CloudSkin

Funded by the European Union

HORIZON EUROPE FRAMEWORK PROGRAMME

CloudSkin

(grant agreement No 101092646)

Adaptive virtualization for AI-enabled Cloud-edge Continuum

D5.3 Integration of the Learning Plane

Due date of deliverable: 31-12-2025
Actual submission date: 29-12-2025

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	50
WP/Task related to this document	WP5 / T5.3, T5.4-T5.7(modeling part)
WP/Task responsible	BSC
Leader	Peini Liu (BSC)
Technical Manager	Peini Liu (BSC)
Quality Manager	Robert Haas (IBM)
Author(s)	Peini Liu(BSC), Joan Oliveras Torra(BSC), Jordi Guitart (BSC), Josep Lluis Berral (BSC), Ramon Nou (BSC), Jose Miguel Garcia (ALT), Raúl Gracia (DELL), Hossam Elghamry (DELL), Alan Cueva (DELL), Reuben Docea (NCT), Marc Sànchez Artigas (URV), Thomas Ortner (IBM)
Partner(s) Contributing	BSC, NCT, DELL, URV, ALT, IBM
Document ID	CloudSkin_D5.3_Public.pdf
Abstract	Report on the integration efforts for the learning methods in the CLOUDSKIN stack, and the validation processes. The produced framework, tools and prototypes will be published as open-source software through public distribution channels.
Keywords	Learning plane, machine learning, model training and validation.

History of changes

Version	Date	Author	Summary of changes
0.1	01-10-2025	Peini Liu	Add TOC.
0.2	25-11-2025	Peini Liu	Add Section 2 Learning Plane.
0.3	05-12-2025	Thomas Ortner	Add Flowstate.
0.4	10-12-2025	Peini Liu	Update Section 2 Learning Plane.
0.5	11-12-2025	Joan Oliveras Torra	Add Section 3 Time series and regression models.
0.6	12-12-2025	Joan Oliveras Torra	Add Mobility use case model usages.
0.7	12-12-2025	Raúl Gracia	Add Surgery use case model usages.
0.8	12-12-2025	Marc Sànchez Artigas	Add Metabolomics use case model usages.
0.9	12-12-2025	Jose Miguel Garcia	Add Agriculture use case model usages.
1.0	15-12-2025	Reuben Docea	Update Surgery use case model usages.
1.1	26-12-2025	Peini Liu	Final version.

Table of Contents

List of Abbreviations and Acronyms

API	Application Programming Interface
CatBoost	Categorical Boosting
CC	Creative Commons
CPM	Contiguous Patch Masking
CSV	Comma-separated values
DOI	Digital Object Identifier
ETS	Holt–Winters Exponential Smoothing
FBD	Functional Basis Decoder
FEDformer	Fourier Enhanced Decomposition Transformer
FFD	First-Fit Decreasing
FMs	Foundation Models
FPS	Frame per Second
GRU	Gated Recurrent Units
LP	Learning Plane
LSTM	Long Short-Term Memory
MAE	Mean Absolute Error
ML	Machine Learning
MSE	Mean Squared Error
NLP	Natural Language Processing
QoS	Quality of Service
SLA	Service Level Agreement
SLO	Service Level Objective
SOTA	State-of-the-art
SSMs	State Space Models
XGboost	eXtreme Gradient Boosting

1 Executive summary

This **Deliverable D5.3** presents the integration of the Learning Plane. In the section 2, we first present the final version of the Learning Plane's architecture, and then propose the implementation architecture of data-connector: an agent-based framework for autonomous ML-based smart management in Cloud-Edge continuum. Moreover, we provide a demonstration of integrating the model outcomes of T5.1 and T5.2 within the data-connector that can be used for the project use cases. In the section 4, we provide different ML models training and evaluating towards different management problems for different usecases. The employment of those models in real scenarios and how this helps with the business is explained in **Deliverable D5.4**.

2 Learning Plane

Artificial Intelligence and Machine Learning (AI/ML, or just ML hereafter) are becoming pervasive and integrated into different kinds of intelligent applications, and the collaborative Cloud-Edge continuum has been introduced as an emerging trend to support their adoption into use cases. However, managing these ML applications in the Cloud-Edge continuum is challenging due to the ML application's dynamic resource usage with different user loads and Cloud and Edge's dynamic resource availability. We envision machine learning methods that can be used for smart management in this dynamic environment, but how to deploy and utilize them for the adaptation scenario in Cloud-Edge continuum is unknown.

We propose an agent-based framework to enable autonomous smart management mechanisms that can be broadly enabled in diverse adaptation scenarios. The agent acts as a data-connector[1], connecting different sources of data, utilizing ML models for decision-making and triggering adaptations in Cloud-edge platforms. A small case study shows the feasibility of our proposed data-connector for smart migration of an ML workload in the Cloud-edge continuum. The result shows that the smart migration-enabled Cloud-edge scenario has 11.9% ML application QoS(prediction time) better than the Cloud scenario without migration. Moreover, with minimal customization, the data connector agent can be adapted for more use cases, which will be shown in Section 4.

2.1 Data-connector Framework

In this section, we introduce the architecture of the Data-connector agent and its features. This agent is a Scanflow agent[1] variant, with enhancements to abstract to different data and platforms, and integrate ML strategies. Specifically, we define agent intelligence with model inference, triggers, and actuators for smart workload management in a dynamic environment.

2.1.1 Agent Architecture

We use the concept of agent, which does not implement a global model or plan but only some simple behaviours. These behaviours allow the agent to observe the environmental changes proactively. An agent includes a sensor that detects changes in internal and external states, an ML model that responds to relevant observations, and an actuator that activates specific processes within the environment or other agents.

Data-connector agents are the fundamental components to implement autonomous ML-based smart management. Each agent is an independent computational unit that is able to run actions according to state changes. Therefore, an agent can be defined as a set of state-to-action mappings (i.e., $Agent = States(s) \rightarrow Actions(a)$), that is, state changes could result in the execution of actions (recommended by ML-based strategies). However, an agent usually cannot directly perceive the states but compute them from observations o_t using a function F. Also, the agent performs actions through rules with the computed states s_t ($a_t = R(s_t)$). Figure 1 shows the agent-environment interaction: At time t, the agent computes the states s_t from the observations o_t using the function F. Then, it chooses actions a_t according to the rules R to achieve the agent's goal.

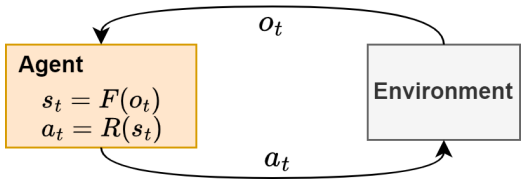


Figure 1: Agent-Environment interaction.

Scanflow agent implements its autonomy by defining strategies that include events, constraints, and actions, expressed as 3-tuples $Strategy = (Events, Constraints, Actions)$, specifically, the autonomic management strategy is described as: when *Event* occurs, if *Constraint* is satisfied, then *Action* will

[1] https://github.com/bsc-scanflow/data-connector

be executed. A Data-connector agent primarily uses ML-based strategies, so that the tuples become $Strategy = (Data, Models, Constraints, Actions)$, specifically, the autonomic ML-based smart management strategy is: load the *Data*, inference them with the *Model* for predictions or recommendations, evaluate rules *Contraints* when generating decision-making, and then execute *Actions* on specific platforms. The workflow of the Data-connector agent to use the strategies is shown in Figure 2.

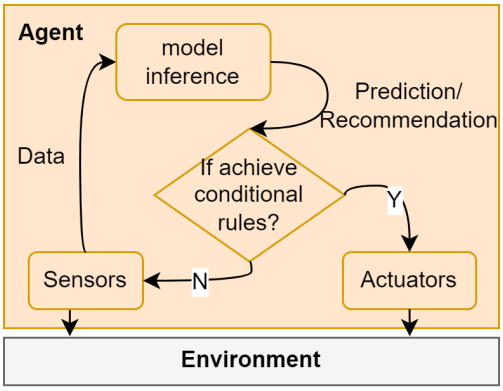


Figure 2: Data-connector agent abstract workflow.

2.1.2 Agent Model Inference

To enable smart management, ML models are used to generate knowledge for the current system. Depending on different use cases and different autonomous management requirements, the developer should gather the history data from their platform and train different models to analyse target scenarios. After that, this model can be used in a pipeline or served as a service for predictions. For example, Figure 3 (steps 1-3) shows that the data-connector agent can trigger an inference pipeline to predict the QoS of an application. This proposed combination of model inference and the agent leverages a distributed decision-making, optimizing performance and reducing the need of extensive telemetry data transfers. This approach is particularly advantageous for application and nodes where statistical predictions can be performed locally, mitigating the necessity for constant data communication with centralized servers.

2.1.3 Agent Sensors

To actively monitor current *States*, data-connector agents are required to trigger tasks to detect useful observations. Data-connector agent has different types of built-in triggers as **Sensors**, namely interval triggers, date triggers, and cron triggers (see Table 1). In addition, basic triggers can be combined 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 *States*. Note that each Scanflow agent contains an asynchronous I/O scheduler with multiple queued tasks. Tasks are run by the scheduler in a thread-pool.

Table 1: 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).

Data-connector has a flexible way to connect with different sources of data, such as real-time data from Prometheus, artifacts from Minio, or different types of data from MLflow. Figure 3 (steps 4-5)

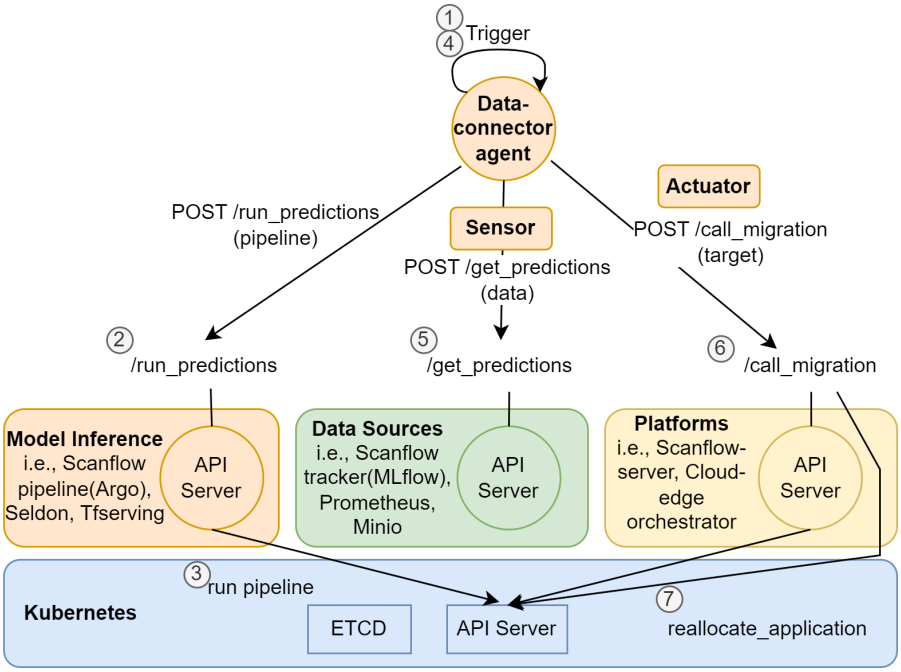


Figure 3: Data-connector agent implementation workflow.

shows a sensor to get_predictions and aggregate prediction results. Different use case will need to prepare the relevant data and define their customized sensors.

2.1.4 Agent Actuators

After a change in *States*, agents need to perform *Actions* (i.e., $a_t = R(s_t)$) through **Actuators** to adapt changes to the environment. Different use cases will need to define their operation primitives for customized actuators and make sure the operations can be implemented or realized by the platform they used in their testbed (i.e., Kubernetes, multi-clustering orchestrator, etc.). For instance, Figure 3 (steps 6-7) shows that an actuator's call to migrate applications, the corresponding Cloud-edge orchestrator/Kubernetes should accomplish this operation through their internal implementations.

2.1.5 Agent Communication

The Data-connector agent's communication methods are aligned with the Scanflow agent. Below, we introduce two different ways for the Data-connector agent to communicate with the environment: through RESTful APIs and shared artifacts[2, 1].

- Interaction through RESTful APIs: In this approach, the sensors and actuators of an agent are exposed as interfaces. The agent is registered into a service discovery system, allowing it to call the well-defined interfaces from the environment data sources to get data through REST. Additionally, the remote call leads to changes in the agent's belief/state and ultimately calls platform APIs to drive an action.
- Interaction through shared artifacts: The data-connector has a database for shared artifacts within a knowledge base, which receives queries from the model inference pipeline and the sensors, and delivers the results from its database. These data include the metadata and logs from the prediction service/pipelines, as well as the metrics, scores, parameters, and different versions of the ML model. For instance, the model inference process will download the model from the database for inference, and the results will be aggregated as knowledge and saved back into a database. This approach is mainly used for model inference within a Data-connector.

2.2 Data-connector Demonstration

The previously proposed Data-connector agent is a framework that can be used for different use cases and satisfy different autonomous management requirements. In this section, we build a use case on top of Scanflow-Kubernetes platform and enable a data-connector agent to perform smart QoS-aware migration for an image classification ML workload.

