IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received 26 June 2024, accepted 17 July 2024, date of publication 19 July 2024, date of current version 1 August 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3431429

RESEARCH ARTICLE

An Agent-Based Data Acquisition Pipeline for Image Data

BENJAMIN ACAR[1], MARTIN BERGER[1], MARC GUERREIRO AUGUSTO[1], TOBIAS KÜSTER[2], 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@tu-berlin.de)

This work was supported in part by the BeIntelli Project of German Federal Ministry for Digital and Transport (BMDV) under Grant 01MM20004, in part by the GoKI Project of German Federal Ministry of Labour and Social Affairs (BMAS) under Grant DKI.00.00032.21, and in part by German Research Foundation and the Open Access Publication Fund of TU Berlin.

ABSTRACT The efficient processing of information plays a critical role in data-driven domains. Data acquisition pipelines, which act as the interface between data collection and subsequent processing, are central to this. Traditional software engineering methods form the foundation for the development of robust data acquisition pipelines, but agent-based systems hold great potential for realizing such pipelines, and this study presents an innovative architecture that uses agent-based components. This modular approach makes it possible to use specialized agents for individual tasks and to increase the effectiveness of the pipelines. The applicability and effectiveness of this architecture are demonstrated using a system for autonomous driving that collects and processes data from edge computers along roads. Our results are on GitHub (https://github.com/BenjaminAcar/Agent-based-Data-Acquisition-Pipeline).

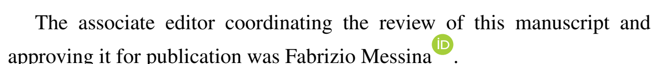
INDEX TERMS Data pipelines, agents, data processing, streaming, autonomous driving.

I. INTRODUCTION

In data-driven domains, it is invaluable not only to collect large amounts of information but also to process it efficiently. Data acquisition pipelines play a crucial role here, as they act as an interface between raw data acquisition and further processing [1], [2], [3]. These pipelines are complex systems consisting of several coordinated components in order to make the processed data available for analysis and model training [4]. In addition, these systems must be flexible enough to adapt to changing data structures and formats as well as changing hardware landscape [5].

Traditionally, developers have relied on proven software engineering methodologies to create data acquisition pipelines. In addition to conventional techniques, however, approaches such as agent-based systems also provide opportunities [6], [7], [8]. These systems use autonomous software units, so-called agents, which perform tasks independently and interact and cooperate with each other [9]. Such systems are remarkably adaptable and greatly benefit in dynamic environments where flexibility and scalability are crucial.

This study presents an architecture that builds on agent-based systems to realize the construction of a data acquisition pipeline. In this concept, agents and their functionalities are independent units that subsequently interact in a coordinated manner. This interaction of the agents leads to a synergetic interplay that results in a highly functional software system. We demonstrate the feasibility and effectiveness of this approach using the example of infrastructure-enhanced autonomous driving. Our presented system collects data from traffic cameras, following a postprocessing step on edge computers co-located with roadside units near the cameras. This data is collected, anonymized, and then aggregated in the cloud to gain insights that are essential for autonomous driving. Our study shows that agent-based systems have considerable potential for the development and deployment of data acquisition pipelines. By using agents, complex data streams are



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

© 2024 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

captured, processed, and integrated more effectively, leading to a significant increase in data quality and availability. The results are applied to our project BeIntelli,[1] which investigates how the enrichment of infrastructure data can improve the capabilities of autonomous vehicles, such as cars or buses. The project uses a large infrastructure in the heart of Berlin, Germany, equipped with sensors and cameras that provide over-the-horizon perception to the vehicles.

II. BACKGROUND

In the following, we will introduce the technical concepts behind agent systems as well as data pipelines. These serve as the basis for our own work.

As information technologies become increasingly distributed, the need for software methodologies that ensure end-to-end functionality, minimize human involvement in design and deployment, and provide flexible software behavior is growing. *Agent-Oriented Software Engineering* (AOSE) offers a promising approach to address these challenges due to its modularity and the ease with which agents are combined. By applying the principles of AOSE, developers can create more flexible and scalable systems that meet the complex needs of today's distributed environments [10]. A common definition describes agents as autonomous entities that can act intelligently and perceive their environment [11]. Multi-agent systems are groups of autonomous agents working independently. Each agent focuses on accomplishing its own tasks while interacting with other agents and the environment to access missing information or coordinate activities. These interactions enable agents to achieve their goals more effectively by leveraging shared resources and knowledge [12].

With the proliferation and growth of data-driven applications, the question of how to manage these data flows arose as well. *Data pipelines* are complex chains of activities that manipulate data, where the output of one component becomes the input of another, enabling an automated flow of data from source to destination. A data pipeline begins with a data source that generates data and ends with a destination that receives the processed data. The final destination of a data pipeline does not need to be a data store. Instead, it can be any application, such as a visualization tool [13]. The need for data pipelines is because, in practice, raw data is rarely ready for consumption and usually needs to be transformed through a series of operations. For example, if the data contains too many variables, feature selection methods are applied [14]. In the area of data collection, topics such as security and analytics also play a particular role [15]. However, in the following, we will focus, in particular, on the architecture of such pipelines.

