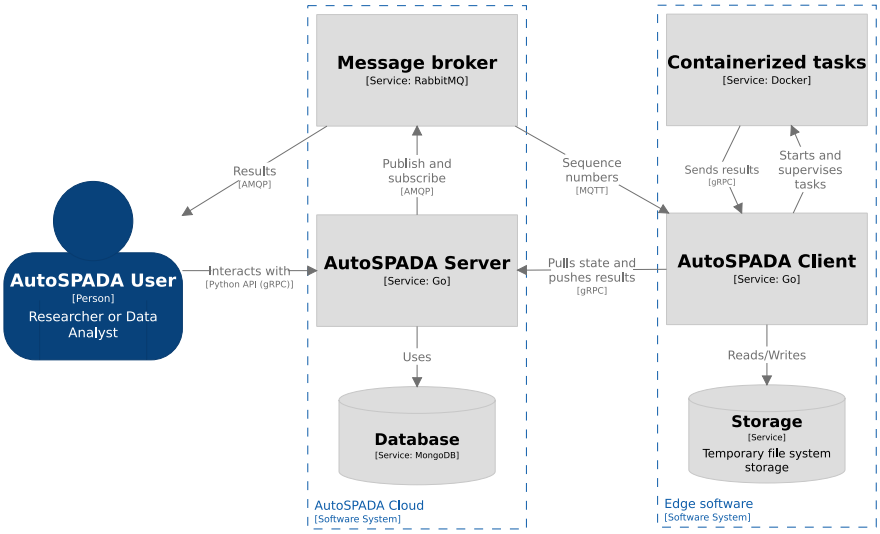

Fig. 2. An overview of the AutoSPADA architecture showing the services, communication paths, languages, and protocols used in the platform. All communication is secured by TLS. Authentication between nodes is performed by OIDC using JSON Web Tokens (JWTs) for user nodes and through mutual TLS (mTLS) using X.509 certificates elsewhere.

3.4.1. Authorization and authentication protocols

All communication in the AutoSPADA network has been designed to be encrypted using TLS, which is the standard for secure communication on the web. TLS further supports mutual authentication (mTLS) of both the server and client using X.509 certificates, which is necessary to establish the trust needed to exchange information between these parts of the network [43,44].

We use OpenID Connect (ODIC) [45] as the authentication protocol to establish user identities. OIDC is a widely used [46] state-of-the-art protocol that builds on the OAuth 2.0 framework [47,48]. The protocol delegates the authentication process to a third-party service, improving security and user convenience. This allows the administrators of AutoSPADA deployments to configure and customize the user authentication to their needs. Also, the OIDC protocol supports privilege systems where user resource access can be arbitrarily controlled.

For the AutoSPADA platform, relying on standard authentication protocols and delegating the difficult task of securing and maintaining user credentials to proven third-party services translates into a higher level of privacy.

3.5. Security

Isolating user-defined tasks from the host environment is an important security concern. This isolation is important to prevent malicious tasks from acquiring sensitive information and compromising the host system through, e.g., excessive resource allocation. Both issues are addressed by running the tasks in containers since they provide a strong boundary between active tasks and the host system and enable per-task resource limits for RAM and CPU usage.

We use the Docker platform [49] for containerization because of its extensive toolkit that includes networking and logging features. These features are unavailable if we interact with the container runtime directly. By using Docker, we limit the number of dependencies and reduce code complexity for better maintainability. However, that is not to say that other container engines, runtimes, or tools are inferior or unsuitable. Nevertheless, internal familiarity, library availability, and compatibility with our existing Kubernetes development platform meant that Docker was natural to use as a starting point.

4. Platform architecture and implementation

Having discussed the various design concerns and the technology choices made to address these, we summarize the roles of the nodes of the platform and the protocols they use. This overall architecture is illustrated in Fig. 2. Users interact with server nodes through a Python library that wraps gRPC calls and provides higher-level functionality. Users can also monitor AMQP topics to receive streaming task updates or query historical results via the Python library. The server nodes implement gRPC services for the client and user-facing APIs and use a MongoDB database for persistent storage. The server produces state update notifications via RabbitMQ and an MQTT bridge that clients consume. The state update notification is a running count of the state revision for the individual client. Through this mechanism, the client is notified about any relevant state updates, triggering a query to the server for the latest revision of the state. Clients cache results locally until they can be delivered to the server via gRPC.

With an understanding of the components of our architecture, the rest of this section is dedicated to its implementation. We especially focus on the details of the client node implementation and omit the server node since it mostly implements the gRPC services. However, we start with a brief discussion of a simplified version of our data model to better understand the resources that users create.

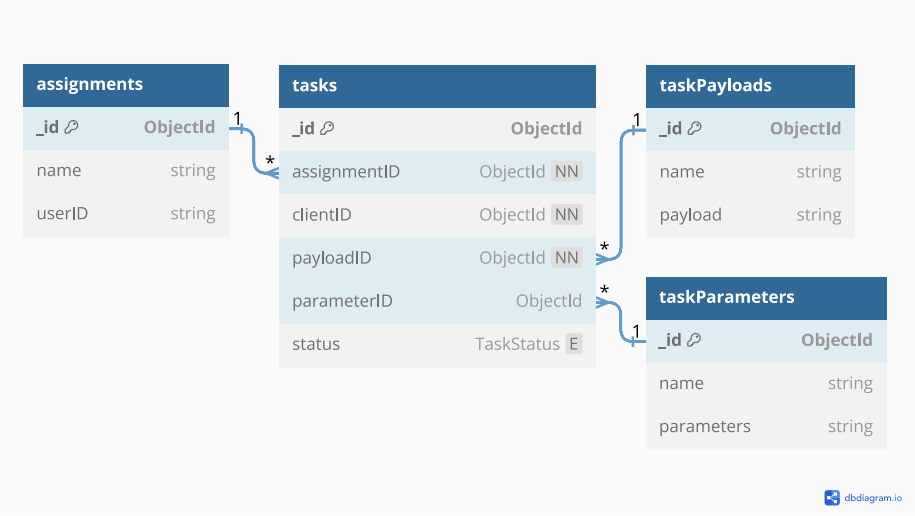

Fig. 3. An entity-relationship diagram showing a simplified view of selected documents in the database, highlighting their relations. An assignment has many client-specific tasks. A task, in turn, has a payload and, optionally, a parameters document. NN is short for *not null* and E is for *enum*.

4.1. Application state data model

Fig. 3 shows a simplified view of the data model for the centralized state to help clarify our distinction between assignments, tasks, payloads, and parameters. Users create assignment documents containing a set of tasks. Tasks, in turn, reference their assignment, a payload (the code to be executed), parameters, and the ID of the client for which the task is intended. The optional parameters document holds a JSON-serializable value that the payload can read via our client Python library. This feature is useful to, for example, distribute a model to many clients or have the same payload use different signal names on different clients.