2.2.1 Experimental Settings

Platform Settings: Our experiments are executed on a three-node Kubernetes cluster, where the control-plane consists of 12 cores, 48Gi; one worker nodes of 8 cores, 16Gi, and one edge node of 4 cores, 8Gi. Each node is a Openstack-based VM with Rocky Linux 9.3.

The Scanflow-Kubernetes platform is built based on Kubernetes v1.28.2 (Network: Calico v3.26.4, Runtime: containerd v1.6.25, DNS: CoreDNS v1.10.1, Storage: etcd 3.5.9). Scanflow corresponding toolkits are Argo workflow v3.5.2, Mlflow 2.9.2, Minio v5.0.11, and PostgreSQL 1.17.

Application Settings: We use a target ML-serving application for image classification. The application is running a Resnet_18 model, wrapped within TorchServe and deployed by Kubernetes Deployment controller. We are using a client sending pictures with a concurrency of two to the TorchServe service and receiving classification results.

2.2.2 Smart Migration Scenario Settings

Figure 4 shows an example of smart migration using a data-connector agent. The data-connector agent proactively calls model inference to predict the QoS of a target application in the cloud and the edge, and also periodically watches the application QoS prediction results with a performance-cost comparison policy to decide if triggering the application migration.

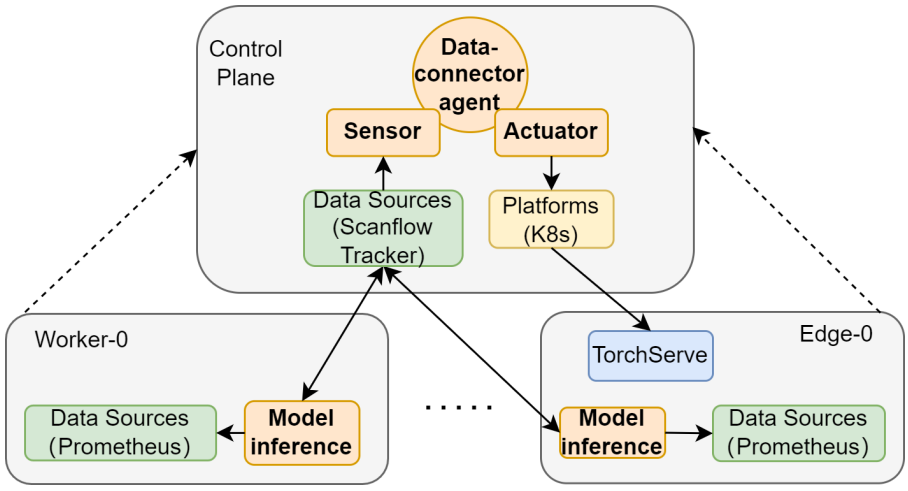


Figure 4: Smart migration of TorchServe using data-connector agent.

The autonomous strategies of the agent for application migration are described in detail in Table 2. The proposed data-connector requires the Data Engineer team to enable the model inference and provide custom functions for sensors and actuators.

2.2.3 Data-connector Agent Implementation

To implement the agent, custom functions of these main components should be developed, i.e., model inference, sensor, and actuator.

Pre-steps: Pre-steps mainly contain data collection and model training. The model used for QoS prediction should be pre-trained. A TorchServe application runs in both cloud and edge and each of the nodes has dynamic resource availability over time. Then we use these node resource usage data and the application prediction latency data to train a Random Forest(RF) model. The trained model

Table 2: Agent autonomous management strategy

Agent	***Analyzing and Planning Strategies***
Data-connector agent	Analyzing Strategy: *QoS_predictions* **WHEN** *intervalTrigger*(*5min*, *QoS_prediction*(*data*)) **IF** *successful_call* **THEN** *runWorkflow*(*prediction-pipeline*, *data*) Planning Strategy: *migrate_app* **WHEN** *intervalTrigger*(*5min*, *watch_app_qos*) **IF** *target_node_app_qos* > *current_node_app_qos* **THEN** *Call*(*Platform-API* : *migrate_app*)

is saved and can be used by the data-connector agent for QoS predictions. More models accepted in the data-connector can be found in the Section 3.

Model inference: Model inference is to use the pre-trained model for predictions. In our implementation, we define a model inference that Loads data uses Prometheus API to get the local node resource usage data in the last time window, preprocesses the unknown values etc., and predicts the application QoS in the edge and cloud using the pre-trained RF model. Note that the model inference can be distributed into the edge and cloud if the data is not aggregated as shown in Figure 4.

Sensor Sensor defines which data in the shared artifacts the agent should watch, and the policy to decide if triggering the actuator. Listing 1 shows the agent is watching the QoS predictions and having a policy of performance cost trade-off (i.e., performance/cost). If QoS constraints are satisfied, chooses the node with the best placement recommendation. Listing 2 shows the frequency of this watch, and how the sensor can mount a timer, in this case 1 minute.

```
#example 1: watch app QoS predictions
@sensor(nodes=["predictor"])
async def watch_qos(runs: List[mlflow.entities.Run], args, kwargs):
    qos = 0
    input_data = get_qos()
    if input_data:
        qos, node_index = choose_better_nodes(input_data)
        if qos_constraints(qos):
            await call_migrate_app(max_qos_index, "icresnet", "torch-deployment")
        else:
            logging.info("all machine can not achieve qos sla, no actions")
    else:
        logging.info("no data in last check")
    return max_qos
```

Listing 1: Custom sensor to get app QoS predictions and enable recommendation policy

```
trigger = client.ScanflowAgentSensor_IntervalTrigger(minutes=1)
sensor = client.ScanflowAgentSensor(
    name='watch_qos',
    isCustom=True,
    func_name='watch_qos',
    trigger=trigger
)
```

Listing 2: Set trigger to watch_qos sensor

Actuator and Platform Actuator is used to connect different platforms to execute migrations. In our use case, we use native Kubernetes to deploy our application, thus the migrate operations should be done by using Kubernetes API. Listing 3 shows the migration operations execution, where the agent