III. RELATED WORK

The potential of agent-based approaches in the context of data acquisition and processing has been demonstrated in many studies. In the following, we provide a brief insight into these, particularly with regard to the architecture that is realized in each case.

Hu et al. [6] analyze various techniques for data collection in manufacturing systems and present an agent-based framework for such tasks, including two use cases in the steel and chemical industries. The framework includes a data acquisition platform, embodying an environment where agents operate both physically and virtually. This platform contains machines and protocols for communication as well as other functionalities. Another component is the sampling workstation, a data acquisition subsystem designed to generate data, consisting of agents that generate data and technical functionalities that are responsible for communication and other crucial components. A further key element is the sensor array, which consists of concrete sensors that collect the data. The platform itself has three layers: a management layer to orchestrate tasks for the system, a data sampling layer to orchestrate specific data sampling tasks, and a basic support layer to support the system components.

Xu et al. [7] introduces an agent-based architecture for data collection and analysis for distributed simulations. The proposed system consists of multiple layers. The lowest layer is responsible for the network communication. At the higher layers, the simulation applications take place. A data agent is in charge of conveying data between the simulation layers and also analyzing the data.

Bodrozic et al. [16] use a multi-agent system to gather data in real-time for fast forest fire reactions. The system consists of several agents. A camera agent is responsible for collecting images and to set up the moving cameras to the desired positions by adjusting the camera's pan and tilt angles. Another agent is responsible for collecting meteorological data. The database agent stores meteorological data and keeps a record of the collected images and alarms. A user agent embodies the interface to the system.

Reichherzer et al. [17] focus on smart home systems that support independent living for the elderly by addressing the challenge of collecting and analyzing data to detect unusual behavior. The system is also agent-driven, where each device in the smart home system has a software agent that acts as an interface for data collection and control of the device itself. The architecture has three types of agents. Sensor agents are used to collect and store raw data. Middleware agents process the collected sensor data. The application agents provide control over the devices, such as controlling the lights.

Yang et al. [18] address the problem of digital libraries in terms of the heterogeneous information stored in them. The authors propose an agent-based framework for managing data access using an XLS-based data model. Stationary agents are used for specific roles, and mobile agents are used to analyze the data repositories. The mobile agent is responsible for assisting the user by performing tasks such as helping the user formulate queries, launching mobile agents, and so on. On the other hand, the static agent handles complex tasks

[1] BeIntelli project: https://be-intelli.com

TABLE 1. API endpoints for the data collection pipeline.

Method	Route	Description	Input	Output
POST	/invoke/CaptureAndProcessData	Starts the recording process in the specified setup. Action is called up directly at the responsible Capture and Process Agent.	rtspUrl, processingUrl, streamSeconds, frameRate, startTime	zip file
POST	/invoke/GetFilesForCameraID	Provides information about the current files for a specific RTSP camera. Can be trigger via the File Manager Agent	rtspUrl	list
POST	/invoke/TriggerAcquisition	Triggers the capture of the dataset. Initiation is provided by the Orchestrator Agent.	action, agentId, rtspUrl, processingUrl, streamSeconds, frameRate, startTime	-
POST	/invoke/DownloadResults	Moves the data to the location of the orchestrator (cloud node), is provided by the Data Stream Agent.	containerId, agentId, fileName	zip file

within the system and communicates with the less complex mobile agents.

In summary, the work presented in this section clearly demonstrates the versatility and effectiveness of agent-based approaches in developing data collection and acquisition pipelines. These approaches provide flexible architectures that can efficiently handle complex data collection and processing tasks. Our architecture innovates on existing methodologies by incorporating the strengths and lessons learned from our own and others' previous research. It significantly transforms data collection, processing, and aggregation through the use of cutting-edge, containerized agent technologies [19], marking a new approach in the field of image data acquisition. We use the modern framework OPACA [20] and evaluate its potential use in the application.

IV. ARCHITECTURE

In the following pipeline, a setup of edge PCs, Real-Time Streaming Protocol (RTSP) cameras is used, and a cloud enables the acquisition, processing, and storage of video data. The edge PCs serve as a central component that can communicate with the RTSP cameras and receive the video streams supplied by them. The edge PCs are also responsible for processing the data. In addition to this local processing, a cloud serves as central data storage and computing power.

Our software architecture includes several types of agents, each with specific responsibilities, as illustrated in Figure 1. The OPACA framework was selected for developing the agent-based data acquisition pipeline due to its inherent capabilities for streaming data, connecting to non-agent external services, and easy-to-use interaction protocols. For more information about the framework used, please see the documentation provided within the respective GitHub.[2] In the following, we provide an overview of the respective responsibilities. The system has four types of agents:

1) Capture and Process Agents,
2) Data Stream Agents,
3) File Manager Agents,
4) Orchestrator Agents.

The *Capture and Process Agents* capture and process data streams from cameras. When calling the action for starting the acquisition, five parameters must be specified:

1) *rtsp-URL*: the camera's address from which the agent receives the data.
2) *processing-URL*: the address of the microservices that anonymizes the images.
3) t: the time when the agent should start collecting the data.
4) t_n: the number of seconds for which the agent should collect the data.
5) n: The frame rate. For example, if $n = 10$, every second 10 frames are collected.

