ASML: Algorithm-Agnostic Architecture for Scalable Machine Learning

DIMITRIOΣ E. DIAMANTIS™ AND DIMITRIOΣ K. IAKOVIDIS™, (Senior Member, IEEE)

Department of Computer Science and Biomedical Informatics, University of Thessaly, 35131 Lamia, Greece

Corresponding author: Dimitrios E. Diamantis (didiamantis@uth.gr)

This work was supported by the European Union and Greek National Funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the call RESEARCH–CREATE–INNOVATE under Project T1EDK-02070.

ABSTRACT Machine Learning (ML) applications are growing in an unprecedented scale. The development of easy-to-use machine-learning application frameworks has enabled the development of advanced artificial intelligence (AI) applications with only a few lines of self-explanatory code. As a result, ML-based AI is becoming approachable by mainstream developers and small businesses. However, the deployment of ML algorithms for remote high throughput ML task execution, involving complex data-processing pipelines can still be challenging, especially with respect to production ML use cases. To cope with this issue, in this paper we propose a novel system architecture that enables Algorithm-agnostic, Scalable ML (ASML) task execution for high throughput applications. It aims to provide an answer to the research question of how to design and implement an abstraction framework, suitable for the deployment of end-to-end ML pipelines in a generic and standard way. The proposed ASML architecture manages horizontal scaling, task scheduling, reporting, monitoring and execution of multi-client ML tasks using modular, extensible components that abstract the execution details of the underlying algorithms. Experiments in the context of obstacle detection and recognition, as well as in the context of abnormality detection in medical image streams, demonstrate its capacity for parallel, mission critical, task execution.

INDEX TERMS Artificial intelligence, parallel processing, distributed computing, machine vision, medical services.

I. INTRODUCTION Deep learning growth has triggered the appearance of frameworks for easy development of ML-enabled applications. Many of these frameworks are supported by tech industry leaders, such as Google, Facebook and Microsoft, which usually provide deep learning Platforms as a Service (PaaS) or Software as a Service (SaaS) on their Cloud Computing infrastructures, specialized in executing deep learning frameworks. For instance, Google, which supports the Tensorflow framework, provides Google Cloud. This is a general purpose cloud computing service, enabled by Tensor Processing Units (TPUs) [1], offering better performance for deep learning applications that use that framework.

While pre-configured virtual machines and containerized ML solutions exist, they still require a technical understanding of the underlying platform; thus, they are not directly applicable to any production environment. For this reason, SaaS providers, such as Amazon and Google offer in their platforms pre-trained deep learning models for specific use cases, typically through a representational state transfer (RESTful) HTTP [2] application programming interface (API). In most cases, it is also possible to deploy pre-trained models such as [3] and [4], as long as they are implemented in a supported framework. The flexibility of such services is limited, as while it is relatively easy to get started, it is difficult to efficiently incorporate ML models based on novel components, such as the fuzzy pooling layer proposed in [5], or complex ML-based data-processing pipelines, such as pipelines that include image preprocessing, integration of multiple heterogeneous ML algorithms with bidirectional data communication. Such pipelines are frequently met in state-of-the-art pattern analysis applications spanning a variety of domains e.g., web content perception [6], obstacle detection and navigation for robotics [7] and assistive technologies [8], real-time analysis of medical image sequences during brain surgery [9] and gastrointestinal (GI) endoscopy [10]. However, their deployment in a SaaS context, using current ML
frameworks, is far from straightforward, especially when high-throughput capacity is required. Today, to deal with this shortcoming, the implementation of such pipelines usually requires from the client to handle the communication and monitor the status of the ML components and implement pre-processing. However, this is not always possible, e.g., in the case of wearable devices and other low-powered embedded systems.

There has been work towards the development of system architectures that aim to encapsulate and abstract the usage of complex business logic for different purposes in various domains. In [11] a system architecture and platform, called Public-oriented Health care Information Service Platform (PHISP) was presented for personalized healthcare services and support remote health care. A system architecture and a framework for discovering content from the web using a RESTful architecture design was presented in [12]. For managing big semantic data in real-time an architecture, called SOLID, was proposed in [13]. This architecture is characterized by its layered design which isolate the real-time and big data specific responsibilities. In the context of time-complemented and event-driven control models, an architecture offering modularity and flexibility of automation software was presented in [14]. That architecture unifies the two models, aiming to preserve the expressiveness of event-driven programming along with the determinism of time-driven logic. In the context of Enterprise Internet of Things (EIoT), a multi-device, multi-task management and orchestration reference architecture was proposed in [15]. The architecture focuses the orchestration on the task-level focusing on the business process modeling of enterprise systems. Similarly for task offloading in IoT applications, an architecture named “EdgeABC” was proposed in [16]. The architecture splits the tasks into multiple subtasks based on the application workflow and then uses blockchain algorithm to ensure the integrity of resource transaction data and the profits of the resource provider. In [17] a scalable system for ML task declaration and learning, called “MLBase”, was proposed. That system aims to make ML accessible to broad audience of users, by simplifying the declaration of ML models in a Pig Latin-like [18] declarative language and automatic ML algorithm selection. In the context of remote Machine Learning as a Service (MLaaS) [19], the “PredictionIO” [20] framework, integrated a variety of ML models into a prediction service, access to which is provided using an API and a graphical user interface. In [21] a framework that aims to provide assistance throughout machine learning task lifecycle, such as training, validation and testing, was proposed with the name “DEEP-Hybrid-DataCloud”. That framework uses a standardized API that enables the functionality of the ML models to be exposed based on known semantics. In [22] a unified component based architecture was proposed primarily focused on utility service deployment in cloud environments. The architecture focused on maximizing the availability of the deployed service with minimum configuration overhead. In [7] a distributed architecture was proposed for motion planning of multi-robotics systems in real-time. Although such architectures can be flexible and sometimes extensible, they are tailored on domain-specific problems, limiting their scope. This paper addresses the problem of remote high-throughput ML task execution involving complex data-processing pipelines. It aims to cope with well-recognized challenges [23], [24] that include the deployment of ML applications in a generic and standard way through a framework that provides the necessary level of abstraction. This framework is independent from the application domain and implementation details, such as the ML algorithms and the different programming languages used for the implementation of different components within these pipelines. To implement this framework, we propose a novel Algorithm-agnostic Scalable architecture for ML applications (ASML) that combines:

