Enabling Parallel Processing at the Edge for Real-Time Video Analysis Applications

Rafael C. Chaves[1,2], **Lucas M. Souza**[1], **Otacílio A. Ramos Neto**[1], **Ruan D. Gomes**[1,2,3]

[1] Smart4i – Polo de Inovação do IFPB
[2]Programa de Pós-Graduação em Tecnologia da Informação (PPGTI)
[3]Programa de Pós-Graduação em Engenharia Elétrica (PPGEE)
Instituto Federal da Paraíba (IFPB) – João Pessoa – PB – Brasil

{rafael.chaves,lucas.mendes}@polodeinovacao.ifpb.edu.br

{otacilio.ramos,ruan.gomes}@ifpb.edu.br

Abstract. *In the context of video analysis systems, distributed processing is a widely recognized strategy for handling large volumes of data and reducing the execution time of complex tasks. In this paper, we present a solution, called Prisma, designed to enable parallel processing by implementing video stream-splitting strategies, which generate multiple derived streams from the original video stream. These derived streams can be distributed to multiple clients, allowing each processing instance to handle only a portion of the original stream while abstracting the complexities of network management and video fragmentation.*

1. Introduction

In recent years, the importance of video analysis systems has increased significantly. One factor contributing to this growth is the adoption of computer vision as a first step in quality control in many industrial plants [Wagner et al. 2023]. Computer vision is also applied in the monitoring of human activities in factory environments. In many cases, such systems have been shown to be more efficient than human-based monitoring, contributing to improved safety and precision in the management of industrial processes [Czimmermann et al. 2020, Meribout et al. 2022]. Object identification and tracking, as well as image classification, are just a few of the applications that highlight the relevance of these systems in the context of Industry 4.0.

However, the increasing use of such systems also brings several challenges that must be addressed, such as the need to efficiently process large volumes of data and reduce the execution time of computationally intensive tasks. In this context, the use of distributed processing is a common approach. Strategies such as MapReduce [Dean and Ghemawat 2008] and Spark [Zaharia et al. 2010] have been widely adopted, with applications such as video encoding in the cloud [Pereira et al. 2010] and identification of relevant information through computer vision [Perafan-Villota et al. 2021], among others.

Few studies have addressed the creation of solutions that facilitate distributed processing for real-time video analysis in small-scale systems. In such cases, distributed processing can be useful for parallelizing complex tasks, not only with the goal of reducing execution time but also of maintaining the same processing rate while using hardware with lower computational resources. In [George and Ravindran 2019], the authors highlight the need for a middleware capable of handling video distribution with real-time response and minimal processing delay. Their solution offers an API for specifying latency and accuracy requirements, and their experiments show that latency increases approximately linearly with frame size, reaching up to 164% higher when two video streams are transmitted simultaneously. The Vision Edge IoT (VEI) middleware, proposed in [Luu et al. 2022], targets edge-based computer vision applications by replicating video streams from cameras to multiple applications using a publish/subscribe API. In [Perafan-Villota et al. 2021], the authors present a parallel video analysis architecture built on a low-cost cluster using Apache Spark, Hadoop, and the MapReduce paradigm, achieving significant reductions in processing time. Finally, [Singh et al. 2023] proposes a real-time video processing framework for multiple streams using distributed computing. The system receives video streams via a Kafka server and applies computer vision algorithms in parallel using Apache Spark.

In a previous work, a middleware named *Espelho* was proposed to enable the implementation of video analysis systems with support for edge computing [Neto et al. 2024]. One of the main features of *Espelho* that distinguishes it from the other works found in the literature is its ability to replicate a video stream to multiple processes on different machines, allowing distributed applications to access video streams through Linux virtual video devices (`/dev/video*`) as if reading from a device physically connected to the machine. This facilitates the development of video analysis applications. Such functionality is particularly useful when the goal is to analyze a video using multiple computer vision algorithms or to run a single algorithm with different parameters in each instance.

Although it provides transparency for applications accessing video streams, the *Espelho* middleware does not implement any functionality to split the video stream for parallel processing, as it always forwards the complete stream to all instances responsible for processing the video. However, in some scenarios, it may be beneficial to perform a temporal or spatial split of the video to accelerate parallel processing, which is an important aspect in edge computing environments and for applications that require low latency. Therefore, a natural evolution is to enable parallel processing by dividing video streams into multiple substreams and sharing them in such a way that each application instance processes only a portion of the video, while still preserving the abstraction provided by *Espelho*. In this scenario, applications consume the substreams as if they were capturing frames from a local camera, without having to deal with the complexity related to substream partitioning and their transmission over the network.