The agents themselves then record the respective frames in a queue. Initially, the data is not immediately written on the hard drive but is kept in memory. This is decided so that the privacy of the captured entities could be maintained. Instead of saving the data immediately, the agent first sends it to a microservice with the address *processing-URL*, which is only responsible for anonymizing the images and then sending them back in processed form. This happens in the same order in which the images are initially captured. The processed images are now successively stored on the local memory with time stamps (see Algorithm 1). The capturing and processing processes happen in parallel, while the sub-processes themselves are in series. However, in theory, also, the processing procedure itself can be done in parallel to enhance the speed of the process. We decided not to do this because the frame capturing itself is very GPU intensive, especially for the camera resolution used (see Appendix). In our experience, putting too much load on the GPU can cause it to hang and require a complete reboot before it is available again. In addition, when processing images in parallel, we would always limit the number of GPU cores that are used; otherwise, failures such as the one described above are more likely. This also requires proper knowledge of the GPU in use. However, in the case of having a heterogeneous

[2] OPACA Core: https://github.com/GT-ARC/opaca-core

hardware stack like we do, maintaining this information for all GPUs is tedious. As a result, we preferred system reliability before speed. Furthermore, it should be noted that the connection to the camera is continuous and is not created anew with every frame. As RTSP connections always take a certain amount of time to be established, too many frames would otherwise be lost simply because the connection has to be established first.

At the end of each acquisition process, two further processes are triggered. Firstly, the folder with all the frames is compiled into a .zip file. Secondly, another agent of type *Data Stream Agent* is created, which is only responsible for being available for requests to the file so that other agents and users can retrieve this file.

The *File Manager Agents* maintain an overview of all generated data records. They provide information on how much data records are available for a particular camera. To do this, the URL of the camera is passed to the agent. The agent then checks for generated data records that are available on the local machine based on the respective URL. These are then returned as a list. Finally, the *Orchestrator Agent* is responsible for initiating those processes and gathering all the potential data records from different machines to a single location in the cloud. For a list of the routes/actions provided by each agent, see Table 1

Algorithm 1 Frame Capture and Processing Algorithm

1: $t \leftarrow$ initial start time
2: $endTime = t + t_n$
3: $frameRate = 1/n$
4: Wait until t is reached
5: **while** $t{+}{+} \leq endTime$ **do**
6: **In Parallel:**
7: **Thread 1:** Capture Frame
8: Capture a frame
9: Store the frame in a queue
10: Wait for $frameRate \times 1$ second
11: **Thread 2:** Process Frame
12: Take a frame from the queue
13: Process the frame
14: Store the processed frame
15: **end while**

To address implementation hurdles when realizing our agent-oriented data acquisition pipeline for image data, ensuring seamless interaction among the various agents is crucial. Firstly, the Capture and Process Agent must be optimized for diverse environmental conditions and hardware variations, using robust preprocessing techniques to maintain data consistency. This can be realized, for example, by always transforming the data to the same size and keeping the resolution as low as possible, depending on the application that needs to be implemented. The Data Stream Agent should utilize efficient data transmission protocols to handle high throughput and low latency. The OPACA API offers generic concepts for streaming data. Therefore, the recommendation here is to use the OPACA framework. For the File Manager Agent, we need an easily navigable directory structure that supports quick data retrieval and storage. The Orchestrator Agent should have advanced scheduling and error-handling capabilities to ensure smooth coordination between different pipeline stages.

A. ARCHITECTURE ADAPTABILITY

The architecture can be transferred to various setups. There are a large number of potential data sources that can be used, and RTSP video streams are just one example; for instance, an adapter for HTTP or Secure Reliable Transport (SRT) input streams can also be used by making a few changes to the Capture and Process Agent. It is also worth mentioning that our dataset is currently to be transferred by design to an Orchestrator Agent, which is embedded in a cloud and is intended to merge the data centrally. This is also not the best choice for every use case. If, for example, a more complex network of storage nodes exists and the aim is to keep the dataset aggregated but still partitioned on different storage nodes, it would be possible to introduce another agent. This agent would then have the task of orchestrating different Orchestrator Agents that are located on different storage nodes in order to realize two levels of abstraction. Optionally, simply communicating directly with the different Orchestrator Agents on the different storage nodes can be applied, but then the interaction with the system would be more cumbersome because then the user has a higher complexity since it now has to use different entities to communicate with the system.

A second use case is employed to validate the pipeline's adaptability. In this instance, the objective is to gather data on object detection. Instead of consuming camera streams, we now consume a Kafka topic, into which edge computers push the results of their object detection. The edge computers will use this data to determine the trajectories of objects on the infrastructure to predict the behavior of these objects (e.g., pedestrians or cars) [21] and then communicate the information about the driving behavior of other road users to our autonomous test car. This should enable the car to optimize its behavior, react to cyclists driving the wrong way, and so forth. Again, the Capture and Process Agent can easily acquire this data and transfer it to the central point of the Orchestrator Agent. Instead of generating a final zip file that contains the image data, this time, we generate a JSON file that documents the objects and their movement, including timestamps, the identifier of the object, and its current geolocation. The pipeline is not benchmarked for this use case because it is important to keep the work within the scope of the paper, which focuses on image data.

