Received 10 December 2023, accepted 3 January 2024, date of publication 12 January 2024, date of current version 22 January 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3353613

RESEARCH ARTICLE

OPACA: Toward an Open, Language- and Platform-Independent API for Containerized Agents

BENJAMIN ACAR[1], TOBIAS KÜSTER[2], OSKAR F. KUPKE[2], ROBERT K. STREHLOW[2], MARC GUERREIRO AUGUSTO[1], FIKRET SIVRIKAYA[2], AND SAHIN ALBAYRAK[1]
[1]Chair of Agent Technology, Technische Universität Berlin, 10623 Berlin, Germany
[2]GT-ARC gGmbH, 10587 Berlin, Germany

Corresponding author: Benjamin Acar (benjamin.acar@dai-labor.de)

This paper is based on the research conducted in the gemeinwohlorientierter KI-Anwendungen (Go-KI) project (Offenes Innovationslabor KI zur Förderung gemeinwohlorientierter KI-Anwendungen), funded by the German Federal Ministry of Labour and Social Affairs (BMAS) under the funding reference number DKI.00.00032.21. Furthermore, we acknowledge support by the German Research Foundation and the Open Access Publication Fund of TU Berlin.

ABSTRACT While multi-agent frameworks can provide many advanced features, they often suffer from not being able to seamlessly interact with the outside world, e.g., with web-services or other multi-agent frameworks. This may be one factor that hinders a broader application of multi-agent systems in production systems. A possible solution to this problem is the combination of multi-agent systems with the concepts of micro-services and containerization, providing language-agnostic open interfaces, as well as encapsulation and modularity. In this paper, we propose an API and reference implementation that can be employed by multi-agent systems based on different languages and frameworks. Each agent component is encapsulated in a container and is accessed through its parent runtime platform, which takes care of aspects such as authentication, input validation, monitoring and other infrastructure tasks. Multiple runtime platforms can then be connected to form systems of distributed, heterogeneous multi-agent societies.

INDEX TERMS Multi-agent systems, microservices, Kubernetes, Docker, API.

I. INTRODUCTION

Over the last few decades, the field of Multi-Agent System (MAS) has advanced considerably, and many frameworks have been developed to implement them [1], [2]. However, despite many advanced features and capabilities, they have mostly remained an academic subject with limited application in productive systems, e.g. in industry. One possible hindrance is the lack of standardized interfaces and interoperability presented by typical multi-agent frameworks. Instead, industry usually favours microservice-architectures, often in combination with virtualization and containerization techniques, decoupling the individual parts of the system from each other and the execution environment.

The associate editor coordinating the review of this manuscript and approving it for publication was Qiang Li.

However, it is noticeable that the development of microservice-based MAS frameworks has been relatively limited. Such frameworks represent a promising area of future research and innovation, since the benefits of scalability, modularity, and fault-tolerance provided by microservices and containerization have the potential to significantly enhance the capabilities and performance of MAS. Therefore, we decided to develop a new framework, called OPACA (Open, Language- and Platform-Independent API for Containerized Agents), embodying the principles of a microservices architecture, where agents are deployed in Docker containers or Kubernetes pods, while retaining typical properties of agents, such as being more autonomous and dynamic than traditional microservices.

Designed to be small, loosely coupled and independently deployable, different microservice components represent individual groups of agents, while the containerization with

Docker and Kubernetes acts to isolate the environment in which an application runs. Thus, each group of agents in the MAS can operate autonomously and make decisions based on its own set of rules and goals. By using each other's REST interface, the agents can communicate with each other, share information and collaborate on complex tasks. The interactions between these agents are facilitated by a platform, which acts as a conductor, responsible for orchestrating the information flow through the MAS.

The main benefits of combining microservices and containerization with multi-agent systems are:

- It greatly improves scalability, allowing the system to easily adapt to changing requirements by simply adding or removing microservices/agents as needed. This dynamic scalability allows the system to efficiently handle both small and large operations, ensuring optimal resource allocation and eliminating bottlenecks.
- The ability to isolate failures increases the robustness and reliability of the MAS as a whole. DevOps practices are enabled by the flexibility of microservice architecture [3]. Updates and enhancements can be made to individual agents without affecting the entire system.

Overall, our MAS is a significant step towards modern MAS. In this paper, we will give a brief overview of our framework and the concepts applied.

The remainder of this paper is structured as follows. In Sections II we will provide more background on the fundamental concepts in this paper, and have a look at similar approaches in Section III. Thereafter, in Section IV we will present the approach we follow in this work and discuss an implementation in Section V. Finally, we will perform a preliminary evaluation of our approach in Section VI before wrapping up in Section VII.

II. BACKGROUND

The popularity of distributed artificial intelligence (DAI) has grown over the past years. Rather than looking at a single entity, DAI looks at many entities interacting with each other, promising to significantly affect many areas, including cognitive science, distributed systems, human-computer interaction, and others [4]. Parallel artificial intelligence (AI), distributed problem solving (DPS), and MAS have been the main areas of DAI research. Parallel AI typically refers to techniques, such as multiprocessing or clustering, that accelerate operations to solve a common task. Distributed problem solving explores how to solve a problem by combining the resources and expertise of many computing entities, similar to parallel AI.

A network of individual agents is defined as a MAS. In such a system, the agents can share information and communicate with each other to solve problems that are beyond the scope of a single agent [5]. The efficiency of MAS results from the intrinsic partitioning of these networks, which divides larger work into numerous smaller tasks and assigns each of these tasks to different agents [6].

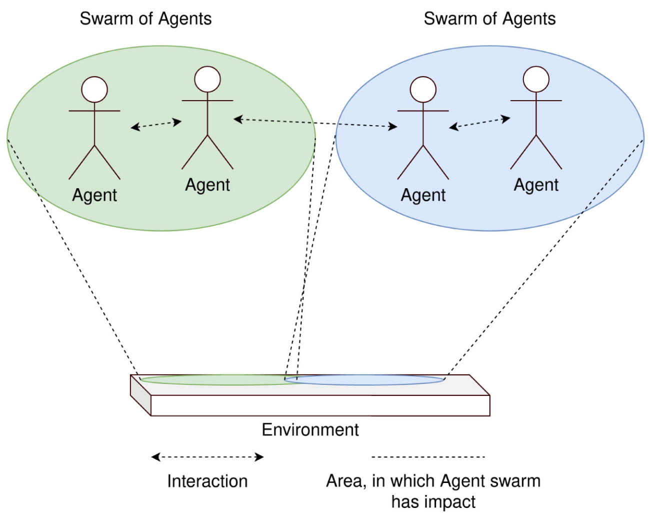


FIGURE 1. Agent swarms, perceiving and changing their environment by performing actions.

