CloudSkin

Funded by the European Union

HORIZON EUROPE FRAMEWORK PROGRAMME

CloudSkin

(grant agreement No 101092646)

Adaptive virtualization for AI-enabled Cloud-edge Continuum

D5.2 Learning methods for Infrastructure and Workload management

Due date of deliverable: 30-06-2024
Actual submission date: 28-06-2024

Start date of project: 01-01-2023 Duration: 36 months

Summary of the document

Document Type	Report
Dissemination level	Public
State	v1.0
Number of pages	40
WP/Task related to this document	WP5 / T5.1, T5.2, T5.3, T5.4-T5.7(Usecase learning part)
WP/Task responsible	BSC
Leader	Josep LL. Berral (BSC)
Technical Manager	Peini Liu (BSC)
Quality Manager	Ardhi Putra Pratama Hartono (TUD)
Author(s)	Peini Liu(BSC), Marc Palacin Marfil (BSC), Joan Oliveras Torra (BSC), Josep Lluis Berral Garcia (BSC), Jose Miguel Garcia (ALT), Marc Sanchez-Artigas (URV), Josep Calero (URV), Raúl Gracia (DELL), Sean Ahearne (DELL), Colin Duggan (DELL), Omar Jundi (DELL)
Partner(s) Contributing	BSC, DELL, NCT, ALT, URV
Document ID	CloudSkin_D5.2_Public.pdf
Abstract	This deliverable presents the first advances and specifications of the infrastructure for the learning plane, and the initial ML-based methods in the use cases. Also, here is described the first version of the API and standard connections, plus the infrastructure for telemetry and data retrieval.
Keywords	Learning Plane, Telemetry Infrastructure, ML-based methods.

History of changes

Version	Date	Author	Summary of changes
0.1	06-03-2024	Peini Liu, Josep Ll. Berral	First draft.
0.9	12-06-2024	Peini Liu, and use case partners contribute each use case modelling and data retrieval	Release Candidate Draft.
0.9	19-06-2024	Marc Sanchez, Josep Carelo	Contributions to sections 4.3 and 5.2.
1.0	17-06-2024	Peini Liu	V1.0 version for internal review.
	17-06-2024	Ardhi Putra Pratama Hartono	First internal review.
2.0	26-06-2024	Peini Liu	Final version after revision.

Table of Contents

List of Abbreviations and Acronyms

API	Application Programming Interface
AWS	Amazon Web Services
CC	Creative Commons
CDF	Cumulative Distribution Function
CSV	Comma-separated values
DOI	Digital Object Identifier
FIFO	First In, First Out
HPC	High Performance Computing
HPDA	High Performance Data Analytics
LP	Learning Plane
LSTM	Long-Short Term Memory
ML	Machine Learning
PoC	Proof of Concept
QoS	Quality of Service
RL	Reinforcement Learning
SLO	Service Level Objective

1 Executive summary

This deliverable, **Deliverable D5.2** "Learning methods for Infrastructure and Workload management", presents the updated specifications of the Learning Plane's architecture, along with demonstration of the initial prototyping implementations on the project use cases, bringing forward the proposed implementations published in previous deliverables **Deliverable D5.1** and **D2.4**. To be specific, this deliverable provides detailed specification and explanations of the Learning Plane's main component; the ML-based methods towards workload analysis, characterization, and the modelling and prediction methods; plus the description of the telemetry components from the Data Plane used for data retrieval to the Learning Plane.

2 Introduction

The progress on the Learning Plane development and evaluation involves the design of the Data Connector specifying the functionalities for modelling and prediction of the acquired telemetry towards recommendations, on different environments for computing. In the CLOUDSKIN project, Task 5.2 is in charge of producing this design, in an iterative-prototyping methodology, defining at this time the updated architecture of the connector and its implementation. Following M18, Task 5.3 will proceed with the instrumentation and deployment of the connector, being able to iterate over the modeling algorithms, the initial adaption of the APIs towards novel use cases, and the publication of the specification towards the general public. During this process, the advances of Task 5.4 to 5.7 also contribute to the definition of the Learning Plane and the expected contents (models and algorithms, along with the related environments such as execution and storage of them), also advancing the machine learning models to be used in such use cases, as the primary purpose of the Learning Plane.

2.1 Development progress

The first design of the Learning Plane is reported on D2.1 and D5.1, in this report we updated the Learning plane design as a function of Data connector(in Section 3) and its implementation. Then for the development stage (Tasks 5.2) and the deployment stage (T5.3), we described the progress of "Workload Characterization - ML-based methods for environment modelling" (in Section 4) and early advance of "Instrumentation and deployment - Instrumentation and Data Retrieval in the Learning Plane" (in Section 5). During the following months, the use cases will integrate the T5.1, T5.2 and T5.3 algorithms with the architecture and the technologies provided by all involved CLOUDSKIN partners. The current progress of the platform and the use case are reported to D2.3. Table 1 shows the stages, along with the corresponding efforts and WP5 tasks, also the sections of this deliverable where progress and experiments are provided.

	Stage 0	Stage 1	Stage 2	Stage 3
Efforts	Learning Plane Design	Workload Characterization	Instrumentation & Deployment	Integration with Use Cases
Ref. Tasks	T2.1, T5.1	T5.1, T5.2	T5.3	T5.4, T5.5, T5.6, T5.7
Report Detail	D2.1, D5.1 and Section 3	Section 4	Early advances: Section 5	D2.3: Section 5
Experiments	NA	Section 4	Preparing	D2.3: Section 5

Table 1: Relation of efforts, tasks and reporting sections in this deliverable.

Tasks T5.4 to T5.7 are currently ongoing (described in D2.3 Section 5), focusing on the development of specific details for each use case, while preparing individual learning methods towards being integrated or collected by the learning plane when properly implemented into data connectors. The following subsection introduces the current status of machine learning methods for each use case, to be described in this report.

3 Updated implementation of the learning plane

This section extends "Learning Plane design and integration" from **Deliverable D2.1** and "Design and Early Prototype of the Learning Plane" from **Deliverable D5.1**. For the fundamental details on the initial design and road-map of the Learning Plane with respect to the rest of the architecture, refer to those deliverables.

3.1 Learning Plane Functionalities

The designed Learning Plane is being integrated by technologies capable to fulfill the different functionalities with regard to retrieving telemetry, modeling the system, managing the catalog of models and serving recommendations, predictions and forecasting. For such, the principal functionalities are listed as follows:

- **Modelling the System**: This functionality involves the set of algorithms for system modelling and characterization. For this, different methods are being explored and benchmarked, starting from naïve methods (heuristics) to statistical and machine learning methods that we have researched previously and then adapt them for use case scenarios, such as ThetaScan [1] and AI4DL [2](Introduced in D5.1). Proposed ThetaScan and AI4DL can detect the statistical properties in the workloads as multi-variate time-series, and also currently we are exploring time-series-based neural networks for workload predictions such as Transformers and Long-Short Term Memory(LSTM). Current efforts are focused on accurate models detecting unexpected and extreme changes in workload behaviour with respect to resource demand, i.e., peaks that could affect the quality of service.

- **Model Federation and Storage**: For model federation, this functionality involves the distribution of the models sharing and aggregation. In this project, we focus on using models for resource provisioning and scheduling. Monolithic models are now generated to utilize data from aggregated monitoring stack to advice holistic scheduling among the platforms. Federated models will be generated in the scenario of resource provisioning, where small models can be used to trace the statistical properties of workloads in the edge, and the cloud can use policies to aggregate predictions and make actions to fit the workload requirements. For model storage, different candidates were considered as distributed file-systems and object storage platforms. Firstly, we use Hadoop HDFS for distributed data with replication [3, 4] and GekkoFS for distributed in-memory volatile data [5] were considered. Such systems are a baseline option without the optimizations required for the current case. Therefore, our preferences are object storage systems, like GEDS [6], providing a distributed storage system with replication and resilience, developed by the IBM partners of CLOUDSKIN.

- **Provisioning Execution**: This functionality involves the execution of prediction and recommendation, and provisioning algorithms with the retrieved data and learned models(i.e., model serving and model inference). Current serving technologies involve Tensorflow serving, Torch serve, Seldon etc., or custom containerized inference engines that fit different libraries such as PyTorch, TensorFlow, ScikitLearn, etc. These model inference applications are containerized thus can be deployed in different platforms, such as Kubernetes, K3S etc.

3.2 Learning Plane as Data Connectors

The design of the Learning Plane (LP) principal component involves LP agents responsible for providing predictions and recommendations post-modelling. These agents function as data connectors, equipped with an API that details their configuration, inputs, and outputs.

The cycle of the Learning Plane Data Connector starts at the submission of an application to be placed in a certain environment (e.g. Kubernetes cluster, managed by an orchestrator). An instance of the connector (the LP agent) is configured, indicating where to run, which application or applications to follow, which sources of telemetry are available, which API should it use to communicate decisions

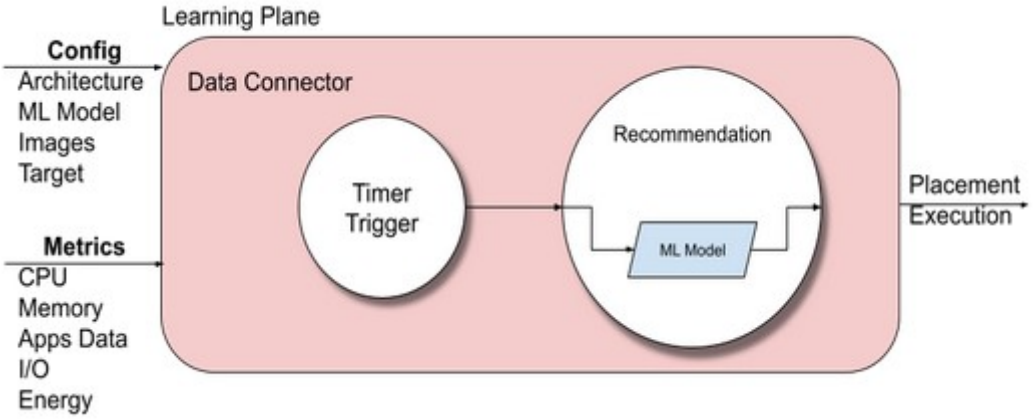


Figure 1: Initial schema of the data connector for the Learning Plane

and actions to the orchestrator or scheduler, and which models should it use (or train). Then, as an active agent, the connector triggers continuously after a configured time, to retrieve data from the telemetry, generate a forecasting or recommendation, and communicate it to the orchestrator or scheduler.

The data connector architecture defines the following elements:

1. A **configuration**, indicating how the instance of the data connector will be run and against which applications. This configuration will indicate which models or algorithms are retrieved for modelling and prediction (this can be either for training, updating, forecasting and recommendation), and which are the hyper-parameters for such models. Also, it will indicate the IDs for the applications to be monitored and made decisions of, in order to produce models and recommendations for them. And finally, the configuration will indicate the data connector instance which are the sources of data to be ingested by the modelling (in the current use cases, the telemetry from the system collector or databases, from a given API REST), also the connection-points where to send produced decisions and forecasting (that is the orchestrator or scheduler API triggering changes on the system).

2. An **input entry-point**, defined by an active data reader receiving the data to be ingested by the model. When the data connector is instantiated, as an agent accompanying the applications and environment to be managed, it will collect data from the telemetry sources continuously, towards performing continuous analytics, in this case the predictions and recommendations, or the training and updating.

3. An **output out-point**, defined by an active data producer, generating the forecastings or recommendations, accordingly to a decided format, to be sent through the REST APIs of the orchestrators or actuators, to perform the recommended actions over the workloads (given the current use cases).

As additional elements to the data connector, there is the model storage to be shared or accessible across LP instanced agents. A data connector prepared to create a model or update a model needs access to the shared storage for retrieving and preserving models, in a collection of available models for specific applications. The current work in progress defines three courses of action:

- Research on workload characterization towards modelling. As indicated in next Section 4.1.1 and 4.1.2, current works are focusing on the discrimination and characterization of workloads, based on their telemetry indicating resource requirements. Through such discrimination, using unsupervised learning techniques to dynamically admit new applications without an strict supervision of system operators, we can classify incoming applications by behavior (resources

consumption along execution), and select the appropriate model for forecasting their consumption and make decisions over them. The model selection can correspond to a user-indicated configuration, allow the Learning Plane agent to observe the initial behavior of the application and then decide which model to use, or a mix of both options letting the user to select a kind of model and allow the LP to refine the selection among similar models (e.g., models based on the same algorithm but with different parameters, or trained on different data).

- Policies for placement making. Also, indicated on Section 4.1.1, current works are focusing on the policies and service-level agreements for use cases, to be adjusted and matched for recommendation on provisioning. This involved models that, once applications have been classified, can produce a recommendation among available options, to be submitted towards the orchestrators and schedulers.

- Proof-of-Concept for the Data Connector. The implementation of a prototype for the Learning Plane data connector is being tested on the Kubernetes + NearbyONE environments, and current works are focusing on defining the APIs for data connector, telemetry mechanisms and orchestrators to communicate. The prototype of the learning plane is detailed 3.3, and in Deliverable 2.3 Section 4.1.1, the usage of it in a usecase is described in Deliverable 2.3 Section 5.1.