This paper presents a proposal for a new module, called *Prisma*, to implement stream-splitting strategies to enable parallel video processing at the edge. Experiments were carried out to evaluate the overhead introduced by a preliminary version of *Prisma*. In this initial version, the module only replicates the video stream instead of splitting it. Taking into account resolutions up to 2560×1440, the addition of *Prisma* resulted in an 11.9% increase in average latency when transmitting a single video stream. In contrast, in the scenario with three replicated video streams in parallel, the use of *Prisma* resulted in a 4.7% reduction in average latency. This result demonstrates that, as the number of streams to be generated increases, the use of *Prisma* provides better performance compared to the use of *Espelho* alone to perform the same function.

2. Solution Architecture

This section describes the architecture of the solution proposed in this work. First, the *Espelho* middleware is described, which is responsible for distributing video streams and providing transparent access to applications. Then, the module proposed in this paper, called *Prisma*, is presented. It is responsible for generating video substreams to facilitate distributed parallel processing while maintaining compatibility with *Espelho*.

2.1. Middleware *Espelho*

Espelho is a video distribution middleware designed to simplify the implementation of video analysis systems in edge computing scenarios, initially described in [Neto et al. 2024]. Figure 1 illustrates a use case of *Espelho*, in which a video captured in an industrial plant is transmitted to a distribution server. This server replicates the video to two clients: one that processes the images using computer vision algorithms, and another located in a monitoring center, which displays the video in real time. The client applications access the video streams through virtual video drivers (`/dev/video*`) created by the middleware.

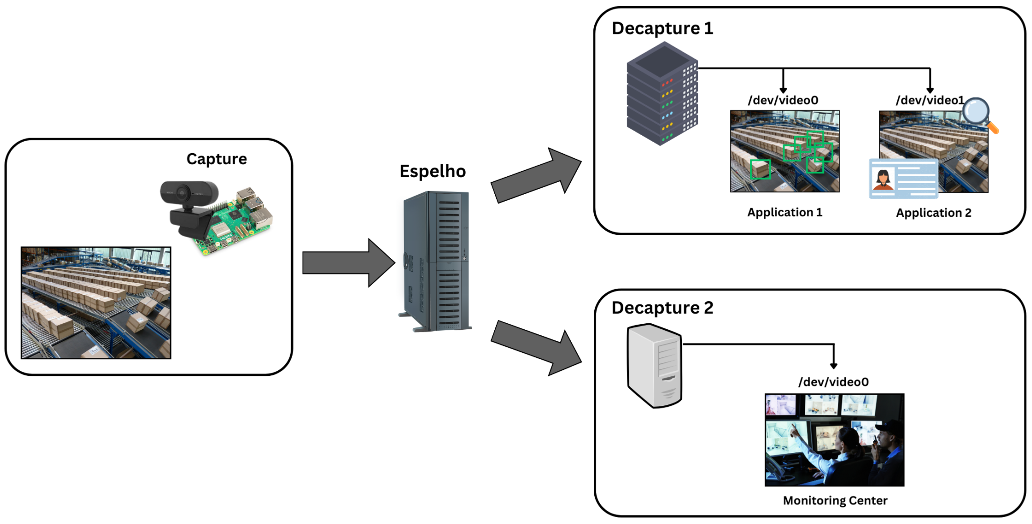

Figure 1. *Espelho* use case in an Industry 4.0 scenario.

The *Espelho* middleware is composed of three modules: *Capture*, *Espelho*, and *Decapture*. The *Capture* module is responsible for reading a video stream from an external camera and transmitting it over the network to the distribution server. It was developed to be compatible with embedded Linux, allowing it to run on edge devices with limited computational resources. The *Espelho* core is the distribution server, which follows the publisher/subscriber model. Each video stream is associated with a topic, and subscribers receive a copy of the stream in real-time. The *Decapture* module receives the video stream, decodes it, and makes it available through virtual video devices. As a result, the video transmitted over the network can be accessed as if it were coming from a local camera. This approach hides the complexity of the transmission process, simplifying the work of developers implementing the processing algorithms.