A popular definition of agents refers to autonomous entities that can act intelligently and perceive their environment. Communication between agents is crucial, whether the communication partners are human or artificial [7].

According to [5], common properties of such agents include:

- Situatedness: the agent is able to interact with its environment.
- Autonomy: the agent acts independently.
- Inferential capability: the agent is able to follow abstract goals.
- Responsiveness: the agent responses to changes on its environment.
- Pro-activeness: the agent pursues its goals, rather than just responding to its environment.
- Social behaviour: the agent interacts with other agents.
- Other properties such as mobility, temporal continuity, collaborative behaviour, etc.

Therefore, what makes MAS unique is that each component is autonomous, self-interested, and focused on achieving its own goals [7]. A visualization of MAS can be found in Fig. 1.

Compared to this, programming classical functional applications is different. Functional software is designed to take input, process it, and then produce output, just like a mathematical function [8]. In practice, however, this simple structure is often insufficient for the realization of complex applications. Instead of an input - compute - output relationship, there is often a continuous interaction between the application and its environment, for example when we think of operating systems [8]. In this case, MAS come into account. The development of MAS requires expertise from various fields, including concurrent programming to deal with task coordination executed on various machines, AI techniques to give systems the ability to deal with unexpected situations and act independently, and software engineering techniques to structure the development process [7]. MAS research focuses on the results of distributed computing. It asks new questions about how agents must communicate with each other to coordinate their actions and solve challenging tasks [7].

Microservices, on the other hand, offer a new concept of deployments. Unlike classic monolithic software approaches, only narrow areas of a software are combined to form a microservice. Every microservice fulfills only a few tasks and is therefore particularly less complex. This has the advantage that microservices are easy to manage, especially during development. Microservice architectures have proven to be reliable and because of their small and independent components. They also address the need for easier maintenance and greater flexibility in the programming languages, frameworks, and technologies used. Decoupling components allows groups to work in parallel on bigger systems, without interfering with each other, granting organizations to achieve fast-paced development cycles [9], [10]. Studies have shown that the usage of microservices in common organizations is high [11]. The REST protocol plays an important role here because it is widely used as an interface in microservice architectures [12], [13]. For more information about microservices, see [14].

On the other hand, Multi Agent Systems (MAS) promote the idea of systems in which agents serve as intelligent entities, able to perceive their environment, make decisions, and take actions to achieve specific goals. Previous studies have shown fundamental similarities between microservices and MAS [15]. Therefore, concepts are established to bring these two development paradigms together, in order to leverage the capabilities of both [15] and [16].

III. RELATED WORK

In the past, many studies have introduced frameworks for developing multi-agent systems.

Charpenay et al. introduces Hypermedea [17], an open source framework designed for Web and Web of Things (WoT) agents. Hypermedea is a powerful and versatile solution built on the foundation of JaCaMo, a multi-agent oriented programming platform. At its core, Hypermedea is a collection of artifacts. Each artifact contributes to the functionality of the framework. Retrieving RDF representations of resources is the purpose of the Linked Data artifact. Next comes the Ontology artifact. It processes OWL definitions found in incoming named graphs and actively listens for changes in the knowledge base. In addition, this artifact performs OWL reasoning. Taking advantage of the meta-programming capability of Jason, the planner artifact is the center of attention. It exposes synthesized plans as observable properties. Finally, there is a Thing artifact class. Its purpose is to reflect the presence of various "things" in the environment.

Boissier et al. [18] explore the fusion of three programming paradigms: agent-oriented, organization-oriented, and environment-oriented. The goal is to create a comprehensive framework that combines these paradigms to facilitate the development of complex multi-agent systems. To accomplish this, the JaCaMo platform is used as the basic infrastructure, utilizing three pre-existing platforms: Jason, Moise, and CArtAgO. Within CArtAgO, software environments can be designed and programmed as dynamic sets of computational entities known as artifacts. These artifacts are assembled in workspaces. These workspaces can be distributed across different nodes in a network. Meanwhile, Jason includes an agent-oriented programming language called AgentSpeak, which serves as a platform for constructing multi-agent systems. Finally, the Moise framework is the implementation of a programming model specifically designed for the organizational aspect of these systems.

However, the frameworks seem cumbersome, with multiple, individually complex components. There is also the question of how to deploy such components in real production environments. This may be one of the reasons why, despite the fact that agent systems are derived from successful concepts in sociology, their application in industry is limited.

According to Dastani et al. [19] it is common to say that industry needs tools and technologies that behave predictably. In contrast, agents and multi-agent systems are generally considered to be intelligent and adaptive systems, which are often considered to behave unpredictably. In addition, complete control over the execution of their software systems is often preferred by developers in the industry. Most existing programming languages rely on interpreters to perform complex choices, like deliberation and control, on behalf of the programmer. Furthermore, the industry has a tendency to depend on standard software technologies and a general resistance to paradigm changes in software technologies and methods.

In contrast, microservices have found their way in broad software engineering [11]. Previous studies have shown fundamental similarities between microservices and multi agent systems [15], such as the isolation of the state, enabling them to function autonomously; working in a distributed manner, across different nodes; and being loosely coupled. However, the studies also report differences, for instance microservices are designed as small units to be used for a narrow purpose, while agents can be considered as arbitrary complex systems. Also, microservices have a purely reactive behavior, while agents can be proactive, able to take initiative.

Despite the differences in those concepts, Collier et al. [15] recognize the many similarities as well and describe their approach for developing microservice-based agents. Therefore, other researches focus on that topic as well. Limón et al. [16] present SagaMAS: a multi-agent framework that is designed to handle distributed transactions in the microservices architecture. It streamlines transaction coordination and relieves microservice developers of this responsibility by acting as a decoupled, autonomous layer. Without a central command unit, the SagaMAS model operates in a semi-orchestrated fashion. Instead, agents request each other to perform subsequent steps, fostering a decentralized workflow. Asynchronous communication allows agents to share and request without having to wait for immediate responses. Each microservice, regardless of its server location, is associated with a dedicated agent in SagaMAS. In order to initiate a transaction, the microservice communicates the start to its associated agent in a variety of ways. For example, the use of a pipe if the microservice and the agent are located on the same server or the use of agent technologies like CArtAgO to react to microservice signals.