3.3 Learning Plane Prototype

The implementation architecture of the data-connector is shown in Figure 2. The data-connector acts as an LP agent in charge of providing QoS predictions of the applications and generating application placement recommendations for the orchestrator. The implementation of the agent is based on Scanflow-k8s [7][8], where it provides different deployments of model predictions(including inference pipeline), tracker of metadata/parameters and model registry, and an agent framework for autonomic management which the engineer only needs to provide custom sensors/actuator based on each scenario. Specific components are described below:

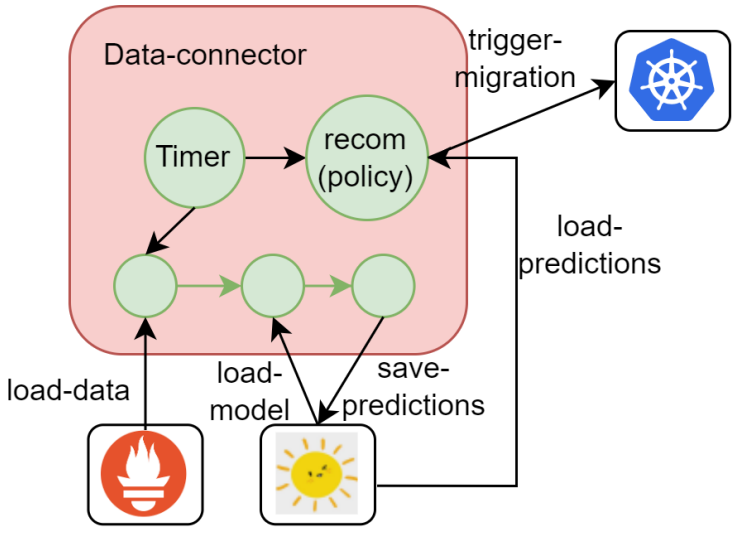


Figure 2: Implementation architecture of the data connector for the Learning Plane

- **Inference pipeline:** The data-connector has the inference pipeline to predict the QoS of a given application on all machines. This pipeline can connect with different models/parameters saved in scanflow. A scanflow pipeline specification is shown :

```
#predictor
executor1 = client.ScanflowExecutor(name='download-model',
```

```
                    mainfile='download.py',
                    parameters={'app_name': app_name,
                                'team_name': team_name,
                                'model_name': 'lstm',
                                 'model_version': 1})

executor2 = client.ScanflowExecutor(name='preprocessing-batch',
                    mainfile='main.py',
                    parameters={'name': 'preprocessing'})

executor3 = client.ScanflowExecutor(name='predictor-batch',
                    mainfile='main.py',
                    parameters={'name': 'preditor'})

executor4 = client.ScanflowExecutor(name='postprocessing-batch',
                    mainfile='upload.py',
                    parameters={'name': 'postprocessing'})

dependency1 = client.ScanflowDependency(dependee='download-model',
                                depender='preprocessing-batch')
dependency2 = client.ScanflowDependency(dependee='preprocessing-batch',
                                depender='predictor-batch')
dependency3 = client.ScanflowDependency(dependee='predictor-batch',
                                depender='postprocessing-batch')

##workflow1 batch-inference-graph
workflow1 = client.ScanflowWorkflow(name='batch-inference-graph',
                   nodes=[executor1, executor2, executor3, executor4],
                   edges=[dependency1, dependency2, dependency3],
                   type = "batch",
                   cron = "*/5 * * * *",
                   output_dir = "/workflow")
```

Listing 1: QoS prediction workflow

To deploy this workflow,

```
deployerClient = ScanflowDeployerClient(user_type="local",
                                deployer="argo",
                                k8s_config_file="/home/rocky/.kube/config")
await deployerClient.run_app(app=build_app)
```

Listing 2: QoS prediction workflow deployment

- **Recommender:** The data-connector has a recommender, where the recommender senses the results of the predictions and triggers the orchestrator. For instance, in Mobility usecase, the sensor gets the application QoS predictions, a policy is enabled to choose the machine id with max_qos, the actuator triggers at this time k8s to patch the placement of the application deployment. The custom function of the sensor and the actuator of the agent is listed below:

```
#example 1: watch app QoS predictions
@sensor(nodes=["predictor"])
async def watch_qos(runs: List[mlflow.entities.Run], args, kwargs):
    max_qos = 0
    if runs:
        max_qos = runs[0].data.metrics['max_qos']
        max_qos_index = runs[0].data.params['max_qos_index']
        if qos_constraints(max_qos):
            await call_migrate_app(max_qos_index, "scanflow-cloudedge-dataengineer", "nginx-deployment")
        else:
            logging.info("all machine can not achive qos sla, no actions")
    else:
        logging.info("no data in last check")
    return max_qos
```

Listing 3: Custom sensor to get maximum app QoS predictions

```
async def call_migrate_app(max_qos_index, namespace, deployment_name):
    nodeName_list=['cloudskin-k8s-control-plane-0.novalocal',
```

```
                'cloudskin-k8s-worker-1.novalocal',
                'cloudskin-k8s-worker-0.novalocal',
                'cloudskin-k8s-edge-worker-2.novalocal',
                'cloudskin-k8s-edge-worker-1.novalocal',
                'cloudskin-k8s-edge-worker-0.novalocal']
    # Prepare the patch
    patch_body = {
        "spec": {
            "template": {
                "spec": {
                    "nodeSelector": {"kubernetes.io/hostname":nodeName_list[int(max_qos_index)]}
                }
            }
        }
    }
    logging.info(f"agent is patch deployment to node - {patch_body}")
    #connect k8s
    config.load_incluster_config()
    api_instance = client.AppsV1Api()
    try:
        api_instance.patch_namespaced_deployment(
            name=deployment_name,
            namespace=namespace,
            body=patch_body
        )
        logging.info("update_deployment_with_patch succeeded")
        return True
    except client.api_client.rest.ApiException as e:
        logging.error(f"update_deployment_with_patch failed: {e}")
        return False
```

Listing 4: Custom actuator to connect k8s platform

- **Timer:** The data-connector has a timer to trigger actions. Data-connector agent has different types of built-in triggers, namely interval triggers, date triggers, and cron triggers (see Table 2). Also, the basic triggers can be combined together using 'and' or 'or' logic to produce more complex hybrid triggers. These triggers can be scheduled at a specific time or time intervals to execute tasks so that agents could get required observations to evaluate the changes of the environment.

Table 2: Types of agent triggers.

Types		Definition
Scheduled Triggers	Interval	Trigger at the specified frequency.
	Date	Trigger once on the given date and time.
	Cron	Trigger when current time matches all specified time constraints (similarly to UNIX cron).

The timer should set a specific trigger and bind to a sensor, the below list shows a definition of the timer binding to a 'watch_qos' sensor of planner agent.

```
trigger = client.ScanflowAgentSensor_IntervalTrigger(minutes=5)
sensor = client.ScanflowAgentSensor(name='watch_qos',
                                    isCustom=True,
                                    func_name='watch_qos',
                                    trigger=trigger,
                                    kwargs={'frequency':300})
```

Listing 5: Set trigger to watch_qos sensor

After implementing the custom sensor and actuator, the agent can be deployed in the platform to proactively make decisions.

```
#planner
planner = client.ScanflowAgent(name='planner',
                               template='planner',
                               sensors=[sensor])
```

Listing 6: Agent deployment

The full example and tutorial is located in a Github repository: Data-connector `https://github.com/bsc-scanflow/data-connector`

4 ML-based methods for environment modelling

Section 4.1, 4.2, 4.3 and 4.4 introduce the machine learning models to be leveraged for efficient resource provisioning and data and workloads allocation for each use case T5.4, T5.5, T5.6, T5.7, respectively.

4.1 Workload scaling and resource allocation

Here we present the methods for scheduling and scaling applications upon resources, and towards modelling and characterising workloads. Section 4.1.1 introduced model training for cloud-edge application QoS predictions for mobility use case. Section 4.1.2 and section 4.1.3 described the progress of the tasks described in Deliverable 5.1, in specific, new ML methods(i.e., transformer) for workload pattern characterization predictions and the smart policy of scheduling HPDA. The later work has been submitted to MIDDLEWARE 2024 and is currently under review.

4.1.1 Cloud-Edge Heterogeneous Modelling

As previously indicated, the connector designed for the learning plane can be considered in this use case as an agent (or set of agents) with a configuration, indicating the kind of metrics to learn and predict, and the models to use with their specific predictions and recommendations. Such recommendations are passed to the scheduler through a specific API (from the scheduler or orchestrator). Given the current scenario and design, completing a first prototype requires: the generation of data to train and test the models, a collection of technologies to implement the telemetry continuous collection, and technologies for acting as back-end for the machine learning methods and processes. For the completion of this prototype, we have developed a methodology depicted in Figure 3. This methodology focuses on the steps needed for the design of the first prototype, but also highlights in blue the steps that will be integrated within the Learning Plane for continuous development and continuous integration of models after the development of the first prototype.

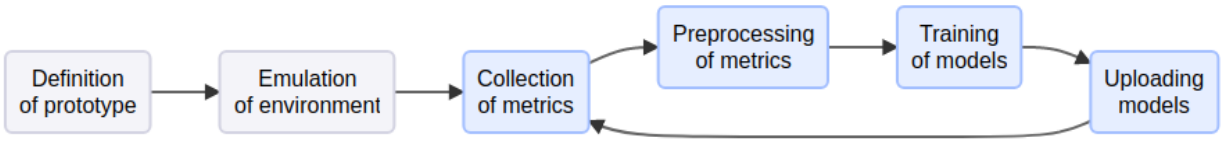


Figure 3: Methodology for the creation of a prototype. First two grey steps are needed for the environment preparation. Blue steps with the last four steps will generate a model for Learning Plane.

Definition of prototype: The initial prototype should integrate predictive placement of the video-analytics application to enhance it's QoS in an environment where other applications are also consuming resources. Such prediction requires modelling of the application and environment, for which data will need to be collected and most importantly, the emulation of a realistic environment is needed.

Emulation of environment: We developed Stress-Profiler, a custom-made Python library, emulating an environment with multiple applications running, stressing CPU nodes and memory using behaviors and patterns from real Cloud systems, where a generic but statistically solid resource usage can be modelled and replicated, putting our target applications in scenarios of resource competition. Stress-Profiler is divided into multiple parts:

- Data collection: using data from real Cloud systems such as Alibaba [9], we feed and preprocess data from multiple sources ensuring multiple generic and real resource behavior telemetry. Apart from the resource telemetry, we also keep statistics such as the distribution of the duration of the jobs, and the distribution of the generation of jobs to emulate a realistic environment where multiple applications are running.
- Profiling: applying AI4DL [2] to the preprocessed data, we focus on discovering behaviors and phases, using this deep learning clustering technique, we are able to translate the time

series telemetry data into time series phases data, which can easily be profiled, allowing us to effectively compare a massive amount of data.

- Pattern compressing: using as input the results from AI4DL [2], we compress the phases sequences in order to detect repeated patterns between applications, this is key as it ensures we will emulate a balanced environment, where all profiles of applications are represented, adding resilience to our future trained models.
- Stress generation: we generate stress based on randomly selecting profiles of applications found when compressing the patterns. We use statistics from real Cloud systems such as the job generation distribution and the job duration distribution to mimic a realistic environment. With this randomized, yet realistic design, we allow for continuous creation of different balanced environments, ensuring that the environment we generate is different yet realistic every single time, which will be key when training models to avoid over-fitting with repeated data.

Collection of metrics: This step focuses on the data retrieval methods for modelling and prediction for the creation of the prototype. Find details of the developed solution in Section 5.1. From this step on, the methodology designed will be integrated within the Learning Plane, as we aim for designing a prototype that can recurrently collect real-time data, preprocess it, retrain and update the developed models.

Preprocessing of metrics: This step focuses on the development of the preprocessing pipeline to be used with the metrics collected. The preprocessing tackles cleaning nulls, duplicates, rearranging the data and normalizing values for training purposes. It also ensures structure consistency keeping data-types and metrics consistent across time, allowing us to store preprocessed data persistently and then use it for training models. Further use of DL tools such as AI4DL [2] is being considered for preprocessing the data, as it allows us to efficiently encode the timeseries data into phases, thus performing feature engineering with our telemetry variables.

Training of models: This step focuses on training Machine Learning and Deep Learning models, with the objective to infer QoS predictions from real-time data in order to maximize the QoS at the placement of the application. Currently, we conducted benchmark in the Cloudskin testbed of the mobility usecase, collected traces and trained an LSTM model for workload QoS predictions as shown in Fig 4.

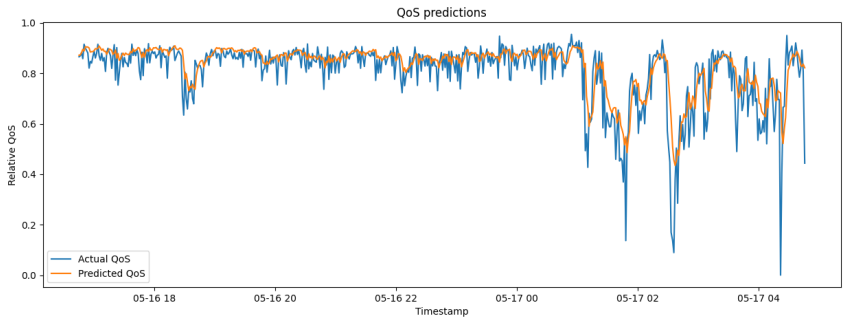


Figure 4: LSTM model for workload QoS predictions.

Uploading models: This step will save models in a model repository for data connector to use in model inference, allowing the data connector to keep a library of multiple trained and updated ML and DL models. This serves as the first step to allow the Learning Plane to smartly choose the best model for every given scenario, keeping a library of models with different inference purposes and capabilities.

In this deliverable, we introduced a pipeline to train a ML baseline model for QoS prediction in a Cloud-Edge environment. The next setup of this work is to add the model to the repository and serve the model by using the Learning Plane, allowing real time inference on the expected QoS and using the knowledge of predictions to generate recommendations and trigger orchestrators(i.e.,

Kubernetes, NearbyOne Orchestrator).

4.1.2 Workload Pattern Characterization through Transformers

In the last report, we indicate that the resource consumption of a workload is not constant, neither stable. Resource usages can have steadiness, variability and abrupt behaviours. In some occasions, the change between different behaviours is abrupt forming what are commonly named spikes, or "burstiness" behaviour. Such varied consumption leads Cloud resource users to demand an allocation of resources far over the average usage. This is an easy but inefficient solution for preventing the highest spike of resource consumption to be underneath the requested limit. Otherwise, the container or virtual machine may be evicted before completing the execution. For this reason, the vast majority of resources in common Cloud infrastructures are on-line but underused, wasting energy while costing users and providers their (idle) running costs.

Research in Cloud computing has been working for years to avoid the waste of computing resources, through co-location and consolidation of applications, dealing with the added risks of overwhelming resources and producing a degradation of the provided Quality of Service. Current approaches focus on co-locating applications with different resource needs in the same computing nodes, avoiding resource competition, or co-locating applications with different demand or tolerance towards lack of resources, allowing low tolerance applications to lend resources to high tolerance ones when required. Such processes are transparent to the user, who only observes the progress of their tasks. However, this approach endures big problems when heavy spikes occur in bursty workloads, as such demand cannot be satisfied when another co-located application is also having a burst or a period of high demand. This impacts the workloads available resources, severely degrading their QoS. For that reason, we are focusing on methods that are able to predict the future consumption of workloads, allowing Cloud providers to anticipate sudden increases of demand and to have an overall good resource management in their computing clusters.

There are plenty of works in the literature dealing with resource usage forecasting, attempting to predict with high accuracy time series on CPU and memory demand, but the vast majority of proposed models are old methods falling back into regression algorithms or variants like Random Forests or Support Vector Machines. Such methods do not incorporate the last advances in time series forecasting, not even multi-variate series, even less the latest advances in deep learning towards time series. One of such advances are the so-called Transformers, proposed by Google in 2017 [10]. Transformers have revolutionized the state-of-the-art of natural language processing, moving from recurrent neural networks to convolutional blocks with full-attention layers. Inputs are processed all at the same time, by different parts of the model, receiving between its multiple layers, all the outputs from their counterparts. These methods have been researched for the past years, showing their advantage over their predecessor models. The current state-of-the-art on Transformers include the Informer [11] and YFormer [12] models, both simplifications of the original architecture with a few extra features for time-series. While the Informer model uses sparse convolutional layers to implement the attention mechanism to reduce drastically the neural network size, the YFormer model merges the Transformer architecture with the well-known U-Net neural network architecture, also allowing it to be used in application domains such as computer vision aside of the already mentioned natural language processing.

The principal target for using Transformers is NOT to forecast a time-series $T_{n+1} \dots T_{n+w}$ from $T_{n-w} \dots T_n$, but to forecast specific behaviours (i.e. spikes and sudden changes on the demand), as we are not interested on knowing the exact consumption but on the changes on its ceiling towards the next provisioning window. Current methods of evaluation, and hence for training and fitting models, are totally oriented to the default forecast matching (find $T_{n+1} \dots T_{n+w}$ with high accuracy), like MSE and MAPE, therefore not really useful for Cloud provisioning. Models trained with such metrics put all focus on matching time series more than our real target, not evaluating abrupt spikes on resource demand correctly, nor conforming a proper analysis of the generated model performance towards our main interests. As an example, Figure 5 shows a resource usage trace alongside a prediction from the default forecasting matching and, in contrast, what would be the desired prediction in a

Cloud provisioning scenario.