can generate a patch of the nodeSelector for a target pod to update the application deployment, so that the application can be finally migrated from one node to another.

```
async def call_migrate_app(max_qos_index, namespace, deployment_name):
    nodeName_list=['cloudskin-k8s-control-plane-0.novalocal',
                'cloudskin-k8s-worker-0.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 3: Custom actuator to connect k8s platform

2.2.4 Data-connector Agent Deployment

Model inference deployment Previously we mentioned the model inference which can be deployed using the Scanflow pipeline, similar to an Argo workflow. If a use case requires customized model inference as well as its deployment, other online serving solutions like Seldon, Torch, or TensorFlow Serving can be used.

Agent service deployment The central agent has an HTTP server, so it can be deployed as a service in Kubernetes environment. Additionally, the shared artifacts knowledge database should be accessible from both the edge and the cloud.

2.2.5 Experimental Results

In this section, we evaluate the effectiveness of our data-connector agent in a scenario of application smart migration. In particular, the migration of a TorchServe-based image classification application is done according to the application QoS smart prediction and performance-cost trade-off policy in a dynamic cloud-edge continuum.

Figure 5 shows the application QoS (i.e., image classification prediction time latency) trends of two runs in Cloud and Cloud-edge scenarios. The red one is a baseline which shows all the requests processed in the Cloud. The blue (the Cloud) and green (the Edge) ones show all the requests processed in the Cloud-edge continuum, where application remains in the Cloud at night due to a high cost of energy in the Edge, and in the daytime, application can be migrated between Cloud and Edge based on the QoS prediction.

Figure 6 shows the distribution of application QoS (i.e., image classification prediction time) in Cloud and Cloud-edge scenarios, and the average prediction time latency of Cloud-edge scenario is 11.9% better than Cloud scenario.

2.3 Tutorial

Learning Plane development and deployment This tutorial demonstrates the development and deployment of the Learning Plane in a Kubernetes environment. The recorded screen (see Figure 7)

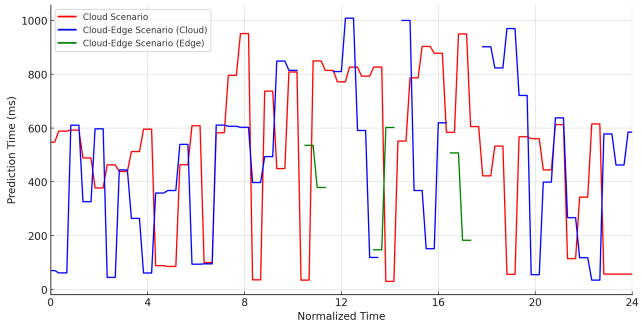


Figure 5: Application QoS (prediction time latency) trends in Cloud and Cloud-edge scenarios.

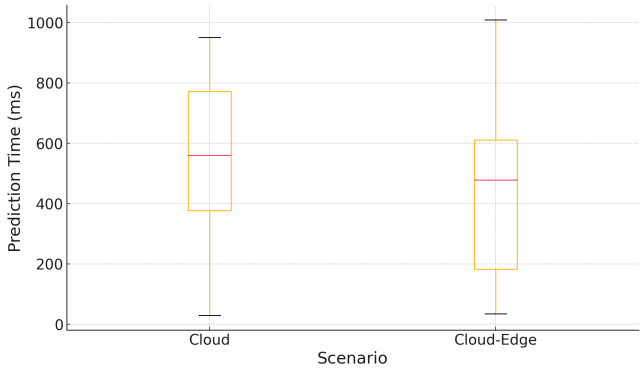


Figure 6: Distribution of application QoS (prediction time latency) in Cloud and Cloud-edge scenarios.

is divided into two sides to showcase the real-time Learning Plane deployment. The left side shows a Jupyter Notebook that contains all metadata of development and deployment of the Learning Plane targeting a mobility use case, and the right side shows a terminal connected to the cloud-edge infrastructure that shows the Learning Plane creation in real time.

Detailed tutorial can be found in the open repository: `https://github.com/bsc-scanflow/data-connector/tree/main/tutorials/cloudedge-proactive-migration`

3 Learning Plane Models

In the Learning Plane, multiple models are developed and used for diverse purposes, demonstrating its adaptability for different use-cases and goals. The section below introduces the different models that are developed or used within the Learning Plane, we provide a brief theoretical overview of each method, deep-diving in FlowState as a novel model architecture created within the project, we highlight the modelling principles and mathematical foundations behind all models used.

3.1 FlowState: A novel State-Space based Time Series Foundation Model

As mentioned in Section 2 ML models are ubiquitously found in many aspects of our daily lives. Especially foundation models (FMs), have received extensive research interest and are today employed in various natural language processing (NLP) tasks. However, despite their astonishing performance in NLP, FMs struggle to be applied to time series processing. Although NLP and time series processing share similarities, for example both are sequence processing tasks, there are major differences. In particular, the atomic unit of information in both tasks, a token in NLP and an individual data point in time series, carries substantially more information in NLP than in time series. Furthermore, time series data can be multivariate and vary strongly across tasks, e.g., the number of incoming requests over time can look entirely different for a metabolomics or a video processing application.

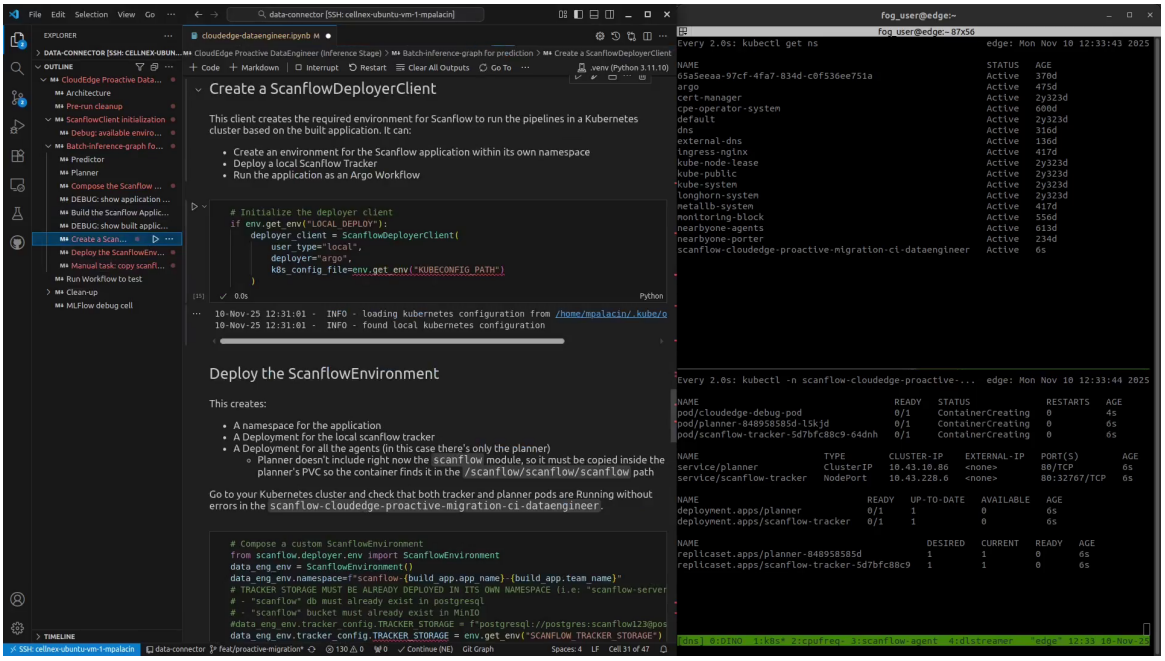


Figure 7: Learning Plane tutorial for mobility use case: Proactive Migration.

Therefore, other model capabilities are required for the time series domain, which resulted in different model architectures to emerge as state-of-the-art (SOTA). For example, while FMs for NLP have so far undoubtedly been dominated by the transformer architecture [3], the same architecture is performing poorly in time series tasks [4]. Researchers have uncovered better architectures, based on linear layers that mix over time and features [5, 6] in an alternating manner. More recently, state space models (SSMs) [7] have emerged as viable alternatives that currently represent the SOTA in several time series tasks.

3.1.1 FlowState Architecture

We developed FlowState [8, 9], an encoder-decoder architecture, employing an S5-based [10] encoder and a functional basis decoder (FBD). Figure 8a shows an overview of its architecture. The input time series with length L is first normalized in a causal manner. Afterwards, the normalized inputs are embedded linearly and then provided to the SSM encoder directly without any patching, see Section 3.1.2. Importantly, whilst the time series before being processed by the SSM are considered to be in the feature space, where each element of the input represents features of the time series, the SSM encodes this information into a coefficient space, where it operates on time-independent coefficients of continuous basis functions. The final output of the SSM encoder forms the basis for the FBD, see Section 3.1.3 for details, whose outputs are then inversely normalized, using the inverse method of the input normalization, and form the forecasts of our model. Importantly, the FBD maps from the coefficient space back to the feature space to provide the forecasts. Furthermore, the SSM encoder, as well as the FBD are controlled by an additional scaling factor s_Δ, that allows to adjust these components to the sampling rate of the input data.

3.1.2 SSM Encoder

FlowState utilizes a stack of S5 layers to form the SSM encoder, see Figure 8b. One S5 layer l consists of an S5 block, followed by an MLP. The dynamics of the S5 block can be described through these governing equations:

$$\mathbf{s}_t^l = \bar{\mathbf{A}}^l \mathbf{s}_{t-1}^l + \bar{\mathbf{B}}^l \boldsymbol{x}_t^{l-1} \tag{1}$$

$$\mathbf{h}_t^l = \bar{\mathbf{C}}^l \mathbf{s}_t^l + \bar{\mathbf{D}}^l \boldsymbol{x}_t^{l-1} \tag{2}$$

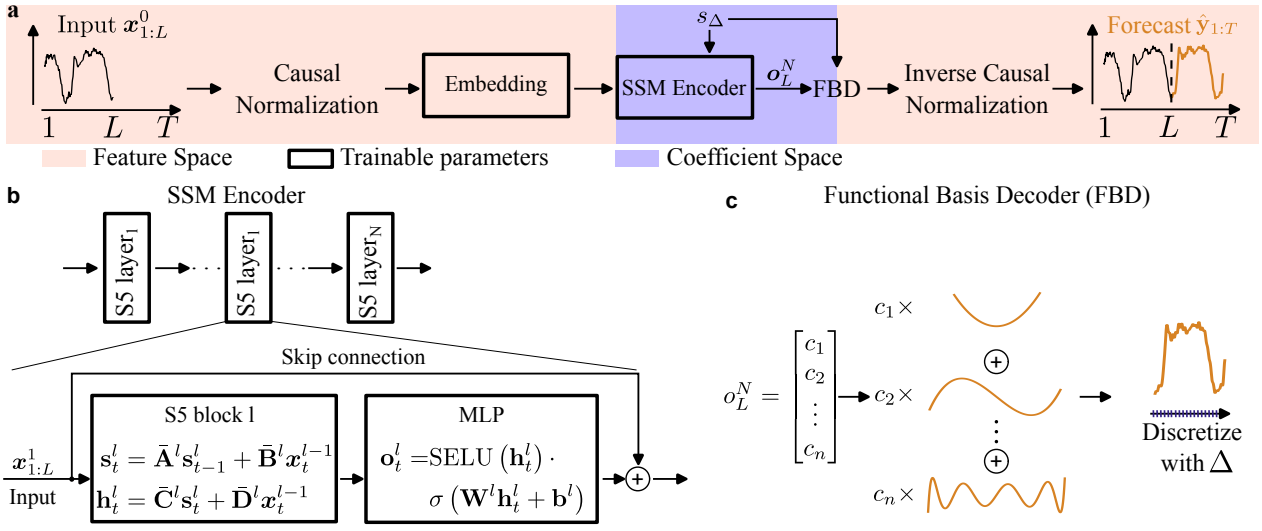


Figure 8: **Architecture overview. a** Overview of the FlowState architecture. The input context in the feature space (orange color) gets normalized, embedded and then processed by the SSM encoder. The SSM encoder transforms the input into the coefficient space (blue color) and provides the final encodings to the functional basis decoder, which then produces the final forecast. Modules with trainable parameters are highlighted in black rectangular blocks. **b** The SSM encoder consists of N S5 layers, each composed of an S5 block extended with an MLP layer. A skip connection is used to allow inputs to propagate also to later encoder layers. **c** The functional basis decoder interprets the outputs $\mathbf{o}^l_t$ of the SSM encoder as coefficients of a functional basis and creates a continuous output, which can be sampled at regular intervals Δ to produce the forecast.

where $\bar{\mathbf{A}}^l \in \mathbb{R}^{P\times P}$, $\bar{\mathbf{B}}^l \in \mathbb{R}^{c\times P}$, $\bar{\mathbf{C}}^l \in \mathbb{R}^{P\times H}$ and $\bar{\mathbf{D}}^l \in \mathbb{R}^{c\times H}$ are the state transition, the input, the output and the skip connections matrices of layer l, m and n are the hidden state size and the output size of the SSM block and $\mathbf{s}^l_t$ and $\mathbf{h}^l_t$ are the state and the output of the SSM block at timestep t. Note that the input is denoted as $\boldsymbol{x}^0_t$. As reported in [10], the matrices of the S5 block l can be computed as

$$\bar{\mathbf{A}}^l = e^{\mathbf{A}^l\Delta}, \bar{\mathbf{B}}^l = \mathbf{A}^{l^{-1}}\left(\bar{\mathbf{A}}^l - 1\right)\mathbf{B}^l, \bar{\mathbf{C}}^l = \mathbf{C}^l, \bar{\mathbf{D}}^l = \mathbf{D}^l,$$

where $\mathbf{A}^l \in \mathbb{R}^{P\times P}$, $\mathbf{B}^l \in \mathbb{R}^{c\times P}$, $\mathbf{C}^l \in \mathbb{R}^{P\times H}$ and $\mathbf{D}^l \in \mathbb{R}^{c\times H}$ are the actual trainable parameters of the S5 and initialized using the HiPPO method [11].

3.1.3 Functional Basis Decoder

For the functional basis decoder, we take inspiration from how SSMs are initialized from an input sequence. The HiPPO approach ensures that their hidden state expresses coefficients of a polynomial basis, which optimally approximates the input sequence. In particular, [12] demonstrated a possibility to use the hidden state of their SSM at timestep t to reconstruct the input sequence until t with a functional basis. We adopt this approach for our decoder, but instead of extracting coefficients that can be used to reconstruct the input, we use a continuous functional basis to construct the forecast from the final outputs of the SSM encoder $\mathbf{o}^N_L$, see Figure 8c. In particular, our proposed FBD interprets the final outputs of the SSM encoder, $\mathbf{o}^N_L$, as coefficients of a functional basis, which can in turn be used to produce a continuous output function. To obtain the forecast with a desired quantization Δ, this continuous output is then sampled at an equally spaced interval, with the spacing Δ. The FBD

can be formalized as follows:

$$c_i = o_{L,i}^N \tag{3}$$

$$\tilde{\boldsymbol{y}} = \sum_{i=1}^{n} c_i p_i(a, b) \tag{4}$$

$$\hat{\boldsymbol{y}} = \text{sample}(\tilde{\boldsymbol{y}}, \Delta), \tag{5}$$

where $p_i(\cdot, \cdot)$ is the i-th basis function evaluated at an interval $[a, b]$, $\tilde{\boldsymbol{y}}$ is the continuous forecast and $\text{sample}(\cdot, \Delta)$ samples the argument equally spaced with Δ.

Our functional basis decoder offers several key advantages. Firstly, it produces a continuous forecast, which can then be sampled with any desired sampling rate. Secondly, it draws inspiration from a well-established procedure to map from coefficient to feature space, and thus can leverage various functional basis functions, depending on the task. For our main experiments, we use the Legendre polynomials to be consistent with the SSM input encoding used by the HiPPO initialization. Another viable option is to use the Fourier basis functions to better match periodic signals. Finally, and most importantly, it enables the decoder to produce forecasts at the correct sampling rate, based on the current parameter Δ. Note that although we introduce the FBD as part of FlowState, it is a separate component and can be combined with other encoder architectures as well.

3.1.4 Foundational model pretraining

We pretrain FlowState as a foundation model on the GIFT-Eval-Pretrain corpus, a large collection of data consisting of time series from various domains and with diverse sampling rates. The individual time series in this corpus may also contain a varying number of channels. Since FlowState is trained as a foundation model, which needs to deal with different number of channels during the pretraining and the later inference phase, we treat all time series during pretraining as univariate, separating the individual channels. However, this approach has the drawback that channel correlations may not properly be taken into account.

In addition to the real time series data from the corpus mentioned above, we added synthetic time series data generated via Gaussian Processes, following the methodology of KernelSynth [13] to both corpora. All data—real and synthetic—is further enhanced using augmentation techniques introduced in [14].

Finally, to enable efficient training of FlowState and to enhance its robustness to varying context lengths, we introduce an advanced foundation training scheme based on multiple parallel predictions, that is described in detail in Appendix A of [9].

3.1.5 Forecasting procedure

FlowState creates patch-based predictions for the target window. If the target windows is larger than the size of an individual forecasting patch, several such patches can be combined in order to form extended forecasts. During this autoregressive forecasting, the outputs produced by FlowState are appended to the previous context. When done naively, this procedure can result in poor performance, as the model will treat the already produced forecasting patches as ground-truth data. In order to counteract this, we employ a technique called contiguous patch masking (CPM), which is described in detail in Appendix A.8 of [9].

3.2 Time-series models

Time-series models aim to predict future values of a sequence by exploiting its temporal structure. These methods operate directly on ordered observations and learn patterns such as trends, seasonality, periodic recurrences, and long-range dependencies. Modern architectures extend classical statistical formulations by incorporating decomposition techniques, frequency-domain representations, and attention mechanisms, enabling flexible modeling of both local and global temporal dynamics. This family of approaches treats forecasting as a sequence learning problem, where the objective is to map past observations to a distribution or point estimate of future values.

3.2.1 Autoformer

Autoformer introduces two core components: a series decomposition block and an Auto-Correlation mechanism. The decomposition block progressively separates each input sequence into trend and seasonal parts. Given an input sequence $\mathbf{x}$, a moving-average operator $\mathrm{MA}(\cdot)$ extracts the trend where $\mathbf{s}$ is the seasonal (residual) component. This decomposition is applied at every layer of the encoder and decoder. The standard dot-product attention is replaced by the Auto-Correlation mechanism, which captures period-based dependencies through time-delay correlation. For query and key sequences $Q, K \in \mathbb{R}^{L\times d}$, the correlation between Q and the τ-shifted K is defined as:

$$\mathbf{t} = \mathrm{MA}(\mathbf{x}), \qquad \mathbf{s} = \mathbf{x} - \mathbf{t},$$

$$\mathrm{Corr}_\tau(Q, K) = \sum_{t=1}^{L} Q_t \cdot K_{t-\tau}.$$

Autoformer selects the top-k dominant delays and the final Auto-Correlation operator aggregates the correspondingly shifted values of V:

$$\mathcal{P} = \mathrm{Top}{-}k\left(\{\mathrm{Corr}_\tau(Q, K)\}_{\tau=1}^{L}\right),$$

$$\mathrm{AutoCorr}(Q, K, V) = \sum_{\tau\in\mathcal{P}} \mathrm{Softmax}(\mathrm{Corr}_\tau(Q, K)) \cdot \mathrm{Shift}(V, \tau).$$

This time-delay aggregation allows the model to emphasize periodic temporal structures, enabling efficient long-term forecasting with sub-quadratic complexity compared to self-attention. For further details, find the original paper [15].

3.2.2 FEDformer (Fourier Enhanced Decomposition Transformer)

FEDformer combines series decomposition with frequency-domain modeling. As detailed in [16], each sequence is decomposed into trend and seasonal components:

$$\mathbf{x} = \mathbf{t} + \mathbf{s}, \qquad \mathbf{t} = \mathrm{MA}(\mathbf{x}), \quad \mathbf{s} = \mathbf{x} - \mathbf{t}.$$

To capture global periodic patterns, the seasonal component is transformed using a discrete Fourier transform:

$$\hat{\mathbf{s}}_k = \sum_{t=0}^{T-1} \mathbf{s}_t \, e^{-2\pi i k t/T},$$

and FEDformer retains only a truncated subset of dominant frequencies $\mathcal{K}$ through frequency sparsification. Self-attention is applied over this compressed frequency representation instead of the full time-domain series, reducing complexity while preserving long-range dependencies. The seasonal output is reconstructed via an inverse Fourier transform:

$$\tilde{\mathbf{s}}_t = \sum_{k\in\mathcal{K}} \hat{\mathbf{s}}_k \, e^{2\pi i k t/T}.$$

3.2.3 Informer

Informer [17] introduces ProbSparse self-attention, which exploits the observation that only queries with large attention magnitudes contribute significantly to the output. Given the standard scaled dot-product attention

$$\mathrm{Att}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right) V,$$