4.1.1. Task life cycle

The simplified data model in Fig. 3 also shows that task documents have a status field. This field has one of four possible statuses: `ACTIVE`, `FINISHED`, `ERROR`, or `CANCELED`. Tasks are said to be `ACTIVE` upon creation, and the only valid transition is from `ACTIVE` to one of the other three statuses.

Transitions from `ACTIVE` to a `FINISHED` or `ERROR` status are client-initiated. Should the payload encounter a runtime error, the task container exits with an error code and the client subsequently uploads the container logs along with an `ERROR` status to the server. If the task instead runs to completion without any errors, the client reports a `FINISHED` status back to the server. The server only accepts results from `ACTIVE` tasks, which means that incoming results for a non-active task are ignored.

Because task payloads are general Python programs, users can define payloads that never terminate. Such indefinite tasks only stop if explicitly instructed to do so. Users do this by canceling the task through a Python library. Canceling a task causes the client to stop the corresponding Docker container, forcing it to exit if needed. Only active tasks can be canceled, however.

4.2. Implementation of the client node

The client application manages its assigned active tasks and serves the client gRPC API used by the tasks. Fig. 4 gives an overview of the main components of the client. The sync loop synchronizes the tasks' states with the centralized server state. As part of this loop, it will spawn a thread to supervise the exit condition for each new task it starts. The tasks can request signal values from the signal handler and send results to the result handler. The signal handler is an abstraction layer to the actual signal source and has a state for each signal containing the latest observed value. Keeping the latest values in memory makes it simple to determine the present value of stateful and infrequent signals, such as binary values representing an on or off state.

4.2.1. Synchronization loop

The primary responsibility of the sync loop is to keep the local state of tasks on a client synchronized with its centralized state stored in the server-side database in a secure, responsive, and lightweight manner. The responsiveness and lightness come from short MQTT messages sent from the server to keep track of user-initiated changes. The client-local state changes when a task is created, canceled, finished, or whenever a result is published. Task creation and cancelation can only be initiated by a user. A pseudo algorithm of the loop is shown in Algorithm 1.

Each client has a centralized logical clock [50] to track changes in its associated tasks. Changes to a task also increment the clock of the corresponding client and the updated value is published over MQTT. On the client device, whenever the sync loop notices

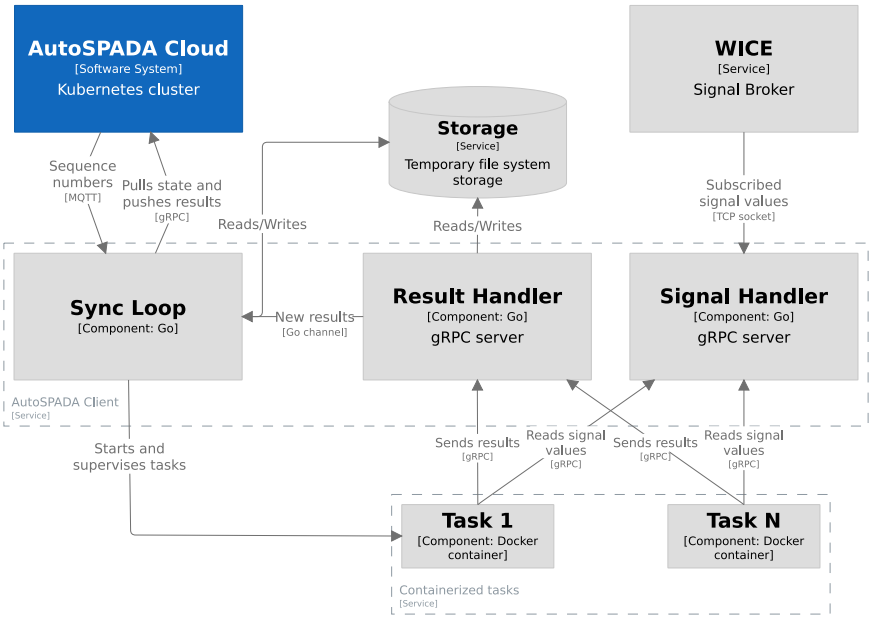

Fig. 4. Detailed view of the software components in an edge device running the AutoSPADA client. The major components are the sync loop, result handler (gRPC), signal handler (gRPC), Docker, and the WICE Signal Broker. A container supervisor thread is also started for each task but is not visualized with a component box.

that its local logical clock has fallen behind, it will request its state from the server, as shown in the first case-clause of Algorithm 1. In response, the client receives its current logical clock and all active tasks. Each task has an ID and the number of submitted results.

Three functions are not defined in Algorithm 1:

- $fetchState$ requests the client state from the server.
- $submit$ uploads results or status changes to the server.
- $syncContainers$ starts and stops containers to match the currently active tasks.

Both $fetchState$ and $submit$ send a new state back to the sync loop. However, only one of them is allowed to run at a time, which is controlled by the $syncingState$ boolean. A dirty state can arise if new results or statuses are received from the result handler or a container supervisor (see Fig. 4) while $syncingState$ is set to true. When this happens, the newly received results or statuses are not visible to the active $submit$ thread. Therefore, when the $dirtyState$ flag is true and the active $submit$ thread sends a new state to the sync loop, the algorithm calls $submit$ again.

4.2.2. Containerization

A new Docker container is started when $syncContainers$ sees a new task not present in the $localTasks$ map. The procedure to run and supervise the container is spawned in a new thread. The supervisor thread sends an `ERROR` status for the task back to the sync loop if the container exits with an error signal, otherwise, it sends a `FINISHED` status. Also, it can stop the container if the task has been canceled or removed.

4.2.3. Result handler

The payload running in the container can communicate with the supervising client through an API implemented as a gRPC server—the Result Handler as seen in Fig. 4. Depending on which RPC is used, results are either forwarded to the sync loop for publication to the server or stored locally as intermediate results to be loaded later for further processing. Storing an intermediate result is a way of handling client restarts.

4.2.4. Signal handler

Our initial hardware target is the Host Mobility MX-4 unit running the Wireless Information Collection Environment (WICE) platform [51]. The WICE platform enables remote deployment of our client binary and exposes a Signal Broker API that allows us to consume signals from the CAN and FlexRay buses through a publish–subscribe model.