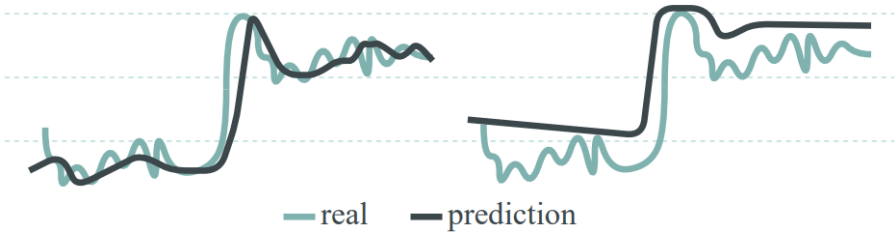


Figure 5: Example that depicts the resource usage of a workload alongside a prediction from the default forecast matching (left) and what would be the desired prediction in a cloud environment (right). Contrary to the left prediction, the example from the right is able to reduce possible cases of resource starvation and to foresee future increases with anticipation.

As stated in the last deliverable, the current works do not focus on specific behaviours and evaluation functions tailored for such, and we were focusing our research on resource prediction towards a new set of functions that allowed attention neural network such as Transformers to learn specific patterns to be discovered or forecasted, specifically:

- The study and research of novel modeling methods based on Transformers for proper prediction and anticipation of sudden behaviours towards resource allocation, putting specific emphasis in the correct prediction of patterns like spikes. The objective is to learn and recognize common patterns that can precede a spike in the resource consumption, while keeping the model agnostic about a specific workload and its dependencies, only obtaining the required information from past values.
- The adaptation of the current evaluation strategies to the Cloud provisioning field, assessing a proper allocation of resources during the execution of a workload, and the correct prediction in time and amplitude of resource consumption spikes.

Since the last deliverable, we did obtain prominent results in both research lines, and we are now preparing a publication with them. Nevertheless, we are still undergoing some experimentation to better support the desired contributions. Specifically, we are working on incorporating the following improvements:

- The employment of state-of-the-art methodologies to explain the reasons behind the predictions of the models. Analysing what important features are detected in the original data to predict future spikes. This will validate the learning of the models, and also will open the room for possible improvements in the efficiency of these models. Once we know which information is decisive for the final predictions, we could reduce the problem to just searching these patterns instead of making use of a full deep learning network.
- The analysis of the adapted evaluation strategies. Checking their relation with standard evaluation methodologies. Understanding in which cases they work better and in which cases they end being not beneficial.

4.1.3 Scalable Scheduling for HPC/HPDA

Leveraging serverless platforms for the efficient execution of large-scale data analytics frameworks, such as Apache Spark, has gained substantial interest since early 2022. The flexibility, elasticity, free-of-management, and on demand scalability offered by serverless have motivated the effort in deploying large-scale data analytics applications to serverless platforms. However, figuring out how to autoscale resources for such complex workloads so that we can fully benefit from the flexibility and

elasticity of serverless is a non-trivial challenge. Mis-configuration can result in severe performance and cost issues arising from resource under- and overprovisioning.

Current systems allocate resources at the per-application level, using general heuristics like fair scheduling, shortest-job-first, and simple packing strategies, ignoring completely application characteristics. For instance, by default, the scheduler of Spark [13] allocates the resources in a FIFO manner: the first stage gets priority on all available resources, then the second stage gets priority, etc. Figure 6 shows an example of a computation over 10 executors when relying on Spark's default FIFO policy. As can be noticed, this leads to a high inefficiency: looking to a given stage as a black-box, all resources are allocated even if a given stage does not need all of them. Considering application-specific characteristics, e.g., stage parallelism levels, systems can efficiently allocate shares of resources to application stages to achieve similar or higher overall performance. Setting the scale-out level at lower granularity, i.e., per-job or per-stage, avoids wasting resources on jobs/stages with little inherent parallelism or running on small input data, leaving more free resources to jobs/stages with large input which can harness additional parallelism. Efficient utilization of resources matters for both the service provider and the user: the provider can save millions of dollars at scale, and the user can benefit from higher performance at the same or even lower cost.

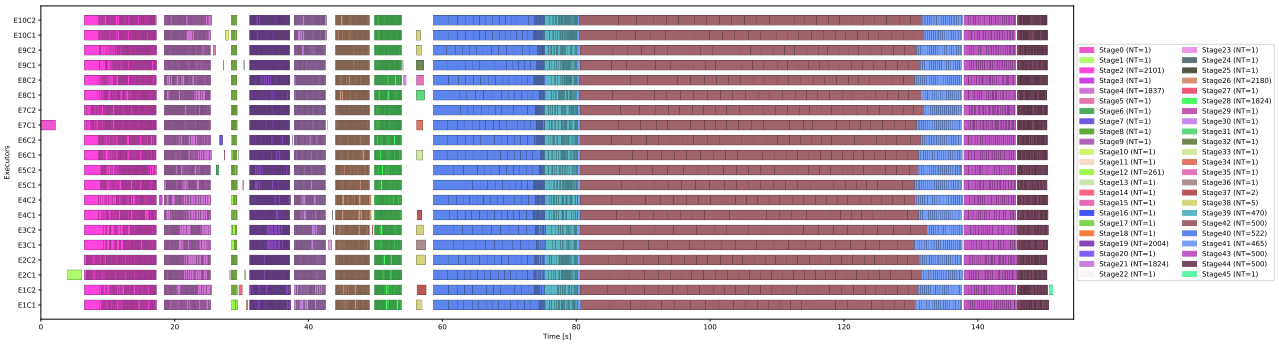


Figure 6: TPD-DS q72 execution on 10 executors (2 cores each) with Spark's default FIFO policy. Stages are identified with different colors, tasks are delimited by vertical bars inside each stage, and white spaces identify time-windows in which executors are free.

Among others, works like [14] tackles this problem at per job-level by presenting Decima, a system using Reinforcement Learning (RL) and neural networks to learn workload-specific scheduling algorithms without any human instruction beyond the high-level objective of minimizing average job completion time. In particular, it uses existing monitoring information and past workload logs to learn sophisticated scheduling policies automatically. Even though Decima outperforms existing heuristics, reducing the average job completion time of TPC-H [15] query mixes by at least 21%, it does not represent a solution for production environments. More precisely, Decima requires a large number of offline simulated experiments to train its RL algorithms. Recent work applies advanced machine learning to schedule workflow tasks but does not fully leverage the resource elasticity in serverless, and the lengthy training time for these approaches limits their practicality in real-world cloud settings.

Since the last deliverable, we have made progress and have presented Dexter, a resource allocation manager, leveraging serverless computing elasticity that continuously monitors the execution of applications, dynamically allocating resources at a fine-grained level to guarantee performance-cost efficiency (optimizing total runtime cost). Dexter is novel in combining predictive and reactive strategies that fully exploit the elasticity of serverless computing to enhance the performance-cost efficiency for workflow executions. Unlike black-box ML models, Dexter quickly reaches a sufficiently good solution, prioritizing simplicity, generality, and ease of understanding. Our experimental evaluation shows that Dexter achieves benefits in terms of both cost, performance-cost efficiency, while saving a significant amount of resources. Furthermore, Dexter represent a robust solution since it

is capable to react to new unseen workloads. Dexter details are under paper review and will be presented in the next deliverable.

4.2 Real-Time Data-Streams Smart Management

We have built a Proof of Concept (PoC) that enables NCT to ingest reliably video streams and perform real-time AI video inference for computed-assisted surgery workloads (see deliverable D2.3). Given that, in this section we focus on a specific problem related to adapting the streaming infrastructure to workload fluctuations that are present in the NCT surgery room utilization traces. In particular, the main problem we address is how to make a Pravega [16] cluster auto-scalable, while minimizing the number of instance auto-scaling events, as they may induce high tail latency that greatly impacts video analytics users.

4.2.1 Problem Statement: The Hidden Latency Cost of Streaming Auto-scaling

In the context of distributed systems, *auto-scaling* or *elasticity* is the property that allows a system to adapt to fluctuating workloads. In many scenarios, including the Cloud and the Edge, elasticity is a vital property required in systems and services. For instance, a Cloud service may grow in popularity quite rapidly, thus requiring the underlying system to increase the number of instances supporting the service to cope with the incoming workload burst. As another example, services at the Edge may need to auto-scale to adapt their capacity for the peak hours of utilization and downscale rapidly to free up resources, as the Edge is typically resource limited. These examples give a sense of why auto-scaling is important in distributed systems across the Cloud-Edge Continuum.

A common approach to system auto-scaling is the reactive one [17]. This approach to auto-scaling is based on building a control loop that captures performance metrics across the system (*i.e.*, latency, throughput) and automatically changes the number of system instances according to some expected performance thresholds. This approach is relatively simple to implement and to operate, as the administrator only needs to define some service level objectives (SLO) to be respected so the system can auto-scale accordingly. Still, a problem of using this approach is that the performance thresholds are not enforced immediately, but in some cases the system may wait for some time to verify that the SLO violation is persistent before taking the auto-scaling decision. The shorter the time is before taking an auto-scaling decision (*i.e.*, *more reactive*), the fewer SLO violations the user may experience.

However, in some cases, there may be a penalty in changing the number of instances running a service on the fly. Naturally, the type of penalty may differ depending on the system at hand. Systems like object stores may require rebalancing objects across storage nodes when increasing or decreasing the number of instances. This rebalancing activity may have associated background traffic for moving or replicating data objects, which could impact the performance of ongoing IO. In this study, we focus on the impact of auto-scaling in streaming systems, which are generally latency sensitive [18, 19, 20, 21, 22]. This is key to satisfy the latency requirements for running Edge analytics workloads (*e.g.*, AI video inference).

In the context of streaming systems, a too reactive algorithm may also represent a performance problem to the system, especially when it comes to tail latency. To wit, it is not uncommon that there is a transient performance cost related to auto-scaling the number of system instances. For example, in Figure 7 we present a real experiment on AWS of the impact of auto-scaling on the IO latency of a Pravega streaming storage system. In this experiment, we started a benchmark tool writing and reading events from Pravega at a rate of 200 events/second. As visible, every time we changed the number of instances, we observed much higher latency values ($> 100x$). This is because the Pravega client may need to reconnect to the new service instances, and such re-connection process induces a latency much higher than normal. Therefore, a too reactive auto-scaling approach may induce many of such high-latency events under a fluctuating workload, which may greatly impact the service for latency-sensitive applications.

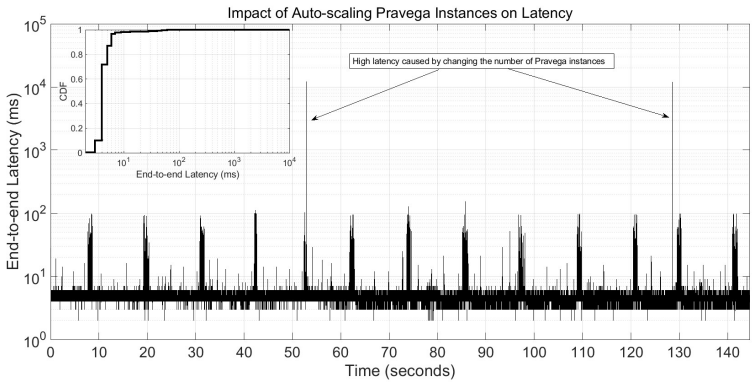


Figure 7: Latency impact of Pravega changing the number of Segment Stores in Pravega during an IO streaming workload.

4.2.2 Analysis of NCT Edge Video Analytics Workload Traces

The need for auto-scaling is evident in the use-case we target in this study: the National Center for Tumor Disease (NCT, Germany). This institution mixes data scientists and surgeons to apply data analytics techniques on surgery-related multimedia. In a nutshell, the use cases of NCT requires video data from surgery cameras to be durably ingested and processed in real time via specialized AI inference models to help surgeons during the procedure. Moreover, video data should be durably stored in long-term storage, so it can be accessed in batch analytics jobs (*e.g.*, AI model training). We addressed this use case by setting up a PoC using Pravega and GStreamer to manage video data for NCT AI jobs (see deliverable D2.3).

However, the utilization patterns of surgery rooms may influence the workload to be handled by Pravega and the resource requirements for AI inference jobs. To this end, we have worked with NCT to get a trace that describes the utilization of surgery rooms that may potentially require video analytics services during surgeries. As a result, Figure 8 shows the complete 2-month trace that NCT collected and anonymized for us. By a simple inspection of the trace, we can observe strong daily patterns in the utilization of the available surgery rooms (10). Even more, weekends tend to exhibit lower utilization compared to weekdays. These kinds of patterns are inherent to human activity and have been observed in many other scenarios [23].

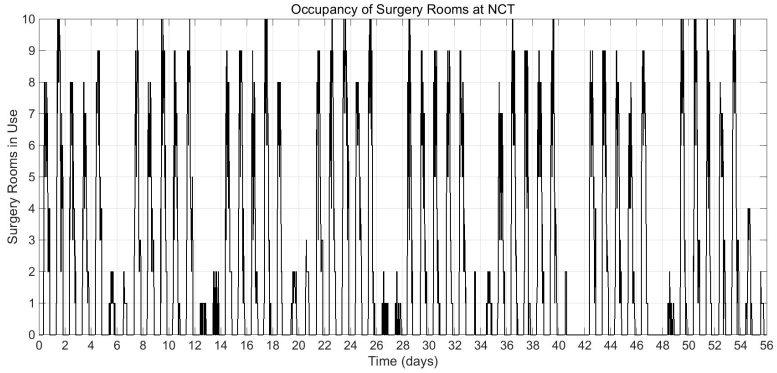


Figure 8: Surgery room occupancy at NCT (complete traces).

To investigate the NCT trace with more detail, Figure 9 shows a one-week period of it. As expected, the trace shows that the utilization of the surgery rooms occurs during the central hours of the day (*e.g.*, 7 : 00 to 18 : 00), which aligns with the normal working schedule of non-emergency surgeries. Moreover, the peak utilization of surgery rooms during weekends is 20%-25% of a weekday's

utilization.

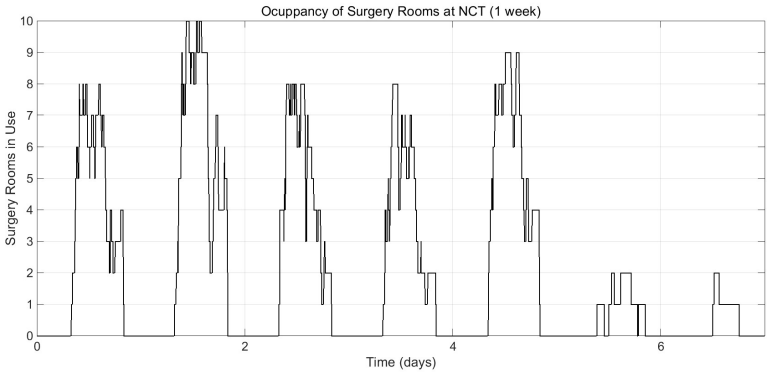


Figure 9: Surgery room occupancy at NCT (1 week).