Informer selects a sparse set of dominant queries $\mathcal{U}$ by keeping only those with top-u largest ℓ_1-norm scores:

$$\mathcal{U} = \mathrm{Top}{-}u\left(\|Q\|_1\right),$$

reducing the attention computation from $O(L^2)$ to $O(L \log L)$. Attention is then computed only on this subset, yielding the ProbSparse operator:

$$\mathrm{Att}_{\mathrm{prob}}(Q, K, V) = \mathrm{softmax}\left(\frac{Q_{\mathcal{U}} K^\top}{\sqrt{d}}\right) V.$$

Informer further employs a generative-style decoder that progressively predicts future points, enabling efficient long-horizon forecasting.

3.2.4 TimesNet

TimesNet [18] models each univariate series by reshaping it into a set of 2D temporal patches and applying period-specific convolutional kernels. For a given period p, the sequence $\mathbf{x}$ is segmented into patches $\mathbf{X}^{(p)}$ and processed by a 2D convolutional block:

$$\mathbf{h}^{(p)} = \mathrm{Conv}_p\left(\mathbf{X}^{(p)}\right),$$

producing multi-period features that serve as learned temporal basis functions. These features are then aggregated across all considered periods $\mathcal{P}$:

$$\mathbf{z} = \sum_{p \in \mathcal{P}} W_p \, \mathbf{h}^{(p)},$$

enabling the model to capture complex multi-frequency and long-range temporal patterns in multivariate sequences.

3.2.5 LSTM (Long Short-Term Memory)

The Long Short-Term Memory (LSTM) network [19] is a recurrent neural architecture designed to capture long-range temporal dependencies by incorporating an explicit memory cell with gated updates. At each time step t, given an input $\mathbf{x}_t$ and the previous hidden and cell states $(\mathbf{h}_{t-1}, \mathbf{c}_{t-1})$, the LSTM computes input, forget, and output gates:

$$\mathbf{i}_t = \sigma(W_i \mathbf{x}_t + U_i \mathbf{h}_{t-1} + \mathbf{b}_i), \qquad \mathbf{f}_t = \sigma(W_f \mathbf{x}_t + U_f \mathbf{h}_{t-1} + \mathbf{b}_f),$$
$$\mathbf{o}_t = \sigma(W_o \mathbf{x}_t + U_o \mathbf{h}_{t-1} + \mathbf{b}_o),$$

together with a candidate cell update:

$$\tilde{\mathbf{c}}_t = \tanh(W_c \mathbf{x}_t + U_c \mathbf{h}_{t-1} + \mathbf{b}_c).$$

The memory cell and hidden state evolve according to

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t, \qquad \mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t),$$

where $\sigma(\cdot)$ denotes the logistic sigmoid and $\odot$ element-wise multiplication. This gated structure mitigates vanishing-gradient effects and enables LSTMs to model nonlinear, long-range temporal patterns in sequential data.

3.2.6 ETS (Holt–Winters Exponential Smoothing)

The ETS framework models a time series through latent level, trend, and seasonal components, providing an interpretable classical baseline. In the additive Holt–Winters formulation, the observation is decomposed as:

$$y_t = \ell_t + b_t + s_t.$$

The components are updated recursively using exponential smoothing:

$$\ell_t = \alpha \, (y_t - s_{t-m}) + (1 - \alpha)(\ell_{t-1} + b_{t-1}),$$
$$b_t = \beta \, (\ell_t - \ell_{t-1}) + (1 - \beta) b_{t-1},$$
$$s_t = \gamma \, (y_t - \ell_t) + (1 - \gamma) s_{t-m},$$

where α, β, and γ are smoothing parameters and m is the seasonal period. The h-step-ahead forecast is:

$$\hat{y}_{t+h} = \ell_t + h \, b_t + s_{t-m+h}.$$

3.3 Regression models

Regression models treat prediction as a supervised learning problem, in which an output variable is modeled as a function of an input feature vector. These approaches assume that the relevant information can be summarized through a set of explanatory variables, and they learn the mapping from features to responses using statistical estimation or ensemble-based procedures. Parametric models, such as linear regression, impose a global functional form on the relationship, while non-parametric methods, such as tree ensembles, capture more general non-linear dependencies. This formulation provides a flexible and widely used framework for predictive modeling across diverse domains.

3.3.1 Linear Regression

Linear regression models a real-valued response as an affine function of an input feature vector. Given $\mathbf{z} \in \mathbb{R}^p$, the model assumes

$$\hat{y} = \beta_0 + \boldsymbol{\beta}^\top \mathbf{z},$$

where $\beta_0 \in \mathbb{R}$ is an intercept term and $\boldsymbol{\beta} \in \mathbb{R}^p$ are coefficients. These parameters are typically estimated by minimizing the empirical squared-error objective over a training dataset:

$$\min_{\beta_0, \boldsymbol{\beta}} \sum_{i=1}^{n} (y_i - \beta_0 - \boldsymbol{\beta}^\top \mathbf{z}_i)^2.$$

This yields a parametric, globally linear predictor with a closed-form solution under mild assumptions, offering interpretability through the sign and magnitude of the learned coefficients and serving as a standard baseline for regression tasks.

3.3.2 Random Forest

Random Forests generalize this idea by replacing the single linear mapping with an ensemble of decision trees, each partitioning the feature space into regions and assigning a constant prediction within each region. A single regression tree implements a piecewise-constant function

$$T_m(\mathbf{z}) = \sum_r c_{m,r} \, \mathbb{I}\{\mathbf{z} \in \mathcal{R}_{m,r}\},$$

where $\{\mathcal{R}_{m,r}\}$ are regions defined by axis-aligned splits on the features and $c_{m,r}$ are leaf predictions. A Random Forest averages the outputs of M such trees trained on bootstrapped samples and random feature subsets:

$$g_\theta(\mathbf{z}) = \frac{1}{M} \sum_{m=1}^{M} T_m(\mathbf{z}).$$

This construction yields a flexible, non-parametric estimator that can approximate complex non-linear relationships between telemetry and latency, while being relatively robust to noise and overfitting.

3.3.3 XGBoost (eXtreme Gradient Boosting)

XGBoost fits an additive model of regression trees $f_t \in \mathcal{F}$ by minimizing a regularized empirical risk:

$$\hat{y}_i = \sum_{t=1}^{T} f_t(x_i), \qquad \min_{\{f_t\}} \sum_{i=1}^{n} \ell(y_i, \hat{y}_i) + \sum_{t=1}^{T} \Omega(f_t),$$

with a common tree regularizer $\Omega(f) = \gamma \, |\mathcal{L}| + \frac{\lambda}{2} \sum_{j \in \mathcal{L}} w_j^2$ (leaves $\mathcal{L}$, leaf weights w_j). At step t, it uses a 2nd-order Taylor expansion around $\hat{y}_i^{(t-1)}$:

$$\sum_{i=1}^{n} \ell\Big(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)\Big) \approx \sum_{i=1}^{n} \Big(g_i f_t(x_i) + \tfrac{1}{2} h_i f_t(x_i)^2\Big),$$

where $g_i = \partial_{\hat{y}} \ell(y_i, \hat{y}_i)$ and $h_i = \partial^2_{\hat{y}} \ell(y_i, \hat{y}_i)$. For a fixed tree structure, each leaf j has aggregate $G_j = \sum_{i \in I_j} g_i$, $H_j = \sum_{i \in I_j} h_i$, and optimal leaf weight

$$w_j = -\frac{G_j}{H_j + \lambda},$$

yielding the split gain used for greedy tree growth:

$$\text{Gain} = \frac{1}{2} \left(\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G^2}{H + \lambda} \right) - \gamma.$$

3.3.4 CatBoost (Categorical Boosting)

CatBoost is also gradient boosting over trees $\hat{y}_i = \sum_{t=1}^{T} f_t(x_i)$ minimizing $\sum_i \ell(y_i, \hat{y}_i)$, but it is designed to reduce target leakage and prediction shift with ordered (permutation-based) techniques. For categorical features it uses ordered target statistics computed along a random permutation π:

$$\text{TS}_k(i) = \frac{\sum_{j:\ \pi(j) < \pi(i),\ x_{j,k} = x_{i,k}} y_j + a\, p}{\sum_{j:\ \pi(j) < \pi(i),\ x_{j,k} = x_{i,k}} 1 + a},$$

where p is a prior (e.g., global mean), $a > 0$ is a smoothing strength, and only "past" examples in the permutation contribute. For boosting, CatBoost uses ordered boosting: when computing gradients for example i, it conceptually uses a model trained only on earlier items in the permutation:

$$g_i^{(t)} = \frac{\partial}{\partial \hat{y}} \ell(y_i, \hat{y}) \Big|_{\hat{y} = \hat{y}_{<i}^{(t-1)}},$$

with $\hat{y}_{<i}^{(t-1)}$ denoting the prediction from a model built without "future" examples relative to i in π.

4 Integration of the Learning Plane

4.1 Overview

Table 3 shows the summary of the usage of LP in the use cases. Each use case has provided a management requirement (*e.g.,* resource allocation, service autoscaling, service migration etc.), how AI can be used for the use case (*e.g.,* predict application latency, predict job execution time etc), and which dataset and models have been used to solve the management issue.

In the following sections, each use case introduces the problem requiring AI, and the data collected for model training, and detailed data preprocessing and model training and evaluation.

4.2 Usecase: Mobility

4.2.1 Problem Definition

Learning Plane has been enabled in a Mobility Usecase, empowering Cellnex businesses undergoing digital transformation to achieve higher operational efficiency. In specific, **we present ML models to predict QoS of Cellnex Video-Analytics application service in a Cloud-Edge continuum**. This model can help Cellnex to anticipate the QoS of the service and do proactive service migration between Cloud and Edge to achieve better service performance and saving costs.

4.2.2 Data Collection

By understanding the Cellnex user history pattern, we have emulated 8 days of workloads with a different distribution on Cellnex infrastructure to collect data. Each emulated day is repeated at least 10 times.

- Each day has a different workload distribution queue.
- The number of concurrent workloads ranges from 0 to 5, corresponding to the number of online cameras actively sending data to the VA application.

Table 3: Summary of the usage of LP in the use cases at M36.

Use case	AI usage	Dataset	Models
Mobility (Service Migration)	• **QoS-based migration of a video analytics application** in cloud-edge environments.	Dataset 1: Application QoS data(i.e., fps, latency), Application and cloud-edge resource usage data(i.e., CPU, memory usage) both 15/05/2024-05/06/2024. Dataset 2: Application QoS data(i.e., fps, latency), Application and cloud-edge resource usage data(i.e., CPU, memory usage) both 29/07/2025-06/10/2025.	The models **FlowState, Informer, Autoformer, FEDformer, TimesNet, LSTM, ETS, Linear Regression, Random Forest** are evaluated to predict the application latency QoS for Dataset 1 and 2.
Metabolomics (Resource Allocation)	• **Cost-driven autoscaling for batch inference on FaaS**, aimed at mitigating cold starts and reducing job completion times under a predefined budget, in contrast to reactive ECS-based scaling without cost control; and • **Privacy-preserving and latency-aware resource provisioning on-premises**, enabling smart pre-allocation of confidential containers.	2024-02-XX.cloudwatch.log; 2024-02_daemon.log; small.0.5k; small.1k; medium.3k; medium.6k; medium.8k; medium.15k; large.30k; large.35k; large.60k.	Exhaustive analysis of the training data using multiple models, including **FlowState**, **LSTM**, and **GRU**, indicates that the `OffSampleAI` workload is highly unpredictable. None of the tested models were able to reliably forecast either job arrival times or the number of requests per job. **Linear regression** is employed to estimate the total aggregated latency, enabling the online determination of the number of serverless executors per classification job, while ensuring that performance is maximized and the cost SLO is respected.
Surgery (Resource Allocation, Service Auto-scaling)	• **Smart resource allocation** of AI models at the Edge. • **Predictive auto-scaling** of streaming infrastructure.	Telemetry collection of GPU/CPU AI model utilization; NCT surgery room usage traces; Cholec80 dataset, GStreamer `videotestsrc`	Multi-dimensional **bin-packing model** to allocate NCT AI models across servers. **LSTM** models trained with NCT surgery room utilization traces for auto-scaling Pravega instances.
Agriculture (Resource Allocation)	• **Automatic resource scaling management**, AI is used to estimate the computing resources required to execute an agricultural and environmental pipeline that combines sensor streams with geospatial datasets.	Agricultural (Agricultural data analysis - Campo de Cartagena). Environmental and infrastructure datasets from the agricultural and environmental dataspace (Experiment 1). Geospatial Sentinel data.	**XGBoost** was used to predict job execution time from input data metadata and resource configuration parameters. The performance was compared against **CatBoost** as an alternative gradient-boosting approach.

- 8 hours per day with 30-minute intervals between workload changes.
- Workload processing duration applies a random deviation between 10 and 60 seconds.

We have collected two datasets from Cellnex infrastructure that reflect different operating conditions and stages of the Cloud–Edge infrastructure.

Dataset 1: Baseline Telemetry Dataset The first dataset consists of the collected telemetry during normal operation of the system at the beginning of the project. It includes the multivariate time series of cluster-level metrics shown in Table 4, together with temporal features (e.g., time of day) extracted to capture periodic patterns in resource usage and latency. This dataset represents the standard configuration of the platform and serves as the primary source for training and validating the forecasting models.

Dataset 2: Reduced-Capacity Edge Scenario The second dataset captures a modified operational setting where the Edge server's computational capacity was intentionally reduced. Due to an upgrade and migration of the physical hardware, we observed that the Edge hardware used by CellNex was significantly more powerful than what is typical in Cloud–Edge deployments. To better reflect realistic conditions, we limited the number of available CPU cores and reduced their clock frequency. This produced a new dataset with different latency dynamics and resource constraints, requiring models to be retrained and adapted to this lower-capacity environment.

4.2.3 Data Preprocessing

The first objective of preprocessing is to transform these raw logs into clean, consistent time series that can be used for forecasting and decision making. We begin by harmonizing the input: selecting a fixed set of relevant metrics, enforcing consistent data types, and standardizing the time axis so that all records align on a common temporal grid. This step removes noisy or irrelevant fields and ensures that all experiments operate on the same, well-defined schema.

Next, we integrate the different sources of information. Pipeline-level records are enriched with node and server telemetry, and then aggregated at the cluster level so that each time step summarizes the overall load and performance of a given cluster. Missing values are handled systematically, and the result is a multivariate time series per cluster (e.g., every 30 seconds) that includes both the latency we want to predict and explanatory signals such as resource utilization and pipeline count, the features from which can be found in Table 4.

Table 4: Final Input features and target for pipeline latency forecasting with exact naming used.

Feature	**Description**
cluster	Processing cluster ID (categorical)
pipelines_status_avg_fps	Average frames per second
pipelines_status_avg_pipeline_latency	Historical avg. pipeline latency (s)
node_cpu_usage	CPU usage of the node (%)
node_mem_usage	Memory usage of the node (bytes)
pipelines_server_cpu_usage	Pipeline server CPU usage (%)
pipelines_server_mem_usage	Pipeline server memory usage (bytes)
number_pipelines	Number of active pipelines
Target	Description
pipelines_status_realtime_pipeline_latency	Real time latency of the pipelines being processed (seconds)

Data preparation For learning, these cluster-level time series are converted into supervised datasets. We split the data chronologically into training, validation and test segments to mimic a real deployment where models are trained on past behavior and evaluated on future unseen periods. On top of the mentioned features in Table 4, we construct sliding windows of recent history and corresponding

prediction horizons, normalize continuous features, and encode categorical variables (such as cluster identity), time features are also extracted based on the granularity of time-of-day so that both deep and classical models can be trained reliably.

In production and offline validation, the same pipeline is applied to short recent windows instead of the full history. Only windows that contain enough data, and no unresolved missing values, are kept, ensuring that the models always receive a meaningful and well-conditioned context for forecasting. Overall, this preprocessing and preparation stage turns heterogeneous raw logs into structured, standardized inputs that capture the key dynamics of the system while respecting temporal causality.