As shown in Fig. 4, the Signal Handler is an AutoSPADA client component that subscribes to signal values from the WICE Signal Broker. Internally, the Signal Handler consists of a gRPC server and a signal broker proxy to consume and cache signals from the

Algorithm 1: The sync loop

Data:
A *state* containing a logical clock *ts*, *tasks* info about active tasks at *ts*, and a map from tasks to results and a status called *localTasks*.
Two booleans, *syncingState* and *dirtyState*, initialized to false.

```
1  loop
2      switch event do
3          case received logical clock tsR from MQTT do
               // Notified about a change made by a user
4              if tsR > state.ts then
5                  state.ts ← tsR
6                  if not syncingState then
7                      syncingState ← true
8                      Spawn fetchState(cloned state)
9          case received new state s do
10             if s.ts ≥ state.ts then
11                 state.ts ← s.ts
12                 state.tasks ← s.tasks
13                 if diryState then
                       // state.localTasks changed while syncing
14                     dirtyState ← false
15                     Spawn submit(clone(state))
16                 else                                   // Done syncing state with server
17                     syncingState ← false
18                     Spawn syncContainers(s)
19             else
20                 Spawn fetchState(clone(state))
21         case received local tasks L from syncContainers do
22             state.localTasks ← L
23             syncingLocals ← false
24         case received result r or status s from container for task t do
               // Local changes
25             append r or change to s for state.localTasks[t]
26             if syncingState then
27                 dirtyState ← true
28             else
29                 syncingState ← true
30                 Spawn submit(clone(state))
```

external WICE Signal Broker. This proxy allows us to normalize the behavior between different external signal sources. That is, the API used by tasks to read signal values stays the same even if support for another signal source, e.g., MQTT, is added.

5. Introduction to using AutoSPADA

The platform is ready for use after it is deployed on a cloud platform and to clients. We provide two libraries for Python that wrap gRPC calls and let users consume message queues. One library enables users to work interactively with the platform, while the other is used in payloads running on the client. This section briefly demonstrates what users can express with the two libraries.

5.1. The Python client library

The client library provides functions used to define AutoSPADA payloads. The core functionality lets users read signal values and publish results. Additionally, one can read task parameters, cache a state as an intermediate result, and read a previously cached state. State caching is local to the task and will be removed upon task completion, but crucially survives client restarts. Listing 1 shows a simple payload using the core functionality and two task parameters.

There is no limit on the number of results a task may publish. This is important because it allows users to define long-running, potentially indefinite, tasks that publish results periodically. In the case of long-running tasks, the need to cache an intermediate result or state becomes evident. For example, consider a task that builds a histogram over time and periodically publishes its progress. Without caching, the task state is lost when the vehicle is turned off, forcing the task to start over from the initial state when the

```python
import autospada
import time

seconds_to_collect = autospada.parameters['seconds']
signal_name = autospada.parameters['signal_name']

total, count = 0, 0
start_time = time.monotonic()
while time.monotonic() - start_time < seconds_to_collect:
    total += autospada.next_signal(signal_name)
    count += 1

mean = total / count
autospada.publish({"Mean": mean, "n_values": count})
```

Listing 1: A payload to compute the mean of value readings collected over a configurable period of time. The result is published as a JSON-serializable Python dictionary.

vehicle starts up again. Because the binning is reset, the user has to go through all published results to find the total bin counts over the lifetime of the task. However, by also caching the histogram binning when publishing results, only the latest result is needed since the historical bin counts now increase monotonically. This simplifies the off-board analysis and reduces the required data transfer from the cloud to the user.

5.1.1. Testing payloads

Writing payloads, or code in general, is an error-prone endeavor. Therefore, testing your code is always a good idea. With an interpreted language like Python, it is easy to run the code directly and see if there are any immediate problems. Users of AutoSPADA can test their Python payloads locally, without access to any clients or even a server node, in two ways: run the script directly or load it into the same kind of container used by clients.

By default, the `autospada` library acts as a dummy library that returns random values for any signal and prints messages to standard output when side effects occur. Payloads can, therefore, run independently like any other Python script. However, the logical flow of a payload may depend on observing specific signal values. In those cases, reading random values for all signals can prevent the payload from exiting normally. Nevertheless, this simple test can still catch unexpected behaviors such as syntax errors.

A more robust testing methodology is to run the payload in a container using the same Docker image as a client would. The user library has functionality that makes this simple to run, provided that the Docker image is also available for the user's local CPU architecture. Moreover, this approach allows users to control the values of signals by providing a CSV file with recorded signal values. This CSV file can contain examples of corrupt or missing data, which will be observed as `None` values by the Python user library. In this way, payloads can be tested offline against captured data or tailored inputs.

5.2. The Python user library

As shown in Fig. 3, our data model has documents for assignments, payloads, parameters, and tasks. The user library provides functions to retrieve and create these documents. With a connection to the RabbitMQ server, the library also lets users subscribe to new results and task status changes through blocking await functions or lazy iterators.

5.2.1. An example workflow

The following section breaks down a simple user workflow. We demonstrate how to define an assignment that instructs online clients — in this case, vehicles — to run a task that computes their mean speed over five seconds.

The Python library for the user is called `autospada_user` and provides the `User` class through which all actions to the server are made. A `User` object is initialized with a TOML configuration file that contains the information required to connect and authenticate with the AutoSPADA cloud.

```python
import autospada_user
user = autospada_user.User('user_config.toml')
```

Below, a payload object is created with the contents of a Python file, which in this example is the same as Listing 1. The library refers to payloads, parameters, tasks, and assignments as document objects. All document objects have a `commit` method that saves the object to the database. The payload object in the example has not yet been committed, meaning that it only exists in local memory for now.

```python
from pathlib import Path
code = Path('mean_payload.py')
payload = user.payload(code.read_text(), name='Average')
```

Parameters are given as a separate object for composability.

```
parameters = user.parameter(
    {"seconds": 5, "signal_name": can_speed_name})
```

Now, we prepare identical tasks for all clients that are currently online.

```
clients = user.get_clients(online_only=True)
tasks = []
for client in clients:
    tasks.append(user.task(
        client.id, payload, parameters))
```

Since these tasks serve the same purpose for each vehicle, they are grouped into one assignment. An assignment does not need to have any related tasks, but every task needs an assignment for the sake of consistency.

```
assign = user.assignment("Mean speed", tasks)
```

The `commit` method commits the assignment document object to the database, including all related documents if they have not been committed yet. Since no document objects have been committed until now, this method call creates the assignment, tasks, payload, and parameter documents in the database. The library is designed with *method chaining* in mind [52], and the assignment object is therefore returned from `commit` so that `await_results` can be called directly. This final method call waits for all tasks to finish and returns all results.

```
results = assign.commit().await_results()
```

6. Evaluation

Since interactivity is a goal for AutoSPADA, we are concerned with latency for short-lived tasks. That is, the time from submitting a task to observing results needs to be low enough to encourage interactive use. However, latency is greatly affected by both the task implementation and client devices' processing power and immediate network environment. These factors are difficult to model in a general way. Instead, our evaluation quantifies the overhead of AutoSPADA under idealized conditions on a Raspberry Pi 3 Model B. The experiment aims to demonstrate that the system can be used interactively on relevant hardware and does not introduce inherent bottlenecks to latency.