2.2. The *Prisma* module

As described in Section 2.1, *Espelho* replicates the complete video stream to all application instances subscribed to receive the video. However, in certain distributed computing scenarios, it is beneficial to split the video to enable parallel processing of different segments of the same stream. To support this distributed processing paradigm within *Espelho*, this paper proposes the creation of a new module called *Prisma*. This module is responsible for dividing the video into substreams, which are then sent to the Espelho server to be distributed to the clients.

Figure 2 illustrates how *Prisma* interacts with the other middleware modules in a distributed processing scenario. *Prisma* was designed to maintain full compatibility with the abstraction provided by *Espelho*, allowing applications to continue reading network-delivered streams as if they were reading from a locally connected camera. By applying *Prisma*, different instances of applications can read different frames or different regions of the frames, without having to deal with the complexity of video splitting and network transmission management.

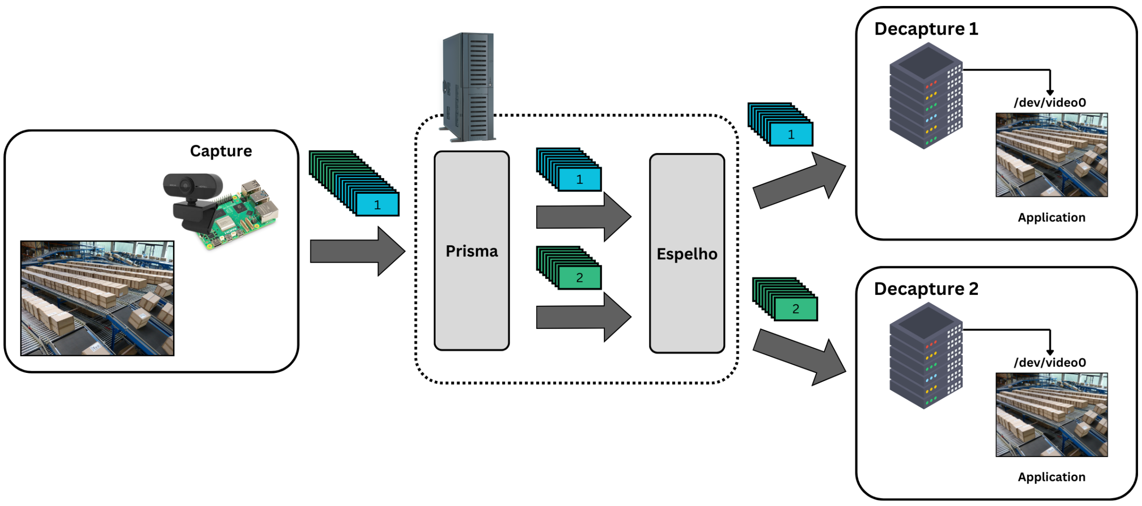

Figure 2. Distributed processing scenario using *Prisma*.

The decision not to incorporate video stream splitting functionality into the distribution server was made to simplify the codebase and facilitate future modifications. Section 3 discusses the impacts of adding this new component, analyzing the results of experiments conducted with a preliminary version of *Prisma*.

Internally, *Prisma* consists of three modules that communicate through queues, allowing each module to be updated without affecting the operation of the others. The *Receiver* module is responsible for establishing connections and communicating with *Capture*. The *Broker* component handles the stream-splitting operations, and can be configured to use different splitting strategies. Finally, the *Sender* component manages communication with *Espelho*. Figure 3 shows the main components of *Prisma*.

Different splitting strategies can be implemented on *Prisma*. For example, in a temporal splitting strategy, the byte stream received by the *Capture* module is analyzed and divided into substreams, each encapsulated separately for transmission to Espelho. In practice, this approach reduces the number of frames that each client must process per second, allowing more time to process the incoming data. Another example is spatial splitting, in which frames are partitioned into quadrants, each sent through a different substream. This enables clients to process received frames more quickly due to their re-