Data Preparation for FlowState FlowState uses the preprocessed time series, but requires additional considerations due to its long-term modeling of seasonal patterns. The input time series spans from 9am to 5pm, resulting in 960 time steps per day (8 hours × 60 minutes × 2 samples per minute). Time intervals outside of this range (5pm–9am) contain missing values, which are filled with NaNs. FlowState interprets these NaNs as missing data, ensuring that the slightly irregular time axis does not affect model inference. The length of the input windows and the seasonal structure are configured accordingly to capture daily patterns.

4.2.4 Model Training

We pursue two complementary modelling directions in this work. From a **time-series forecasting** perspective, we treat latency as a univariate or multivariate sequence and model its temporal dynamics explicitly. From a **regression** perspective, we focus on learning a direct mapping from aggregated telemetry and time features to future latency.

Time-Series Forecasting Perspective In the time-series view, we consider a family of neural sequence models for multivariate forecasting, including transformer-based architectures, frequency-domain models and lightweight linear variants as detailed in 3.2. Training relies on mini-batch gradient descent with an adaptive optimizer and early stopping on a held-out validation set. The standard mean squared error loss (MSE) is used as a main loss function, however, we have also explored custom loss functions such as an SLA-aware loss, seen in Equation 6, which increases the penalty for underestimating high-latency events and down weights small errors in low-latency regimes, encouraging the models to focus on accurately predicting potential SLA violations.

$$\mathcal{L}_{\text{SLA}} = \frac{1}{N}\sum_{i=1}^{N} w_i \cdot (y_i - \hat{y}_i)^2 \tag{6}$$

$$w_i = \begin{cases} 30 & \text{if } y_i > 200ms \text{ and } \hat{y}_i < 150 \\ 15 & \text{if } y_i > 200ms \\ 3 & \text{if } y_i < 100ms \\ 1 & \text{otherwise} \end{cases} \tag{7}$$

We use as input recent windows of cluster-level telemetry and produce short-term forecasts of pipeline latency over a few minutes horizon, using the same input–output format, normalization scheme and temporal splits to ensure a fair comparison. From a functional point of view, models learn a mapping

$$f_\theta : \mathbb{R}^{T\times d} \to \mathbb{R}^{H},$$

that maps a window of T past multivariate observations $\mathbf{X}_{t-T+1:t} \in \mathbb{R}^{T\times d}$ to a horizon-H forecast of future latency values $\hat{\mathbf{y}}_{t+1:t+H} \in \mathbb{R}^{H}$. We instantiate this framework with several model families, each grounded in a different theoretical view of temporal dependence.

All models were trained using a full node of the Accelerated Partition of the Marenostrum 5 [20], the latest supercomputer from the Barcelona Supercomputing Center, using identical data splits and optimization settings.

Regression Perspective with Time Features In the regression view, we compress recent history into feature vectors and predict future latency directly, without explicitly modelling the full temporal trajectory. For each prediction time t we build a feature vector $\mathbf{z}_t \in \mathbb{R}^p$ aggregating recent telemetry (e.g., mean CPU and memory usage, number of pipelines) and time-related attributes (e.g., hour of day, quarter of the hour, cluster identity), and learn a mapping

$$g_\theta : \mathbb{R}^p \to \mathbb{R}, \qquad \hat{y}_{t+1} = g_\theta(\mathbf{z}_t),$$

whose output $\hat{y}_{t+1}$ is then replicated over the short forecast horizon used by the decision logic.

As a baseline, we use a fixed-coefficient linear regression model that operates on a small set of intuitive cluster-level features (average node CPU and memory, number of pipelines, and a CPU interaction term between node and server CPU usage), defining separate coefficients for edge and cloud clusters. To capture more complex, non-linear dependencies, we also employ a Random Forest regressor using the same type of summary statistics and time features; both detailed in 3.3. Random Forest outputs a single latency prediction for the next interval, which we replicate over the short forecast horizon used by the decision logic. Empirically, this Random Forest has been the strongest performer among the regression-based models, substantially improving upon the linear baseline while preserving moderate complexity and offering insights into feature importance. It thus represents our main regression-oriented benchmark and a practical alternative to full sequence modelling.

4.2.5 Model Evaluation and Comparison

FlowState Results As mentioned in Section 4.2.2, we use FlowState to predict the realtime latency, represented as the feature *pipelines_status_realtime_pipeline_latency*. To do so, we provide the historic pipeline latency time series to FlowState and use it to make predictions of the future latency within a particular target window. We found that FlowState works best for this scenario when using all data available. Thus we provided all available datapoints for each channel separately, without considering the specific train/validation/test split. There are about 33.6k steps for an individual channel and FlowState would support up to 82k steps in this setting, thus we can provide the entire context and enable FlowState to "learn in-context" and find useful patterns for prediction.

Starting from the second day, to ensure some minimum context for the predictions, FlowState then produces forecasts from each consecutive context step. Since FlowState has been optimized for predictions of size $\geq$ the size of the seasonality, we configured FlowState to produce forecasts of 960 steps, which is equivalent to 1 day. Therefore, FlowState produces about 32.6k forecasts, each for a full day. We compared the performance of FlowState in this setting to a running median baseline, which produces a constant prediction for the given target window, that is computed based on the median of the last n context steps. We found that n in the range between 1 and 100 works best. The result comparison can be found in Figure 9. One can see that the predictions of FlowState match the ground-truth target much better in the long target scenario.

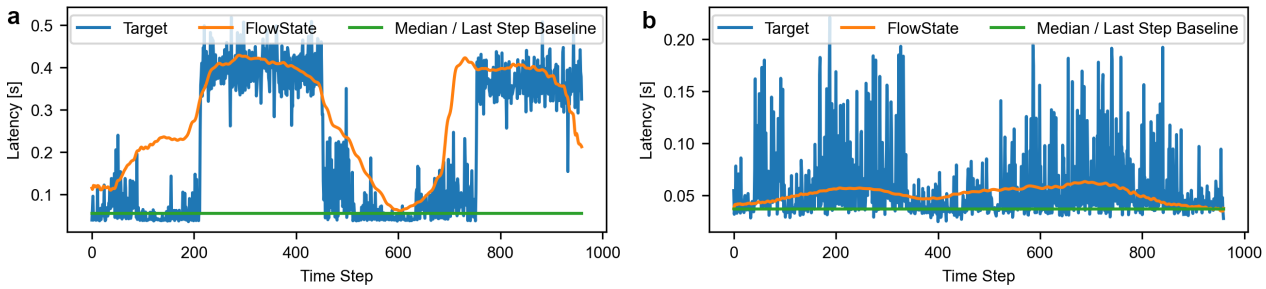


Figure 9: Prediction of the pipeline latency for 960 steps. **a** Prediction for the edge node and **b** for the cloud node.

Furthermore, we can also compare the performance for a shorter target window of 20 steps. The results can be found in Figure 10. We can observe that the data in this short forecasting horizon

becomes quite unpredictable and thus also the quality of the forecasts of the baseline and of FlowState degrades. As mentioned above, this is a general trend that we observe for this dataset, the shorter the prediction window is, the less information the predictions carry.

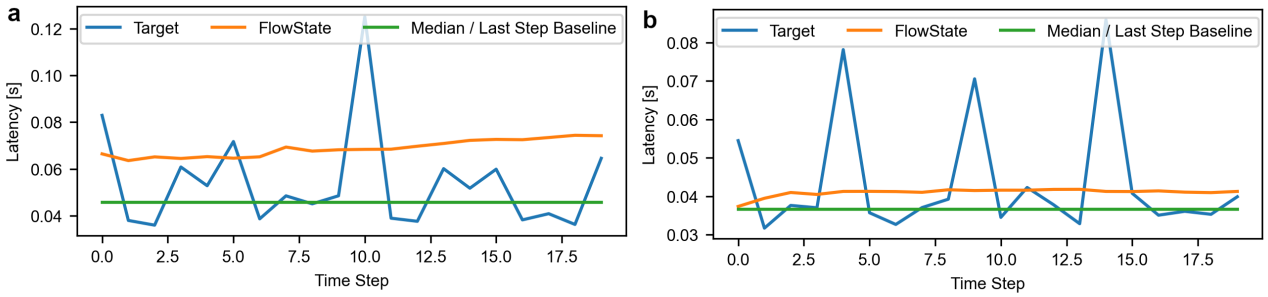


Figure 10: Prediction of the pipeline latency for 960 steps. **a** Prediction for the edge node and **b** for the cloud node.

Time Series Model Results We also tried other models we have trained, all models are evaluated on the held-out test set using the standard forecasting metrics: mean squared error (MSE), mean absolute error (MAE). Since the application requires an accurate prediction of latency spikes, and many of the points do not contain a lot of variation, we focus on MAE as the most representative indicator of both stability and short-term deviation. Table 5 summarizes the performance of the different architectures and sequence-length configurations explored.

Model Configuration	MSE	MAE
FEDformer (sl=60, ll=60, pl=20, dm=32)	0.00261	0.03349
Autoformer (sl=60, ll=60, pl=20, dm=32)	0.00453	0.04664
Informer (sl=120, ll=120, pl=20, dm=256)	0.00366	0.03942
Informer (sl=60, ll=60, pl=20, dm=256)	0.00394	0.03709
TimesNet (sl=60, ll=60, pl=20, dm=64)	0.00289	0.03325
TimesNet (sl=120, ll=120, pl=20, dm=64)	**0.00279**	**0.03176**

Table 5: Comparison of time-series forecasting models on the test set. Best models shown after hyperparameter tuning.

To ensure a fair comparison, each model was trained using a controlled hyperparameter search. We systematically varied key temporal parameters such as:

- **Sequence length (sl)**: 60 to 2880 time steps. Going from a few (30 minutes) to a large context (24 hours).
- **Label length (ll)**: 60 to 2880 time steps.
- **Prediction horizon (pl)**: fixed to 20 time steps. We focus on predicting the 5 min range of latency from the future 5-10 minutes.
- **Model dimensionality (dm)**: 32, 64, 256 depending on architecture.
- **Encoder/decoder depth**: commonly 2/1 or 3/1.
- **Feed-forward expansion (d_ff)**: 128 or 512.

These hyperparameters correspond to the internal settings used in our training scripts (e.g., `seq_len`, `label_len`, `d_model`, `d_ff`, and architectural depth values). All runs share the same optimizer (Adam, learning rate 0.001), batch size (32), patience (2), and 20 epochs.

Time Series Forecasting Result Interpretation Overall, the models show only modest differences in performance. TimesNet obtains the lowest errors in our experiments, followed by FEDformer and the two Informer variants, while Autoformer consistently performs the worst. However, all models face the same main limitation: they struggle to capture the sudden, short-term fluctuations in our latency data. These fast changes, often caused by unpredictable workload spikes or system noise, make accurate short-term forecasting very difficult, regardless of sequence length or model size. For this reason, even though the experiments allow us to compare the different architectures, none of the models provides predictions that are reliable enough for short-horizon decision making in our environment. The results reflect the challenging nature of the problem more than the quality of the models themselves, and they suggest that additional or alternative techniques may be needed to handle such highly variable QoS signals. An example forecast obtained with the best-performing time-series model (TimesNet) is shown in Figure 11.

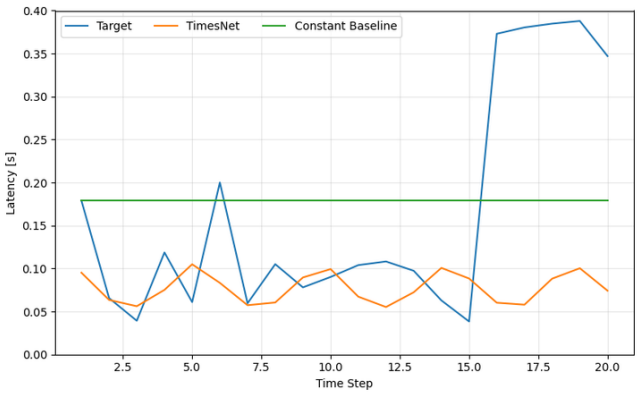


Figure 11: Example short-horizon latency forecast on the test set using the best-performing time-series model (TimesNet).

Regression Model Results As an alternative to sequence models, we evaluate regression approaches that predict future latency directly from aggregated telemetry and time features. All models are trained and tested on the same feature–target pairs, and evaluated using mean squared error (MSE) and mean absolute error (MAE) on the held-out test set. Table 6 summarizes the performance of the linear baseline and three non-linear tree-based regressors.

Model	MSE	MAE
Linear Regression (baseline, averaged edge/cloud)	0.0035	0.0341
Decision Tree Regressor	0.0029	0.0266
Random Forest Regressor	**0.0028**	**0.0266**
Gradient Boosting Regressor	0.0028	0.0272

Table 6: Comparison of regression models on the test set using aggregated telemetry and time features.

Regression Result Interpretation The linear regression baseline, built on a small set of engineered features, significantly reduces error compared to naive predictions but remains limited by its global linear form. Tree-based methods that model non-linear interactions between telemetry and time features achieve consistently lower errors, with the Random Forest regressor providing the best overall performance in terms of both MSE and MAE. An example short-horizon forecast using the Random Forest model is shown in Figure 12. These results indicate that, in our setting, relatively simple regression models with well-chosen features can match or outperform more complex sequence models for short-horizon latency prediction, while remaining computationally efficient and interpretable

enough for deployment.

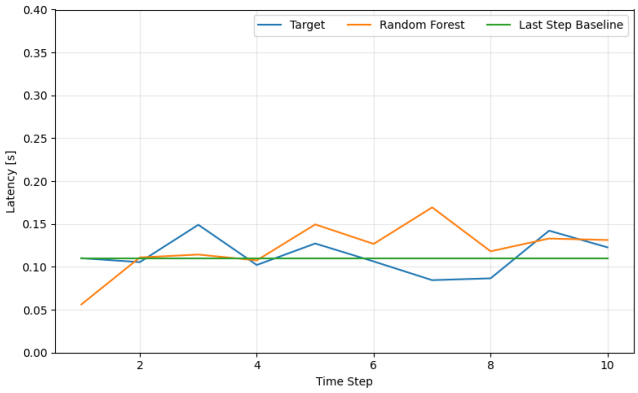


Figure 12: Example short-horizon latency forecast on the test set using the best-performing regression model (Random Forest).

4.3 Usecase: Metabolomics

4.3.1 Problem Definition

As discussed in preceding deliverables, the current production-grade `METASPACE` inference pipeline leverages AWS ECS to perform image classification in a serverless environment. While this solution ensures scalability, the `METASPACE` DevOps team realized soon that it was not fully cost-efficient due to several contributing factors.

On the one hand, AWS ECS provides no mechanism to specify a **cost-based SLO**, meaning that it is not possible to provision instances while explicitly enforcing a pre-defined budget. On the other hand, the scaling strategy for `OffSampleAI` instances relies on **reactive feedback-control autoscaling**, in which AWS ECS continuously monitors running instances and adjusts their number based on an observed metric like average CPU utilization. In the current deployment of the `OffSampleAI` service, a single instance remains active at all times. When the average CPU utilization surpasses 80%, the auto-scaler triggers the deployment of four additional instances, up to a maximum capacity of nine. Due to this reactive design, scaling decisions occur only after load increases have already observed. As a result, the platform experiences substantial instance provisioning delays when serving highly variable workloads such as those in the Metabolomics use case. This, in turn, leads to unnecessarily long job completion times and further exacerbates the overall cost inefficiency.

These limitations motivate the first challenge **(C1)**: the need for a more **intelligent**, cost-driven solution that can dynamically adjust resource allocation while maintaining high performance.

However, this is not the only challenge. In parallel, for confidential image datasets, an alternative inference pipeline has been implemented on an **on-premises Kubernetes** cluster by re-engineering `Lithops` Serve with **confidential containers** using SCONE. Since this cluster is managed directly by the `METASPACE` platform, the cost of running inference in the public cloud is no longer a limiting factor. In this scenario, the challenge shifts from controlling cloud costs to ensuring data privacy **(C2)**. As a result, a complex AI-driven solution was not required; instead, a simple heuristic proved sufficient. Specifically, we profiled both the batch processing time and the confidential-container startup latency, and used these measurements to provision the required number of instances to satisfy a relaxed SLO. Further implementation details are provided in D5.4.

In this deliverable, we present our approach for automatically extracting workload traces from the pre-project `OffSampleAI` service, and we show how this workload exhibited high variability that rendered predictive scaling ineffective. Built upon these insights, we shifted to a simpler and more robust solution grounded in parametric regression models to estimate the number of executors that can be provisioned within a predefined budget.

4.3.2 Data Collection

Building on D5.2, we briefly summarize the data collection process, avoiding redundancies to focus on the final resource provisioning approach. The collection process spanned a period of five months, from January 1, 2024, to May 31, 2024, containing a total number of 9,298 jobs and +35 million image requests.

In a nutshell, the training data was collected by interfacing with the `METASPACE` platform daemons, combining multiple sources. In particular, we instrumented the daemon managing the `OffSampleAI` service. This daemon receives notifications from the prior stage of the `METASPACE` annotation pipeline and handles off-sample classification jobs. Each job contains a variable number of `.png` images, which are submitted to the AWS ECS service for classification.

The `OffSampleAI` daemon processes up to four datasets concurrently, with one dataset per off-sample thread. Images are grouped into batches of 32 and sent to the ECS HTTP endpoint, with each thread managing up to eight synchronous batches to avoid overloading the containers.

From this daemon and AWS CloudWatch, we extracted the following set of training files:

- **YYYY-MM_datasets**: Information from `METASPACE` regarding dataset ID, image resolution (x, y), number of annotated molecules, number of generated images and public/private status as shown in Table 7.

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

Table 7: 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 8.

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 8: Start and finish times for the datasets.

- **YYYY-MM_daemons**:These files are generated by the `OffSampleAI` daemon, internally called '`update-dameon`', and contain information about the actual start and end times of classification jobs. An example of the file format is shown below:

```
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 files are a number of logging 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 9 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 9: Log entry details.

The above files were subsequently processed and consolidated to support further analysis and to provide structured input for model training. More concretely, we generated three types of data files: *datasets*, *batches*, and *metrics*. The fields contained in each file, along with their descriptions, are listed in Tables 10, 11, and 12, 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 `OffSampleAI` 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 10: 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 11: Batch information.

The consolidated dataset serves two primary objectives:

- **Comparison of implementations**: to evaluate latency, speedup, cost, and performance-per-dollar between the previous and current system implementations.
- **Workload forecasting**: to train an AI model capable of anticipating activity spikes, dynamically adjusting the function pool, mitigating cold starts, and pre-allocating on-premises resources, including containers.

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 12: Metric information.

4.3.3 Model Training

After a detailed analysis of the training files using several established time-series forecasting models, including LSTM, GRU, and Prophet, we observed that the `OffSampleAI` workload is **unpredictable**, both in terms of **job arrival times** and **the number of requests per job**. This extreme unpredictability results from a combination of heavy-tailed job sizes, variable inter-job arrival times, and extended idle periods, producing workloads with no recurrent patterns.

The next section demonstrates that even advanced time-series forecasting approaches, such as FlowState built on foundation models, failed to accurately predict the number of requests per time bin. To this end, we utilized the information contained in the `n_images` and `started` fields.

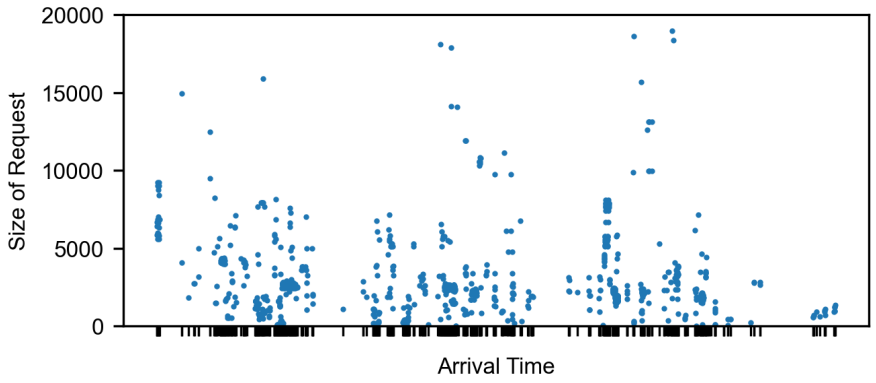


Figure 13: Example of the data showing the size of the requests at each time point.

Data Preprocessing for FlowState Figure 13 shows an example of number of requests per time. As one can see, the data points are not spread equally over time, but FlowState has been pretrained with equally-spaced data. Thus, we first have to preprocess the data.

Since our goal was to predict the number of incoming requests over a certain period with a time series foundation model, the pairs of (`n_images`, `started` times) needed to be binned, transforming them into an equally spaced time series with granularity determined by the size of the time bins. For example employing a bin size of 1 hour the data is transformed into a time series of length $\sim 150 \cdot 24$. Figure 14 shows an example of the binned data.

Since in most hours no requests have arrived, this time series is relatively sparse, consisting of mostly zeros. Using larger time bins (for example 12 hours per bin) changes this property, and the time series becomes less sparse, but on the other hand the entire data is now a single series of length 300, making it harder to evaluate due to the small size.

Forecasting the number of incoming requests with FlowState As a first step, we tried to forecast the number of incoming request for a certain period of time. To quantify forecasting performance, we split the resulting time series into context/target pairs of size 512/96 for the hourly and 4 hourly

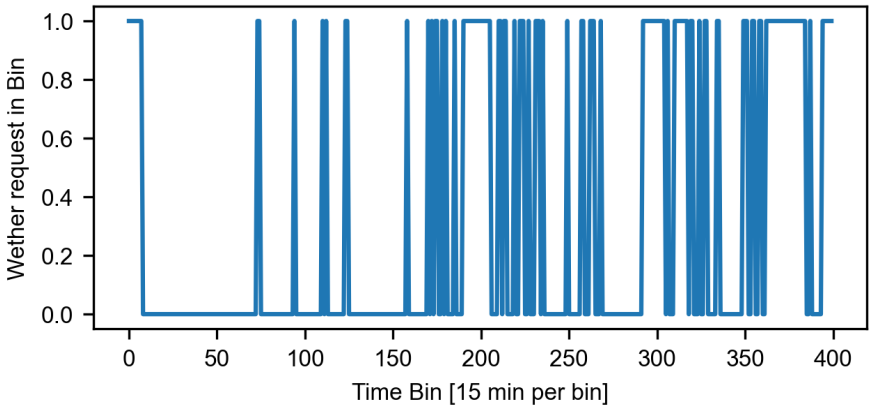


Figure 14: Example of the binned data, where a value of 1 indicates that at least one request was received during the last 15 minutes.

binned data. For 8/12h bin size, we had to reduce the context length, because there were not enough time points.

The forecasts were evaluated with the mean squared error (MSE) and mean absolute error (MAE), averaged over all context/target pairs we could construct from the data. As baselines, we selected 3 very simple forecasting approaches: a zero predictor, a mean predictor and a median predictor. These three baselines receive the same context as FlowState, but all three predict a constant value for the entire target sequence. The zero predictor constantly predicts zero, the mean predictor constantly predicts the mean of the context, and the median predictor the median of the context.

Overall, we found that the data is very difficult to predict. Visually looking at the data there does not seem to be consistently reoccurring patterns and most of it looks unpredictable. Our forecasting results also support this assessment, given at best small improvements over very simple baseline forecasting strategies.

If we use small time bins, 1h or smaller the data becomes very sparse such that always predicting zero becomes the best strategy. For larger bins, FlowStates performance becomes better, but still, we cannot consistently beat the mean/median predictors. The following table summarizes the results of FlowState vs the three baselines for varying time in sizes.