V. EVALUATION

To evaluate the efficiency and reliability of our data acquisition pipeline, we implement four quantitative tests: measuring the generation time of datasets, determining the number of lost frames, measuring the maximum length of the

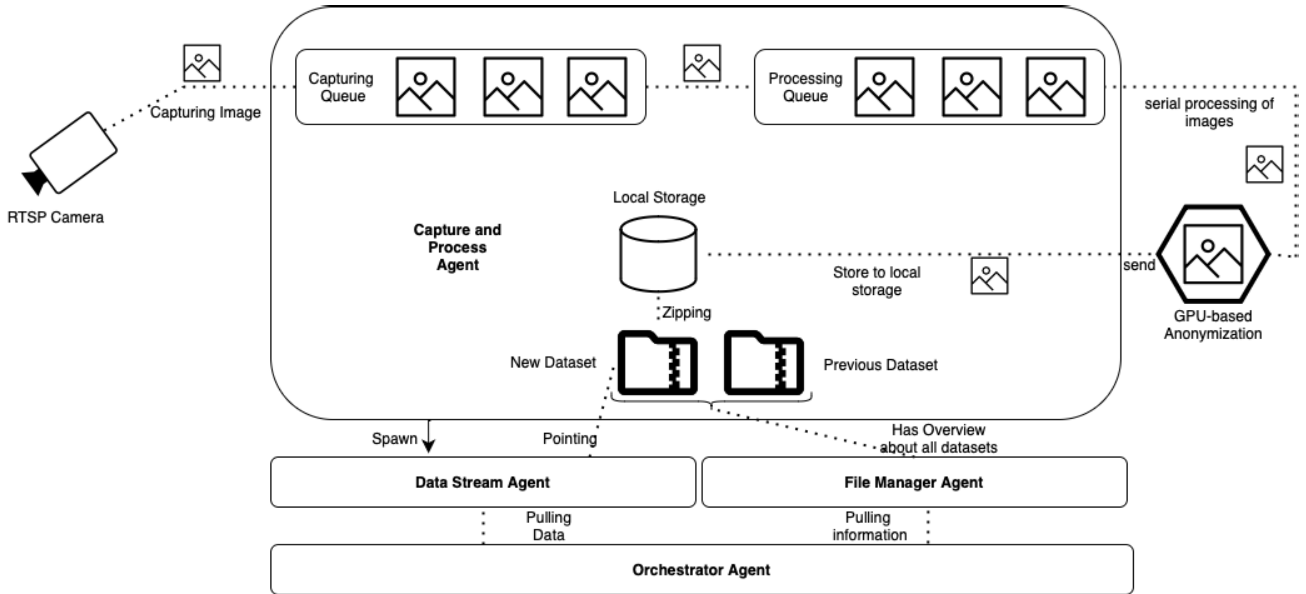


FIGURE 1. Architecture of our system. The Orchestrator Agent initiates the pipeline by calling the action of the Capture and Process Agent, passing initial parameters such as the time when the agent should start the process. The agent then starts to capture and process frames in parallel. When done, it zips the images to a single file and spawns a Data Stream Agent, which is solely responsible for providing stream access to the file. A File Manager Agent has an overview of the local storage and provides users and other agents with information about existing datasets.

queue of images waiting to be processed and the processing time for each image.

First, we focus on the duration of data generation. Here, we measure the time required to create datasets for different time periods. These time periods vary; for example, we could generate data for an hour, a half hour, and so forth. It is also important to consider different frame rates. By testing different frame rates, such as 30 fpm (frames per minute), 60 fpm, or 120 fpm, we can better understand how the performance of our pipeline is affected by data density. Higher frame rates could lead to longer generation times and losses of frames, which is critical. Information about our duration and frame setups is in Table 2 as well as the results. The present results show that even taking into account high frame rates and extended acquisition periods, the time required for the final generation of the dataset is acceptable, with around four hours for a high-density dataset of 240 frames per minute and one hour in total of capturing time. This result is achieved despite the very high resolution.

The second test focuses on the number of frames lost during the acquisition process. This is a crucial factor for data quality. Lost frames can indicate various problems, such as network delays, storage bottlenecks, or processing errors. By systematically capturing and analyzing these losses at different frame rates and over different time periods, we gain valuable insights into the robustness and reliability of our pipeline. The present results reveal that the developed data acquisition pipeline, even when confronted with high frame rates and extended time intervals, has an impressively low error rate in terms of missing frames. This finding is particularly noteworthy as it highlights the robustness and reliability of the system under different and

TABLE 2. Overall setup of our experiments and the corresponding execution time results. ***Duration*** describes the time a dataset is recorded. ***Time taken*** is the result in terms of how long it took to complete the experiment for the given duration and frame rate.