6.1. Measurements

The overhead is measured in terms of latency from the user's perspective. The design of the experiment is such that we interpret the measurements as proxies for events in the system. This is achieved using two unconventional tasks—one that publishes empty results twice in succession and then exits, and another that does nothing. None of the tasks specify parameters, which saves an additional communication round to the server. Using these two tasks, we proceed to take four measurements that we will refer to as $t_{start}, t_{delay}, t_{exit}$, and t_{cycle}. We will now discuss each measurement in turn.

6.1.1. Task startup

The first measurement, t_{start}, begins when the task is committed and ends when the user receives the first result. This is a critical measurement since it gives an approximate lower bound for the time a user must wait to receive initial feedback from their task.

To minimize overhead, we use a payload that immediately publishes empty results. The user subscribes to receive assignment results on an AMQP queue before issuing the task. Hence, results from the server are received as fast as possible. Because the task publishes an empty result as soon as it starts, we interpret this measurement as the time to start the task plus the time to propagate a result back to the user. The total amount of network communication included in this measurement is three gRPC calls (one from user to server and two calls from client to server), one MQTT message, and one AMQP message. The measurement also includes the time to start a Docker container.

6.1.2. Delay between results

In an idealized scenario, the time between observing two results published immediately after one another, t_{delay}, should be small. We expect this to show that the system is highly responsive once the task is running in its container. This is important because it demonstrates that results are promptly delivered to users, contributing to an interactive user experience. Because t_{delay} measures a difference between two publishing events, it only includes the inter-process communication from the task container to the client program, plus any difference in propagating the result from the AutoSPADA client to the user. Since this approximates the overhead of inter-process communication between the client and the task container, it should be considerably smaller compared to the other measurements.

6.1.3. Container shutdown

The time from receiving the second result to observing a `FINISHED` status, t_{exit}, is also measured. The resulting number approximates the time it takes to exit the task container. This has less impact on the overall user experience since users are less interested in the task's status once they have collected all the results they need. However, t_{exit} combined with t_{start} will help us interpret the results for t_{cycle}. The communication overhead included in this measurement is between the Docker daemon and the AutoSPADA client to detect shutdown, plus any difference in propagating the status from the client to the user.

Table 1
Specifications of the experiment participants. The server node was provisioned through Google Kubernetes Engine (GKE).

Node	Host	Specifications
User	Local VM	Ubuntu 22.04 LTS, Ethernet internet connection
Server	E2 x86-instance in GKE	Limited to 1 vCPU, 1 GiB memory
Client	Raspberry Pi 3 Model B Rev 1.2	Broadcom BCM2837, 1 GB RAM, Raspberry Pi OS (32-bit Bullseye), WiFi internet connection

6.1.4. Task cycle

Lastly, we measure the whole task life cycle, t_{cycle}, from `task.commit()` to a user-observed `FINISHED` status. This includes the time it takes for the task container to exit, which differentiates it from t_{start}. The goal of this measurement is to see how fast the smallest possible task propagates through the system from start to finish. When put into relation to the other metrics, this helps us understand the impact of task containerization on latency.

Whereas the previous measurements were taken at different stages of the same task, t_{cycle} uses another task with a payload that only imports the `autospada` library and then exits. The import is included to make the measurement comparable to t_{start} since the import loading time is not negligible, mainly due to the transitive import of the gRPC library. The amount of network communication included in this measurement remains the same as for t_{start}. The measurement additionally includes the time to start and stop a Docker container.

6.2. Experiment setup

A minimal deployment of one client, one server node, and a user participated in the experiment. The relevant specifications of each node are given in Table 1. A Raspberry Pi acted as the client node to better represent the processing capabilities of hardware that we envision could run the AutoSPADA client. In particular, the Raspberry Pi 3 Model B has similar specifications to the Host Mobility MX-4 T30 used during the AutoSPADA project.

The AutoSPADA client is deployed as a binary with a configuration file and valid certificates. The compiled AutoSPADA client binary for ARM is 32 MiB. The `top` utility reported 26.0 MiB resident (RES) and 20.5 MiB shared (SHR) memory sizes for a newly started idling client program. The Docker image used was 48 MiB and contained Python 3.8 plus our client library. This image was downloaded to the local Docker image registry before the start of the experiment.

The experiment was repeated 100 times to give aggregated statistics. The user starts by preparing payloads and assignments so that their submission time is not included in any measurement. Before tasks are committed, the user also creates connections to message queues to which results and task statuses are published.

New payloads were created in each iteration since the client keeps a cache of the ones most recently used. This means that if payloads are reused in the experiment, they are not downloaded again. Hence, caching improves the t_{start} and t_{cycle} measurements noticeably. Although payload reuse is efficient, including the payload download in the measurement is more representative of an iterative development process where payloads frequently change. Therefore, we include the time to download the payload in our measurements.

6.3. Results

The experiment results are presented in Table 2. The difference between t_{start} and t_{cycle} suggests that container shutdown invokes a delay of approximately one second, which is confirmed by the measured t_{exit}. The time from submitting a task to observing the first result, t_{start}, is roughly between 4 to 4.5 s, which is justifiable considering that it includes the time for container creation and startup. The delay between two results, t_{delay}, is stable during the experiment, which is expected because identical gRPC calls between a task container and the client program should be nearly constant in time. All measurements besides t_{delay} involve container creation, teardown, or both, which is less predictable than an inter-process gRPC call and is observed as larger variances.

The peak resident memory size during the execution of the experiment was approximately 29.0 MiB—up 3.0 MiB from the idle client. This number was taken from the *high water mark* field (VmHWM) in the `status` file of the AutoSPADA client process in the `/proc` filesystem after the experiment was completed. Importantly, this only measures the AutoSPADA client program and does not account for the memory used by Docker and the containerized task, c.f. Fig. 2.

7. Related work

Various edge computing platforms already exist on the market—some build on known architectures such as Kubernetes, some focus on stream processing, while others have more unique designs. This section highlights a few relevant actors and reviews their similarities and differences compared to the AutoSPADA platform. A summary is given in Table 3.

Table 2
Results of the latency experiment. The statistics are derived from one hundred data points per measurement. Measurement times are given in seconds, and SD is a shorthand for the sample standard deviation. We also show the 5th and 95th percentiles since there are a few outliers at both extremes.

	Measurements (s) | $n = 100$			
	t_{start}	t_{delay}	t_{exit}	t_{cycle}
Mean	4.282	0.261	1.198	5.640
SD	0.260	0.080	0.316	0.377
$P_{5\%}$	3.973	0.233	0.830	4.940
$P_{95\%}$	4.605	0.271	1.695	6.191

Table 3
A comparison of properties for different edge computing platforms. Memory footprints are either measured (KubeEdge [53] and AutoSPADA) or taken from the specified system requirements of the respective platform. The memory footprint of Stream Analyze ranges from just a few kilobytes to five megabytes.