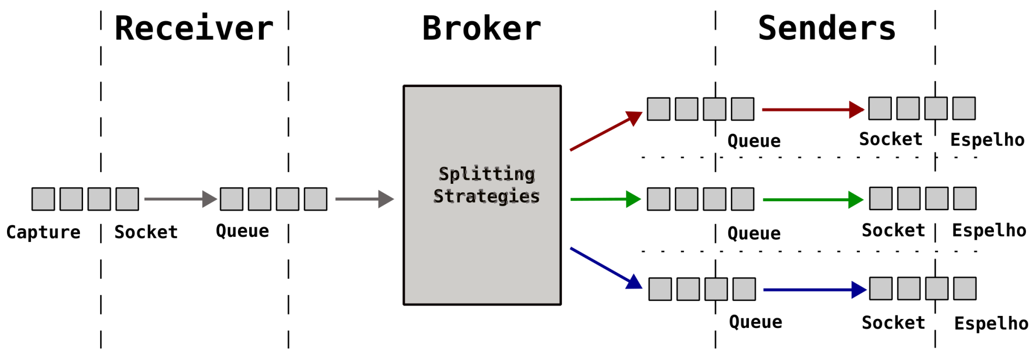

Figure 3. Prisma Architecture

duced size. Depending on the chosen approach, decoding or transcoding of streams in the *Prisma* module may be required. In other cases, *Prisma* may simply split an already encoded stream, sending different frames or groups of encoded frames into separate substreams.

3. Experimental Evaluation

This section describes the experiments proposed to evaluate the influence of using *Prisma* on the performance of *Espelho* middleware. It is important to note that the tests were conducted using a preliminary version of *Prisma*, which only implements video stream replication. This scenario was designed to verify the feasibility of adding an external component to the distribution server without imposing a high overhead. Furthermore, the impact of running multiple instances of *Decapture* on the same machine was explored, evaluating how this scenario could benefit parallel video processing on a single device.

To evaluate each of these questions, the average transmission latency and the delivered frame rate (frames per second - fps) were measured. To ensure the accuracy of these metrics, the clocks of the machines used in the experiments were synchronized through a software implementation of the Precision Time Protocol (PTP). Furthermore, to avoid distorting the results, the experiment was conducted by transmitting a two-minute video encoded in H.264 format at multiple resolutions (ranging from 800×480 to 4096×2160) at a constant frame rate of 30 fps.

The experiments were carried out using three distinct machines connected to the same gigabit Ethernet network, ensuring that middleware performance was not affected by network bandwidth limitations. The source device, running the *Capture* module, is equipped with a AMD A10 PRO-7800B processor and 16 GB of DDR3 RAM. The distribution machine, hosting both *Espelho* and *Prisma*, has a Intel Core i5-12400 processor and 8 GB of DDR4 RAM. Lastly, the client machine, which ran the instances of *Decapture*, has a AMD PRO A10-8750B processor and 16 GB of DDR3 RAM.

3.1. Results

Figure 4 shows the average latency values for resolutions up to 2560×1440. On the left are the results for transmissions involving a single instance of *Decapture*, while on the right the results for transmissions with three instances running simultaneously are presented. From the results, it can be observed that the addition of an external component to *Espelho* did not significantly impact performance. In fact, under higher workloads (with

3 clients), *Prisma* integrated to *Espelho* achieved a slightly better performance than using *Espelho* alone. This can be explained by the fact that the replication routine implemented in *Prisma* is lighter and simpler than the one found in *Espelho*.

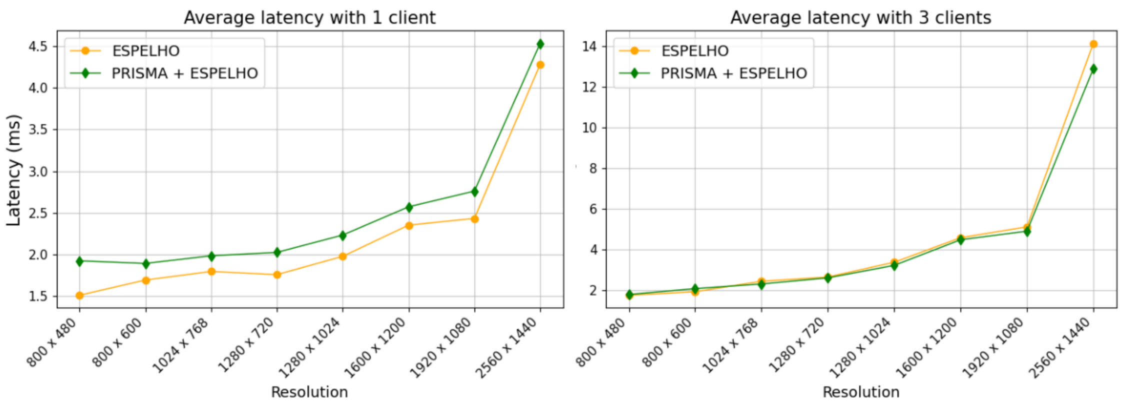

Figure 4. Latency per frame at resolutions up to 2560×1440 (with 1 client on the left, and 3 clients on the right).