Duration (minutes)	Frames per Minute	Frames in Total	Time taken (minutes)
1	60	60	1.5
1	120	120	2.0
1	240	240	3.0
10	60	600	10.5
10	120	1200	21.5
10	240	2400	41.0
30	60	1800	32.0
30	120	3600	63.0
30	240	7200	124.0
60	60	3600	65.5
60	120	7200	133.5
60	240	14400	255.0

potentially challenging conditions. In particular, it shows that the pipeline maintains consistent precision regardless of the speed or duration of data collection. Only in one experiment, there is a loss of 18 frames. This happened for the case of one minute with 120 frames per minute. Instead of the expected 120 frames, as described in Table 2 row 3 under the column *Frames in Total*, we only collected 102 frames. For all other experiments, the *Frames in Total*, which describes the expected number of frames given the duration of the experiment, and the *Frames per Minute*, the resulting number of frames met the expectations. When capturing frames from an RTSP camera, the network may become congested due to high traffic. We suspect that this led to network overload, resulting in packet loss and missing frames.

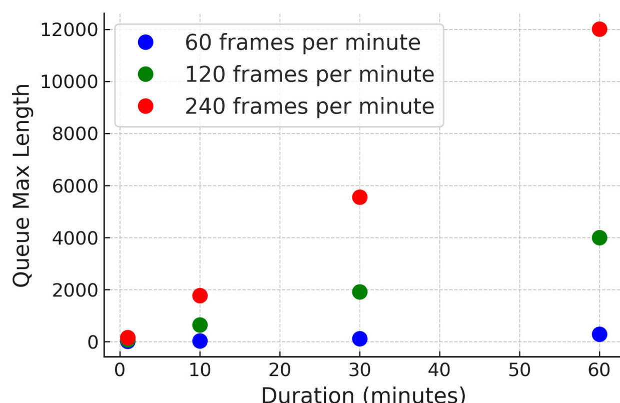


FIGURE 2. **Measuring the maximum queue length of images waiting to be processed.**

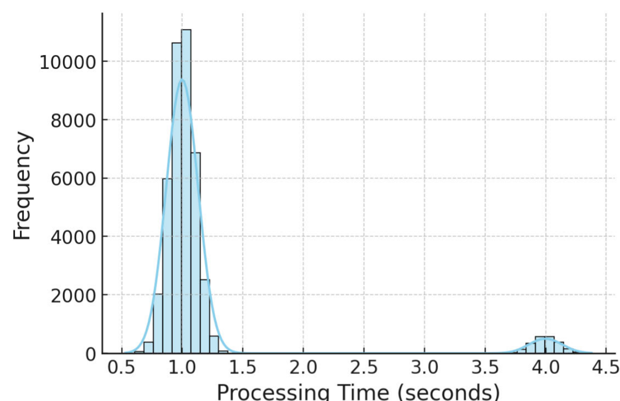


FIGURE 3. **Capturing the distribution of the processing time for each image.**

Next, we look at the maximum length of the queue of images waiting to be processed. This value, particularly the behavior about increasing the duration of the experiment, gives us information about the extent to which the system works consistently. If the queues become very large in some cases, the system will not be reliable, and a large amount of RAM would have to be provided to cope with the fluctuations in the queue. As the images are only saved once they are anonymized, only the RAM is affected up to this point. Here, the queue increases linearly depending on the frames per minute, indicating that the system works consistently, see Figure 2. For the experiment with 60 frames per minute, there is a negligible increase, which is realized without problems using smaller RAM. For the experiment with 240 frames per minute and a length of 60 minutes, there is a peak value of approx. 12,000 images in the queue, which is due, in particular, to the serial processing of the images. Such a large queue can overload the system for smaller systems with little RAM. Here, it is important to adapt the pipeline to the respective hardware possibilities to keep the system stable. The observed linear increase in the queue size with higher image processing rates can significantly impact the edge computing system. In scenarios where large volumes of data are processed–such as during peak traffic times—the inability of the system to handle these spikes efficiently, in addition to the data acquisition itself that takes quite many machine resources, might cause a bottleneck. However, when the queue size increases linearly with the number of images processed per minute, it provides a transparent and predictable resource usage pattern. This predictability is crucial for effectively managing and allocating system resources on the edge computer. Therefore, the issue explained can be mitigated by applying a proper strategy (such as reducing the allocation of machine learning tasks for a period). Overall, the results meet expectations and show consistent behavior.

In our last test, we look at the average time in which an image is processed. This value is important to determine the extent to which the overall performance of the system is affected by the anonymization step. If it is too slow, the entire architecture is affected. For this purpose, we created a histogram across all experiments; see Figure 3. The results show that the vast majority of images take around 1 second to be processed and returned to the Capture and Process Agent. Some images reach peak values of approx. 0.6, while a larger proportion takes approx. 1.2 to 1.3 seconds. This is roughly normal behavior. Surprisingly, a second high-frequency range exists, in the range of 3.5 to 4.5 seconds processing time. It seems that the system has some phases in which the processing step does not run optimally, and then the processing time is increased. However, since the results are such that the outliers themselves form a second, independent normal distribution, we conclude that although the system is slower than usual, it is still stable, and no significant downtimes have occurred. In principle, this can be critical if the pipeline is also used for real-time data acquisition pipelines, i.e., if the anonymized data is transferred continuously. In such a case, if one of the components experiences excessive delays, it inevitably follows that certain use cases can no longer be realized. One example of this is timely online learning. Overall, the results show that the processing step still has room for improvement but is performing efficiently enough to meet the system architecture.