The work of Jagutis et al. [20] focuses on traffic simulation systems (TSS), an established field of study with the goal of improving the planning, design, and operation of transportation systems. They are using agent-based modeling (ABM) as one of the means to achieve this goal. ABM is an approach that analyzes how complex systems behave by modeling them as a population of interacting individuals in a shared environment. To construct ABM simulations, the environment is represented as a collection of interconnected hypermedia resources using Hypermedia Multi-Agent System (MAS) simulation. Each hypermedia resource acts as a micro-environment. It is connected to other related micro-environments through hyperlinks. These micro-environments are implemented as microservices. They communicate using REST. Building complex agent-based simulations by combining loosely coupled reusable components is the key principle behind the Hypermedia MAS Simulation approach. Thus, a simulation is constructed by integrating multiple subsimulations, each being an instance of a microservice.

Despite the similarities between microservices and agent systems, and the adoption of emerging design principles such as the use of REST as an interface between agents, we could not find any framework that has emerged that effectively implements these microservices agent systems while seamlessly integrating with popular microservices frameworks such as Kubernetes. For this reason, we propose our framework, which can be used to take advantage of both the agent principles as well as the microservice paradigm.

IV. APPROACH

In a nutshell, our approach combines multi-agent systems with microservices and container technologies in order to integrate different multi-agent frameworks and other types of software services, with different strength and scopes of application, allowing them to run on different hosts, both in the cloud and on premise, each encapsulated inside containers and communicating with each other and the "external world" using a common API.

A. REQUIREMENTS

The API was build around a set of simple design principles and requirements:

1) **Open, Standardized Interfaces:** It should be easy to implement and allow the combination of different systems and different architectures to allow the integration of legacy components. Established protocols by the Internet Engineering Task Force (IETF) acknowledge the importance of standardization and interoperability [21].
2) **Language Agnostic:** It should be a generic interface that can be implemented in different languages and run on different hardware. As shown in [22], different programming languages use different concepts and therefore have different strengths. By being language agnostic, the API allows developers to take advantage of these different strengths.
3) **Modularity and Reusability:** It should be modular and allow reusability of existing components within the system. As described in [23], modularity is an important design principle for software design. Concepts such as Docker images also have shown that modularity often comes along with reusability [24].
4) **Self-Descriptive:** It should be self-describing in the sense that the components using the API provide information about their functionality and how to use them. Concepts such as the Web Service Description Language (WSDL) [25] have shown the benefits of providing an abstract layer of information to describe expected input parameters and result types, as well as a basic description of the function's intent.
5) **Open, Multi-Tenancy:** It should allow new tenants to enter the system, connect services/agents by different vendors, and create new value out of them. The benefits of such a multi-tenancy include efficiency through shared resources and competitive costs [26].
6) **Distribution:** It should allow the seamless distribution of computing tasks, on the one hand connecting different types of devices (e.g. microcomputers and servers), and on the other hand allowing for e.g. replication, being an enabler of high performance and reliable computing [27].

In particular, the API does not require the use of a particular agent framework. While the use of *some* agents framework or language is encouraged, and the reference implementation is using one, individual agent components can also be implemented as simple web services, or wrap existing web services or algorithms, thus making it easier to e.g. wrap legacy components or deploying components on resource-limited devices.

The overall goal of the API is to help in the development of autonomous, scalable, reusable and secure distributed components.

B. THE OPACA API

A multi-agent system in the OPACA approach consists of two types of components:

- *Agent Containers* are containerized applications (that may or may not contain actual "agents") that implement the OPACA Agent Container API.

- *Runtime Platforms* are used to manage one or more Agent Containers, deploying those in Docker or Kubernetes, acting to connect different Agent Containers (and different Runtime Platforms) while also providing basic services such as yellow pages, authentication, monitoring, etc.

Table 1 shows an overview of the different routes for the Agent Container and Runtime Platform components. In the following we will describe the different routes (REST services) the OPACA API expects from those two types of components.

1) AGENT CONTAINER API

The Agent Container provides REST routes that allow the outside world to find out about the agents running inside the container, and to interact with those agents by means of sending asynchronous messages (unicast and broadcast) and invoking synchronous actions.

- The `agents` route is used to get information on one or all agents and their actions running in the component.
- The `send` and `broadcast` routes are used to send an asynchronous message to a specific agent, or to all agents subscribed to the given channel or topic.
- The `invoke` route is used to invoke an action or service provided by either any or the given agent and get the result (synchronously). Expected parameter and output types are given in the action description.

2) RUNTIME PLATFORM API

The Runtime Platform provides REST routes for managing (adding, listing, updating, removing) both, containers running in the container runtime (Docker or Kubernetes) associated with that particular Runtime Platform, and connections to other Runtime Platforms (and thus their containers) running on a different host.

- The `info` route is used to get information on the Runtime Platform, its currently deployed containers and connections to other platforms.
- The different `containers` routes are used to get information on one specific or all agent containers currently running on this platform as well as to deploy new containers or update or remove existing ones.
- Likewise, the `connections` routes are used to get a list of connected platforms, to establish or remove connections to other platform or update the information on those platforms.

In addition to those, the Runtime Platform provides all the routes of the Agent Container, such as `invoke`, or `send`, forwarding the calls to the Agent Container providing the respective agent or action (if it exists), both within the platform's own containers and within connected platforms.

When an Agent Container is started, the Runtime Platform will pass certain environment variables into the container, such as the container's own ID, information needed for communicating with the platform, as well as application-specific parameters for, e.g., interfacing with external resources, defined in the Agent Container's description.

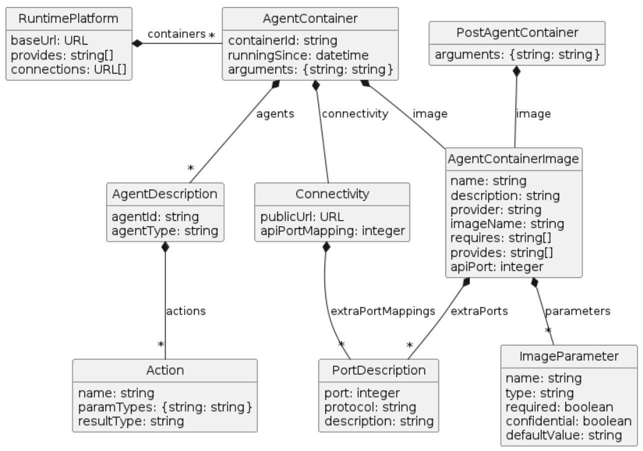


FIGURE 2. UML class diagram of models used in API routes.