- Algorithm-agnostic architecture design, that enable arbitrary ML applications to be modeled
- Modular design and extensible components that allow extensibility both in terms of the supported tasks and the input and output of the architecture
- Highly scalable architecture multi-client and parallel execution task support, enabling SaaS deployment scenarios
- Synchronous and asynchronous task execution.

To the best of our knowledge, no such ML-oriented system architecture has ever been proposed, despite the emerging needs for remote artificial intelligence (AI) services in different application domains. The main contributions of this paper include:

- It provides a performance evaluation indicating its efficiency.
- It provides technical details and application scenarios that can be used as examples for implementation of other ML application pipelines.
- It provides an answer to the open research question of how to design and implement an abstraction framework, suitable for the deployment of end-to-end ML pipelines in a generic and standard way.

To evaluate the performance and the flexibility of the proposed system architecture, we conducted experiments for two SaaS use case scenarios, where pattern recognition is provided as a cloud service. The first use case addresses the complex task of multi-user obstacle avoidance in the context of visually impaired navigation, using a state-of-the-art obstacle avoidance framework [8]. The second use case includes synchronous and asynchronous task execution in the context of abnormality detection in gastrointestinal endoscopy images [10]. It should be noted that ASML is applied for the first time for the SaaS implementation of these use case scenarios.

The rest of the paper consists of three sections. The proposed ASML architecture and its components are described in Section II. The evaluation methodology along with the use case scenarios details and results are included in Section III. Conclusions derived from this study are summarized in
II. ASML ARCHITECTURE

The proposed system architecture is task-oriented. A task is defined as a self-contained series of actions that is required to be completed to achieve a goal. A goal can be thought as the output of a procedure such as, image classification, object detection, object tracking etc. A task can contain multiple actions that can be executed in parallel or sequentially, depending on the goal needs. When actions are executed sequentially, the execution of the next actions is postponed until the previous ones are completed. Each action defined in the series is receiving the output of all previously completed actions, enabling complex use case scenarios to be defined. When an action in the series results into multiple outputs which are required to be processed separately by the next actions in the series, the task can create multiple tasks to parallelize the process. This is important, especially on high throughput use cases, where a significant performance improvement can be achieved by process parallelization. The input and the output of a task, including data transport, is implemented by data source handlers, and the actions are implemented by processors. Both data source handlers and processors are instantiations of reusable, abstractly defined, software components. The data of all input and output data source handlers are tagged by unique identifiers, which enable dynamic routing of the data to processors. The routing of both data source handlers and processors, is facilitated by a specialized, reusable software component, called interceptor.

Overall, the ASML architecture (Fig. 1) consists of four logical components:

- The RESTful API, which acts as the entry point of the overall architecture
- The worker, which handles the task execution
- The task scheduler
- The data storage and system monitoring module

To ensure redundancy, all the components of the ASML architecture can be deployed in a cluster configuration [25], which considers multiple instances per component, as illustrated in Fig. 1. Assuming that a client has the required access rights to access and dispatch tasks to the proposed system architecture, a typical task flow can be summarized as follows. Initially the client obtains an access token using the API of the architecture. Using this token, the client can make calls to the API to create, cancel, or obtain information about a task. Historical data, such as the output of previous tasks can be obtained using the same API. When a new task is submitted by a client to the API, a record is created in the database of the system containing the task information along with meta data such as, the status of the task, the user who created it etc. In parallel, a record is created in a key-value pair data store, which is used to track the progress and other temporal data about the task lifecycle. This data store offers high throughput read/write operations and is used internally by the system as a temporal meta data storage medium instead of a conventional Relational Database Management System (RDBMS). This design decision was made because temporal data, such as the progress of a task, usually requires frequent updates (can be thousand times per second), which can degrade the time-performance of the system and increase the resource requirements, such as CPU and memory use, of a conventional RDBMS store [26]. The task is then registered to a message queue to be delivered to a worker. At this point a response is issued to the client by the API, containing identification information about the newly created task, which can be used by the client as a reference for future requests, such as tracking the task progress etc. When the task is enqueued, a worker which monitors the message queues, consumes the task and initializes the execution. Depending on the input data source handlers, the processors and the output data source handlers, the worker unravels the task, pulls the appropriate modules and starts the execution of the task. Depending on the task configuration, the worker can
communicate directly with the client, receiving and dispatching information or asynchronously inform the client about the progress and the output of the task. When the task execution is completed the API receives a request from the worker informing about the outcome. At this stage, all temporal data are deleted and the meta-data stored in the key-value pair store are permanently written to the RDBMS store.

A more detailed description of the architecture components and their interaction for the implementation of a complex data processing task is provided in the following paragraphs.