Given the characteristics of NCT's surgery room occupancy traces, an Edge/Cloud infrastructure running a streaming workload following such patterns may benefit from auto-scaling, either in terms of resource savings or performance improvements. However, as mentioned in the problem statement, using a pure reactive auto-scaling approach for the streaming infrastructure may lead to additional latency spikes due to frequent instance scaling events derived from the observed strong workload patterns at NCT. For this reason, we would like to explore a "predictive" approach to auto-scaling the streaming infrastructure, so we can adapt the streaming infrastructure in advance while minimizing the scaling events. This would reduce high tail latency, which is critical in a computer-assisted surgery use case.

4.2.3 Predicting workload patterns in NCT

In this section, we analyze the NCT trace and explore AI/ML techniques to forecast workload in the near future. We are interested in this avenue of research given the visible workload correlations present in the NCT trace. Effectively exploiting workload prediction may not only be beneficial for predictively auto-scaling streaming services, but other types of infrastructure in the Cloud Edge continuum.

In this sense, to be able to predictively auto-scaling streaming service instances we need to resort to prediction models that can capture seasonality in a time series [24]. One of these models we are interested in is the Long-Short Term Memory (LSTM) model. LSTM is a type of artificial recurrent neural network (RNN) architecture used in the field of deep learning [25]. Unlike standard feed-forward neural networks, LSTM has feedback connections, making it a "recurrent" network. It is well-suited for tasks that require the network to learn from data that spans over long sequences and where maintaining context over time is crucial. LSTMs are specifically designed to overcome the limitations of traditional RNNs, particularly the problem of "vanishing gradients", which makes it difficult for RNNs to learn long-range dependencies in sequential data. LSTMs can maintain and learn from data over long periods, making them ideal for tasks such as time series forecasting, natural language processing (NLP), speech recognition, and more.

We proceeded by using the NCT trace as the input for an LSTM model. To this end, we have trained the LSTM model with 1-week worth of data and evaluated its effectiveness on the remaining 7 weeks of trace. The procedure to do so involves the normalization of the values and the configuration of the LSTM neural network (*i.e.*, 2 hidden layers, 2 feed forward layers, 10-15 neurons per layer). It is worth noting that for this piece of work, we have not spent a huge amount of effort in evaluating the accuracy of the LSTM model based on different parameters for the NCT trace, as that exercise alone would be worth a dedicated study. Still, even with some basic configuration, Figure 10 shows that the forecast of LSTM is quite close with the actual trace, even with a 1-week training period.

To inspect the actual accuracy of the LSTM prediction, Figure 11 shows the absolute errors of the

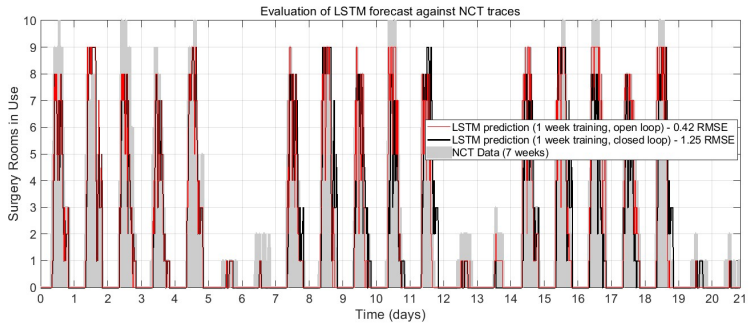


Figure 10: Evaluation of the LSTM model on the NCT trace.

LSTM forecast compared to the actual traces. We observe a significant difference in accuracy when using a closed and open loop version of LSTM. In an open loop, also known as "teacher forcing", during training, the model is provided with the true previous output (from the training data) as input for the next time step, rather than using its own previous predictions. In a closed loop, also known as "free running" mode, during both training and inference, the model uses its own predictions as inputs for the next time step.

As can be observed, the errors in the closed loop version of LSTM are significantly larger compared to the open loop version, especially regarding extreme error values. This is not surprising, as the closed loop model does not fix predictions based on actual real measurements, yielding a potential accumulation of errors. Conversely, most of the errors for the open loop version of the LSTM model range within 1-2 units. This implies that, when the model misses, it forecasts 1 to 2 surgery rooms more to be occupied/free compared to the actual trace.

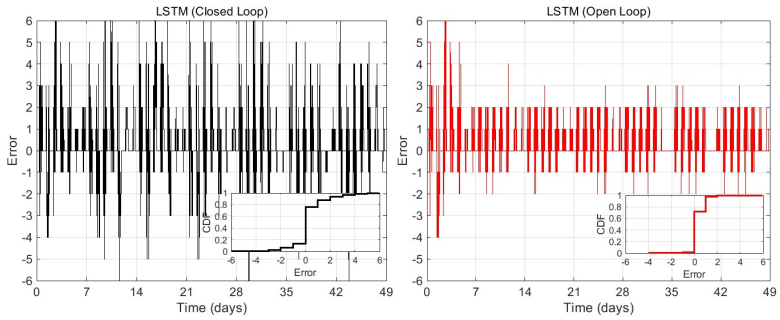


Figure 11: Errors from the LSTM forecast compared to the actual NCT trace.

Overall, the observed range of errors seems tolerable for being exploited in a predictive auto-scaling solution for streaming analytics [26, 27]. Therefore, we could exploit this model for auto-scaling the streaming infrastructure predictively, so we minimize both periods of time suffering from under-provisioning number of auto-scaling events. In principle, this should reduce tail latency in streaming video analytics.

4.2.4 Predictive auto-scaling of streaming service instance for video analytics

Next, we explore the application of the LSTM model in the auto-scaling of the streaming infrastructure (*i.e.*, Pravega). To this end, we design a simple algorithm that relies on the LSTM predictions and abides by the following two observations:

- *Prioritize latency over resource usage*: We are targeting latency-sensitive sensitive streaming applications, like video analytics. For this reason, we prioritize in our algorithm to meet latency requirements over minimizing resource usage (*e.g.*, amount of instance execution time). Re-

call that, in terms of latency, we pursue minimizing tail latency events related to auto-scaling, which could be quite disruptive when running video analytics applications.

- *Coarse grained prediction windows*: In the previous section, we have observed that LSTM may have some degree of inaccuracy comparing prediction to the original trace. But, at the same time, we also noticed that workload patters are structurally quite consistent. This leads us to exploit coarse grained prediction time windows of the near future to make auto-scaling decisions. Moreover, we also realize that the wider the time window for predictions we use, the fewer auto-scaling events we may induce.

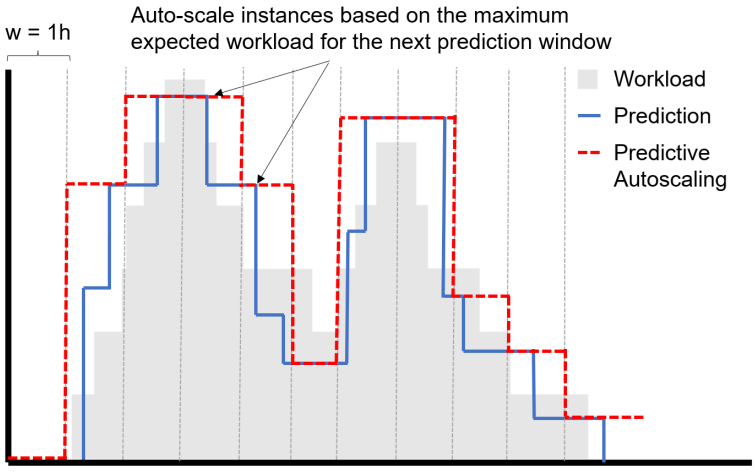


Figure 12: Predictive auto-scaling algorithm based on LSTM forecasts.

With these observations in mind, we use a predictive auto-scaling algorithm as shown in Figure 12. Our predictive algorithm is based on LSTM forecast traces about the near-term workload. The algorithm works as follows. First, the algorithm is expected to satisfy a latency SLO goal (*e.g.*, latency at p95 under 20ms). This goal is expected to be always satisfied, irrespective of the auto-scaling events. Second, it also establishes a time window w. The size of the time window refers to the period of time in the near future that the algorithm will consider from the LSTM prediction. Based on that, the algorithm will pick the maximum expected workload within w. In the NCT trace, the workload may be described as number of ongoing surgeries using video stream analytics. The algorithm also assumes some modeling or performance-related information about the latency of a system instance under parallel streams. Based on such performance information, the algorithm looks for the number of streaming system instances that satisfy the required latency SLO assuming the maximum predicted workload within w. Once the system gets to the next time window, the algorithm runs again. As can be noticed, the algorithm gives priority to meeting latency requirements to minimizing resource usage. Moreover, with a sufficiently large prediction time window, the algorithm is expected to greatly reduce the number of auto-scaling events inducing high tail latency.

4.2.5 Simulation-based analysis of streaming infrastructure auto-scaling

Next, we validate the exploitation of a predictive approach for auto-scaling stream analytics via simulation. To this end, we have developed a trace-based simulator based on real performance measurement data. First, we characterized the end-to-end latency of a Pravega Segment Store instance handling video streams. This characterization has been done by deploying a Pravega cluster on AWS EKS (4 i3.2xlarge nodes with local NVMe drives). The Pravega cluster consisted of 1 Controller instance, 1 Segment Store, and 1 Bookkeeper instance (using a dedicated NVMe for the ledger and journal volumes). The main Pravega instances were running in 2 Kubernetes nodes, and we prevented Kubernetes from scheduling any further pods in these nodes to avoid performance interferences. Given that, we deployed pairs of benchmark pods writing and reading video streams in

the two remaining Kubernetes nodes (1240x780, 30 FPS per video stream). The performance metric we are interested in is the "end-to-end latency". To measure this metric, we modified the GStreamer Pravega connector to attach the current timestamp as metadata when writing every video frame. Then, the video reader obtains the "write time" of each frame and calculates the delta with its local time when reading a video frame. This measurement is accurate given that the Kubernetes cluster uses a time synchronization service. As visible in Figure 13, this methodology allows us to have an accurate measurement of the end-to-end latency for video streams depending on the number of video writer and reader pairs.

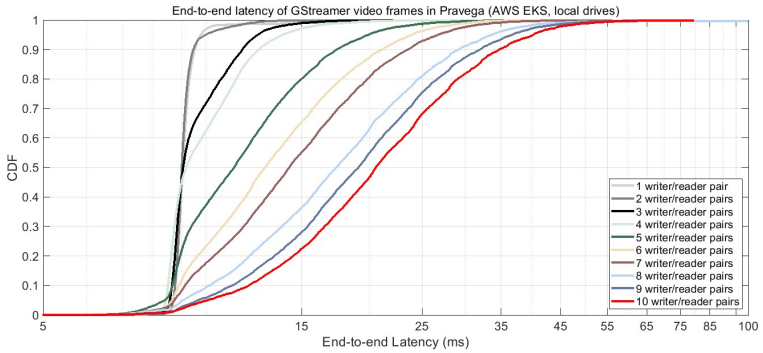


Figure 13: Measured end-to-end latency of GStreamer video frames in Pravega (AWS EKS deployment with local drives). The cluster was formed by 1 Pravega Segment Store, 1 Pravega Controller, and 1 Bookkeeper instance. Each video stream is made of 1280x740 frames and 30 FPS.

We use the latency distribution values in Figure 13 to feed our simulator. That is, depending on the number of video streams we expect on a surgery and the number of ongoing surgeries at any given time, we could estimate the end-to-end latency of video frames based on the available Pravega instances. To wit, if in a given moment there are 8 video streams according to the NCT trace and we have 2 Pravega instances at that time, the IO latency the simulator will report would be based on extracting values from the empirical distribution of 4 writer/reader pairs in Figure 13. For simplicity, we assume an even distributions of video streams across instances.

In Figure 7, we showed that auto-scaling Pravega instances may induce latency spikes. To capture this behavior, our simulator allows us to define a latency penalty for every instance auto-scaling event (*e.g.*, standard distribution with $\mu = 100$ and $\sigma = 10$). The number of high latency frames impacted will be proportional to the extent of the auto-scaling event. For instance, if there are 2 Pravega instances handling 4 video streams each and the system decides to auto-scale to 4 Pravega instances, half of the video streams will be potentially reallocated to the new instances and, therefore, will likely suffer from a latency penalty.

With such simulator functionality in place, we next overview the auto-scaling methods to be evaluated. In all cases, the algorithms target to achieve an end-to-end latency SLO (*e.g.*, 20ms at percentile 95). Concretely, to help the algorithms to being less reactive, we provide a latency SLO higher and lower tolerance bounds, defined as a percentage of the main SLO value (*e.g.*, 10%). Moreover, the goal of the proposed predictive algorithms is to achieve the same as well as to minimize the number of instances auto-scaling events that may induce high latency spikes. The algorithms evaluated are:

- *Reactive auto-scaling vanilla*: This method embodies the most common approach to auto-scale distributed systems. There is a feedback loop that takes performance metrics as input and reacts to the ongoing workload by scaling up or down the number of service instances. In our case, this algorithm takes the last m minutes as the "time window" to compute the end-to-end latency for all the streams in the system. If the current end-to-end latency of video streams is over the SLO higher bounds (*e.g.*, 22ms at percentile 95), the algorithm scales up the number of

Pravega instances. Similarly, the algorithm may scale down the number of Pravega instances if the current latency is below the lower SLO bound in the last time window.

- *Reactive auto-scaling with memory*: The most basic version of the reactive algorithm may lead to situations of instability. That is, it may detect that latency is below the threshold for the current window period and then decide to downscale the number of Pravega instances. However, it may be the case that fewer Pravega instances may lead to an end-to-end latency over the higher latency SLO bound. As the algorithm does not have memory, it may be scaling up and down the number of instances continuously, which is undesirable. To mitigate this problem, we evaluate a version of the reactive auto-scaling algorithm with memory. This means that the algorithm will record the number of writer and reader pairs in the system in the previous scaling event, which is used to prevent downscaling the system if that would lead to violating the latency SLO again.

- *Predictive Oracle auto-scaling*: In this study, we propose to exploit recurrent workload patterns to minimize the number of auto-scaling events while meeting the expected streaming system latency. To have a reference for comparison, we use an "oracle". The oracle will use the predictive algorithm described in the previous section (see Figure 12) but using as prediction the actual NCT trace. Therefore, the "oracle" provides us an upper bound of the effectiveness of the proposed algorithm when using a perfect prediction.

- *Predictive LSTM auto-scaling*: In this case, the predictive algorithm uses LSTM forecasts based on a 1-week period of the NCT trace. Note that we execute our experiments starting after the training period, which should give a sense on the accuracy of the predictions.