3) MODEL ELEMENTS

An overview of the different model classes and their relations is shown in Figure 2. Most attributes should be self-explanatory. The `requires` and `provides` lists can be used to declare certain features an Agent Container requires from or provides to the platform. The `PortDescription` element is used to describe additional ports (besides the main API port) where Agent Containers provide additional services, and the `Connectivity` element denotes which ports on the host those have been mapped to, since multiple containers may request the same port.

V. IMPLEMENTATION

In the following, we will shed light on the technical realization of our approach, described in section IV. At its core, the OPACA API is just that: An API. There is no one definite implementation, as its purpose is to connect different agent components implemented in different languages, using different (agent) frameworks, running on different hardware platforms (Figure 3). In the following we will describe merely the reference implementations of both, the Runtime Platform (RP) and Agent Container (AC). The implementation has been released as open-source under https://github.com/gt-arc/opaca-core.

A. REFERENCE IMPLEMENTATION: RUNTIME PLATFORM

The Runtime Platform has been implemented in Java using Spring Boot.[1] It is in itself not an agent-based application but just a web service, whereas the actual agents are situated in the agent containers. It provides an Open-API compatible Swagger web UI for inspecting and invoking the different REST services provided by the OPACA API. For the routes of the Agent Container API, the RP will look up the AC (or connected RP) containing the requested agent or action and

[1] Spring Boot: https://spring.io/projects/spring-boot

TABLE 1. Different REST routes of the API. First block: Agent container routes; second block: Runtime platform routes.

Method	Route	Description	Input	Output
`GET`	`/agents[/{agent}]`	Get description of one specific or all agents running in the Agent Container, or on the Platform.	-	list of agents
`POST`	`/send/{agent}`	Send asynchronous message to an agent.	message	true/false
`POST`	`/broadcast/{channel}`	Send asynchronous message to all agents subscribed to channel.	message	-
`POST`	`/invoke/{action}[/{agent}]`	Invoke action/service provided by either any or the given agent and get result (synchronously).	map of parameter names to values	action result
`GET`	`/info`	Get information on the Platform, its containers and connections.	-	description of the platform
`POST`	`/containers`	Deploy Agent Container onto this platform.	post-container	id of container
`GET`	`/containers[/{container}]`	Get information on one specific or all agent containers currently running on this platform.	-	list of agent containers
`POST`	`/containers/notify`	Notify about changes in one of its containers.	id of container	true/false
`DELETE`	`/containers/{container}`	Undeploy Agent Container with given ID	-	true/false
`POST`	`/connections`	Connect platform to another platform.	platform URL	true/false
`GET`	`/connections`	Get list of connected platforms.	-	list of URLs
`POST`	`/connections/notify`	Notify about changes in connected platform.	platform URL	true/false
`DELETE`	`/connections`	Disconnect from another platform.	platform URL	true/false

forward the REST call to those. Parameters like the execution environment to use, or containers to create on startup, are defined in a properties file or as environment variables.

Agent containers can be deployed either on Docker (on the same or a remote host) or Kubernetes. Both the Docker and Kubernetes client will check if the image exists, pull it if necessary (with appropriate credentials as required), run the image, establish the connection between the AC and the RP, and finally stop the container when requested or when the RP itself is shut down.[2] For productive use, the Kubernetes client should be used; for testing and if no Kubernetes cluster is available, the Docker client can be used. Clients for alternative execution environments can easily be added.

Besides implementing the basic API routes, the RP also provides additional functionality that is automatically available to all its ACs, such as logging and monitoring of certain events and interactions, and basic authentication and authorization using JSON Web Tokens (JWT).[3] Further features, such as a service registry and a better generic UI (on top of the basic Swagger UI), are currently under development.

1) KUBERNETES INTEGRATION

One of our objectives was the integration of our MAS into Kubernetes (K8s), used as our underlying infrastructure for production environments. This decision was mainly due to the recognition of the widespread application of Kubernetes in common microservices architectures. One of the most striking differences to Docker lies in the way we deploy our platform. Instead of deploying it natively on the host machine, the platform is deployed within a K8s pod. Since

[2]The Runtime Platform can also be configured to restart all containers that were running when it stopped, or to leave the containers running and re-connect to them when restarted.

[3]JSON Web Token: https://jwt.io/

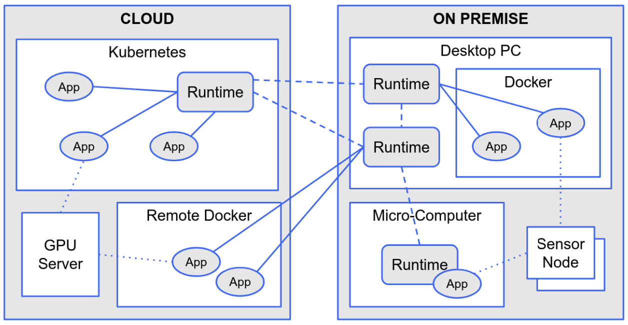


FIGURE 3. Visualization of the deployment strategies within our framework: The runtime platform can be deployed natively on a computer, in Docker or in Kubernetes. Agent containers typically run in Kubernetes or a (remote) Docker engine, but may also be deployed natively and bundled with a Runtime platform for resource-limited devices. Multiple RPs (and their containers) can be connected to each other. Very limited devices such as microcontroller sensor nodes may be connected via the REST API.

the platform itself is used to deploy new agent container pods, and such practices are not very common in microservice environments, the platform needs special privileges within its environment. By using K8s Service Accounts and Role Bindings, the platform pod has the capabilities to monitor, create and delete pods. However, to ensure best practices, all components (agents and platform) are deployed within a dedicated namespace, and the platform's special permissions are limited to that namespace, rather than the full K8s environment. In addition to the deployment of the platform, a service is created that maps to the platform pod, enabling effortless communication with the platform by just using the service DNS. The remaining functionalities and principles are similar to the Docker-centric approach, thanks to the comparable concepts of pods and containers.