A. RESTFUL API

The API component facilitates the communication with the clients and is the entry point of the ASML architecture. To maintain high compatibility and easy integration with most clients, the API is implemented as a RESTful HTTP service [2]. To enable ASML architecture to be used in a SaaS deployment, the authentication and authorization of the clients is handled by following the OAuth 2.0 [27] protocol. As a result, clients using existing OAuth 2.0 service providers can make use of the architecture without the need of providing their private credentials. Depending on the use case scenario new service providers can be added or disabled dynamically. To ensure high service availability, multiple instances of the API can run simultaneously in an HTTP load balanced environment.

The API exposes four endpoints. The first endpoint is responsible for the creation of a task. The request must contain a payload, which describe the task by identifying the processors, input and output data source handlers that will be used. Along with the payload, the request can contain other parameters, such as the desired priority of the task, the remote callback endpoints that the system will request when the status of the task changes, and flags indicating if a task should be re-processed in case of a failure.

When a task is created, a unique identifier is generated and returned to the client along with two endpoints; one that can be used to track asynchronously the history of all the tasks that have been created along with their output. While the protocol for the creation of the task depends on the architecture, the requests can be encoded using JavaScript Object Notation (JSON) or Extensible Markup Language (XML), depending on the content type of the HTTP request. This is done to maximize the client compatibility.

As many client applications are nowadays executed on web browsers, the API implements cross-origin resource sharing (CORS) [28]. Depending on the use case scenario, the API can be equipped with request quote thresholds that can be enforced on per user basis. Such thresholds can be applied on the number of requests that a user can issue within specific time period, resources allocated per user etc. This capability can be used as a pricing schema or to ensure a fair use of the system and prevent denial of service (DOS) [29] attacks.

B. WORKER

A worker is an extensible component that can be thought as a handler, subscribed to one or many queues, and it consumes tasks. It is equipped with one or more input and output data source handlers and it can have multiple processors, which are responsible for the execution of several actions (Fig. 2). As a result, the capabilities of a worker and the information about the queues to which it will subscribe, are derived from the types of actions that it can process. This enables the worker to process pipelines that otherwise would be incompatible to each other, based on their software and hardware dependencies, e.g., a worker could execute a pipeline with two ML processors, one capable of executing models implemented in PyTorch models and the other in Tensorflow. This battles the limitations of “MLBase”-like [17] models, where a single framework must be used. Furthermore, the parallel processor design, enables the implementation of complex use-cases, where more than one models are used in parallel to produce results for the next processor in the pipeline, which is not possible by systems such as [20] and [21].

When a task is consumed, a worker initially loads the input data source handlers along with the processors and their output data source handlers and instantiates them using the parameters found in the payload of the task. An example payload with multiple input data source handlers, sequential and parallel processors is illustrated in Fig. 2. Upon initialization, it executes the input data source handlers found in the task and passes their output to the processors identified in the payload. A processor may or may not have one or more output data source handlers, which are executed when the processor execution step finishes. In the special case where the output
of one processor is needed as an input for the execution of the next one, the processor execution is delayed until all previous processors finish their execution. The output of all processors along with the initial input data source output handler is then piped to the processor as input. In the case where an action can be parallelized, the processor can create new tasks using the API component of the architecture and wait for their output. This enables the worker to use the available resources of the system, when available, and increase its throughput. The scheduling of these tasks is handled by the task scheduling component of the architecture. Considering that processors are re-usable components, not all the outputs of all previous processors are always needed. For this reason, interceptors can be used to select the input of the processors. Finally, when all processors finish their execution, the worker informs the API about the completion of the task.

In all steps of the process, the worker is updating the progress of each processor in a key-value pair database. This is used by the API when a request about the status of the task is received by a client. In some cases, a client might require intermediate information about the execution of some processors; for this reason, the worker can inform the client using remote endpoints after the execution or the failure of a processor.

While a worker can be extended to support any type of processors and data source handlers, the architecture already contains a series of predefined modules that cover most use case scenarios. The default input and output data source handlers include, HTTP, FTP, SCP, S3, Swift protocols along with Web Real-Time Communication (WebRTC) [30] and Real-time Streaming Protocol (RTSP) [31] for real-time input and video streaming. ASML architecture is equipped with general purpose processors that enable image, audio and video processing along with machine learning. For example, in the case of the scenarios described in Section III, the image processor is implemented as a wrapper around the widely used ImageMagick library API. Similarly, the audio and video processor act as wrappers around the FFmpeg library API. The machine learning processor can be used for inference (not for training) on pre-trained ML models coming from a variety of ML libraries, including Pytorch and Tensorflow frameworks. The later supports the majority of the popular deep learning frameworks such as Tensorflow, Pytorch, CNTK and Darknet and provide variable configuration depending on the use case. The trained models can be provided to the processor using an input data source handler. For extensibility purposes the worker exposes a well-defined API and documentation that can be used for the development of new modules.

Nowadays web applications, i.e., applications that run solely on web browsers, are becoming more and more common. While they offer the flexibility of running on web browsers, most devices are equipped with, they are limited by the APIs exposed by the browser. For this, common real-time protocols such as the Real-time Messaging Protocol (RTMP) [32] and the RTSP [31] cannot be used. Recently, web browsers adopted the WebRTC [30] standard for real-time audio and video communication.

The ASML architecture supports WebRTC peer-to-peer communication between the workers and the client by using a Traversal Using Relay NAT (TURN) [33] server, since typically, workers and clients are behind a Network Address Translation (NAT) service. Signaling between workers and the WebRTC clients is handled via WebSockets [34], which is an open standard for real-time messaging. WebSockets can also be used by the workers to communicate messages to the client in real-time. Authentication and authorization to the TURN server and the WebSockets is handled by the API component of the architecture using OAuth 2.0 Bearer Tokens [35].