Experiment 1: Evaluating the auto-scaling behavior of the proposed algorithms. The goal of this experiment is to evaluate the number of auto-scaling events that induce high tail latency for the proposed algorithms. To this end, Figure 14 shows a time-series view of the Pravega instance auto-scaling events for the proposed algorithms. Visibly, the Reactive "vanilla" auto-scaling shows a significant degree of instability. To wit, the algorithm reacts only based on the last window (5 minutes) of latency observations without considering past auto-scaling events (*i.e.*, memoryless). This leads the system into situations in which the number of instances goes up and down, given than a certain number of Pravega instances meet the latency SLO for the workload at hand, but reducing them makes the system to miss the latency SLO. In this sense, we observe that providing this algorithm with memory about past auto-scaling events and the latency SLO achieved at that moment greatly prevents such situations of instability. As visible, the Reactive "memory" algorithm performs much better than the "vanilla" one by reducing the number of auto-scaling events. On the other hand, the Predictive approaches ("oracle" and "LSTM") can perform even better than the reactive ones when it comes to minimize the number of auto-scaling events.

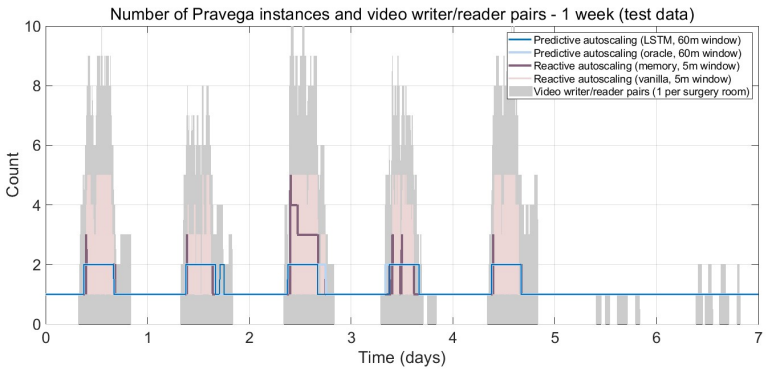


Figure 14: Time-series view of Pravega instance auto-scaling events for the proposed algorithms.

To inspect the number of instance auto-scaling events more in detail, we suggest to look at Figure 15. In Figure 15, we observe that the Reactive "vanilla" auto-scaling method exhibits 1310 auto-scaling events during 1 week of simulation of the NCT trace. By adding memory about past auto-scaling events (Reactive "memory"), we have been able to reduce this number in almost $30x$ (44 instance auto-scaling events). Still, we can confirm from this figure that the predictive approaches can drastically reduce instance auto-scaling compared to reactive ones. For instance, the Predictive "LSTM" method reduces instance auto-scaling events in $3.6X$ compared to Reactive "memory" (12 instance auto-scaling events). Interestingly, the Reactive "LSTM" only incurred 2 instance auto-scaling events more than the Predictive "oracle", which is the ideal case in which the prediction the same as the experienced workload.

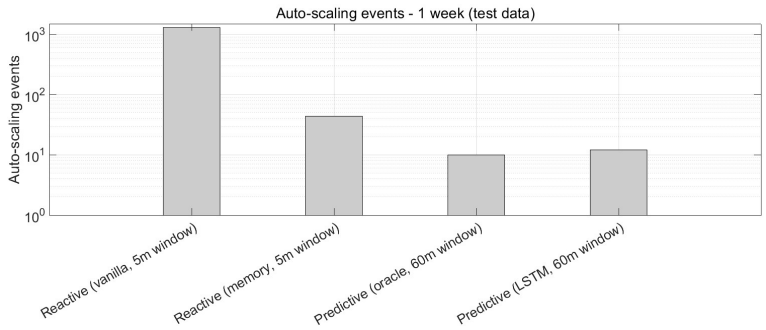


Figure 15: Number of Pravega instance auto-scaling events depending on the algorithm at hand.

The goal of minimizing the number of instance auto-scaling is to reduce long tail-latency values, which could impact real-time video analytics applications. This is illustrated in Figure 16. In our simulations, we induce a high latency event (*i.e.*, standard distribution with $\mu = 100$ and $\sigma = 10$) for all the writers impacted by an instance auto-scaling event. Figure 16 clearly shows these high latency events as part of the end-to-end latency distribution tail. Visibly, the predictive methods have a much "smaller" tail, given that the latency percentiles impacted with auto-scaling events are p99.89 and p99.91 for "LSTM" and "oracle", respectively. Conversely, reactive algorithms are impacted much more heavily by high-latency auto-scaling events: p87 and p99.56 for Reactive "vanilla" and Reactive "memory", respectively.

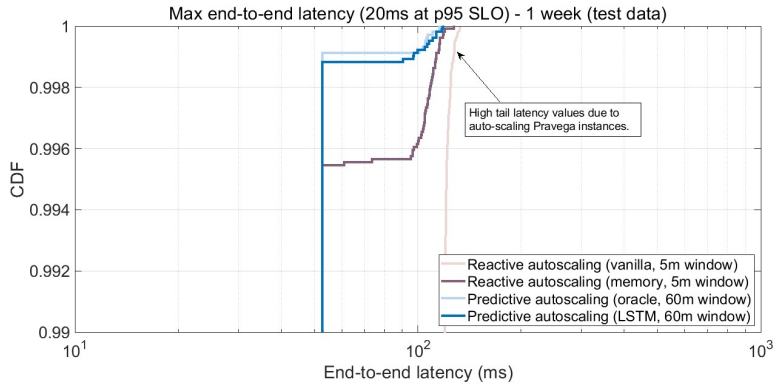


Figure 16: High tail-latency values related to instance auto-scaling events for the proposed algorithms.

Conclusion: Our predictive auto-scaling algorithm for streaming systems can greatly reduce the number of instance auto-scaling events, and therefore, reduce tail latency associated to autos-scaling.

Experiment 2: Analysis of the end-to-end latency SLO enforcement. In this experiment, our goal is to analyze if the end-to-end latency SLO is enforced by the proposed algorithms. In this sense, Figure 17 provides a time-series view of the end-to-end latency for 1 week of NCT trace simulation. Note that all the algorithms have a tolerance threshold (10%) relative to the end-to-end latency SLO defined (p95 end-to-end latency under 20ms) to make them less sensitive to workload changes. Such tolerance bounds are depicted as back lines in Figure 17. Visibly, the highest rate of SLO latency violations is related to Reactive "vanilla" method. There are at least two reasons for this behavior: i) the sudden changes in workload may induce SLO violation during the observation window (5 minutes), and ii) the lack of memory in the algorithm may induce to mistakenly downscaling instances, which would induce additional misses in the latency SLO. Similarly to the evaluation of the number of instance auto-scaling events, adding memory to the reactive algorithm can greatly improve its stability, which translates into meeting SLO more frequently (Reactive "memory"). At first glance, the Reactive "memory" method seems to achieve the latency SLO in a similar way to the predictive ones.

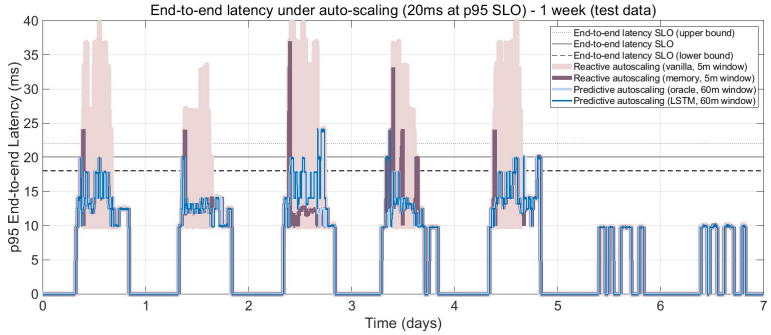


Figure 17: Time-series representation of the p95 end-to-end latency during 1-week of NCT trace.

To inspect end-to-end SLO violations more in depth, we recommend looking at Figure 18. Visibly, the Reactive "vanilla" approach is by far the worst performing one, as it violates the (upper bound) SLO latency in 5.82% of the (1-minute) time slots. On the other hand, we observe that the Reactive "memory" algorithm achieves a slightly lower (upper bound) SLO violation rate (0.36%) compared to the Predictive "LSTM" method (0.62%). The main reason for this is that, if the LSTM prediction is significantly lower compared to the actual workload, the number of provisioned Pravega instances will not meet the expected latency SLO. This is supported by the fact that the Predictive "oracle" method achieved a 0% of end-to-end latency violations.

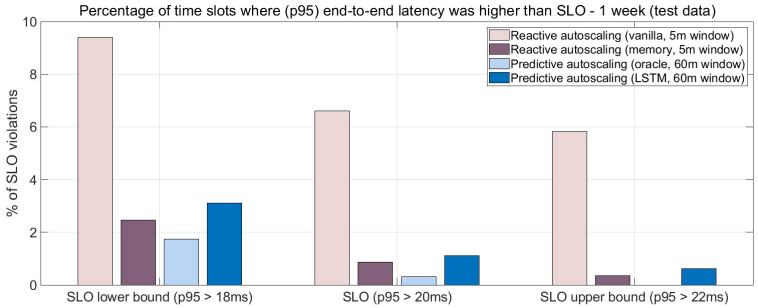


Figure 18: Rate of end-to-end latency SLO violations for the proposed algorithms.

Conclusion: The Predictive "LSTM" method achieves a slightly worse SLO violation rate compared to the Reactive "memory" method, as it may be misled if the prediction underestimates the ongoing workload. This may be improved by adding some fallback reactive behavior to take over in the case of a wrong workload prediction.

Experiment 3: Inspecting differences in resource usage. In this battery of experiments, we aim at quantifying the time of execution of Pravega instances during auto-scaling workload. Figure 19 illustrates the number of execution time (in minutes) of Pravega instances handling the same 1-week workload period from the NCT trace under different auto-scaling algorithms. Interesting, the Reactive "memory" algorithm induces an overall execution time of 2.79% higher than the Predictive "LSTM" (and 1.79% higher than Predictive "oracle"). This result may be related to some underestimation in workload forecasts, which also contributed to increase the number of SLO misses for Predictive "LSTM". But, overall, the Reactive approaches do not achieve any significant resource savings compared to the Predictive approaches.

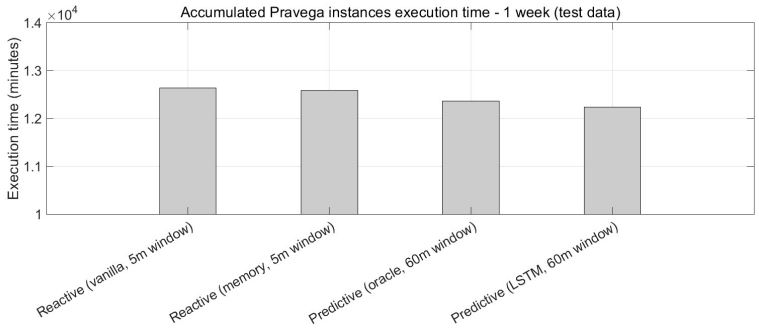


Figure 19: Execution time of Pravega instances for a 1-week period of the NCT and the proposed algorithms.

Conclusion: The Reactive approaches tested do not significantly save more instance execution time compared to Predictive approaches. This supports the usage of Predictive algorithms, given the reported improvements in tail latency.

Experiment 4: Accuracy and time-granularity in predictive auto-scaling. In this battery of experiments, we analyze the impact of the time window size in the Predictive algorithm. In Figure 20, we show the number of instance auto-scaling events depending on the prediction window size. As can be inferred, a larger time window helps to minimize the number of auto-scaling events. This seems natural, as a larger prediction time window reduces the frequency in which the auto-scaling algorithm may decide to change the number of instances in the system. Additionally, this makes the algorithm coarser grained and less sensitive to short-term deviations between the forecast and the actual workload. In this sense, we observe that a prediction window of 3 hours achieves the same result as the "oracle" for the NCT trace. This is interesting because a larger prediction window seems to make the prediction accuracy less critical to obtain good results.

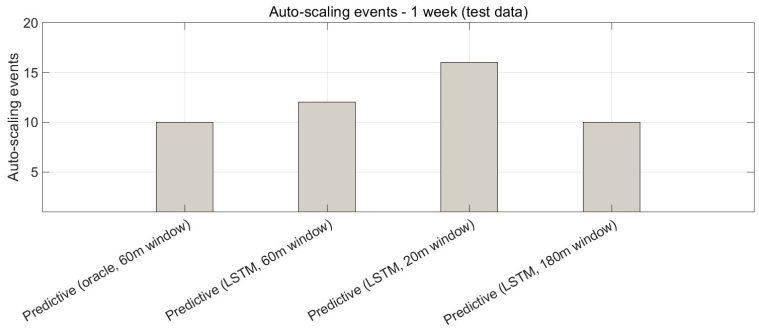


Figure 20: Number of Pravega instance auto-scaling events for different Predictive algorithm time windows.

As a direct consequence of the number of instance auto-scaling events, Figure 21 shows the tail

latency for the different prediction windows tested. Visibly, the tail latency related to instance auto-scaling is very similar for the Predictive "oracle" and Predictive "LSTM" with a prediction window of 3h. In this sense, we also observe that tail latency may get significantly impacted if we use a shorter prediction window (*e.g.*, 20 minutes).

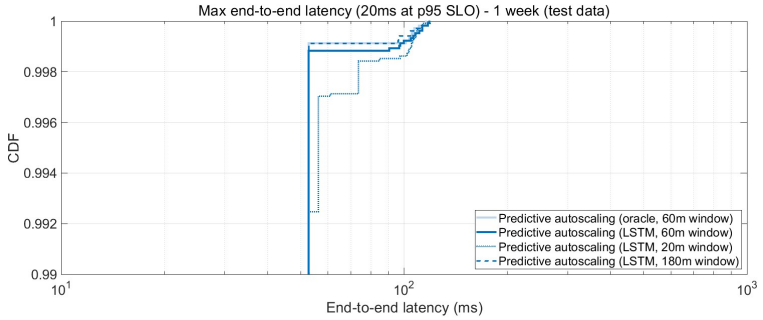


Figure 21: High tail-latency values related to instance auto-scaling events for different Predictive algorithm time windows.

Finally, we are interested in inspecting the number of end-to-end latency violations for the different prediction windows tested. In Figure 22, we also observe that latency SLO is also better preserved when using a larger prediction window. The reason for that is likely the fact that the algorithm picks the maximum workload for the prediction window and adapts the number of instances to it. With a larger prediction window, the system is likely to use a number of Pravega instance capable of handling the workload for the time window, even though if there are variations within the window itself.

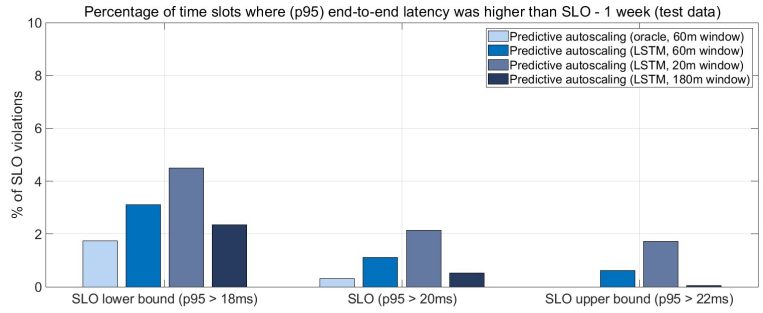


Figure 22: Rate of end-to-end latency SLO violations for different Predictive algorithm time windows.

Conclusion: Larger prediction window (*e.g.*, 3 hours) provides the best results compared to smaller ones. This is especially true if we focus on improving latency, which is the main goal of this study.