Figures 5 and 6 show the latency per frame and the frame rate for resolutions up to 4096×2160 (4K). Up to resolution 2560×1440 the clients were able to process a frame rate equivalent to the one generated at the source (30 fps). However, it can be observed that there is a significant performance degradation at the two higher resolutions. This occurs because of the use of a machine equipped with a processor that offers low performance by current standards. Because the decoding implemented is software-based, the CPU cannot process frames at the incoming rate, resulting in dropped frames and increased latency for the remaining ones. These results encourage the use of parallel processing, mainly in scenarios in which a set of machines is being used with little resources available to execute the applications.

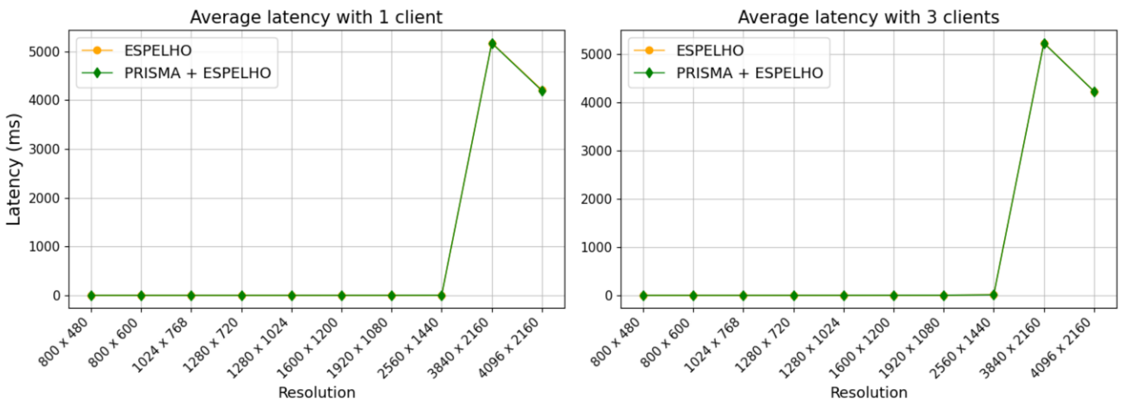

Figure 5. Latency per frame at resolutions up to 4K (with 1 client on the left, and 3 clients on the right).

Another interesting observation is that running multiple *Decapture* instances on the client machine does not cause as severe a performance drop as processing video at higher resolutions (3840×2160 and 4096×2160). This occurs because *Decapture* runs serially, with each instance only slightly affecting others as long as processor cores are avail-

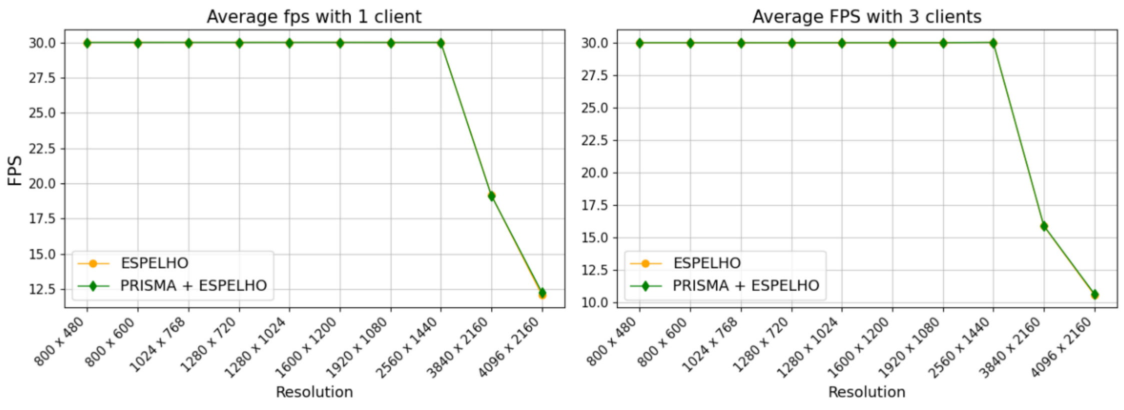

Figure 6. Frames per second at resolutions up to 4K (with 1 client on the left, and 3 clients on the right).

able. Thus, increasing the workload beyond the capacity of a single core has a more significant impact than increasing the number of parallel instances. Splitting video streams into substreams to distribute workload across multiple instances could be an effective solution to handle higher resolution streams on low-performance hardware.