The worker module is designed so that it allows the implementation of any ML application pipeline, as new processors and new input and output data source handlers can be added. In ASML architecture, an ML pipeline implementation can be summarized into four steps:

1) Deploy the pre-trained ML model in the storage module of the architecture
2) Define which input data source handler(s) are going to provide input to the pipeline
3) Configure the ML processor to use the pre-trained model
4) Define one or many output data source handlers which are going to be used as the output of the processor.

Given an ML model created using a common ML framework, such as Tensorflow or PyTorch, the system can automatically load it and use it. Otherwise, a new ML processor should be implemented to enable support of less popular frameworks. In Section III, two complex use-case scenarios are examined along with the steps followed to implement them as ASML pipelines.

### C. TASK SCHEDULER

When a client creates a task using the API component, the tasks are placed in a queue and recorded in a database. The task scheduler acts as an intermediate between the task queue and the API, handling the priority in which the task will be executed. Depending on the priority and the requirements of the task, it will be placed in the appropriate queue. For queuing, the architecture uses Advanced Message Queuing Protocol (AMQP) [36] compliant servers in cluster mode to ensure redundancy. The requirements of the task are derived from hardware and software dependencies of the task actions. Such requirements can be e.g., the need for a GPU or for a specific software, such as FFmpeg [37]. Upon the registration of a new worker, its capabilities are announced to the system through the monitoring component. The queues from which the worker consumes messages are then marked as capable of executing tasks with these requirements. Although software dependencies can be included in the list of the worker requirements, we consider that for a given ASML deployment, a subset of libraries or utilities will be available as common resources to all workers.
The priority of a task is determined by the client, upon the creation of a task, and it can be one of the following types: low, normal, high and critical. This flag indicates the urgency of execution of the task and it is used as a method to weight the task priority in the corresponding queue. In case of a critical task, the scheduler guarantees that the task will be executed immediately, whereas for the other types, the task will be executed in a first-in-first-out (FIFO) order. The scheduler performs a series of steps in order to guarantee the execution of critical tasks. Initially the scheduler tries to find an empty queue. If that fails, it communicates with the monitoring component to create and register a new worker. When the resources are saturated, the scheduler checks if a worker with proper capabilities is busy executing a lower priority task. In that case the scheduler places the task in the appropriate queue and signals the worker to halt the execution of the task which is then placed back in the queue. Only tasks with priority marked as low or normal are eligible for halting. In the unlikely event that the scheduler is unable to allocate resources for a new critical task, the task creation will fail, and an error is returned to the client.

Scheduling tasks derived from parallelized actions are considered a special case for the scheduler. These tasks are queued only when workers with the action requirements are idle, otherwise they fail before creation. This enables the workers to continue processing without waiting parallelized actions to be picked by a worker. It can also be used to create scenarios where resources become available after the parallelization, the parent worker can retry to parallelize the action at specific time intervals, this option is only available to tasks with high and critical priorities. To avoid resource stagnation, all tasks that derive from task parallelization are marked with low or medium priority depending on their parent task priority.

D. DATA STORAGE AND SYSTEM MONITORING MODULE

The architecture uses two types of data storage; one for heavy Input/Output (I/O) load use cases and one for file storage. For heavy I/O load use cases, Ceph [38] is used as a network file system mounted on all components of the architecture. For client file storage, such as pre-trained neural networks, image masks and the output of the workers, an object-store storage is used. The object-store storage enables meta-data to be saved along with the actual files. This property is used by ASML architecture to deal with the problem of model versioning [24], by saving the model version along with the pre-trained models. The redundancy of the network file system is achieved using ZFS [39] filesystem format in RAID-Z2 (the ZFS version of RAID-6) configuration, while the object-store redundancy is handled by software.

Monitoring is integrated at the core of the proposed system architecture and it can be broken down into two categories: one specific to system load monitoring, and one for general events such as hardware failure. In both cases of monitoring, all events are stored and are accessible by the system administrator. The information gathered by the monitoring component include, CPU, memory and GPU. As all components of the architecture are deployed as containers [40], the load monitoring system offers the ability to activate workers or API instances on-demand, depending on the hardware available to the architecture.

The monitoring component monitors the utilization of the system resources in combination with the number of tasks queued for processing and when this number exceeds a configurable threshold, it instantiates a new worker. As tasks are prioritized based on their urgency, the monitoring component is capable of reserving hardware for the execution of critical tasks. High and normal priority tasks are eligible for additional hardware allocation when available, while tasks that are marked as low priority are not. Similarly, tasks that are created by other tasks, typically derived from the parallelization of actions, are also not considered eligible for additional resource allocation in order to avoid resource saturation. This also ensures that parallelized tasks will not get affected by the latency imposed by the hardware resource allocation, such as the virtual machine or container startup. To increase the reliability of the system, the monitoring component takes into consideration worker failures and the corresponding tasks that are processed by these workers. In the case of a failure due to hardware or network issues, the monitoring component will try to allocate new hardware resources to recover. Tasks with critical, high or normal priority and failed due to this error, are automatically prioritized to be assigned on the new allocated hardware. This is important for critical tasks with high-throughput requirements, as their service is not interrupted, while tasks with high and normal priority, experience only the initial latency of the hardware initialization.

In commercial cases, where a cloud provider is used, this feature enables cost-effective deployment, where the resources are allocated according to the real-time needs of the system. As each cloud service provider offer different access to its cloud infrastructure, the load monitoring component generates events in a form of HTTP requests to a configurable endpoint.