The system offers potential for optimization. As already mentioned, parallel processing can significantly accelerate the anonymization task. In addition, the possibility of simply connecting agents developed in the OPACA framework with other non-agent microservices is used (cf. [20]), which is why an agent did not take over the anonymization process. In principle, this could also be modeled as such. Also, the transfer of the large dataset files to the cloud can be done with more proper protocols, such as FTP if needed. Finally, we have decided to transfer the data to the cloud as a ZIP file only on request. In principle, OPACA also allows the data to be transferred continuously as soon as it has been processed without waiting for the ZIP to be created. However, as we operate many edges and do not have a real-time requirement for the creation of data records, we have decided against this.

A. COMPARISON

In the following, we will compare our architecture against the related work, presented in section III.

Our current modeling approach does not make a clear separation between agents in charge of handling physical devices and those in charge of data processing tasks such as anonymization. In our framework, a main agent is tasked with both collecting raw data and directing the anonymization process. This contrasts to approaches such as that of Hu et al. [6], which use a component called the Sensory Array specifically to connect to physical sensors. Their model's clear separation of responsibilities improves its usability within more complex architectures. Without a clear separation, agents can adapt to multiple roles, allowing them to perform different tasks, including sensor interactions, data processing, and communication. This flexibility can simplify development because a single agent might be used in different contexts. However, it can also lead to complex code as agents grow to accommodate different functionalities, making maintenance and debugging more difficult. This is particularly critical when you consider the processing of lidar data, for example, which can become particularly complex. Mixing these with other functionalities would then quickly become very critical. Without dedicated agents for sensor interactions, the same agents performing other tasks may face resource conflicts. For example, an agent may struggle to balance sensor-related responsibilities with data processing or communication tasks, potentially causing performance issues. The processing of lidar data is also a good example here, as this data is particularly complex and requires a lot of computing power. This can lead to inconsistent handling of sensor data, especially under heavy load, which affects overall system reliability. Having dedicated agents for sensor interactions ensures that these agents are highly specialized and optimized for this specific task. This specialization can lead to more efficient and reliable sensor data handling because the agents have a focused purpose. It also makes the system easier to maintain since any problems with sensor interactions can be traced back to these specific agents. This clear separation requires robust mechanisms for coordination between the sensor-interacting agents and other agents responsible for data processing or communication. It requires well-defined communication protocols and interfaces, adding complexity to the system design. Developers must ensure seamless data flow and synchronization between agents, which can be time-consuming and require careful planning.

Compared to Xu et al. [7] there is also a big difference in the way the data agent collects data. In Xu et al.'s approach, the data agent should have some intelligence to decide whether the collected data is sufficient or not. In contrast, in our system, we have a centralization in the way that the acquisition itself is orchestrated, rather than up to the individual agents. While the authors' approach is more flexible than ours, the purpose is also different. We are interested in creating specific datasets, so we want to be able to predefine our setup and have it orchestrated by an agent. However, this also has a disadvantage. If, for example, an object suddenly appears and data acquisition is paused due to the frequency of the sensor readout rate, this object is not included in the data record. In the related work, the data agent is also responsible for independently finding data in, for example, databases. Therefore, the data agent needs more flexibility for its discovery tasks. This also has its pitfalls. The operation of intelligent agents necessitates the implementation of sophisticated algorithms and the availability of substantial computational resources, both of which are essential for formulating well-informed decisions. To illustrate, in heavy traffic conditions, an intelligent agent would be required to process data from several sensors, analyze traffic patterns, and determine, in real time, which data is most important.

Reichherzer et al. [17] used a very simple architecture. This can, therefore, also be transferred well, but it is poorly developed in terms of how information is exchanged between agents, namely via databases. This approach can be very cumbersome, depending on the database chosen. If many agents are involved who all want to share information with each other but which may not be of interest for persistent storage, this generates a high load on the databases. On the other hand, it simplifies data synchronization across the system, as many agents can access the same up-to-date information from the database. Peer-to-peer approaches between the agents can perform considerably better if information is exchanged directly between them. Of course, this requires more effort in terms of architecture, as the interfaces have to be defined. Also, the bandwidth might be more loaded if one agent shares the same information with multiple agents; for instance, objection detection data detected by one vehicle is sent to the surrounding vehicles that might be affected by the same obstacle.

In terms of design, our approach is closest to Bodrozic et al. [16]. The distinction here is particularly important concerning those agents that interact directly with the sensors to collect data and those that deal with data storage. Nevertheless, a clear modeling of the agent system is missing here, particularly a systematic structuring of functionalities in actions and the embedding in the environment where agents share information about themselves and their functionalities. Developing a similar system would require a complete re-modeling in advance, which is why the transfer can only be realized to a limited extent. In comparison, our agents are embedded in an environment, share information about their functionalities, but also about how these functionalities are used, share their activity, the networking of the individual functionalities is easily realized and by using the OPACA API all the interfaces are clearly defined. This makes our approach easier to transfer to other applications in comparison to Bodrozic and the rest of the related work. Furthermore, using containerized agents makes it easy to embed our architecture into popular software architectures, such as Kubernetes.[3]