4. Conclusion

This paper describes the *Prisma* module, which emerges as a solution to enable parallel processing on the edge, maintaining a high level of transparency for applications, due to its integration with the middleware *Espelho*. The solution described in this paper allows for straightforward application development by abstracting the complexities of video transmission over networks and parallel processing. Applications can be easily replicated and configured to process different video segments simultaneously, without increasing code complexity.

Experimental results demonstrated that adding an external component to the distribution server of *Espelho* introduces minimal overhead under low load conditions and delivers superior performance as load increases. Another key observation from the experiments is that increasing the workload for a single *Decapture* instance beyond the capacity of a single processor core has a more severe impact than increasing the number of parallel instances on a device. This finding supports the idea that the proposed stream-splitting approach can effectively enable distributed video processing even on low-performance devices.

As future works, we will implement multiple splitting strategies in the Prisma module, and evaluate their performance across different edge-based video processing scenarios, and for applications in the context of Industry 4.0. Scenarios considering the use of 5G private networks will also be evaluated.

5. Acknowledgments

This work is supported by EMBRAPII (BFA 2301.0001), Cisco, Prysmian, and MPT Cable. The authors also thank CNPq (305536/2021-4), CPQD, Inatel, Taggen, Data Machina, and the IFPB Innovation Hub.

References

Czimmermann, T., Ciuti, G., Milazzo, M., Chiurazzi, M., Roccella, S., Oddo, C. M., and Dario, P. (2020). Visual-based defect detection and classification approaches for industrial applications—a survey. *Sensors*, 20(5):1459.

Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

George, A. and Ravindran, A. (2019). Distributed middleware for edge vision systems. In *2019 IEEE 16th International Conference on Smart Cities: Improving Quality of Life Using ICT & IoT and AI (HONET-ICT)*, pages 193–194. IEEE.

Luu, S., Ravindran, A., Pazho, A. D., and Tabkhi, H. (2022). VEI: a multicloud edge gateway for computer vision in IoT. In *Proceedings of the 1st Workshop on Middleware for the Edge (Quebec, Quebec City, Canada) (MIDDLEWEDGE '22)*, pages 6–11. Association for Computing Machinery, New York, NY, USA.

Meribout, M., Baobaid, A., Khaoua, M. O., Tiwari, V. K., and Pena, J. P. (2022). State of art iot and edge embedded systems for real-time machine vision applications. *IEEE Access*, 10:58287–58301.

Neto, O. d. A. R., Chaves, R. C., Nascimento, A. P., and Gomes, R. D. (2024). Middleware para aplicações distribuídas de vídeo com suporte à computação na borda na indústria 4.0. In *Brazilian Symposium on Multimedia and the Web (WebMedia)*, pages 215–222. SBC.

Perafan-Villota, J. C., Mondragon, O. H., and Mayor-Toro, W. M. (2021). Fast and precise: parallel processing of vehicle traffic videos using big data analytics. *IEEE transactions on intelligent transportation systems*, 23(8):12064–12073.

Pereira, R., Azambuja, M., Breitman, K., and Endler, M. (2010). An architecture for distributed high performance video processing in the cloud. In *2010 IEEE 3rd international conference on cloud computing*, pages 482–489. IEEE.

Singh, T., Rajput, V., Satakshi, Prasad, U., and Kumar, M. (2023). Real-time traffic light violations using distributed streaming. *The Journal of Supercomputing*, 79(4):7533–7559.

Wagner, R., Matuschek, M., Knaack, P., Zwick, M., and Geiß, M. (2023). Industrialedgeml - end-to-end edge-based computer vision system for industry 5.0. *Procedia Computer Science*, 217:594–603. 4th International Conference on Industry 4.0 and Smart Manufacturing.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster computing with working sets. In *2nd USENIX workshop on hot topics in cloud computing (HotCloud 10)*.