The object-store storage of the architecture is accessible by the clients through the API component of the architecture. Upon initial authorization for each client is created, an object container (user space) to which only the authorized client has access to. A common use case scenario for this storage, is its use as a primary data-store in which a client stores trained ML component, such as trained neural networks, with meta-data information for the trained models. These are used primarily for version control and backward compatibility, which are important and still open issues with respect to the deployment of ML applications [23]. For this reason, the object-store component allows the client to issue signed URLs to these resources, granting access to the internal components of the architecture, with the ability to set expiration date for them (TTL). The monitoring component, records access to the object-store storage and all requests, enabling limiting access or bandwidth use of the store configurable on a client base manner.
The described ASML architecture is generic, suitable for the time-efficient SaaS deployment of ML-based data processing applications. In the following section, its capabilities are experimentally demonstrated with two contemporary use cases, where its capacity for real-time video processing is evaluated.

III. EXAMPLE USE CASES AND EVALUATION

To demonstrate the effectiveness of the proposed system architecture, we conducted experiments on two different ML-based data processing use cases. The first use case considers the problem of obstacle detection and recognition in the context of an assistive system for navigation of Visually Challenged (VC) individuals. Considering that such a system is meant to be used by people with disabilities, it must be accurate, fast and reliable. The second use case considers the problem of abnormality detection in gastrointestinal (GI) tract videos, in an effort to provide a solution for real-time assistance to the physicians during GI endoscopy. When the endoscopic modality used does not require real-time streaming capabilities, such as in the case of WCE [41], we demonstrate how the same processor can be used to process the videos asynchronously.

A. REALTIME OBSTACLE DETECTION, RECOGNITION AND TRACKING

Recently we presented a methodology for the detection and recognition of obstacles, and evaluated its effectiveness [8]. In this paper, we consider this as an indicative scenario to show how such a methodology can be implemented using the generic ASML architecture described in Section II, how the ASML-based implementation can be extended by incorporating an obstacle tracking algorithm, and we assess its efficiency and scalability.

1) OBSTACLE DETECTION AND RECOGNITION

The methodology presented in [8], considers that color RGB-Depth (RGB-D) image streams are captured using a stereoscopic camera. Each image is processed by two parallel components and their results are aggregated to determine image regions where high-risk obstacles are located. The first component uses a Generative Adversarial Network (GAN) [42] to generate human eye fixations that highlight salient image regions. The second component uses the depth channel of the RGB-D image, to compute three risk maps, representing high, medium and low risk obstacles, based on fuzzy logic. Following the fuzzy aggregation of the outputs of these components, the resulting sub-images corresponding to obstacle regions are provided to a CNN, called Look Behind Fully Convolutional Neural Network (LB-FCN) light [43], to perform the obstacle recognition step. The processing steps required by the methodology [8], are illustrated in Fig. 3.

The two deep learning inference steps described, are performed on a GPU-enabled server, remotely accessible for the navigation of the VC individuals. Each individual, carries a mobile phone-based wearable system, running a lightweight client application that performs image acquisition and communication with the server. Considering that not all users use the same mobile phones and that native applications are platform-dependent, as a client we considered a conventional web browser. To enable real-time streaming between the client and the system, we used WebRTC capabilities of the architecture, as it is natively supported by all major web browsers. Similarly, for real-time messaging, we used WebSockets to communicate the output of the obstacle detection and recognition back to the client. For the implementation of the obstacle avoidance schema, initially a new processor is required that splits the original RGB-D image into two parts; one with the RGB channels, and one with only the depth information. For the salient region detection and obstacle recognition, the ML processor of the proposed architecture can be used directly; thus, only the pre-trained networks need to be deployed on the object-store storage of the system. For the high-risk map generation, a new processor is required that takes as an input the depth channel of the RGB-D image and computes the high-risk map. For the detection of obstacle subimages, a processor was added following the methodology proposed [8], which takes as input the output of the second the third processor and performs the aggregation (Fig. 4, step 4). Finally, the obstacle regions along with the...
RGB image is piped to the ML processor which performs the obstacle recognition (Fig. 4, step 5). The worker configuration with the corresponding processors, input and output data source handlers is illustrated in Fig. 4. Interceptors are used to select the RGB and depth channels for the ML and high-risk map processors. The pre-trained ML models are provided to the worker as input data source handlers from the object-share storage of the architecture, the selection of which is performed using interceptors.

In this use case scenario, the performance of the worker is highly dependent on the number of obstacle regions found by the aggregation processor. As a result, in images where multiple obstacle regions are identified in a single image, the performance of the worker drops exponentially (Fig. 5). For this reason, we considered parallelization of the object recognition component (Fig. 4, step 5). To achieve this, the obstacle region processor creates a new task for each detected obstacle region and submits it to the API component for processing, effectively performing the same operation as a client would do. The output of each object recognition task is received by the worker directly, using an HTTP request, produced by the output data source handler of each object recognition task. Each request contains the label of the recognized object along with an identification number which is used by the processor to identify the obstacle region to which the label belongs to. Accounting for the latency that is introduced by the network, this enables higher frame rate compared to the conventional approach where all the information is processed on a single instance.

To perform our experiments and evaluate the performance of the proposed architecture, we used a typical smartphone device with 4 ARM based CPUs and 2 GB of RAM each one paired with Intel RealSense [44] D435i RGB-D camera. To maximize mobile cross-platform compatibility, our experiments were conducted using Google Chrome web browser as the client. For the deployment of the architecture, we used virtualization and more specifically containers through Docker [45]. For the HTTP load balancing we used NGINX [46] to distribute the incoming requests on multiple instances of API components deployed on lightweight containers. As RDBMS we used master-master deployment of MariaDB. Caching and message queueing was implemented using Redis and RabbitMQ [47] running in cluster mode respectively. The API and monitoring components, the TURN server and the workers are implemented in Go programming language.