	Open source	Containerized workloads	Memory (MB)	Implementation
AutoSPADA	–	Yes	27.3	Go
IoFog	Yes	Yes	256	Java
Stream Analyze	No	No	0.017–5	C
KubeEdge	Yes	Yes	40	Go
Azure IoT Edge	Partially	Yes	–	C#
AWS IoT Greengrass	Partially	Possibly	96	Java

7.1. Stream analyze

Stream Analyze Sweden AB offers a commercial platform for collecting and aggregating data from edge devices, with a focus on stream processing [54]. Analytical models are sent to clients that execute the requests and send streaming results to a backend that users can query. Data analysis is performed in the Object Stream Query Language (OSQL), their proprietary language for data stream processing, or using a graphical tool for query editing and result visualization. In addition to streaming, Stream Analyze also supports offline retrieval of task results.

7.2. KubeEdge

Initially proposed by researchers at Huawei in 2018 [55], KubeEdge is an open-source edge computing framework built on Kubernetes. The goal of the framework is to extend Kubernetes clouds to also include edge hosts, meaning that existing Kubernetes workloads can be applied to both cloud and edge workers. The framework was accepted as a Cloud Native Computing Foundation (CNCF) incubating project in September 2020.

On a high level, KubeEdge has two components: CloudCore and EdgeCore. The CloudCore is a centralized component that orchestrates edge devices. The EdgeCore consists of several components that synchronize device status with the cloud, enable containerized execution of workloads, and mappers to allow external communication with the EdgeCore over common IoT protocols. The EdgeCore connects to the cloud through a WebSocket client and has local storage to let the EdgeCore function when offline.

7.3. Eclipse IoFog

Eclipse IoFog is an open-source edge computing platform initially developed by Edgeworx and later donated to the Eclipse Foundation [56]. Edge devices run an IoFog component called Agent—a daemon service responsible for running containerized microservices. Devices running the Agent are orchestrated by the so-called Controller component that constitutes the platform's control plane. The Controller can be placed anywhere, even on the same device as an Agent, as long as it is reachable by all devices running the Agent. Moreover, the control plane can be deployed to a Kubernetes cluster where the number of Controller instances can be scaled for high availability.

The components of IoFog are Java programs, meaning that they run on the Java Virtual Machine rather than natively. Running a virtual machine generally imposes a performance penalty, especially regarding memory. This is also observed in benchmarks where virtual machine languages used 2.28 times more memory on average compared to natively compiled languages [33]. Another evaluation showed that the IoFog Agent component uses approximately 240 MB, nearly five times more memory than the corresponding component in KubeEdge and K3s [53]. This memory usage may be prohibitively large.

7.4. Big tech platforms

Of the so-called Big Tech companies, three (Alphabet, Amazon, and Microsoft) offer cloud computing services with various edge or IoT-oriented products. Amazon's AWS IoT Greengrass and Microsoft's Azure IoT Edge are edge computing platforms with open-source software components. Their client runtimes are open source, but cloud integration relies on the respective provider's

commercial services. Having retired the Google Cloud IoT Core, Alphabet now offers Google Distributed Cloud Edge (GDC Edge) instead. However, GDC Edge is a fully managed service, meaning that Alphabet supplies both hardware and software, making it less relevant to include in our survey.

The Azure IoT Edge and AWS IoT Greengrass platforms are in many aspects similar to each other. Both share the goal of bringing computation closer to data sources and support the deployment of Docker containers to edge devices. Also, both platforms extend the respective cloud provider's services, e.g., AWS Lambda Functions. The software running on edge devices is packaged into AWS IoT Greengrass Components or Azure IoT Edge Modules, respectively. Amazon provides a library of prebuilt Greengrass Components and Microsoft similarly provides prebuilt IoT Edge Modules such as Azure Stream Analytics. Microsoft provides Software Development Kits (SDKs) for users to write custom IoT Edge modules. Likewise, Amazon provides SDKs to allow users to develop custom Greengrass Components. One difference is that IoT Edge Modules run as containers, whereas AWS Greengrass Components are not containers by default.

7.5. Comparison

AutoSPADA distinguishes itself from other cloud or Kubernetes-based platforms because of its focus on interactive use. In particular, AutoSPADA users work entirely in Python, without the need to write Kubernetes manifests or build Docker images for foreign CPU architectures. This simplifies the development cycle, encouraging exploration and rapid prototyping.

AutoSPADA and Stream Analyze share a focus on computational tasks rather than microservices. As a result, they are not designed to deploy general services such as API servers or databases, and instead run their computational tasks in isolation. For example, AutoSPADA has no network path between tasks, and users cannot directly access running task containers. In contrast, microservice architectures, such as KubeEdge or ioFog, require device-to-device communication because services must be able to reach each other even if they run on separate devices. In general, microservice orchestrators view connected nodes as processors to which services can be freely scheduled. Likewise, client nodes are agnostic to the services they run. However, in edge computing, where data comes from a multitude of heterogeneous sources, prior knowledge of which devices a workload can be scheduled for is necessary. This includes, for example, that the device has the expected sensors. No matter the platform, this information has to be known or sourced before creating the task or service.

Among the surveyed platforms, Stream Analyze requires the least amount of client resources by a wide margin. The very small memory footprint of its client runtime allows it to run in extremely resource-constrained environments. The runtime can also exist without an operating system in a bare-metal deployment. The other platforms, AutoSPADA included, need sufficient client resources to run a Linux environment to containerize their workloads.

8. Discussion and future work

The AutoSPADA platform has been deployed in various stages during its development process. The client has been deployed to the cloud, standalone hardware, and the same hardware used in vehicles through the WICE remote binary deployment. We also held a final demonstration and workshop, where we remotely deployed the platform onto live vehicles and guided our collaborators from Volvo Cars to use it. During the workshop session, we explored how our collaborators could express their use cases within AutoSPADA. Two use cases were considered: one related to durability analysis and another to measuring the impact of one-pedal driving on energy consumption in an electric car. On the European Union's scale of Technology Readiness Levels (TRL) [18], the platform reaches level seven because it was demonstrated running on vehicles. This grading is further motivated by a European Commission website that exemplifies TRL 7 for system technologies: "Testing is moved to operational environments such as a vehicle or machines" [57].