B. SAMPLE IMPLEMENTATION: AGENT CONTAINER

A reference/sample Agent Container has been implemented in Kotlin using the JIAC VI framework [28] as a library

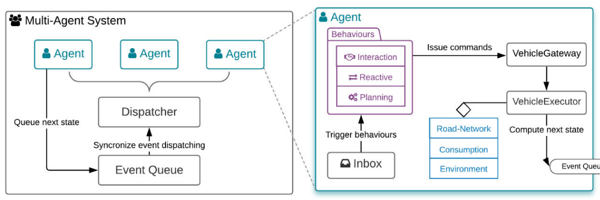


FIGURE 4. Architecture of JIAC VI used inside the sample agent containers [28].

that can be used as the basis for different ACs. The general architecture of a JIAC VI application is shown in Figure 4. The AC uses a simple HTTP Servlet for implementing the REST routes of the Agent Container API. Since the API routes of the AC are not called directly (and in fact not exposed at all), features such as a web UI, input validation, and security can be provided by the RP.

Inside the container, a Container Agent establishes contact to the parent RP, accepts all incoming communication and forwards messages and requests to the appropriate agent(s). Those agents can extend an abstract super-class, Containerized Agent, collecting information on the agent, the actions it provides and messages it will respond to, and registering the agent at the Container Agent when it starts. Afterwards, it runs in its own life-cycle, executing regular behaviors, reacting to messages (from the Container Agent or other agents within the container), etc.

As pointed out earlier, agent containers can be implemented in any language, and they do not have to include a full agent life-cycle as well, but could be mere microservices. A second, simpler Agent Container library has been created for Python, e.g. for using Python's rich set of machine learning libraries.

VI. EVALUATION

In this section, we try to provide a preliminary evaluation of our approach. While we do not yet have enough data to fully assess its feasibility, we will (a) show how it can be applied in a number of realistic application examples, and (b) explain how it fulfills the different requirements we stated earlier in this work.

A. SAMPLE APPLICATIONS

The API and reference implementation are being developed in the course of the *Go-KI* research project.[4] We are currently applying and evaluating the approach in three application examples. Those are still work-in-progress, but the results so far are promising.

1) SMART DESK-BOOKING SYSTEM

As the first example, a smart desk booking system is being developed for the workstations at the *ZEKI Reallabor*,[5] a testing site for hard- and software projects as well as a shared co-working space, which leverages environmental sensors to provide users with a better understanding about the environmental conditions of each workplace, such as CO_2 or noise levels and illumination.

The desk booking system consists of several components that utilize the OPACA API for the communication between these components.

- The first component is a container for managing the sensors and their recorded data. It has a Manager Agent that can add and remove Sensor Agents, representing the individual sensors. These sensor agents regularly collect and persistently store data from the actual sensors and provide actions to update the sensor settings, get the recorded data for the latest or a specific time, or an average over a subset of the recorded data samples.
- The second component is a container for managing the desks and their reservations. Similar to the sensor container, it has a Manager Agent, which provides actions for adding and removing Desk Agents, which represent the individual desks and provide actions for making and deleting reservations or updating the desk's settings. Each desk also has the sensor registered which provides the best environmental data for it, usually this is the sensor closest to it.
- The third component provides a web interface and uses the OPACA API to tie the two other containers together by requesting data from them to enable the users to easily make reservations for desks, see existing reservations and examine each desk's environmental conditions to find the one that suits them the most. It also provides interfaces for deleting reservations, as well as adding, updating and deleting desk and sensor agents, using the actions described above.

In this example, using the OPACA API allowed to dynamically add and remove agents for desks and sensors, and to quickly combine the desk- and sensor-components to a smart, environmental-aware desk-booking system.

2) FEDERATED LEARNING NETWORK

As a second example, we implemented a federated learning (FL) network, which generally consists of multiple participants and a single director working together in a cooperative manner to create one global neural network model.

The director initially sends its model parameters to each participant, which then start training their own, identical neural network models with data only available to the respective participant. After successful training, the participants send their updated model parameters back to the director, which then aggregates said parameters and applies them to the global model on the director's side. This process can be repeated until a satisfying model has been developed.

This design allows the distribution of computational power required to train machine learning models, while also limiting the network traffic by only training with locally available data. It also protects a participant's data, since no data gets shared outside the participant's scope.

[4]Go-KI Project: https://go-ki.org/
[5]ZEKI Reallabor: https://ze-ki.de/zeki-spaces/

With the OPACA API, the director as well as the participants can be modeled by the same type of agent, providing multiple actions to start and participate in an FL process. These agents make use of a popular FL framework called Flower,[6] which creates a dedicated server on the director's side, multiple client agents (incl. from other containers) can connect to, exemplifying the scalability concept of the OPACA framework. All the communication regarding the FL process will be managed by this internal server.

As a demonstration, a sample FL network was created to classify images of the FEMNIST[7] dataset, which was specifically designed for an FL environment, including handwritten digits and letters for classification. This dataset is an extension of the commonly used MNIST dataset and additionally contains uppercase and lowercase letters as well as information about the author of each digit/letter. Splitting the dataset based on the author's information simulates the limited access of domain-specific data for each participant, introducing a domain-bias while training the participants' models.

Implementing the FL network as an agent container for the OPACA API inherits useful functionalities for an FL environment. Adding and removing agents from the FL model is done in a dynamic manner by (dis-)connecting additional containers to the runtime platform. Agents can communicate by making requests to the OPACA API, allowing them to exchange information such as collected metrics during the model training. Finally, the language independent implementation allowed us to easily integrate common ML libraries, in this scenario *PyTorch*, into the agent container written in Python.

3) DATA ACQUISITION

In order to evaluate the effectiveness of our MAS, we have integrated it into our project *BeIntelli*,[8] which focuses on autonomous driving enhanced by digital road infrastructure. We strategically deployed road side units (RSU) with various types of sensors and edge computers to enable over-the-horizon capabilities along the BeIntelli test track. Generating large datasets that cover the entirety of our testbed is one of the goals of BeIntelli. This data is mainly camera data collected from various sources. Our overall aim is to enable other researchers and developers working on infrastructure-enhanced autonomous driving without needing direct access to our physical test bed. In order to achieve this goal, we have designed and implemented a network of microservice-based agents that are deployed both on the edge computers and in our cloud infrastructure. A visualization of the architecture is provided in Fig. 5.