To demonstrate the performance improvement that the proposed system architecture offers compared to conventional synchronous approaches, we conducted four experiments using different number of workers. In all experiments the workers infrastructure was equipped with an NVIDIA GTX-1080 TI GPU. The implementation of deep learning algorithms was performed using the TensorFlow framework, which enables the ML algorithms to be executed on GPUs, significantly increasing the performance. The results of the frame rate achieved by the proposed architecture using different number of workers are illustrated in Fig. 5. The single worker experiment demonstrates the performance of a conventional deployment without the use of the proposed architecture. On average, 3.2ms are required for the obstacle detection task while the obstacle recognition requires 2.1ms for each obstacle region. We found that in a typical scenario, the obstacle region processor detects 12 objects on average per image. Any increase in that number can significantly decrease the performance of a single instance deployment. A visual representation of the parallelized obstacle region classification procedure is illustrated in Fig. 6. The performance improvements that the proposed ASML architecture offers become even more apparent when multiple clients are required to be processed in parallel. In this case, a single instance deployment would have been insufficient and would require scaling, which the proposed architectures offer. Communication between multiple workers introduce network latency which can degrade the overall performance. In our experiments, using intranet connections, the network latency was on average 1.3ms. Using the ASML architecture, Fig. 5 shows that when more than 3 obstacle regions are required to be recognized, the overall performance improvement overcome the network latency.
2) OBSTACLE TRACKING

In the context of an obstacle avoidance application, tracking of the objects found in previous frames is important as it can be used to avoid re-informing the user multiple times for the same obstacle, or compute the trajectory of a moving object, such as a vehicle. The methodology presented in [8] does not include tracking; however, several single [48] and multiple [49] object tracking methodologies have been proposed over the years, which can be introduced to the system as an additional processing step. These approaches usually rely on conventional handcrafted features (such as color, shape and texture), deep learning [50], or follow the tracking-by-detection [51] approach. In tracking-by-detection methodologies, the state of the algorithm, which contains the history of the detected objects from previous frames need to be preserved in order to be compared with the detected objects of the current frame. This can be challenging when this state must be shared across multiple workers.

To overcome this, ASML architecture uses the key-value pair store to share the state across multiple workers. To demonstrate that, we enhanced the obstacle detection and recognition framework [8] methodology with object tracking [51].

As the one used in [8]. The algorithm of [51] relies on the fact, that in high frame rate scenarios consecutive frames have significantly overlapping detections. When the intersection-over-union (IOU) of the consecutive detections is lower than a certain threshold, the detection belongs to the same object.

To include the method proposed in [51] in the obstacle detection pipeline, a new processor was created (Fig. 7 step 6). This processor accepts as input the detected obstacles and their corresponding classes and outputs their tracking information. This is illustrated in Fig. 7. As processors are stateless, the previously computed tracking information, are stored in the key-value pair store. The overhead of this is minimal (on average 4.7ms per frame). This includes the data transmission time, deserialization and post processing data serialization. Fig. 8 demonstrates how the performance (FPS) is affected when a different set of workers is used in comparison to the number of bounding boxes that need to be tracked.

B. REALTIME MULTI-USER ENDOSCOPIC VIDEO ANALYSIS

In the context of computer-aided detection of abnormalities in GI endoscopy, we used the state-of-the-art LB-FCN deep CNN architecture, pre-trained to detect abnormalities in flexible endoscopy (colonoscopy) videos and Wireless Capsule Endoscopy (WCE) [52], [53]. To use this model in the proposed architecture the ML processor of ASML was utilized. Considering that the CNN architecture requires an input with spatial dimensions $224 \times 224$ pixels, the video frames received as input, have to be resized accordingly. For this reason, an image processor was used to resize the video frames to the required size using zero-padding to prevent the aspect ratio as suggested in [52].

The same CNN architecture can be used in both flexible endoscopy and WCE abnormality detection applications. In the latter case the videos are not usually streamed in real-time; instead they are obtained only after the capsules are excreted from the GI tract. For this reason, in the case of flexible endoscopy RTSP input data source handler is used, while in the case of WCE the video is stored in the object-store storage and provided to the worker via the HTTP input data source handler. In both cases, when an abnormality is detected by the ML processor, an HTTP request is sent, informing the client about the detection and the frame at which the abnormality was detected. As multiple medical instates can be benefited by such a service, in both cases we considered a SaaS cloud deployment of the ASML, were multiple physicians can access it simultaneously. Fig. 9 illustrates the worker configuration. Fig. 10 includes samples of WCE image classification results, from the KID [54] dataset, obtained using the proposed ASML architecture.

To evaluate the real-time performance of the proposed ASML architecture, multi-user experiments were conducted using different number of workers on the problem of flexible endoscopy abnormality detection. In our experiments, on average 2.8ms were required for frame classification using a single worker. Fig. 11, shows that when more than 12 endoscopes are streaming in parallel, such a singular deployment
becomes insufficient. ASML is capable of scaling horizontally by increasing the number of workers according to the required number of streams and the resources available to the architecture. This can be proven particularly useful in the case of a SaaS deployment, where abnormality detection is offered as a service to multiple clients, e.g., several clinics with one or more endoscopy units using the abnormality detection service.