Table 13: Initial Results: MSE / MAE for FlowState and Baselines at Different Bin Sizes

Bin Size	FlowState	Zero-Predictor	Mean-Predictor	Median-Predictor
1 hour	322 / 18.5M	**311** / 19.7M	440 / **16.9M**	312 / 19.7M
4 hours	**1181** / 159M	1240 / 193M	1506 / **146M**	1189 / 170M
8 hours	2418 / 470M	2521 / 612M	2771 / 446M	**2308** / 477M
12 hours	**3736** / 1040M	4459 / 1471M	4199 / **946M**	3770 / 1040M

Forecasting the presence of an incoming requests with FlowState A second attempt at forecasting consisted of ignoring the number of requests per dataset, and instead forecast whether or not there will be a new job arriving within the time frame of 15 minutes. This time frame corresponds to the timeout limit of the serverless functions (AWS Lambda).

Predicting whether an event happens within a certain time frame is not directly how FlowState was pretrained. Nevertheless, there are ways in which we can apply FlowState to perform this task. The approach we tried was to bin the data in time steps of 15 minutes, and transforming the series to be '1' if there was a job arrival in the corresponding bin and else '0'. The task then translates to forecasting only 1 time step.

To assess how accurate the predictions were we used the MAE, which resembles the loss function FlowState was trained on, but we also calculated the **accuracy** by interpreting all forecasts $>= 0.5$ as `True` and < 0.5 as `False` and then calculating the accuracy (how many percentages of 15 min. intervals we guessed correctly. Overall, in the data there were $\sim 25\%$ of bins in which a job arrived. Therefore, always guessing 'no job' would result in accuracy 75%. Since multiple jobs sometimes occur in close temporal proximity, another reasonable baseline is to predict that there will be a job in the next time bin, if there was one in the current bin. Expanding the baseline to include more context and forecast incoming jobs, if it was the case for a majority of previous 15 min. time windows in the context only decreased baseline performance. The result of the strongest baseline is 81.2% accuracy and FlowStates accuracy using all of the available context is 81.6%. You can find the confusion matrix, accuracy, precision, recall and F1 scores summarized in the table below.

Table 14: Confusion matrix and metrics for FlowState vs Baseline.

Actual	**FlowState**		**Baseline**	
	Pred. Positive	Pred. Negative	Pred. Positive	Pred. Negative
Positive	941	1082	1261	762
Negative	409	5664	762	5311
Accuracy	81.6%		81.2%	
Precision	69.7%		62.3%	
Recall	46.5%		62.3%	
F1 Score	27.9%		31.2%	

Overall, the results stay the same: the workload appears chaotic, with few discernible patterns, and performance improvement over simple baselines are minimal. As shown in the confusion matrix, FlowState struggles most with **false negatives**. Lowering the prediction threshold can help balance this trade-off. In the following table, a threshold of 0.3 yields more balanced predictions, though with a slight reduction in overall accuracy.

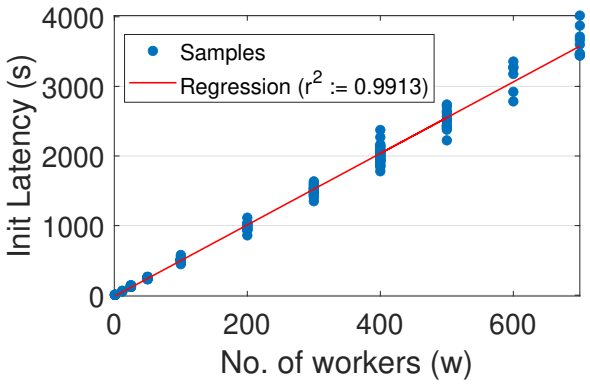
Table 15: Confusion matrix and metrics for FlowState with threshold 0.3

	Predicted Positive	**Predicted Negative**
Actual Positive	1249	774
Actual Negative	766	5307
Accuracy	80.98%	
Precision	61.99%	
Recall	61.74%	
F1 Score	30.93%	
MAE	0.209	

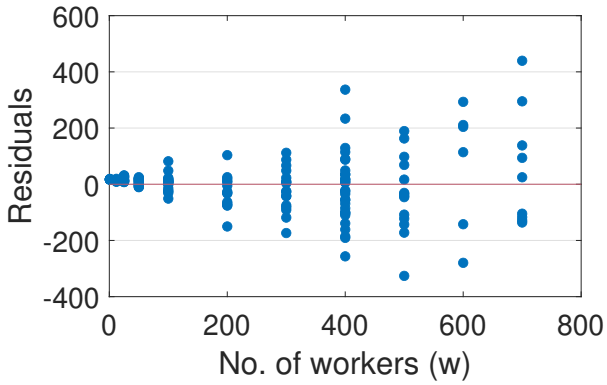
4.3.4 Model Comparison and Evaluation

Given the inherent unpredictability of the workload, we reformulated the initial strategy. As outlined above, our goal was to design an online heuristic capable of determining the appropriate number of serverless executors for a classification job in order to maximize performance while respecting budget constraints. To achieve this, we developed an estimator of the total aggregated latency, enabling us to approximate the associated cost and ensure that the cost SLO is not violated at any time.

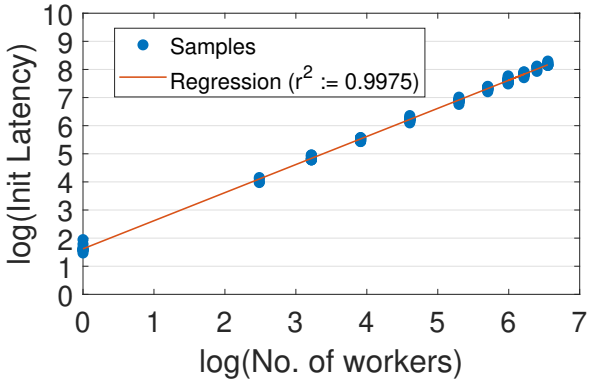
More formally, let $T(w, r)$ denote the total aggregated latency incurred by the w executors in order to process the r inference requests in the job. That is, $T(w, r) = \sum_{i=1}^{w} T_i(r_i)$, where $T_i(r_i)$ is the latency contributed by executor i in processing r_i requests. Note that $\sum_{i=1}^{w} r_i = r$. Fortunately, we found that $T(w, r)$ can be accurately approximated via regression using a small number of samples, and that



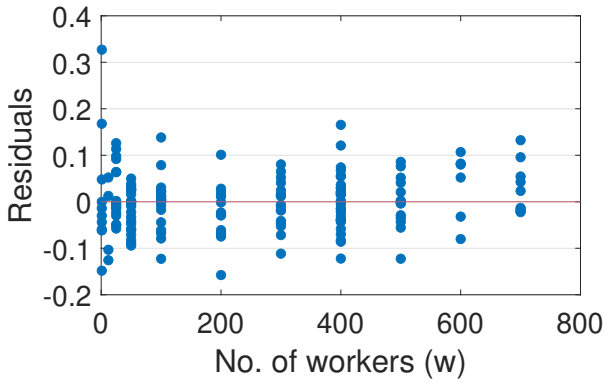
(a) Init time ($T_{\text{Init}}(w)$).



(b) Residual plot of Init time.



(c) Log-log Init time.



(d) Residual plot of Log-log Init time.

Figure 15: Linear regression for the **Init** phase using the classification CNN model [22] used by the `OffSampleAI` service.

this approximation can subsequently be used to estimate the cost per request (CPR). We describe this regression approach in what follows.

Estimation of aggregated latency. To approximate $T(w, r)$ accurately, we needed to account for the two components of the total aggregated latency in Function-as-a-Service (FaaS) platforms: (a) the **Init** phase and (b) the **Invoke** phase [21]. In what follows, we describe the online parametric regression process used to estimate both phases for AWS Lambda.

Init phase. The Init phase creates the execution environment for our executors by downloading the function code and dependencies and starting the chosen runtime. This phase takes place only during a "cold start". Importantly, the aggregated Init latency depends solely on the number of workers and not on the number of requests. After analysis, we found that the aggregated **Init** latency, $T_{\text{Init}}(w)$, can be accurately approximated with univariate linear regression:

$$T_{\text{Init}}(w) = b_0 + b_1 w + \varepsilon, \tag{8}$$

where $b_0, b_1 \in \mathbb{R}$ are learned parameters minimizing the residual sum of squares (RSS). Figure 15a shows the fitted linear model for the classification CNN model [22] used by the `OffSampleAI` service. Despite a high coefficient of determination ($r^2 := 0.9913$), the residuals (ε) display increasing variance with larger w, as seen in Figure 15b, indicating heteroscedasticity. To overcome this, we applied a log-log transformation:

$$\log T_{\text{Init}}(w) = b_0 + b_1 \log w + \varepsilon. \tag{9}$$

This transformation improved both prediction accuracy ($r^2 := 0.9975$) and residual behavior, as shown in Figure 15c and Figure 15d.

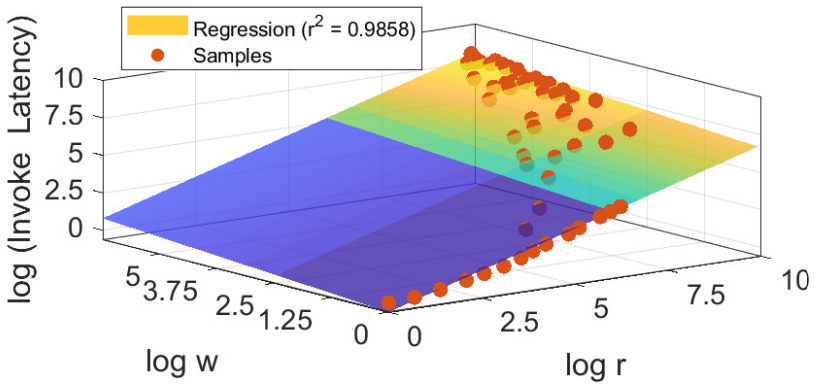


Figure 16: Log-log transformation of the Invoke latency (Eqn (11)) using ResNet50.

Invoke phase. The Invoke phase executes the function code after initialization completes, which includes the code of the `Lithops` Serve executors. The executors perform three operations: fetching batches of images from object storage, preprocessing them, and storing batch inference results back to object storage. Once the final batch is processed, the executors terminate. The **Invoke** latency thus encompasses the time from when the serverless executor starts until the completion of the last batch. The cumulative **Invoke** latency, $T_{\text{Invoke}}(w, r)$, can also be well approximated by linear regression:

$$T_{\text{Invoke}}(w, r) = a_0 + a_1 w + a_2 r + \varepsilon, \tag{10}$$

with parameters $a_0, a_1, a_2 \in \mathbb{R}$. The dependency on executor count emerges because executors start asynchronously, creating workload imbalance. Similar to initialization latency, heteroscedasticity is present; we corrected it via log-log transformation:

$$\log T_{\text{Invoke}}(w, r) = a_0 + a_1 \log w + a_2 \log r + \varepsilon. \tag{11}$$

Fig. 16 shows the improved fit after transformation.

Combining phases. By combining Eqns. (8) and (10), the total latency $T(w, r)$ can be approximated as:

$$\begin{aligned} T(w, r) &\approx e^{\log T_{\text{Init}}(w)} + e^{\log T_{\text{Invoke}}(w,r)} \\ &\approx A \cdot w^{b_1} + B \cdot w^{a_1}, \end{aligned} \tag{12}$$

where $A = e^{b_0}$ and $B = e^{a_0} \cdot r^{a_2}$.

Cost estimation. From Eqn. (12), the total cost of an inference job is given by

$$C(w, r) \approx T(w, r) \cdot p(m),$$

where $p(m)$ denotes the price of a Lambda instance with memory size m ($/GB-s). We recall that AWS Lambda bills execution time based on the amount of memory allocated to the function and the total duration of its execution. When a function is invoked, AWS charges for every millisecond of runtime from the moment the code starts executing until it returns or terminates, rounded up to the nearest millisecond. The cost per millisecond increases with the configured memory size, as higher memory allocations automatically provide proportionally more CPU, network, and I/O resources. Consequently, selecting the appropriate memory configuration is a key performance–cost trade-off: allocating more memory can significantly reduce execution time, but at a higher per-millisecond rate, while lower memory settings reduce the rate but may lead to slower execution and increased overall cost. This trade-off is precisely what the new implementation of the `OffSampleAI` service leverages to achieve higher efficiency and better cost–performance balance.

Finally, we computed the empirical cost per request as:

$$\hat{\mathrm{CPR}}(w,r) \approx \frac{C(w,r)}{r} \approx \frac{p(m)}{r} \cdot \left(A \cdot w^{b_1} + B \cdot w^{a_1}\right). \tag{13}$$

The estimated cost per request derived in Eqn. (13) is used as the primary decision metric in the `Lithops` Serve auto-scaler. For any incoming batch job of size r, the auto-scaler evaluates $\hat{\mathrm{CPR}}(w,r)$ across the feasible range of w and selects the smallest number of executors that satisfies the target CPR SLO. This ensures that `Lithops` Serve provisions executors dynamically while guaranteeing that no configuration exceeds the allowed CPR. As detailed in D5.4, this AI-driven scaling strategy allows `OffSampleAI` to balance performance and budget constraints efficiently, replacing "purely" reactive heuristics with a principled, cost-aware approach.