4.3 Modeling in Metabolomics Serverless Environments

METASPACE operates in production on an EC2 instance (r6a.2xlarge), processing datasets in 14 steps. After these steps, there is an extra step called off-sample, that runs classification of datasets using AWS ECS, a fully managed container orchestration service from AWS. AWS ECS runs within Docker containers, as detailed in D2.3.

Here we describe the approach for automatic workload extraction from the pre-project off-sample service, followed by a first characterization of the workload. Equipped with this data, we plan to train a number of AI models that can help to achieve the following two tasks:

1. **Mitigation of cold starts** that can cause severe performance degradation of the inference ser-

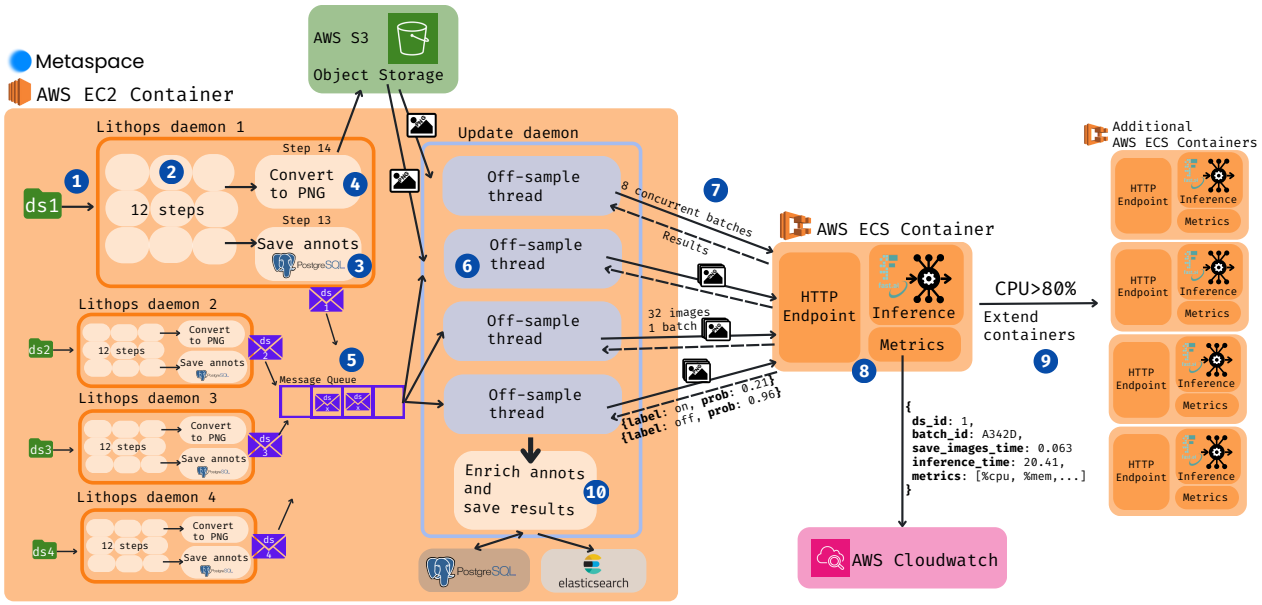


Figure 23: METASPACE pre-project off-sample architecture built upon AWS ECS.

vice; and

2. **Optimal resource provisioning** in order to find the optimal "sweet spot" between the public cloud and on-prem edge resources in this use case.

At M18, we have set the monitoring infrastructure to extract the workload characteristics of the METASPACE service in production and begun the investigation of the right AI models. We defer to the next deliverable the detailing of the AI models and the corresponding improvements of the KPIs, as this task is sill under intense investigation.

4.3.1 Workload Extraction

Here we describe the process to extract the workload information for model training. The extraction of this information comes from several sources and is a representative example of information fusion. As a first step, we describe how the pre-project off-sample in production works to better understand where information comes from.

As of today, several daemons operate within METASPACE, with two key daemons responsible for trace collection: four Python Lithops daemons and a control daemon called "`update`". In particular, the `update` daemon is the one in charge of running the off-sample service.

The execution flow is depicted in Figure 23 and follows the following steps:

1. The four Lithops daemons facilitates the simultaneous processing of four datasets, with each daemon processing a separate dataset.
2. The Lithops daemons manage the 14 steps and perform some inter-step calculations.
3. The output of this pipeline is a collection of annotated molecules that are saved to a PostgreSQL database.
4. The service generates a `.png` image for each annotated molecule. The image is archived in AWS S3 object storage.
5. Upon the completion of a dataset processing by a Lithops daemon, the Lithops daemon sends a message to a queue where the `update` daemon is subscribed.

6. The `update` daemon handles the off-sample processing of the incoming datasets. The input are the `.png` images generated by the prior step. These images are pushed to the AWS ECS service for off-sample classification. The `update` daemon is able to process up to 4 datasets at the same time, a dataset per off-sample thread.
7. To process datasets, the images are grouped into batches of 32 images. Batches are transmitted to an HTTP endpoint in the AWS ECS service. To not overload the containers, every off-sample thread manages up to 8 concurrent synchronous batches.
8. There is always a minimum of one AWS ECS container running the service, with a Falcon HTTP server that listens to all incoming batches. Each AWS ECS container includes a parallel process that sends resource usage values (e.g., CPU and RSS memory) once per second. These values, along with the dataset ID, batch ID, and the time taken to load each image and classify it, are logged by AWS CloudWatch for further analysis.
9. As described in deliverable 2.3, the auto-scaler can boot up additional containers if CPU usage exceeds a certain threshold.
10. The Lithops daemon previously stored the annotated molecules in PostgreSQL. The SQL database is normalized and the number of fields for each annotation is several dozens, resulting in very slow SQL queries. For this reason, the `update` daemon is in charge of retrieving the annotations, denormalizing the data and saving it to AWS Elasticsearch. Elasticsearch has a more optimized way of handling read requests, especially for full-text search. When off-sample of the entire dataset finishes, results are stored both in PostgreSQL and ElasticSearch.

From the daemons and AWS CloudWatch, the following set of workload files are produced:

- **YYYY-MM_datasets**: Information from PostgreSQL regarding dataset ID, image resolution (x, y), number of annotated molecules, number of generated images and public/private status as shown in Table 3.

ds_id	x	y	annots	imgs	is_public
small.1k	132	148	1890	7369	False
medium.3k	157	147	2582	5806	True

Table 3: Dataset general information.

- **YYYY-MM_dataset_start_finish**: includes the date and time when a notification is sent from the Lithops daemon to the update daemon, indicating the start of dataset processing as reported in Table 4.

ds_id	start	finish
small.1k	2023-03-01 01:29:15.221671	2023-03-01 01:32:28.484376
medium.3k	2023-03-01 01:50:29.554324	2023-03-01 01:53:07.631446

Table 4: Start and finish times for the datasets.

- **YYYY-MM_daemons**: This file is generated by the `update` daemon and contains information about the actual start and end times of processing through the off-sample service. An example of the file format can be found here:

```
2024-01-01 16:18:55,904 - INFO - update-daemon[Thread-1] - queue.py:532 - [v] Sent {"ds_id": "2023-12-20
    _03h38m48s", "action": "classify_off_sample", "stage": "STARTED"} to sm_dataset_status

2024-01-01 16:43:07,530 - INFO - update-daemon[Thread-1] - queue.py:532 - [v] Sent {"ds_id": "2023-12-20
    _03h38m48s", "action": "classify_off_sample", "stage": "FINISHED"} to sm_dataset_status
```

- **YYYY-MM_cloudwatch_logs**: These are a number of logs files from AWS CloudWatch. These files include log information from the AWS ECS containers. It includes info on the dataset ID, the batch ID, container ID (@logStream), number of images (default 32), and execution times and metrics. See Table 5 as an example.

@timestamp	@message	@logStream
2024-05-21 13:24:28.599	2024-05-21 13:24:28,598 - off-sample - INFO - Perf: {'**ds_id**': 'small.8k', '**batch_id**': '0AFE4699', '**n_images**': 32, '**start_ts**': 1716297848.116, '**deserialization_time**': 0.009, '**save_images_time**': 0.062, '**predict**': 20.41, '**metrics**': [{1716297848.494: {'**cpu**': [37.8, 33.3], '**memory**': 20.8, '**inf_rss_mb**': 224.62890625}}, ...], '**end_ts**': 1716297868.599}	`ecs/58c95beb`

Table 5: Log entry details.

4.3.2 Workload Characterization

The above files are then transformed and merged to facilitate further analysis and become useful for training a model. Concretely, we generate three type of files: datasets, batches and metrics. The concrete fields on each file, along with their description can be found in Table 6, Table 7, and Table 8, respectively.

Field	Description	Example
ds_id	Unique identifier of the dataset	medium.8k
ds_name	Name of the dataset	2023-skin-cancer
message_sent	Time at which the classification message is sent to the off-sample service	2024-01-02 10:15:20.443575
started	Time of start of dataset processing	2024-01-01 16:18:55,904
finished	Time of end of dataset processing	2024-01-01 16:43:07,530

Table 6: Dataset information.

Field	Description	Example
ds_id	Unique identifier of the dataset to which the batch belongs	medium.8k
batch_id	Unique identifier of the batch	9356b57d2db1
n_images	Number of images of the batch	32
container_id	Identifier of the container processing the batch	ecs/a8ec806db
start_ts	Start time of batch processing	1709248592.674
deserialization_time	Time taken to transform images from base 32 to PNG	0.003
save_images_time	Time taken dumping images to disk	0.004
predict	Time taken for inference	11.107
end_ts	End time of batch processing	1709248603.788

Table 7: Batch information.

Field	Description	Example
batch_id	Unique identifier of the batch to which the metric belongs	9356b57d2db1
timestamp	Local timestamp of the metric	1704189381.019
cpu[1...X]	List of usage percentage of each CPU	[51.5, 50.0]
memory	Percentage of memory used	21.5
inf_rss_mb	Memory RSS used	562.265625

Table 8: Metric information.

The collected data serves two primary objectives:

- **Comparison between old and new implementation**: in terms of latency, speedup, cost, as well as performance per dollar between both implementations.
- **Forecast spikes in activity**: to train an AI model to anticipate and adjust dynamically the pool of functions to mitigate cold starts and pre-allocate on-premises resources such as containers.

4.4 Modeling in Smart Agriculture Infrastructures

AI models play a crucial role in enhancing performance and optimizing resource usage within smart agriculture infrastructures.

Agricultural data sources vary in complexity and resource demands. Simple environmental data such as temperature, humidity or soil moisture require minimal processing power, while more complex data like geospatial images demand significant computational resources.

These models need to leverage both cloud and edge resources, incorporating containers, virtual machines (VMs), or serverless functions in the cloud; and containers or C-Cells at the edge.

The models aim to achieve two primary objectives:

- **Effectively distribute workloads**: The system must dynamically scale resources up or down based on current workloads, redistributing tasks to edge resources or reallocating them to the cloud as needed.
- **Predict future workloads**: By anticipating future demands, the system can provision resources in advance to ensure optimal performance. For example, lower activity levels at night may allow edge cells to process datasets at a slower pace, while higher workloads or latency-sensitive tasks during the day can be managed by more powerful cloud-grade machines.

There are different types of key metrics to be extracted, with different levels of granularity and relation:

- **General metrics about the dataspace**: this includes number of requests, users logged in, dataset providers and locations.
- **Metrics from each dataset**: format, size, type of processing tasks (e.g., sensor data processing, NDVI index calculation, radiation calculation, geospacial image analysis etc.) and processing time.
- **Metrics from each resource**: resource consumption patterns related to CPU, memory, disk, and network usage. These metrics may be facilitated by KIO Networks platform.
- **Dataset-resouce assignation**: it is crucial to maintain a registry that tracks which datasets were processed by which resources. Some datasets may be divided into chunks to be processed by multiple resources, maximizing the usage of idle resources.

By analyzing these metrics, we can establish patterns linking dataset processing requirements and resource needs, aiming to minimize latency and maximize resource efficiency.

Prometheus can be employed to save metrics from various agricultural data sources. It supports a wide range of exporters to gather metrics from different types of resources such as sensors, edge devices, and cloud services. Furthermore, Grafana can be used to create comprehensive dashboards to visualize the metrics, displaying real-time data, historical trends, and predictive insights. The use of Prometheus will facilitate integration with the Learning Plance.

To enhance performance and optimize the dataspace design, we have studied two analysis models:

- **Time Series Model for Performance Monitoring**: This model is ideal for analyzing resource consumption patterns and predicting future resource needs by examining their variations. It helps identify trends, demand peaks, and potential bottlenecks, enabling proactive resource management.
- **Regression Analysis Model**: This type of model is capable of predicting resource consumption by establishing relationships between various performance metrics and resource usage. It provides insights into how changes in the number of requests impact CPU and memory usage, which is valuable for anticipating the effects of future design or workload changes.

Other modeling methods studied, such as container-based queuing models, machine learning, and deep learning for automatic optimization, were initially discarded because their complexity exceeds the scope of the experiment.

As of now, the best model candidate is the Long-Short Term Memory (LSTM), due to the greater data generation capacity offered by the KIO virtual platform. LTSM will has also been tested in other use cases and can also be highly effective in the context of smart agriculture for several reasons:

- **Predictive Accuracy**: LSTM networks are capable of matching complex relations, which improves their predictive accuracy.
- **Handling Sequential Data**: Agricultural data often exhibit temporal patterns due to seasonal variations. LSTM models are designed to handle sequential data and can learn patterns over long time periods.
- **Proactive Resource Management**: By predicting future workloads and resource requirements, LSTM models enable proactive management of resources. For example, if a surge in data processing is anticipated during harvest season, additional resources can be allocated in advance to handle the increased load.
- **Adaptability**: The agricultural environment is dynamic, with conditions changing due to weather, soil health, and crop cycles. LSTM models can adapt to these changes by continuously learning from new data, thus providing more reliable predictions.

The focus will be in the development of a LSTM model, leaving the regression analysis (or other candidate models) for a later phase of the project, if data is representative enough to be able to make realistic predictions.

5 Instrumentation and Data Retrieval in the Learning Plane

All the learning methods for decision making and resource management rely on retrieving data from the different environments, including resource usage, QoS of applications, data usage and transmission, etc. This section introduces and advances the early efforts in Task T5.3 on instrumentation towards data retrieval for modelling and prediction for smart management. Each subsection corresponds to one use case in T5.4-T5.7.

5.1 Cloud-Edge Instrumentation using Kubernetes

The learning plane implemented for Kubernetes based cloud-edge application orchestration considers using the telemetry from different levels. For instance:

- **Infrastructure level:** From the infrastructure level, the learning plane can retrieve the system and application resource usage from Prometheus, such as CPU, memory etc., Also, based on the availability of the energy application and understand of the energy data, in the following months we will enable the learning plane to take as well the energy consumption data of each system.
- **Application level:** From the application level, the learning plane can monitor the application stream QoS, such as fps, elapsed_time and pipeline_latency.