C. USE CASE RESPONSE TIME ANALYSIS

To demonstrate the performance of the proposed architecture design we conducted two experiments based on the presented use cases. In both experiments three workers were used to measure the system’s response times over a period of time. In the case of obstacle detection, recognition and tracking (Fig. 12), the experiment recorded the average response time of the workers and the number of bounding boxes found at a 15 second sampling interval, over a period of 30 minutes. To show the behavior of the system in the case of a worker failure, different workers were shortly removed from the system, at different times. Fig. 12 shows that upon a worker failure, one of the running workers is successfully compensating for the loss, with the expected expense of higher average response time, due to the extra computation overhead. In total the average response time of the system was 28.1ms, and the maximum and minimum response times 48.2ms and 22.1ms respectively.

A similar system behavior can also be observed in the second use case scenario. For the abnormality detection in GI tract images, the experiment measured the average response time of the workers when used by different number of users with a sampling interval of 1 minute for 2 hours. The average response time of the workers was 3.6ms, and the maximum and minimum response times were 6.1ms and 2.4ms, respectively. When each of the two workers was removed from the system for a short period of time, the system had the same behavior as in the first use case. The response time fluctuations that are observed in both use cases (Figs. 12, 13), can be attributed to the network latency, whereas the higher response times are due to worker initialization.

IV. CONCLUSION AND FUTURE WORK

There has been work towards scalable system architectures and application frameworks that aim to provide scalable task execution. When it comes to ML, the deployment of such systems tends to be complicated and usually coupled to specific domains and use cases. The lack of an abstraction framework for the whole ML pipeline and need for a generic and standard deployment approach has been highlighted in the recent literature [23], [24]. Architectures such as [11] and [12], although scalable, are domain specific and thus they do not allow arbitrary ML task declaration and execution. The needs of ML task execution are also not satisfied by the generic architecture proposed in [15] as it focuses on task orchestration in EIoT applications which limits the scope to periodic or event-driven task modeling and it does not include an abstraction framework that can be used as a standard solution for ML pipeline task modeling. Although the declarative ML task execution system “MLBase” [17] has the advantage of automatic scaling, the platform is coupled with a specific ML framework and language [55], which is a limitation for use in production environments. The “DEEP-Hybrid-DataCloud” framework proposed in [21], although it satisfies the needs for a generic ML task deployment and execution environment, it does not offer flexibility in terms of generic ML pipeline modeling, as models with non-standard functionality exposure semantics are not compatible. Furthermore, the architecture presented in [21] does not include any standard ML task input and output handling. To cope

---

**FIGURE 9.** Diagram of a worker implementing the steps required for the flexible endoscopy and WCE abnormality detection. In flexible endoscopy, the RTSP input data source handler is used while in WCE, the HTTP input data source handler is used to provide the video from the object-store storage.

**FIGURE 10.** WCE frame classification using the proposed architecture. (a) Normal, (b) Polypoid, (c) Blood, (d) Inflammatory condition.

**FIGURE 11.** Comparison of frame rate achieved by the proposed architecture using different number of workers and number parallel endoscopes. The dotted line illustrates the real-time performance threshold of 30 fps.
with these issues in the context of remote, high throughput ML task execution, involving complex data-processing pipelines, we proposed ASML as a novel algorithm-agnostic and platform-independent system architecture. The architecture achieves that by:

- Providing a standardized, extensible and unified algorithm-agnostic task-oriented pipeline framework, with interchangeable platform independent processing units;
- Handling the pipeline execution in a highly scalable system architecture;
- Task scheduling for task parallelization.

Access to the architecture is provided through a RESTful API, enabling platform independence. This in combination with the use of open technologies, such as WebRTC, enables thin clients, such as a web browser, to use the system without special software requirements.

The results obtained from the deployment of two state-of-the-art SaaS ML application scenarios indicated that the ASML architecture is suitable for high throughput applications in different domains. The extensibility of ASML architecture was also investigated, where in the first use case the obstacle detection pipeline was extended to include obstacle tracking by the addition of an object tracking processor.
There are several other domains where ASML architecture is applicable, including robotics, transportation, and security, e.g., for SaaS deployment of ML-assisted navigation of autonomous robots and vehicles, and recognition of suspicious patterns from multiple surveillance cameras.

As a limitation of the proposed architecture, one could consider its inability to automatically identify the software requirements of each task, which can result into dependency issues. To solve this problem, the tasks are required to include labels to indicate which software dependencies are required by the task. Another limitation of the architecture is that processors are considered as black boxes and only the workers are informed about what input and output can be accepted. As a result, the API has no way of knowing if an input or output data source handler is compatible with a declared processor. This can result into situations where invalid tasks are successfully created and queued to the system, failing later, when they are picked by a worker. Although this can be resolved by including documentation for each processor, we plan to include automation validation prior task execution.

The proposed ASML architecture can utilize platforms specifically designed for parallel process execution in a multi-host environment, such as Spark [56]; however, configuring these platforms still requires advanced technical knowledge and skills. To cope with this issue, within our future research prospects is to develop the proposed system architecture to include automated host clustering, enabling parallel task execution without special configuration. An open access implementation of the proposed system architecture is planned for distribution to the wider research community.

REFERENCES

[1] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, and others, “In-datacenter performance analysis of a tensor processing unit,” in Proc. 46th Annu. Int. Symp. Comput. Archit., 2017, pp. 1–12.

[2] L. Richardson and S. Ruby, RESTful Web Services. Newton, MA, USA: O'Reilly Media, 2008. [Online]. Available: https://www.oreilly.com/library/view/restful-web-services/9780596529260/

[3] V. Badrinarayanan, A. Kendall, and R. Cipolla, “SegNet: A deep convolutional encoder-decoder architecture for image segmentation,” IEEE Trans. Pattern Anal. Mach. Intell., vol. 39, no. 12, pp. 2481–2495, Dec. 2017, doi: 10.1109/TPAMI.2016.2664615.