At the edge level, we have two types of agents. The first is dedicated solely to communicating with nearby cameras, and capturing real-time streams from these. It receives the

[6]Flower Framework: https://flower.dev/

[7]FEMNIST Dataset: https://github.com/TalwalkarLab/leaf

[8]BeIntelli Project: https://be-intelli.com/

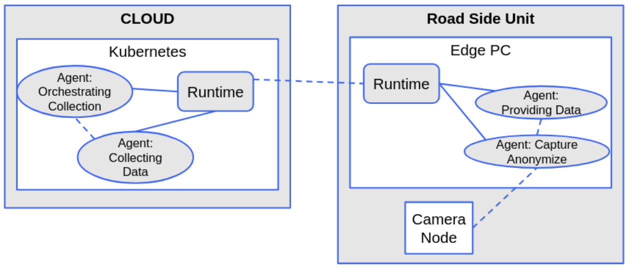


FIGURE 5. Visualization of the data acquisition architecture based on OPACA. An Edge PC is shown here as an example, which is responsible for acquiring and processing the image data. The cloud infrastructure, which is responsible for orchestrating and collecting the processed data from the various Edge PCs, is also shown.

video streams and captures frames based on the specific requirements and specifications. The same agent also ensures that the captured frames are anonymized before storing. The second edge agent is responsible for providing a data stream download, so that the cloud can retrieve the collected edge data.

A separate set of agents is deployed for each corresponding edge computer in the cloud environment. These cloud-based agents orchestrate the entire process of collecting data by working with their corresponding edge agents. They effectively manage the flow of data from the edge to the cloud infrastructure by requesting the necessary frames from the edge agents. To simplify the collection process, a final agent orchestrates these cloud agents.

By seamlessly integrating these microservice-based agents, our MAS successfully acquires, processes, and manages the vast amounts of camera data from our infrastructure-enhanced autonomous driving testbed in Berlin. The resulting dataset can serve as a valuable resource for the research community. It supports the advancement of autonomous driving and infrastructure-integrated technologies.

B. REQUIREMENTS REVISITED

In the following, we will evaluate our framework against the requirements, formulated in Section IV:

1) OPEN/STANDARDIZED INTERFACES

The utilization of REST/HTTP ensures a common and standardized approach to data exchange. Furthermore, according to a Statista evaluation (2022), the adoption rate of Kubernetes in organizations is 61% [29]. By leveraging microservices and using them in Kubernetes-based architectures, our components can be easily integrated into popular software architectures.

2) LANGUAGE-AGNOSTIC

By using REST-based web-services for the different interactions and container technologies for encapsulation, agent components can be written in virtually any programming language for which there are suitable REST and JSON libraries, interacting seamlessly with each other. While there is a full-features reference implementation of the runtime platform, that, too, could be rewritten in another language, as could be external tools for e.g. runtime monitoring and interaction.

3) MODULARITY & REUSABILITY

The basis of our framework is a microservice architecture in which the agents are deployed containerized. Container technologies have intrinsic properties in the sense of modularity, through which they are encapsulated with all their dependencies in images. These are particularly easy to transfer across environments and are easy to deploy overall, especially in microservice-optimized environments, such as Kubernetes clusters. That makes them especially reusable, due to ease of encapsulation and transfer. To make reusability more practical, we are currently developing a registry service that can be used to search agent components running on other platforms or available for installation, and to connect to or install them onto the current platform.

4) SELF-DESCRIPTIVE

The proposed API includes routes providing information on different levels: the deployed containers, their agents, actions, and parameters of the actions. Further, each container can specify what functionalities it requires and what it provides to other containers. The exact format of those requirements' descriptions is still to be finalized.

5) OPEN, MULTI TENANCY

The API allows both, to add and remove agent containers at runtime, and connect or disconnect other runtime platforms. Individual Agent Containers can also spawn (or remove) agents or actions and notify their parent runtime platform. The platform's description is automatically updated with the new information.

6) DISTRIBUTION

The platform allows for distribution on two levels: First, the agent-containers registered at a single runtime-platform can run on different physical hosts, e.g. different remote Docker hosts, or Kubernetes nodes. Second, multiple runtime-platforms running in different environments can be connected, connecting all their respective agent-containers as if they were running on the same platform.

C. LIMITATIONS

While our API itself is highly flexible and with the "extra-ports" feature can be extended with arbitrary additional features, the core API may be limited in what it is able to express. We are currently investigating ways to improve the expressiveness towards, e.g., using JSON Schema to describe used data types, or related projects in the context of inter-agent protocols. Further limitations, and ways to address them, may be revealed as our API is used in more practical projects.

VII. CONCLUSION

MAS have formed a large field of research for many years, but despite their notoriety and popularity, agent-oriented programming principles have found their way into software engineering only to a limited extent. This was especially due to the fact that those systems often do not conform to modern software paradigms. Through the OPACA API, we have taken a first step towards applying MAS in production-level environments, making it possible to easily develop and connect agents. By exploiting basic concepts, our approach is easily tangible and extensible. Due to principles of microservice-based software architectures, our system inherits properties such as reusability, service-orientation, open-interfaces, language-agnosticism, and others.

The use of the MAS in the context of data acquisition for the BeIntelli project illustrates how coordination and collaboration between microservice-based agents can successfully gather, process, and manage large amounts of camera data from test beds such as ours. The provisioning of a comprehensive data set is made possible by the MAS, allowing for seamless communication between the edge and cloud agents, and showing that our approach is a viable solution for such data tasks. Furthermore, the usage in the context of our smart desk booking system has shown the benefits of our framework in the context of hardware-related applications. Moreover, we applied our framework within the context of federated learning, demonstrating its seamless combination capabilities with state-of-the-art technology. Our framework provides multiple examples of agent-containers that can be used as a blueprint for own containers, allowing an entry point with a low barrier.

A. FUTURE WORK

The approach presented in this paper can be extended in several directions. These include establishing a sophisticated user/role management system and advanced security concepts, as well as integrating more general Machine Learning (ML) and distributed problem solving approaches. Furthermore, it is necessary to further explore the integration within larger application projects, to ensure the performance, security, scalability, and applicability of the system in real-world scenarios.