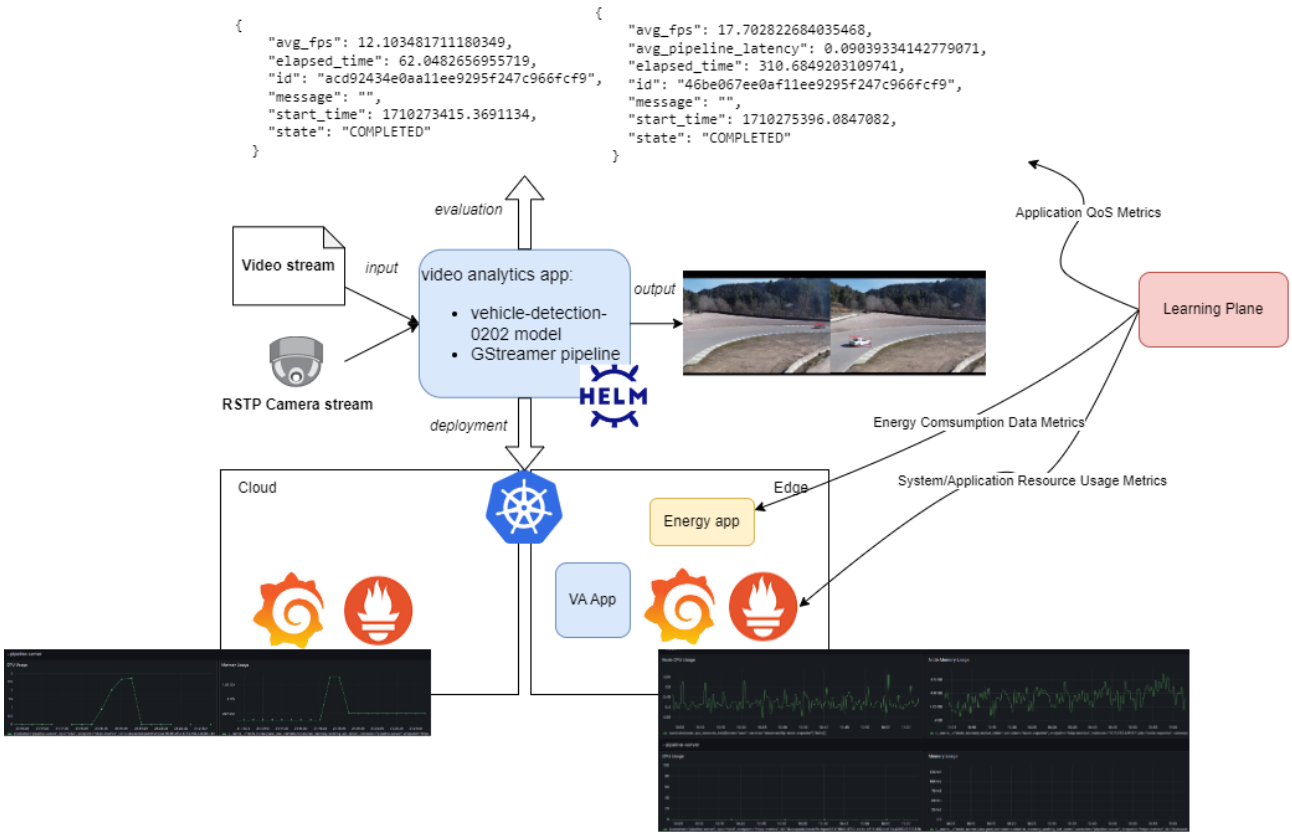


Figure 24: Telemetry for Learning Plane.

Currently, these data are extracted through CNX infrastructure and NBC observability stack (Described in D2.3) for the mobility use case. The required ETL pipeline comprises the following components:

- **Prometheus JSON exporter**[1]**:** this exporter is responsible of formatting the JSON objects that

[1]https://github.com/prometheus-community/json_exporter

the DLStreamer pipeline server application returns when querying its pipelines status API endpoint to Prometheus-compatible metrics and labels. When deployed with the Kubernetes Service Monitor CDR enabled, the Prometheus operator stack available in the same cluster discovers it and configures a scrape target to the pipeline server, retrieving new metric values on predefined scraping intervals

- **Prometheus SNMP exporter**[2]**:** same as with the JSON exporter, this component is responsible of walking through the provided OIDs of one or more scrape targets and formatting its content into Prometheus-compatible metrics and labels. This exporter is used by the Prometheus operator stack to periodically retrieve the energy data from Cellnex Orion platforms
- **PromCSV library**[3]**:** a custom-made Python library that streamlines the process of querying multiple Prometheus metrics via PromQL syntax and merging the results into a single Pandas DataFrame or exporting it into a CSV file. This library relies on the *promql-http-api*[4] library for the heavy-lifting task of converting the Prometheus API response data to Pandas DataFrames, and the *maya*[5] library for dealing with natural datetimes. A Docker image is also available for scheduled data export from Prometheus databases, useful when creating training datasets for the Learning Plane
- **Grafana dashboards:** custom dashboards have been created to properly visualize and monitor the extracted application data and cluster resources, only list a few in Figure 25:

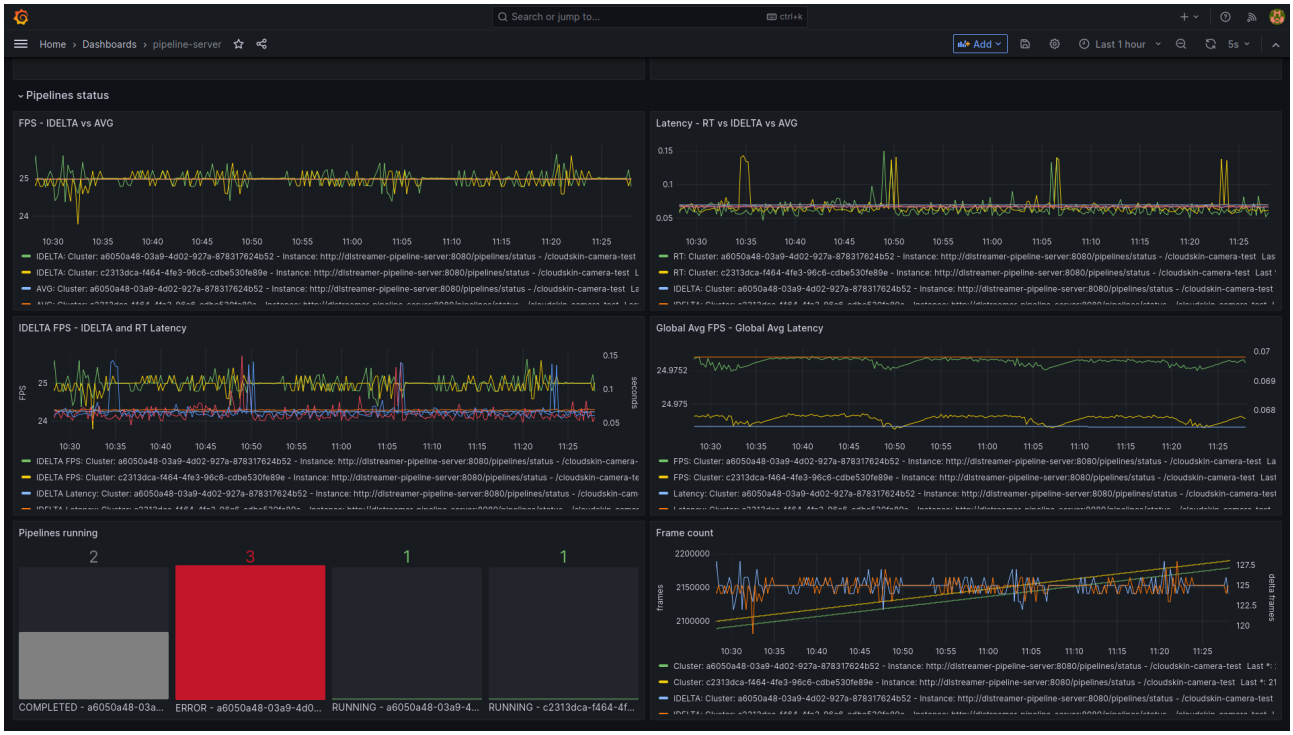


Figure 25: Grafana - Pipeline panels.

5.2 Serverless Instrumentation using Lithops

Lithops Serve provides a comprehensive trace collection system, providing insights at various levels of operation: executors, batches and images. Each dataset processed through Lithops Serve not only

[2]https://github.com/prometheus/snmp_exporter
[3]https://gitlab.bsc.es/datacentric-computing/cloudskin-project/cloudskin-learning-plane/-/tree/main/libs/promcsv
[4]https://github.com/nir-arad/promql-http-api
[5]https://github.com/kennethreitz/maya

yields classification results but also captures detailed traces of its execution process.

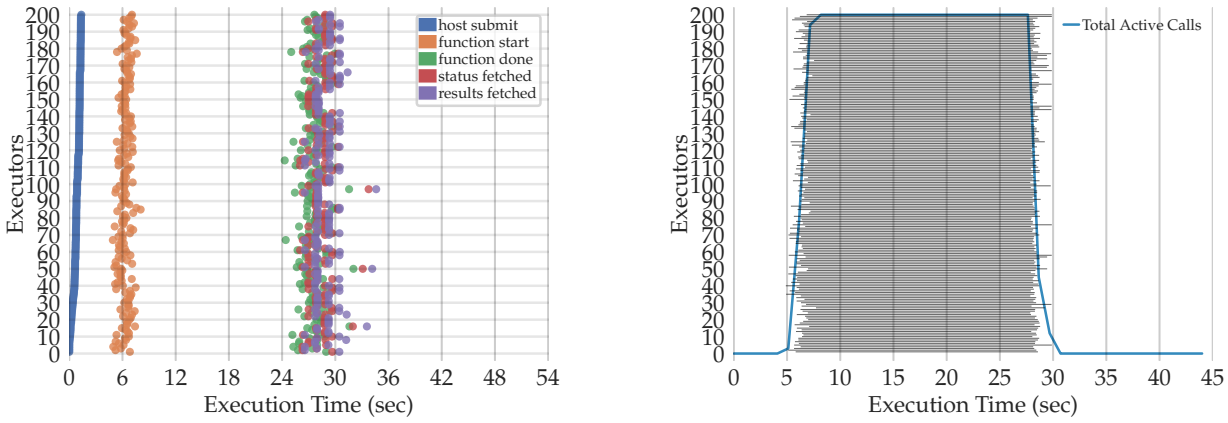
Although the original Lithops framework offered basic plotting functionalities, the exported data from these plots was not available for telemetry monitoring and visualization. However, Lithops Serve addresses this gap and outputs a timeline plot, represented as a sequence of bars in the form of a Gantt chart, to visualize the start, duration, and completion of the executors over time. Figure 26a presents a detailed timeline illustrating every step of the execution process. The steps are described below:

- **Host submit:** when invoking multiple executors, a thread is created for each asynchronous call to the AWS boto3 API. This timestamp is recorded just before the call within every thread.
- **Function start:** records the time where the serverless executor starts to execute the DL code. This occurs after initialization and the loading of dependencies.
- **Function done:** records the disposal time of the executor.
- **Status fetched:** prior to function completion, the executor uploads its status to AWS S3. Lithops periodically monitors the status of an executor by polling AWS S3, recording the time when an executor is marked as terminated.
- **Results fetched:** Likewise, final results are automatically uploaded to AWS S3 and retrieved by the Lithops orchestrator before the executor gracefully terminates. These results correspond to the data returned by the Python `return` statement that terminates the executor, similar to what happens with an AWS Lambda function:

```
def lambda_handler(event, context):
    message = 'Hello␣{}␣{}!'.format(event['first_name'], event['last_name'])
    return {
        'message' : message
    }
```

  Intermediate results corresponding to the classification of batch of images are saved to AWS S3, or can be sent back to the Lithops Serve orchestrator.

Alternatively, the histogram just shows the time that the function was running; in this case, in AWS Lambda.



(a) Timeline with all Lithops steps

(b) Histogram of function execution

Figure 26: Timeline and Histogram of Lithops Serve on AWS Lambda classifying a dataset (large.35k).

Prior to classification, the dataset undergoes partitioning into batches of a predetermined size (default is 32). Each batch is assigned a unique ID. Batches are sequentially dispatched to executors,

with the executor requesting additional batches from the orchestrator upon completion. Figure 27 illustrates the batch distribution at the end of the process, showing how batches are allocated as executors become idle. Executors finish only when all batches are completed or assigned.

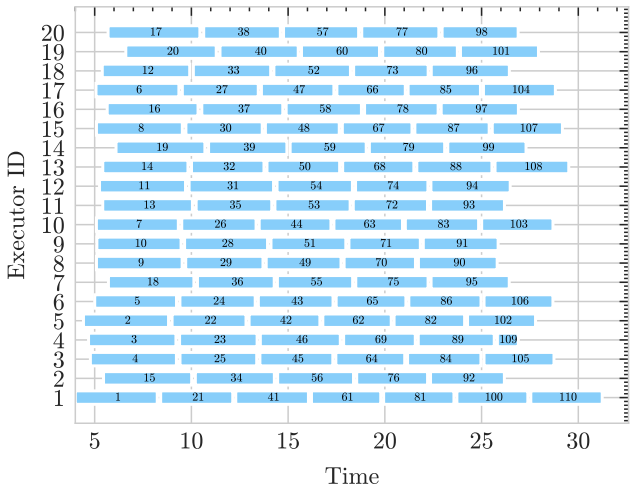


Figure 27: Batch distribution among executors (medium.3k)

Executors also have an internal logging system to record every step of classification: downloading, transformation and inference of every image in batch. This information is recorded internally in a file that after is transformed and incorporated in the results of the dataset. This data can then be plotted as in Figure 27.

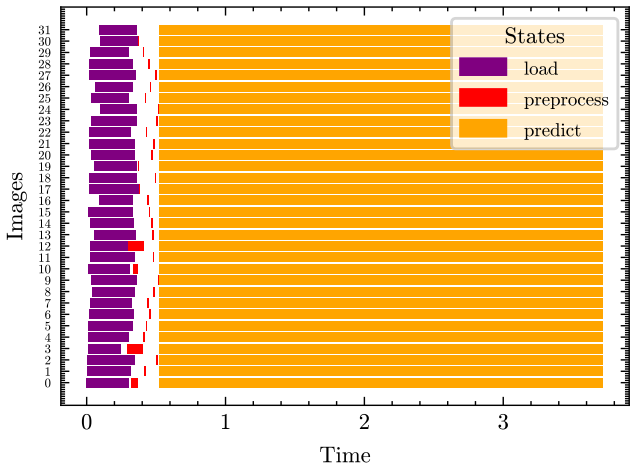


Figure 28: Loading, preprocess and prediction of every image in a batch running on an AWS Lambda executor

Furthermore, the new trace collection system provides data that can be processed to extract latency, throughput, price and performance / $. This is the information used to develop the use case Figures in D2.3.

Integrating the Learning Plane with Lithops Serve offers significant advantages, particularly focusing on the modeling and provisioning functionalities. While Lithops Serve collects data on the status of running images, batches, and executors, it currently lacks a real-time monitoring system. This capability will be implemented to enable the Learning Plane to effectively share critical information, empowering it to make informed resource adjustments not only for incoming datasets but also for existing datasets.

A Prometheus integration for Lithips is currently under development, and will be employed to collect and store metrics on Lithops Serve. Prometheus is well-suited for this task due to its powerful data collection capabilities.