Throughout the project, we implemented and tested various simulated use cases for testing and demonstration purposes. Two such use cases were *active learning* and *hazardous spot mapping*. Active learning is a machine learning field where the training model itself is used to decide what data is most informative to collect for labeling [58]. This case demonstrated how machine learning inference could be used on edge devices. In the hazardous spot mapping case, we updated an external database to create a map of slippery roads. Although this particular type of service already exists, e.g., the Road Surface Alerts by NIRA Dynamics [59], our use case demonstrated the efficiency and simplicity of interactively prototyping fleet-wide services using AutoSPADA. These use cases also highlight that vehicular data is not limited to diagnostics, but can be used as an asset to create new software services that add customer and business value [11].

Rapid prototyping in AutoSPADA is facilitated by the provided Python libraries designed to be intuitive for anyone with Python experience. Users interact with AutoSPADA entirely through Python, allowing them to work conveniently in, e.g., interactive Jupyter notebooks [60]. Our experiment shows that the latencies in the system are sufficiently low to suggest that the platform is suitable for interactive use. Moreover, our client program performs well compared to other platforms on the market.

The AutoSPADA platform helps teams of data scientists to develop their data practices and answer complex case-specific questions. These questions may cover general topics such as whether a federated approach can improve performance and reduce data storage costs or how to pre-process data from heterogeneous sources before using it as input to a machine learning model. They may also include specific questions such as how the use of regenerative braking compares to coasting in terms of energy consumption in electric vehicles. Some of these questions can initially be assessed through historical data, but at some point, they need to be evaluated in the real world. In many cases, however, it is best to start live tests as soon as possible. Additionally, some questions are better answered using real-time data, especially if an external data source needs to be merged with vehicle data. For

example, historical weather data may be unavailable, but requesting a current weather report is straightforward.

Given AutoSPADA's current TRL, a natural next milestone for a follow-up project would be to deploy it to an operational test fleet. This would also allow us to understand the platform's operational characteristics and better quantify its performance beyond the idealized experiment presented in this paper.

8.1. Future considerations

Other languages may offer better performance or compatibility for running any of the services on the platform. Since the interfaces between the user, server, and client nodes are specified using protobuf, they can easily be compiled to several other target languages. This allows for a switch of implementation language should it ever be needed.

Several platform features and enhancements could not be implemented within the scope of the AutoSPADA project. We conclude by highlighting some valuable future additions.

8.1.1. Data abstraction

Because our project partners are in the automotive industry, our work has focused on connected vehicles. Specifically, reading device data through the `get_signal` function, as shown in Listing 1, only works if the clients are WICE-equipped vehicles. For other hardware platforms, users must include code to read the desired signals in their payloads. Making the data subscription service configurable to support more protocols, such as MQTT, would expand its functionality to a broader range of client devices.

8.1.2. Resource management

The AutoSPADA platform cannot manage clients through the user API. For example, you cannot prune Docker images on clients in the current user API. Redundant Docker artifacts should be removed regularly since client devices are resource-constrained.

Because tasks run in Docker containers, each task's resource usage can be limited. The maximum amount of resources used per task should be a configurable parameter on the assignment level. Moreover, AutoSPADA currently does not assess the busyness of clients, which means that they are vulnerable to simple denial-of-service attacks from malicious or simply inattentive users. A mechanism to refuse or schedule pending tasks for clients with exceeded resource quotas should be implemented.

8.1.3. Customizable container environment

Only the Python standard library and the AutoSPADA library are available for use in payloads. However, users surely want to leverage Python's rich ecosystem of third-party packages when writing payload code. This means that the Docker image used to containerize tasks on clients needs to be customizable. Though it is already possible to change the image, this is impractical since it requires a manual reconfiguration and redeployment of the AutoSPADA client on each edge device. A more flexible approach is to configure the Docker image used for task containerization *per assignment*.

Allowing customizable Docker images raises the question of how to implement Python package customization without increasing the user complexity beyond an acceptable level. A possible way forward is to support up to three customization alternatives with increasing flexibility and complexity:

1. Provide users with pre-built images with different sets of common packages, such as the ubiquitous NumPy package for array programming [61].
2. Allow users to specify their required dependencies via, e.g., a `pyproject.toml` file and build the image centrally.
3. Allow users to build custom Docker images using an AutoSPADA base image.

The second alternative adds server-side complexity and cost since a central build server is required. The third and most flexible alternative adds user-side complexity, especially if images need to be built for foreign CPU architectures. Building custom images is, therefore, most suited to expert users willing to invest more time and effort. Related to device resource management, placing custom Docker images on edge devices also makes image pruning an essential feature.

8.1.4. Client metadata

Users are fully responsible for assigning tasks to clients and making sure that the client is equipped with the expected hardware and sensors. This typically means that a user queries the platform for all online clients and cross-references that list with external information about each client device. However, it would be convenient for users if they only need to specify client requirements instead of forcing task placement onto a specific device. After all, the user might only require that a specific sensor exists. Also, if the server can decide on task placement, tasks can be held in a pending state until a compatible client becomes available. This would avoid forcing users to guess which client to choose if all compatible clients are offline. Besides the additional server-side logic, such a mechanism requires clients to register metadata about themselves, such as the device type, CPU architecture, and available sensors.

CRediT authorship contribution statement

Adrian Nilsson: Writing – original draft, Visualization, Software, Investigation. **Simon Smith:** Writing – original draft, Software, Investigation. **Jonas Hagmar:** Writing – original draft, Supervision, Conceptualization. **Magnus Önnheim:** Writing – review & editing, Supervision, Conceptualization. **Mats Jirstrand:** Writing – review & editing, Project administration, Funding acquisition.

Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this work, the authors used GrammarlyGO and Microsoft Copilot in order to quickly improve the readability and flow of the text. After using either tool, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research was financially supported by the projectAutomotive Stream Processing and Distributed Analytics (AutoSPADA) in the funding program FFI: Strategic Vehicle Research and Innovation (DNR 2019-05884), which is administered by VINNOVA, the Swedish Government Agency for Innovation Systems.

We are grateful to AI Sweden for providing remote access to MX-4 units through the Edge Learning Lab. The Edge Learning Lab has been a valuable test environment during the development of the AutoSPADA platform. Special thanks to Anders Nord and Viktor Larsson at Volvo Cars for contributing the durability and one-pedal drive use cases, respectively, and for their collaboration in testing the platform on Volvo cars.

Data availability

Code may be made available on request subject to license and collaborative conditions.