First, while the system already includes basic authentication, implementing more advanced security concepts is crucial to protect sensitive data and ensure secure inter-agent communication. The MAS should establish secure and authenticated interactions by integrating public/private key concepts and cryptographic techniques [30], thus providing a solid foundation for secure and reliable operation by strengthening the system's resistance to unauthorized access, data leakage, and malicious attacks. In addition, for production-level deployments of MAS, it is essential to establish a user management system, including role management capabilities that allow, e.g., administrators to define and assign permissions to users and agents (cf. e.g. [31]). With such a system in place, the system will ensure proper access control, security, and efficient administration, thereby facilitating its use in real-world scenarios.

Second, there is great potential for enhancing the capabilities of the MAS through the integration of more general ML techniques, in addition to the already included federated learning that allows agents within the system to jointly train machine learning models using data that is distributed across a number of containers, and to learn from a variety of data sets while preserving data privacy [32]. In addition, to enable collaborative problem solving among agents, the use of distributed problem solving approaches is essential [33]. The MAS will facilitate agents to collaborate, share information, coordinate actions, and collectively search for optimal solutions by applying distributed algorithms. This collaborative approach enables the MAS to efficiently and effectively tackle optimization problems by using the collective intelligence of the agents.

Further, the framework is being showcased and discussed in different workshops in the Go-KI project, collecting feedback from different development teams associated with the initiative. Finally, the integration of the MAS into larger application projects will provide an opportunity to evaluate its performance and identify potential limitations of the approach in production environments. We started testing and refining our framework in different contexts, among others for data acquisition in the BeIntelli project (see example in section VI), which will also be the topic of an upcoming paper, providing a comprehensive analysis of its performance on big data sets. As more will follow, we expect to find limitations for production-level usage of our framework, directing us to define and implement new concepts to allow a wider application.

REFERENCES

[1] X. Feng, K. L. Butler-Purry, and T. Zourntos, "A multi-agent system framework for real-time electric load management in MVAC all-electric ship power systems," *IEEE Trans. Power Syst.*, vol. 30, no. 3, pp. 1327–1336, May 2015.

[2] F. Bellifemine, A. Poggi, and G. Rimassa, "Developing multi-agent systems with jade," in *Proc. Int. Workshop Agent Theories, Archit., Lang.* Boston, MA, USA. Cham, Switzerland: Springer, Jul. 2000, pp. 89–103.

[3] M. Waseem, P. Liang, and M. Shahin, "A systematic mapping study on microservices architecture in DevOps," *J. Syst. Softw.*, vol. 170, Dec. 2020, Art. no. 110798.

[4] A. H. Bond and L. Gasser, *Readings in Distributed Artificial Intelligence*. San Mateo, CA, USA: Morgan Kaufmann, 2014.

[5] P. G. Balaji and D. Srinivasan, "An introduction to multi-agent systems," in *Innovations in Multi-Agent Systems and Applications1*. Cham, Switzerland: Springer, 2010, pp. 1–27.

[6] A. Dorri, S. S. Kanhere, and R. Jurdak, "Multi-agent systems: A survey," *IEEE Access*, vol. 6, pp. 28573–28593, 2018.

[7] V. Julian and V. Botti, "Multi-agent systems," *Appl. Sci.*, vol. 9, no. 7, p. 1402, 2019. [Online]. Available: https://www.mdpi.com/2076-3417/9/7/1402

[8] R. H. Bordini, J. F. Hübner, and M. Wooldridge, *Programming Multi-Agent Systems in AgentSpeak Using Jason*. Hoboken, NJ, USA: Wiley, 2007.

[9] D. Taibi, V. Lenarduzzi, C. Pahl, and A. Janes, "Microservices in agile software development: A workshop-based study into issues, advantages, and disadvantages," in *Proc. XP Sci. Workshops*, May 2017, pp. 1–5.

[10] M. Viggiato, R. Terra, H. Rocha, M. T. Valente, and E. Figueiredo, "Microservices in practice: A survey study," 2018, *arXiv:1808.04836*.

[11] Statista. (2021). *Microservices Adoption Level by Organizations*. Accessed: May 25, 2023. [Online]. Available: https://www.statista.com/statistics/1233937/microservices-adoption-level-organization/

[12] E. Al-Masri, "Enhancing the microservices architecture for the Internet of Things," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2018, pp. 5119–5125.

[13] S. Baškarada, V. Nguyen, and A. Koronios, "Architecting microservices: Practical opportunities and challenges," *J. Comput. Inf. Syst.*, vol. 60, no. 5, pp. 428–436, Sep. 2020.

[14] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*, M. Mazzara and B. Meyer, Eds. Cham, Switzerland: Springer, 2017, pp. 195–216. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-67425-4_12

[15] R. W. Collier, E. O'Neill, D. Lillis, and G. O'Hare, "MAMS: Multi-agent MicroServices," in *Proc. Companion World Wide Web Conf.*, May 2019, pp. 655–662.

[16] X. Limón, A. Guerra-Hernández, A. J. Sánchez-García, and J. C. Peréz Arriaga, "SagaMAS: A software framework for distributed transactions in the microservice architecture," in *Proc. 6th Int. Conf. Softw. Eng. Res. Innov. (CONISOFT)*, Oct. 2018, pp. 50–58.

[17] V. Charpenay, A. Zimmermann, M. Lefrançois, and O. Boissier, "Hypermedea: A framework for web (of things) agents," in *Proc. Companion Web Conf.*, Apr. 2022, pp. 176–179.

[18] O. Boissier, R. H. Bordini, J. F. Hübner, A. Ricci, and A. Santi, "Multi-agent oriented programming with JaCaMo," *Sci. Comput. Program.*, vol. 78, no. 6, pp. 747–761, Jun. 2013.

[19] M. Dastani, "Programming multi-agent systems," *Knowl. Eng. Rev.*, vol. 30, no. 4, pp. 394–418, 2015.

[20] M. Jagutis, S. Russell, and R. Collier, "Simulating traffic with agents, microservices and REST," in *Intelligent Distributed Computing XV*. Cham, Switzerland: Springer, 2023, pp. 89–99.

[21] R. Morabito and J. Jimenez, "IETF protocol suite for the Internet of Things: Overview and recent advancements," *IEEE Commun. Standards Mag.*, vol. 4, no. 2, pp. 41–49, Jun. 2020.

[22] A. Y. Bahar, S. M. Shorman, M. A. Khder, A. M. Quadir, and S. A. Almosawi, "Survey on features and comparisons of programming languages (Python, JAVA, AND C#)," in *Proc. ASU Int. Conf. Emerg. Technol. Sustainability Intell. Syst. (ICETSIS)*, Jun. 2022, pp. 154–163.