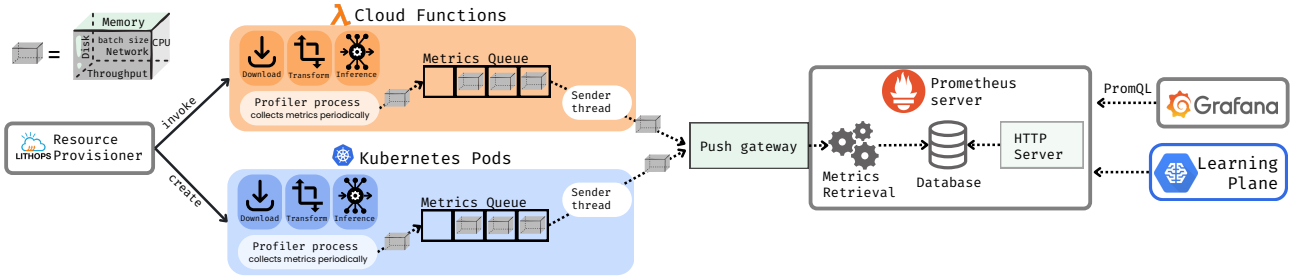
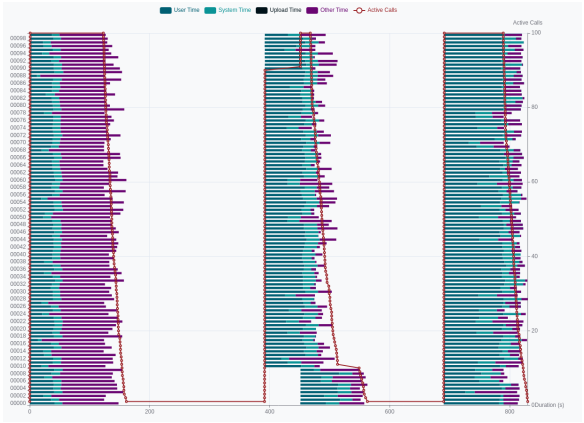


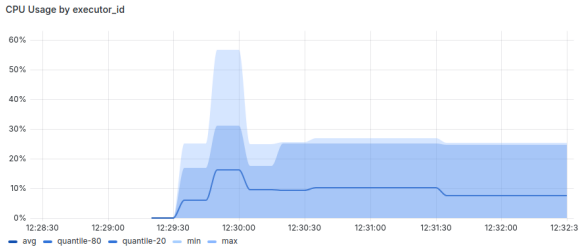
Figure 29: Architecture of Prometheus on Lithops.

As can be seen in Figure 29, Lithops Serve orchestrator will include a Prometheus server to collect metrics from executors. The Resource Provisioner invokes cloud functions and/or kubernetes pods, each of them with an executor. Each executor has a profiler process that collects real time metrics on CPU, memory, network and disk usage, latency, throughput, dataset size and batch size. These metrics are registered for every executor and process within the executor. These metrics, once collected are put in a Queue. In parallel, a sender thread gets the metrics from the queue and sends them to the prometheus server via a configured endpoint (push gateway). Once in Prometheus, metrics are saved in the internal database.

Unlike previous visualizations created with Python's matplotlib library, the collected metrics will be visualized using Grafana. Grafana offers great integration with Prometheus and excels in creating dynamic, real-time dashboards. Figure 30 displays two types of metric representation.



(a) Timeline of consecutive executions.



(b) CPU usage of an executor in time.

Figure 30: Examples of Grafana Dashboard metrics

The LP will query real-time data through the Prometheus HTTP API, providing information about the entire orchestrator.

Furthermore, a predictive model is under development to use historical data on dataset arrival times and sizes to forecast the arrival of the next dataset. Based on current resource (AWS Lambda and Kubernetes pods) usage and historical trends, resources can be pre-warmed to be ready for the next dataset.

5.3 Data Streaming Instrumentation with Pravega

In Pravega, we have built a complete metrics system that allows administrator and orchestrators to reason about the performance of the system and react accordingly. In the following, we provide some

technical details about the Pravega metrics service and the compatible systems and APIs to consume and visualize metrics from Pravega:

- *Metrics exporting (InfluxDB, Prometheus, StatsD)*: Pravega uses Micrometer[6] as the underlying metrics library. Pravega also provides our own API to simplify metrics access within the system. The Pravega metrics service is initiated using the `StatsProvider` interface: it provides the start and stop methods for the metrics service. It also provides `startWithoutExporting()` for testing purposes, which only stores metrics in memory without exporting them to external systems. Currently we have support for InfluxDB, Prometheus, and StatsD registries. The internal API that Pravega uses to interact with Micrometer metrics can be inspected in detail in the official documentation[7].

- *Metrics visualization (Grafana, Prometheus)*: The metrics that Pravega produces can be visualized via Grafana and Prometheus dashboards, among others. It is important that there are multiple sources of metrics in Pravega that can be visualized and exploited by the Learning Plane in CloudSkin. For instance, the system exports metrics about performance, including latency of operations, throughput of the write-ahead log (*i.e.*, Bookkeeper), or the data rate at which Pravega is moving data to long-term storage. Moreover, Pravega exports metrics about streams, so the Learning Plane may reason about the number of streams in the system and their utilization (*e.g.*, bytes or events per second), even at the segment level. Finally, Pravega also exports metrics about the resource usage of Segment Store and Controller processes (*e.g.*, JVM statistics).

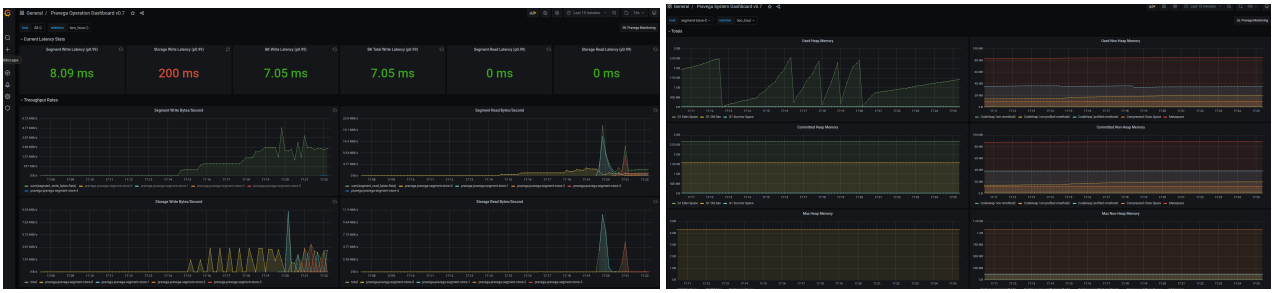


Figure 31: Dashboards to visualize metrics in Pravega.

In summary, Pravega metrics can be a powerful substrate to build intelligence for managing the streaming infrastructure across the Cloud-Edge Continuum. Moreover, the technologies that the system uses for metrics management and visualization are aligned with the proposal for the CloudSkin Learning Plane, which facilitates integration.

5.4 Usage Instrumentation using Dataspace

The learning panel implemented for the dataspace considers two levels of telemetry:

- Infrastructure level. Operations Telemetry: Data will be collected about the infrastructure of the data space, including CPU usage, memory, storage, and bandwidth consumption. System crash and error logs will also be included.

  KIO, as a partner involved in the CloudSkin project, provides the necessary resources for the testbed utilized by ALT. These resources are based on the IaaS services we offer from one of our certified TIER IV Data Centers, which is part of the network of cloud Edge computing platforms we operate. The service provides a wide range of metrics within the management

[6] `https://docs.micrometer.io/micrometer/reference/`
[7] `https://cncf.pravega.io/docs/latest/metrics/`

panels that are accessible from the console that ALT has access to, allowing monitoring of the testbed's status and performance.

Among the metrics available, the following ones stand out:

- CPU usage percentage rate (%)
- Memory-Guest Demanda
- RAM Memory usage percentage rate (%)
- Network-Data Receive Rate (KBps)
- Network-Data Transmit Rate (KBps)
- Network-Usage Rate (KBps)
- Virtual Disk-Aggregate of all Instances-Total IOPS
- Virtual Disk-Aggregate of all Instances-Total Latency (ms)

- Application level. Although the creation of an analysis and monitoring software panel is not contemplated, values could be obtained from the application to facilitate the telemetry of the platform, which could be crossed with infrastructure consumption data. Application level telemetry can be focused on several levels to ensure efficient operation and optimal security and performance:
  - Security Telemetry: Access attempts (successful and failed), changes in permissions and any other activity that may indicate a security risk will be monitored. This may include monitoring for unauthorized access and detecting unusual patterns that could suggest an intrusion attempt. This telemetry is not, in this mockup phase, one of the priorities of the project, although it will be incorporated at a basic level.
  - Performance Telemetry: Data will be obtained about the pages and actions launched (data source loading, project creation, API calls). Cross-referencing this information with server performance logs can help identify bottlenecks or performance issues that impact the user experience.
  - Usage Telemetry: This telemetry allows us to understand how users interact with the data space and their frequency. In the current model it is a secondary objective, so only those that are considered most relevant for the optimization of the platform will be applied, such as access frequencies, leaving out telemetries of commercial interest.
  - Integration Telemetry. Usage and performance of integrations will be analyzed through data access mechanisms. This will include both access and integration error control.

When implementing these levels of telemetry, and although this use case is part of a research project, compliance with related legislation, such as the GDPR in Europe, will be studied, reflecting our concern for user privacy. Consequently, all personal data will be managed and stored securely, with users informed about what data is collected and its motivation.

6 Conclusions

This deliverable presents the learning methods to be used for infrastructure and workload management. Firstly, we introduced the Learning Plane design and implementation and also came up with a prototype specifying the usage of each component and how to customize them. Then, we proposed different ML-based models for environment modelling, which can be used for different use cases. In the end, the last section defined for each use case or scenario the data can be retrieved for infrastructure and workload management.

References

[1] J. L. Berral, D. Buchaca, C. Herron, C. Wang, and A. Youssef, "Theta-scan: Leveraging behavior-driven forecasting for vertical auto-scaling in container cloud," in 2021 IEEE 14th International Conference on Cloud Computing (CLOUD), pp. 404–409, 2021.

[2] J. L. Berral, C. Wang, and A. Youssef, "AI4DL: Mining behaviors of deep learning workloads for resource management," in 12th USENIX Workshop HotCloud, 2020.

[3] T. White, Hadoop: The Definitive Guide. O'Reilly Media, Inc., 1st ed., 2009.

[4] Y. Zhai, J. Tchaye-Kondi, K.-J. Lin, L. Zhu, W. Tao, X. Du, and M. Guizani, "Hadoop perfect file: A fast and memory-efficient metadata access archive file to face small files problem in HDFS," Journal of Parallel and Distributed Computing, vol. 156, pp. 119–130, oct 2021.

[5] M.-A. Vef, N. Moti, T. Süß, M. Tacke, T. Tocci, R. Nou, A. Miranda, T. Cortes, and A. Brinkmann, "Gekkofs — a temporary burst buffer file system for hpc applications," J. Comput. Sci. Technol., vol. 35, p. 72–91, jan 2020.

[6] IBM, "GEDS Distributed Ephemeral Data Store." https://github.com/IBM/GEDS.

[7] P. Liu, G. Bravo-Rocca, J. Guitart, A. Dholakia, D. Ellison, and M. Hodak, "Scanflow: An end-to-end agent-based autonomic ml workflow manager for clusters," in Proceedings of the 22nd International Middleware Conference: Demos and Posters, Middleware '21, (New York, NY, USA), p. 1–2, Association for Computing Machinery, 2021.

[8] P. Liu, G. Bravo-Rocca, J. Guitart, A. Dholakia, D. Ellison, and M. Hodak, "Scanflow-k8s: Agent-based framework for autonomic management and supervision of ML workflows in kubernetes clusters," in 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid), pp. 376–385, 2022.

[9] Alibaba Cloud traces. https://github.com/alibaba/clusterdata/blob/master/cluster-trace-v2017/README2017.md.

[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017.

[11] H. Zhou, S. Zhang, J. Peng, S. Zhang, J. Li, H. Xiong, and W. Zhang, "Informer: Beyond efficient transformer for long sequence time-series forecasting," 2021.

[12] K. Madhusudhanan, J. Burchert, N. Duong-Trung, S. Born, and L. Schmidt-Thieme, "Yformer: U-net inspired transformer architecture for far horizon time series forecasting," 2021.

[13] Apache Spark. https://spark.apache.org.

[14] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," CoRR, vol. abs/1810.01963, 2018.

[15] TPC-H Benchmark. https://www.tpc.org/tpch/.

[16] R. Gracia-Tinedo, F. Junqueira, T. Kaitchuck, and S. Joshi, "Pravega: A tiered storage system for data streams," in Proceedings of the 24th International Middleware Conference, pp. 165–177, 2023.

[17] M. D. de Assuncao, A. da Silva Veith, and R. Buyya, "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions," Journal of Network and Computer Applications, vol. 103, pp. 1–17, 2018.

[18] V. Cardellini, F. Lo Presti, M. Nardelli, and G. R. Russo, "Runtime adaptation of data stream processing systems: The state of the art," ACM Computing Surveys, vol. 54, no. 11s, pp. 1–36, 2022.

[19] V. Cardellini, F. L. Presti, M. Nardelli, and G. R. Russo, "Decentralized self-adaptation for elastic data stream processing," Future Generation Computer Systems, vol. 87, pp. 171–185, 2018.

[20] R. Gracia-Tinedo, F. Junqueira, B. Zhou, Y. Xiong, and L. Liu, "Practical storage-compute elasticity for stream data processing," in Proceedings of the 24th International Middleware Conference: Industrial Track, pp. 1–7, 2023.

[21] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic scaling for data stream processing," IEEE Transactions on Parallel and Distributed Systems, vol. 25, no. 6, pp. 1447–1463, 2013.

[22] B. Lohrmann, P. Janacik, and O. Kao, "Elastic stream processing with latency guarantees," in 2015 IEEE 35th International Conference on Distributed Computing Systems, pp. 399–410, IEEE, 2015.

[23] R. Gracia-Tinedo, Y. Tian, J. Sampé, H. Harkous, J. Lenton, P. García-López, M. Sánchez-Artigas, and M. Vukolic, "Dissecting ubuntuone: Autopsy of a global-scale personal cloud back-end," in Proceedings of the 2015 Internet Measurement Conference, pp. 155–168, 2015.

[24] Wikipedia, "Time-series seasonality." https://en.wikipedia.org/wiki/Seasonality.

[25] Wikipedia, "Long-short Term Memory." https://en.wikipedia.org/wiki/Long$_s$*hort* − *term*$_m$*emory*.

[26] K. Gontarska, M. Geldenhuys, D. Scheinert, P. Wiesner, A. Polze, and L. Thamsen, "Evaluation of load prediction techniques for distributed stream processing," in 2021 IEEE International Conference on Cloud Engineering (IC2E), pp. 91–98, IEEE, 2021.

[27] R. K. Kombi, N. Lumineau, and P. Lamarre, "A preventive auto-parallelization approach for elastic stream processing," in 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), pp. 1532–1542, IEEE, 2017.