[4] H. Fabelo, S. Ortega, S. Kabwama, G. M. Callico, D. Bulters, A. Szolna, J. J. F. Ponge, and R. Sarmiento, “HELICOID project: A new use of hyperspectral imaging for brain cancer detection in real-time during neurosurgical operations,” Proc. SPIE Hyperspectral Imag. Sensors, Innov. Appl. Sensor Standards, vol. 9860, May 2016, Art. no. 986002, doi: 10.1117/12.2223075.

[5] D. K. Iakovidis, S. V. Georgakopoulos, M. Vasilakakis, A. Koulouzidou, and V. P. Plagianakos, “Detecting and locating gastrointestinal anomalies using deep learning and iterative cluster unification,” IEEE Trans. Med. Imaging., vol. 37, no. 10, pp. 2196–2210, Oct. 2018, doi: 10.1109/TMI.2018.2837002.

[6] P. Wang, Z. Ding, C. Jiang, and M. Zhou, “Design and implementation of a Web-service-based public-oriented personalized health care platform,” IEEE Trans. Syst., Man, Cybern. Syst., vol. 43, no. 4, pp. 941–957, Jul. 2013.

[7] J. I. Fernandez-Villamor, C. A. Iglesias, and M. Garjio, “A framework for goal-oriented discovery of resources in the RESTful architecture,” IEEE Trans. Syst., Man, Cybern. Syst., vol. 44, no. 6, pp. 796–803, Jun. 2014.

[8] M. A. Martinez-Prieto, C. E. Caesta, M. Arias, and J. D. Fernandez, “The solid architecture for real-time management of big semantic data,” Future Gener. Comput. Syst., vol. 47, pp. 62–79, Jun. 2015.

[9] C. Pang, J. Yan, and V. Vyatkin, “Time-complemented event-driven architecture for distributed automation systems,” IEEE Trans. Syst., Man, Cybern. Syst., vol. 45, no. 8, pp. 1165–1177, Aug. 2015.

[10] S. Ahmad and D. Kim, “A multi-device multi-tasks management and orchestration architecture for the design of enterprise IoT applications,” Future Gener. Comput. Syst., vol. 106, pp. 482–500, May 2020.

[11] X. Xiao, Z. Gao, W. Shi, X. Qiu, Y. Yang, and L. Rui, “EdgeABC: An architecture for task offloading and resource allocation in the Internet of Things,” Future Gen. Comput. Syst., vol. 107, pp. 498–508, Jun. 2020.

[12] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan, “MLbase: A distributed machine-learning system,” in Proc. 6th Biennial Conf. Innov. Data Syst. Res. (CIDR), 2013, pp. 1–2. [Online]. Available: http://cirdb.org/cidr2013/Papers/CIDR13_Paper118.pdf

[13] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, “Pig latin: A not-so-foreign language for data processing,” in Proc. ACM SIGMOD Int. Conf. Manage. Data, 2008, pp. 1099–1110.

[14] M. Ribeiro, K. Greqlinger, and M. A. M. Capretz, “MLaaS: Machine learning as a service,” in Proc. IEEE 14th Int. Conf. Mach. Learn. Appl. (ICMLA), Dec. 2015, pp. 896–902.

[15] S. Chan, T. Stone, K. P. Szeto, and K. H. Chan, “PredictionIO: A distributed machine learning server for practical software development,” in Proc. 22nd ACM Int. Conf. Inf. Knowl. Manage., 2013, pp. 2493–2496.

[16] T. Kraska, A. Garcia, J. M. De Lucas, M. Antonacci, W. Zu Castell, and M. David, “A cloud-based framework for machine learning workloads and applications,” IEEE Access, vol. 8, pp. 18681–18692, 2020, doi: 10.1109/ACCESS.2020.2964386.

[17] M. Habiba, M. R. Islam, and A. B. M. S. Ali, “A component based unified architecture for utility service in cloud,” Future Gener. Comput. Syst., vol. 87, pp. 725–742, Oct. 2018.

[18] D. Schelter, F. Biessmann, T. Januschowski, D. Salinas, S. Seufert, and G. Szarvas, “On challenges in machine learning model management,” in Proc. Bull. IEEE Comput. Soc. Tech. Committee Data Eng., 2018, pp. 5–15.

[19] L. Baier, F. Jüren, and S. Seebacher, “Challenges in the deployment and operation of machine learning in practice,” in Proc. 27th Eur. Conf. Inf. Syst. (ECIS), 2019, pp. 1–15.

[20] A. H. Bader and R. Pennington, Applications, Int. J. High Perform. Comput. Appl., vol. 8, pp. 18681–18692, 2020, doi: 10.1109/ACCESS.2020.2964386.

[21] Y. Li and S. Manoharan, “A performance comparison of SQL and NoSQL databases,” in Proc. IEEE Pacific Rim Conf. Commun., Comput. Signal Process. (PACRIM), Aug. 2013, pp. 15–19.

[22] D. Hardt and others, The OAuth 2.0 Authorization Framework, document RFC 6749, Oct. 2012.

[23] H. Buder and R. Pennington, “Applications,” Int. J. High Perform. Comput. Appl., vol. 15, no. 2, pp. 181–185, 2001.

[24] A. Bergkvist, D. C. Burnett, C. Jennings, A. Narayanan, and B. Aboba. (2010). [Online]. Available: https://www.w3.org/TR/2010/WD-oidc-20100727/

[25] A. Van Kesteren and others. (2010). [Online]. Available: https://www.w3.org/TR/2010/WD-cors-20100727/