[23] F. Beck and S. Diehl, "On the congruence of modularity and code coupling," in *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng.*, Sep. 2011, pp. 354–364.

[24] M. Vanegas Ferro, A. Lee, C. Pritchard, C. M. Barton, and M. A. Janssen, "Containerization for creating reusable model code," *Socio-Environ. Syst. Model.*, vol. 3, p. 18074, Mar. 2022.

[25] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web services description language (WSDL) 1.1," W3C, Mar. 2001. [Online]. Available: https://www.w3.org/TR/2001/NOTE-wsdl-20010315

[26] N. Goyal, A. K. Pandey, S. K. Gupta, and R. Pandey, "Suppleness of multi-tenancy in cloud computing: Advantages, privacy issues and risk factors," in *Proc. Int. Conf. Sustain. Comput. Sci., Technol. Manage. (SUSCOM)*, 2019, pp. 2185–2190. [Online]. Available: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3358249

[27] M. Van Steen and A. S. Tanenbaum, *Distributed Systems*. Leiden, The Netherlands: Maarten van Steen Leiden, 2017.

[28] C. Rakow, "A framework for simulating mobility services in large scale agent-based transportation systems," in *Proc. 2019 Summer Simulation Conf.* San Diego, CA, USA: Society for Computer Simulation International, 2019.

[29] Statista. (2022). *Kubernetes Adoption Level by Organizations*. Accessed: Jul. 13, 2022. [Online]. Available: https://www.statista.com/statistics/1233945/kubernetes-adoption-level-organization/

[30] H. Subramanian and P. Raj, *Hands-On RESTful API Design Patterns Best Practices: Design, Develop, Deploy Highly Adaptable, Scalable, Secure RESTful Web APIs*. Birmingham, U.K.: Packt Publishing Ltd, 2019.

[31] J. Ferber and O. Gutknecht, "A meta-model for the analysis and design of organizations in multi-agent systems," in *Proc. Int. Conf. Multi Agent Syst.*, Sep. 1998, pp. 128–135.

[32] S. Banabilah, M. Aloqaily, E. Alsayed, N. Malik, and Y. Jararweh, "Federated learning review: Fundamentals, enabling technologies, and future applications," *Inf. Process. Manage.*, vol. 59, no. 6, Nov. 2022, Art. no. 103061.

[33] E. H. Durfee and J. S. Rosenschein, "Distributed problem solving and multi-agent systems: Comparisons and examples," in *Proc. 13th Int. Distrib. Artif. Intell. Workshop*, 1994, pp. 94–104.

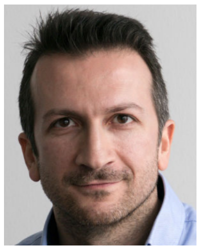
BENJAMIN ACAR holds a master's degree in technomathematics (mathematics with a minor in physics) from the Karlsruhe Institute of Technology (KIT), Germany. He is currently pursuing the Ph.D. degree in computer science (Technische Universität Berlin), focusing on multi-agent systems. Previously, he worked as a Risk Analyst in one of the largest German banks. His research interests include encompass distributed systems, machine learning, and software engineering.

TOBIAS KÜSTER received the Diploma degree in computer science from Technische Universität Berlin (TU Berlin), in 2007, and the Ph.D. degree, in 2017. He is currently the Head of the Go-KI Project with GT-ARC gGmbH. He is also directing the Competence Center "Agent Core Technologies" (ACT), DAI-Labor, TU Berlin. He has worked on different research projects on multi-agent systems, process modeling, and the optimization of industrial processes and schedules.

OSKAR F. KUPKE is currently pursuing the bachelor's degree in computer science with Technische Universität Berlin (TU Berlin). Since February 2023, he has been a Student Assistant with the German–Turkish Advanced Research Centre for ICT (GT-ARC), Berlin. His research interests include the applications of agent technologies in everyday scenarios.

ROBERT K. STREHLOW received the B.Sc. degree in computer science from Technische Universität Berlin (TU Berlin), in 2022, where he is currently pursuing the master's degree in computer science. He has the bachelor's thesis centered around the application of neural networks to NP-complete problems. Since 2023, he has been a Student Assistant with the German-Turkish Advanced Research Centre for ICT (GT-ARC), Berlin. His research interests include AI and machine learning.

MARC GUERREIRO AUGUSTO is currently pursuing the Ph.D. degree in computer science with the DAI-Labor/Technische Universität Berlin (TU Berlin), focusing on platform economy and distributed AI for CCAM solutions. He is a Computer Scientist with the Port Logistics Industry with an emphasis on process optimization and automation. He also leads the BeIntelli Research Project, exploring AI in mobility based on platform economy, a lighthouse project on autonomous driving in Berlin, Germany. He acts as a Partner and the Program Manager with the Center for Tangible AI and Digitalization (ZEKI). His research interests include automation, artificial intelligence, and digital platforms, with an application focus on autonomous mobility and transportation.



FIKRET SIVRIKAYA received the bachelor's degree in computer engineering from Boğaziçi University, Istanbul, Turkey, in 2000, and the Ph.D. degree in computer science from the Rensselaer Polytechnic Institute, Troy, NY, USA, in 2007. Since 2008, he has been a Senior Researcher and a Lecturer with Technische Universität Berlin (TU Berlin), Berlin, Germany. Since 2016, he has also been the Research Director with the German–Turkish Advanced Research Center for ICT (GT-ARC), an affiliated institute of TU Berlin. His research interests include future mobile networks, the Internet of Things, and artificial intelligence, with an application focus on intelligent transport systems and smart cities.

SAHIN ALBAYRAK received the Ph.D. and Habilitation degrees in computer science from Technische Universität Berlin (TU Berlin), Germany, in 1992 and 2002, respectively. He is currently a Full Professor in business applications and telecommunication (AOT) with the Chair of Agent Technology, TU Berlin. He is the Founder and the Head of the Distributed Artificial Intelligence Laboratory (DAI-Labor), TU Berlin. He is also the Founding Director of the Connected Living Association, the German-Turkish Advanced Research Centre for ICT (GT-ARC), and the Center for Tangible AI and Digitalization (ZEKI), Berlin, Germany. His research interests include distributed systems, machine learning, cybersecurity, multi-agent systems, and autonomous systems, with their particular applications in autonomous driving, smart cities, smart energy systems, telecommunications, and preventive health.