[3]https://kubernetes.io

B. LIMITATIONS

Despite its advantages, our architecture also has some gaps that need to be filled in the future. Resource contention among agents for CPU, memory, and network bandwidth can break down the system if the container configurations are not well chosen. Adding additional concepts to avoid such behavior is crucial for large-scale applications based on our architecture. To mitigate the risk of system failures, a robust monitoring and alerting system capable of tracking resource usage and performance metrics in real-time must be implemented. Furthermore, the Orchestrator Agent, as the central coordinating entity, represents a single point of failure. Therefore, it is posing a risk to the system reliability and availability unless complex redundancy and failover mechanisms are implemented. This limitation is mitigated because the other agents can be used without the Orchestrator Agent. In practice, however, users want to use only one interface to the system, not several. Understanding the principles of all entities can be a hurdle. To mitigate this issue, orchestration tools such as Kubernetes can be employed to implement redundancy for the Orchestrator Agent. Instead of relying on a single agent, multiple agents are deployed to overtake the role of the Orchestrator Agent in the event of a failure. This approach is not limited to the Orchestrator Agent but can also be employed for all the other agents, such as the Capture and Process Agent. In addition, robust error handling is essential. At the moment, more concrete information about data losses and their reasons are not given, especially because the monitoring of network issue, can be cumbersome. Overcoming these limitations will enable a broader application of our architecture in the future.

VI. DISCUSSION

In the following, we will shed light on the concrete contribution we made in the paper, as well as the conclusions we can draw from the results and future directions of work.

A. CONTRIBUTION

This paper presents an innovative system for a data pipeline based on an agent-based approach. This system consists of different specialized agents, each performing dedicated tasks. One agent is solely responsible for capturing and processing data from a camera stream, while another agent provides this processed data as a stream. A third agent handles file management, and a fourth agent orchestrates the entire data collection. Each agent exposes its functionality through a REST interface, enabling seamless interaction and integration. This modular approach not only streamlines the data collection process but also provides flexibility and scalability, making the system easily extensible and adaptable to different data acquisition tasks. Furthermore, due to its properties, the pipeline is also well-suited to large-scale infrastructure, such as IoT testbeds and other use cases.

VII. CONCLUSION

The results of our work clearly demonstrate the effectiveness and efficiency of the agent-based architecture. Particularly noteworthy is its suitability for the acquisition and processing of large datasets. Furthermore, results have shown that the data acquisition pipeline is robust and yields accurate datasets even for high frame rates and long durations. The error rate in terms of missing frames is noticeably low. The system is also consistent and scales linearly. The processing of individual frames follows a narrow normal distribution, with a deviation in the area of higher processing times. Our work, therefore, confirms the assumptions of the related work. At the same time, it provides further approaches as to how potential modeling of possible systems could look. Our approach and the agent structures are generic and can be used for different use cases, depending on the domain, by slightly adjusting those.

A. FUTURE WORK

The incorporation of sensor fusion into the existing data acquisition pipeline can significantly enhance its capabilities by integrating distributed sensor data to make the data more comprehensive. This involves extending the Capture and Process Agent to handle inputs from multiple sensors, such as cameras and LiDAR sensors, and fusing their data to produce more detailed and versatile datasets. The File Manager Agent manages the increased volume and complexity of data, ensuring efficient data overview. The Orchestrator Agent would coordinate the acquisition process across different sensor types, ensuring synchronized data acquisition. Finally, the Data Stream Agent will deliver the fused sensor data in real-time or in a historical manner.

APPENDIX

A. SETUP

For the experiment, we used a GeForce RTX 3080 Lite Hash Rate graphic card, deployed on a Nuvo-8108GC-XL edge PC. The used camera has a resolution of 1920×1080 (Full HD).

REFERENCES

[1] M. Kejriwal and Y. Gu, "A pipeline for rapid post-crisis Twitter data acquisition, filtering and visualization," *Technologies*, vol. 7, no. 2, p. 33, Apr. 2019.

[2] Í. M. Collado, D. M. Miracle, R. V. Robles, and A. B. Salat, "A data acquisition pipeline for home energy management systems," in *Proc. III Congreso Iberoamericano de Ciudades Inteligentess*, 2020, pp. 195–209.

[3] G. Grosso, N. Lai, M. Migliorini, J. Pazzini, A. Triossi, M. Zanetti, and A. Zucchetta, "Triggerless data acquisition pipeline for machine learning based statistical anomaly detection," in *Proc. EPJ Web Conf.*, vol. 295, 2024, pp. 20–33.

[4] A. Mumuni and F. Mumuni, "Automated data processing and feature engineering for deep learning and big data applications: A survey," *J. Inf. Intell.*, vol. 1, no. 2, pp. 1–17, Jan. 2024.

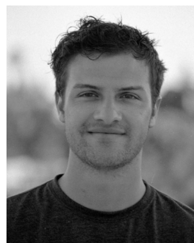
[5] A. Kumar Sarker, A. Alsaadi, N. Perera, M. Staylor, G. von Laszewski, M. Turilli, O. Ozan Kilic, M. Titov, A. Merzky, S. Jha, and G. Fox, "Design and implementation of an analysis pipeline for heterogeneous data," 2024, *arXiv:2403.15721*.