References

[1] Cisco, Cisco Annual Internet Report (2018–2023), White Paper, Cisco, 2020.
[2] M. Johanson, S. Belenki, J. Jalminger, M. Fant, M. Gjertz, Big automotive data: Leveraging large volumes of data for knowledge-driven product development, in: 2014 IEEE International Conference on Big Data (Big Data), 2014, pp. 736–741, http://dx.doi.org/10.1109/BigData.2014.7004298.
[3] M. Petersson, M. Papatriantafilou, M. Axelson-Fisk, M. Jirstrand, A. Nord, A. Koppisetty, M. Johanson, BADA – On-board Off-board Distributed Data Analytics, Public Report, VINNOVA - FFI, 2020.
[4] W. Shi, S. Dustdar, The promise of edge computing, Computer 49 (5) (2016) 78–81, http://dx.doi.org/10.1109/MC.2016.145.
[5] F. Liu, G. Tang, Y. Li, Z. Cai, X. Zhang, T. Zhou, A survey on edge computing systems and tools, Proc. IEEE 107 (8) (2019) 1537–1562, http://dx.doi.org/10.1109/JPROC.2019.2920341.
[6] Y. Mao, C. You, J. Zhang, K. Huang, K.B. Letaief, A survey on mobile edge computing: The communication perspective, IEEE Commun. Surv. Tutor. 19 (4) (2017) 2322–2358, http://dx.doi.org/10.1109/COMST.2017.2745201.
[7] D.C. Nguyen, M. Ding, P.N. Pathirana, A. Seneviratne, J. Li, H. Vincent Poor, Federated learning for internet of things: A comprehensive survey, IEEE Commun. Surv. Tutor. 23 (3) (2021) 1622–1658, http://dx.doi.org/10.1109/COMST.2021.3075439.
[8] A.R. Elkordy, Y.H. Ezzeldin, S. Han, S. Sharma, C. He, S. Mehrotra, S. Avestimehr, Federated analytics: A survey, APSIPA Trans. Signal Inf. Process. 12 (1) (2023) http://dx.doi.org/10.1561/116.00000063.
[9] S. Deng, H. Zhao, W. Fang, J. Yin, S. Dustdar, A.Y. Zomaya, Edge intelligence: The confluence of edge computing and artificial intelligence, IEEE Internet Things J. 7 (8) (2020) 7457–7469, http://dx.doi.org/10.1109/JIOT.2020.2984887.
[10] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, J. Zhang, Edge intelligence: Paving the last mile of artificial intelligence with edge computing, Proc. IEEE 107 (8) (2019) 1738–1762, http://dx.doi.org/10.1109/JPROC.2019.2918951.
[11] J. Bosch, H.H. Olsson, Digital for real: A multicase study on the digital transformation of companies in the embedded systems domain, J. Softw.: Evol. Process 33 (5) (2021) e2333, http://dx.doi.org/10.1002/smr.2333, e2333 JSME-20-0122.R2.
[12] J. Wan, J. Liu, Z. Shao, A.V. Vasilakos, M. Imran, K. Zhou, Mobile crowd sensing for traffic prediction in internet of vehicles, Sensors 16 (1) (2016) http://dx.doi.org/10.3390/s16010088.
[13] L. Zhao, H. Qian, A. Hawbani, A.Y. Al-Dubai, Z. Tan, K. Yu, A.Y. Zomaya, Overtaking feasibility prediction for mixed connected and connectionless vehicles, IEEE Trans. Intell. Transp. Syst. (2024) 1–16, http://dx.doi.org/10.1109/TITS.2024.3398602.
[14] L. Hu, J. Ou, J. Huang, Y. Chen, D. Cao, A review of research on traffic conflicts based on intelligent vehicles, IEEE Access 8 (2020) 24471–24483, http://dx.doi.org/10.1109/ACCESS.2020.2970164.
[15] H. Grimmemyhr, B. Havers-Zulka, A. Koppisetty, V. Gulisano, M. Papatriantafilou, R. Duvignau, E.M. Schiller, M. Jirstrand, M. Johanson, AutoSPADA: Automotive Stream Processing and Distributed Analytics Analytics, Public Report, VINNOVA - FFI, 2023.
[16] G. Ulm, S. Smith, A. Nilsson, E. Gustavsson, M. Jirstrand, OODIDA: On-board/off-board distributed real-time data analytics for connected vehicles, Data Sci. Eng. 6 (1) (2021) 102–117, http://dx.doi.org/10.1007/s41019-021-00152-6.
[17] G. Ulm, S. Smith, A. Nilsson, E. Gustavsson, M. Jirstrand, Facilitating rapid prototyping in the distributed data analytics platform OODIDA via active-code replacement, Array 8 (2020) 100043, http://dx.doi.org/10.1016/j.array.2020.100043.
[18] M. Héder, From NASA to EU: the evolution of the TRL scale in public sector innovation, Innov. J. 22 (2017) 1.
[19] S. Cass, SQL should be your second language, IEEE Spectr. 59 (10) (2022) 20–21, http://dx.doi.org/10.1109/MSPEC.2022.9915547.
[20] J. Armstrong, A history of Erlang, in: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages, in: HOPL III, Association for Computing Machinery, New York, NY, USA, 2007, 6–1–6–26, http://dx.doi.org/10.1145/1238844.1238850.
[21] E. Rescorla, The transport layer security (TLS) protocol version 1.3, 2018, http://dx.doi.org/10.17487/RFC8446, RFC 8446.
[22] A.J.B. Rodrigues, V. Fördős, Towards secure Erlang systems, in: Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang, in: Erlang 2018, Association for Computing Machinery, New York, NY, USA, 2018, pp. 67–70, http://dx.doi.org/10.1145/3239332.3242768.
[23] S. Lehrig, H. Eikerling, S. Becker, Scalability, elasticity, and efficiency in cloud computing: A systematic literature review of definitions and metrics, in: 2015 11th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA), 2015, pp. 83–92, http://dx.doi.org/10.1145/2737182.2737185.

[24] The Kubernetes Authors, Kubernetes, [online]. URL https://kubernetes.io/. (cited 10 September 2024).
[25] M. Stonebraker, R. Cattell, 10 rules for scalable performance in 'simple operation' datastores, Commun. ACM 54 (6) (2011) 72–80, http://dx.doi.org/10.1145/1953122.1953144.
[26] R. Cattell, Scalable SQL and noSQL data stores, SIGMOD Rec. 39 (4) (2011) 12–27, http://dx.doi.org/10.1145/1978915.1978919.
[27] U. Cubukcu, O. Erdogan, S. Pathak, S. Sannakkayala, M. Slot, Citus: Distributed postgreSQL for data-intensive applications, in: Proceedings of the 2021 International Conference on Management of Data, SIGMOD '21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 2490–2502, http://dx.doi.org/10.1145/3448016.3457551.
[28] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels, Dynamo: Amazon's highly available key-value store, in: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, SOSP '07, Association for Computing Machinery, New York, NY, USA, 2007, pp. 205–220, http://dx.doi.org/10.1145/1294261.1294281.
[29] DB-engines ranking, 2023, [online] URL http://web.archive.org/web/20230709014024/https://db-engines.com/en/ranking. (cited 2023-07-19).
[30] MongoDB, MongoDB Architecture Guide, Tech. rep., MongoDB, Inc, 2018.
[31] W. Khan, T. Kumar, C. Zhang, K. Raj, A.M. Roy, B. Luo, SQL and noSQL database software architecture performance analysis and assessments—A systematic literature review, Big Data Cogn. Comput. 7 (2023) http://dx.doi.org/10.3390/bdcc7020097.
[32] MongoDB, Multi-Document ACID Transactions on MongoDB, Tech. Rep., MongoDB, Inc, 2020.
[33] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J.P. Fernandes, J. Saraiva, Energy efficiency across programming languages: How do energy, time, and memory relate? in: Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, in: SLE 2017, Association for Computing Machinery, New York, NY, USA, 2017, pp. 256–267, http://dx.doi.org/10.1145/3136014.3136031.
[34] MQTT Version 5.0, Standard, Organization for the Advancement of Structured Information Standards (OASIS), 2019, URL https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html. Edited by Andrew Banks, Ed Briggs, Ken Borgendale, and Rahul Gupta.
[35] N. Naik, Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP, in: 2017 IEEE International Systems Engineering Symposium, ISSE, 2017, pp. 1–7, http://dx.doi.org/10.1109/SysEng.2017.8088251.
[36] ActiveMQ, [online]. URL https://activemq.apache.org/. (cited 10 September 2024).
[37] EMQX, [online]. URL https://www.emqx.com/en. (cited 10 September 2024).
[38] R.A. Light, Mosquitto: server and client implementation of the MQTT protocol, J. Open Sour. Softw. 2 (13) (2017) 265, http://dx.doi.org/10.21105/joss.00265.
[39] RabbitMQ, [online]. URL https://www.rabbitmq.com/. (cited 10 September 2024).
[40] E. Longo, A.E.C. Redondi, M. Cesana, P. Manzoni, BORDER: A benchmarking framework for distributed MQTT brokers, IEEE Internet Things J. 9 (18) (2022) 17728–17740, http://dx.doi.org/10.1109/JIOT.2022.3155872.
[41] S.a. Popić, D. Pezer, B. Mrazovac, N. Teslić, Performance evaluation of using protocol buffers in the internet of things communication, in: 2016 International Conference on Smart Systems and Technologies, SST, 2016, pp. 261–265, http://dx.doi.org/10.1109/SST.2016.7765670.
[42] J.C. Viotti, M. Kinderkhedia, A benchmark of JSON-compatible binary serialization specifications, 2022, arXiv:2201.03051.
[43] K. Walsh, J. Manferdelli, Mechanisms for mutual attested microservice communication, in: Companion Proceedings of The10th International Conference on Utility and Cloud Computing, in: UCC '17 Companion, Association for Computing Machinery, New York, NY, USA, 2017, pp. 59–64, http://dx.doi.org/10.1145/3147234.3148102.
[44] T. Yarygina, A.H. Bagge, Overcoming security challenges in microservice architectures, in: 2018 IEEE Symposium on Service-Oriented System Engineering, SOSE, 2018, pp. 11–20, http://dx.doi.org/10.1109/SOSE.2018.00011.
[45] N. Sakimura, J. Bradley, M.B. Jones, B. de Medeiros, C. Mortimore, OpenID Connect Core 1.0 Incorporating Errata Set 2, Specification, The OpenID Foundation, 2023, URL https://openid.net/specs/openid-connect-core-1_0.html.
[46] P. Siriwardena, Openid connect (OIDC), in: Advanced API Security: OAuth 2.0 and beyond, A Press, Berkeley, CA, 2020, pp. 129–155, http://dx.doi.org/10.1007/978-1-4842-2050-4_6.
[47] D. Hardt, The oauth 2.0 authorization framework, 2012, http://dx.doi.org/10.17487/RFC6749, RFC 6749.
[48] M.B. Jones, D. Hardt, The oauth 2.0 authorization framework: Bearer token usage, 2012, http://dx.doi.org/10.17487/RFC6750, RFC 6750.
[49] Docker, [online]. URL https://www.docker.com/. (cited 10 September 2024).
[50] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM 21 (7) (1978) 558–565, http://dx.doi.org/10.1145/359545.359563.
[51] M. Johanson, WICE: Automotive telematics, fleet management, rapid prototyping and remote software download for connected vehicles, White Paper, Alkit Communications AB, n.d..
[52] A.M. Keshk, R. Dyer, Method chaining redux: An empirical study of method chaining in Java, Kotlin, and Python, in: 2023 IEEE/ACM 20th International Conference on Mining Software Repositories, MSR, 2023, pp. 546–557, http://dx.doi.org/10.1109/MSR59073.2023.00080.
[53] I. Čilić, P. Krivić, I. Podnar Žarko, M. Kušek, Performance evaluation of container orchestration tools in edge computing environments, Sensors 23 (8) (2023) http://dx.doi.org/10.3390/s23084008.
[54] Stream Analyze Resources, [online]. URL https://www.streamanalyze.com/resources. (cited 07 July 2023).
[55] Y. Xiong, Y. Sun, L. Xing, Y. Huang, Extend cloud to edge with KubeEdge, in: 2018 IEEE/ACM Symposium on Edge Computing, SEC, 2018, pp. 373–377, http://dx.doi.org/10.1109/SEC.2018.00048.
[56] F. Desbiens, Edge computing, in: Building Enterprise IoT Solutions with Eclipse IoT Technologies: An Open Source Approach To Edge Computing, A Press, Berkeley, CA, 2023, pp. 271–296, http://dx.doi.org/10.1007/978-1-4842-8882-5_11.
[57] About technology readiness levels, 2020, [online]. URL https://euraxess.ec.europa.eu/career-development/researchers/manual-scientific-entrepreneurship/major-steps/trl. (cited 27 September 2023).
[58] B. Settles, Active Learning Literature Survey, Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009.
[59] Precise data for greater safety: NIRA dynamics launches road surface alerts with audi to improve slippery roads warning system, 2022, URL https://niradynamics.se/precise-data-for-greater-safety-nira-dynamics-launches-road-surface-alerts-with-audi-to-improve-slippery-roads-warning-system/. (cited 11 September 2024).
[60] M. Beg, J. Taka, T. Kluyver, A. Konovalov, M. Ragan-Kelley, N.M. Thiéry, H. Fangohr, Using jupyter for reproducible scientific workflows, Comput. Sci. Eng. 23 (2) (2021) 36–46, http://dx.doi.org/10.1109/MCSE.2021.3052101.
[61] C.R. Harris, K.J. Millman, S.J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N.J. Smith, R. Kern, M. Picus, S. Hoyer, M.H. van Kerkwijk, M. Brett, A. Haldane, J.F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, T.E. Oliphant, Array programming with NumPy, Nature 585 (7825) (2020) 357–362, http://dx.doi.org/10.1038/s41586-020-2649-2.