[6] Y. M. Hu, R. S. Du, and S. Z. Yang, "Intelligent data acquisition technology based on agents," *Int. J. Adv. Manuf. Technol.*, vol. 21, nos. 10–11, pp. 866–873, Jul. 2003.

[7] Y. Xu, S. Sen, and F. W. Ciarallo, "An agent-based data collection architecture for distributed simulations," *Int. J. Model. Simul.*, vol. 24, no. 2, pp. 55–64, Jan. 2004.
[8] G. Lyu and R. W. Brennan, "Evaluating a self-manageable architecture for industrial automation systems," *Robot. Comput.-Integr. Manuf.*, vol. 85, Feb. 2024, Art. no. 102627.
[9] P. G. Balaji and D. Srinivasan, "An introduction to multi-agent systems," *Innov. Multi-Agent Syst. Appl.-1*, vol. 1, pp. 1–27, Jul. 2010.
[10] F. Bergenti, M.-P. Gleizes, and F. Zambonelli, *Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook*. Cham, Switzerland: Springer, 2004.
[11] V. Julian and V. Botti, "Multi-agent systems," in *Foundations of Artificial Intelligence*. Amsterdam, The Netherlands: Elsevier, 2019.
[12] F. Zambonelli, N. R. Jennings, A. Omicini, and M. J. Wooldridge, "Agent-oriented software engineering for internet applications," in *Coordination Internet Agents: Models, Technology Application*. Heidelberg, Germany: Springer, 2001, pp. 326–346.
[13] A. Raj, J. Bosch, H. H. Olsson, and T. J. Wang, "Modelling data pipelines," in *Proc. 46th Euromicro Conf. Softw. Eng. Adv. Appl. (SEAA)*, Aug. 2020, pp. 13–20.
[14] A. Quemy, "Data pipeline selection and optimization," in *Proc. DOLAP*, 2019.
[15] A. Kumari and S. Tanwar, "A secure data analytics scheme for multimedia communication in a decentralized smart grid," *Multimedia Tools Appl.*, vol. 81, no. 24, pp. 34797–34822, Oct. 2022.
[16] L. Bodrozic, D. Stipanicev, and M. Stula, "Agent based data collecting in a forest fire monitoring system," in *Proc. Int. Conf. Softw. Telecommun. Comput. Netw.*, 2006, pp. 326–330.
[17] T. Reichherzer, S. Satterfield, J. Belitsos, J. Chudzynski, and L. Watson, "An agent-based architecture for sensor data collection and reasoning in smart home environments for independent living," in *Proc. 29th Can. Conf. Artif. Intell.*, 2016, pp. 15–20.
[18] Y. Yang, O. F. Rana, D. W. Walker, C. Georgousopoulos, G. Aloisio, and R. Williams, "Agent based data management in digital libraries," *Parallel Comput.*, vol. 28, no. 5, pp. 773–792, May 2002.
[19] R. W. Collier, E. O'Neill, D. Lillis, and G. O'Hare, "MAMS: Multi-agent MicroServices," in *Companion Proc. World Wide Web Conf.*, May 2019, pp. 655–662.
[20] B. Acar, T. Kauster, O. F. Kupke, R. K. Strehlow, M. G. Augusto, F. Sivrikaya, and S. Albayrak, "OPACA: Toward an open, language- and platform-independent API for containerized agents," *IEEE Access*, vol. 12, pp. 10012–10022, 2024.
[21] R. Yan, Y. Gu, Z. Zhang, and S. Jiao, "Vehicle trajectory prediction method for task offloading in vehicular edge computing," *Sensors*, vol. 23, no. 18, p. 7954, Sep. 2023.

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

MARTIN BERGER received the Diploma degree in computer science, in 2013. He is currently pursuing the Ph.D. degree with Technische Universität Berlin (TU Berlin), with a focus on multi-agent systems, computer vision, and reasoning. He is a Researcher with TU Berlin. Previous research projects included topics from (e-)mobility, robotics, augmented reality, communication, and infrastructure for autonomous mobility.



MARC GUERREIRO AUGUSTO is currently pursuing the Ph.D. degree in computer science with the DAI-Labor/Technische Universität Berlin, focusing on platform economy and distributed AI for CCAM solutions. He is a Computer Scientist who has been working in the port logistics industry with an emphasis on process optimization and automation. He leads the BeIntelli Research Project, exploring AI in mobility based on platform economy, a lighthouse project on autonomous driving in Berlin, Germany. He also 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.

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

FIKRET SIVRIKAYA received the bachelor's degree in computer engineering from Bogazici University, İstanbul, Türkiye, in 2000, and the Ph.D. degree in computer science from Rensselaer Polytechnic Institute, NY, USA, in 2007. Since 2008, he has been a Senior Researcher and a Lecturer with Technische Universität Berlin, Germany. He has been the Research Director with German-Turkish Advanced Research Center for ICT (GT-ARC), an affiliated Institute of TU Berlin, Berlin, Germany, since 2016. 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. degree in computer science and the Habilitation degree from Technische Universität Berlin, Germany, in 1992 and 2002, respectively. He is currently a Full Professor in business applications and telecommunication (AOT) with the Chair of Agent Technologies, TU Berlin. He is a 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, 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.