We observe that although Eqn (13) is a sum of power functions in w with no general closed-form solution, the optimal number of workers can be efficiently found using binary search. The time complexity is $\mathcal{O}\left(\log \min\left(W_{\max}, \lceil r/b \rceil\right)\right)$, very small since $W_{\max}$ is limited to a thousand concurrent Lambda executions.

4.4 Usecase: Surgery

4.4.1 Problem Definition

The National Center for Tumor Diseases (NCT) requires a compute continuum capable of supporting real-time AI-assisted surgical workflows, where latency and resource efficiency are critical. Endoscopic video streams must be processed at high frame rates (*e.g.*, $\geq$ 30 FPS) for AI models performing instrument detection, phase recognition, and segmentation. Moreover, endoscopic video data should be durably stored for downstream analytics while supporting fluctuating workloads related to the daily activity of surgery rooms. This dual requirement introduces two major challenges:

- **Smart Resource Allocation (C1):** Surgical AI workloads exhibit highly heterogeneous resource demands, combining GPU-intensive segmentation models with lighter detection tasks. These workloads must run concurrently on limited edge hardware without violating strict real-time guarantees. The challenge lies in efficiently allocating CPU and GPU resources to maximize utilization while preventing oversubscription and ensuring that latency-sensitive models maintain their required frame rates.
- **Predictive Streaming Auto-Scaling (C2):** Video ingestion and storage systems must adapt to significant workload fluctuations driven by operating room schedules. Scaling the streaming infrastructure reactively often causes *latency spikes during reconfiguration*, which can disrupt real-time video analytics. The challenge is to anticipate workload variations and adjust resources proactively to maintain stable ingestion performance and avoid Service Level Objective (SLO) violations during critical surgical procedures. The challenge for the LP is to drive predictive auto-scaling decisions on Pravega segment store instances by consuming metrics (*e.g.*, write latency, number of segment stores published via Prometheus), along with other relevant system metrics (*e.g.*, number of active surgery rooms). Importantly, Pravega segment store instances can be horizontally scaled through Kubernetes APIs.

4.4.2 Data Collection

C1 – Smart CPU & GPU Allocation For the CPU & GPU bin-packing challenge, telemetry was collected from the Kubernetes-based surgical edge cluster running NCT AI inference pipelines (instrument detection, phase recognition, liver segmentation). GPU metrics - including utilization percentage, memory consumption, and power draw - were retrieved via direct `nvidia-smi` queries executed within pods. CPU usage per pod was monitored through `kubectl top` and pod resource request specifications, while frame rate (FPS) per stream was measured via GStreamer appsink callbacks to ensure compliance with real-time Service Level Objectives ($\geq$30 FPS). GPU allocation was controlled through custom per-GPU resources (`nvidia.com/gpu-index`) with time-slicing configurations (up to

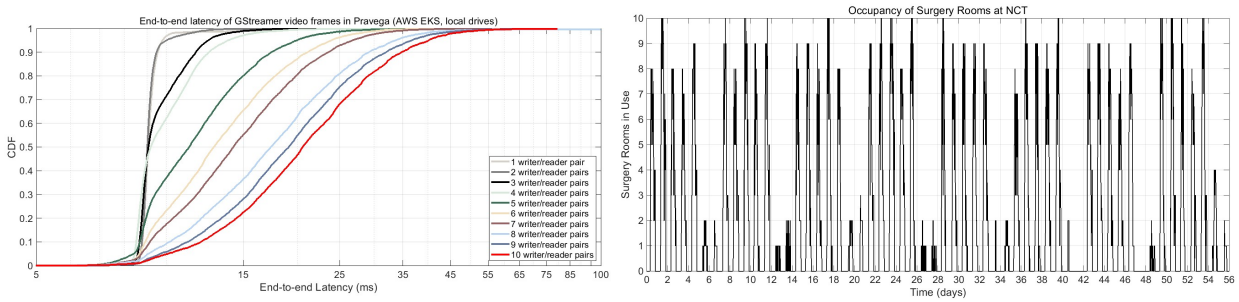


Figure 17: Data collected for solving the Pravega predictive auto-scaling challenge (C2). On the left, the performance profiling of Pravega segment stores based on the number of parallel video streams. On the right, the anonymized surgery room utilization traces collected by NCT.

6 concurrent processes per GPU) and `nodeSelector` constraints for node targeting. These measurements were sampled at 1-second intervals during bin-packing experiments - using both baseline (one stream per GPU) and consolidated (first-fit decreasing) strategies - to characterize resource profiles and validate consolidation under multi-stream workloads.

C2 – Predictive Auto-Scaling of Pravega When considering auto-scaling a streaming system like Pravega based on workload patterns, there are two sources of data that are required: i) performance profiling of the system, and ii) workload traces to be used for predictions. Regarding the former, data was gathered from a full Pravega deployment integrated with GStreamer pipelines and monitored through Prometheus. Collected metrics included segment store write latency distributions (p50, p90, p99), stream ingestion throughput, and number of active segment store instances. Collectively, this information help us building a performance profile of the ingestion latency a single segment store can provide for a number of parallel video streams. This is key for taking auto-scaling decisions to guarantee a specific write latency SLO (see Fig. 17, left). Moreover, scaling events were tracked alongside tail latency spikes during reconfiguration.

On the former data source, we used two months of anonymized NCT operating room utilization traces to train and validate the LSTM-based forecasting model (see Fig. 17, right). replayed at accelerated speed to emulate real-world workload fluctuations. These telemetry streams were complemented with resource usage data from Kubernetes nodes and Pravega performance counters, enabling correlation between workload patterns and system elasticity behavior.

4.4.3 Data Preprocessing

C1 – Smart CPU & GPU Allocation The raw telemetry collected during profiling experiments consists of per-stream frame rate (FPS) measurements and CPU usage (mCPU) for each workload type. Before applying the bin-packing optimization, FPS samples collected at 1-second intervals are aggregated over stable execution windows (excluding warm-up periods) to identify the minimum CPU allocation that sustains $\geq$30 FPS.

Initial profiling measured CPU consumption in millicores (mCPU), but allocating fractional CPU resources introduced non-linearity when scaling the number of concurrent workloads. To preserve linear scaling behavior, CPU allocations were rounded to whole cores, enabling predictable resource consumption as stream counts increase. From this processed telemetry, we derive *resource profiles* for each workload type, encoding the minimum whole-core CPU allocation required to meet the frame rate SLO. Profiling established that Instrument Detection and Phase Detection each require 4 CPU cores, while Liver Segmentations requires 8 CPU cores due to its higher computational complexity. The bin-packing solver operates on these profiles alongside cluster capacity constraints: 16 CPU cores allocatable per GPU (derived from the 64-core node hosting 4 GPUs) and a maximum of 6 time-sliced pods per GPU. This time-slice limit is derived from the most GPU-intensive workload (Liver Segmentation), which consumes approximately 12–18% GPU utilization per stream; beyond

6 concurrent streams, GPU saturation causes frame rate degradation. Workloads are sorted by CPU demand in descending order (first-fit decreasing) before placement to improve packing efficiency.

C2 – Predictive Auto-Scaling of Pravega The raw input for this use case consists of operating-room utilization traces collected in the previous section, stored as a univariate time series (`nct.csv`). Before training, the data undergoes normalization and chronological splitting to preserve temporal causality. Specifically, the series is converted to `float32` and scaled to the range $[0, 1]$ using a `MinMaxScaler` fitted on the training partition. This ensures consistent feature scaling across training/testing phases.

To emulate real deployment conditions, the dataset is divided into two segments: the earliest 12.5% (1 week) of samples for training and the remaining 87.5% (7 weeks) for testing. This chronological split prevents leakage and reflects the operational scenario where models learn from past behavior and predict future unseen intervals. The normalized data is then transformed into supervised learning pairs using a sliding-window approach: each input consists of a short historical context (one time step), and the target is the immediate next value. This design aligns with the goal of near-term forecasting for proactive scaling decisions. Finally, the input arrays are reshaped into the format required by sequence models, namely `[samples, timesteps, features] = [N, 1, 1]`, ensuring compatibility with the LSTM architecture.

4.4.4 Model Training

C1 – Smart GPU Allocation The GPU bin-packing strategy was designed to address heterogeneous resource demands of NCT AI models (instrument detection, phase recognition, liver segmentation) under strict real-time constraints. These models exhibit different computational profiles: lightweight detection tasks are CPU-bound, while segmentation workloads are GPU-intensive and memory-hungry. A naive one-stream-per-GPU approach leads to severe under-utilization ($\geq$ 10% GPU usage), whereas aggressive time-slicing without constraints degrades frame rates below the 30 FPS Service Level Objective (SLO).

Baseline Allocation. Let $\mathcal{W} = \{w_1, \ldots, w_n\}$ denote a set of workloads and $\mathcal{G} = \{g_1, \ldots, g_m\}$ a set of available GPUs. The baseline strategy assigns each workload to a dedicated GPU:

$$x_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \qquad \forall\, w_i \in \mathcal{W},\, g_j \in \mathcal{G} \tag{14}$$

where $x_{ij} \in \{0, 1\}$ indicates assignment of workload w_i to GPU g_j. This approach guarantees isolation but limits concurrency to $|\mathcal{G}|$ streams regardless of actual resource consumption.

First-Fit Decreasing Bin-Packing. To improve utilization, we formulate workload placement as a bin-packing problem with two capacity constraints per GPU: CPU cores ($C_{\max} = 16$) and time-sliced pods ($T_{\max} = 6$). Each workload w_i has a CPU requirement $c_i \in \{4, 8\}$ cores (4 for detection/phase, 8 for segmentation). The objective is to minimize the number of GPUs used while respecting capacity constraints:

$$\min \quad \sum_{j=1}^{m} y_j \tag{15}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} c_i \cdot x_{ij} \leq C_{\max} \cdot y_j \qquad \forall\, g_j \in \mathcal{G} \tag{16}$$

$$\sum_{i=1}^{n} x_{ij} \leq T_{\max} \cdot y_j \qquad \forall\, g_j \in \mathcal{G} \tag{17}$$

$$\sum_{j=1}^{m} x_{ij} = 1 \qquad \forall\, w_i \in \mathcal{W} \tag{18}$$

$$x_{ij}, y_j \in \{0, 1\} \tag{19}$$

where $y_j = 1$ if GPU g_j is used. The time-slice limit $T_{\max} = 6$ is derived from the GPU utilization of the most demanding workload (Liver Segmentation $\approx$12–18% per stream); exceeding this threshold causes GPU saturation and SLO violations.

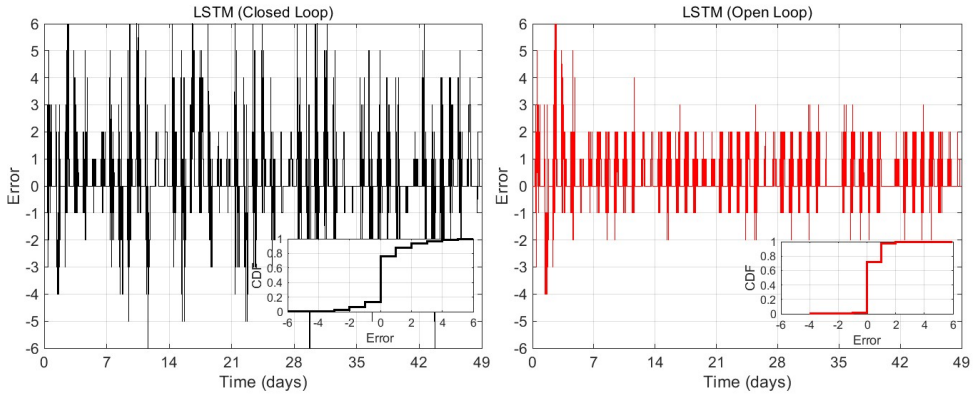


Figure 18: Mean squared error for LSTM predictions in open and closed loop modes.

Rather than solving the integer program exactly, we employ the *first-fit decreasing* (FFD) heuristic: workloads are sorted by CPU demand in descending order, then each workload is assigned to the first GPU with sufficient remaining capacity. While being efficient, FFD provides near-optimal solutions [23], making it suitable for real-time scheduling decisions.

Training for this component did not involve a predictive model but rather profiling-based characterization. We collected telemetry on GPU utilization, memory footprint, and FPS under varying CPU allocations to build resource profiles for each model. These profiles were then encoded into the bin-packing policy, which was validated through controlled experiments comparing baseline deployments against optimized packing strategies. GPU metrics were collected via direct `nvidia-smi` queries, while FPS stability and pod scheduling behavior were monitored through GStreamer callbacks and Kubernetes events.

C2 – Predictive Auto-Scaling of Pravega To reduce tail-latency spikes during reconfiguration and avoid oscillations typical of reactive scaling, the LP employs a lightweight Long Short-Term Memory (LSTM) forecaster that anticipates short-term changes in workload intensity derived from operating-room (OR) utilization traces. The forecast is converted into a recommended number of Pravega Segment Stores via the performance profile (streams per instance vs. write latency) and applied proactively through Kubernetes APIs. This places the LSTM at the "sense–predict–act" center of the LP pipeline: ingest recent telemetry, predict near-term load, and trigger scale-out before the surge.

LSTM has been chosen for its ability to capture temporal dependencies and mitigate vanishing gradients in sequential data. The model architecture follows a simple yet effective design: an input layer feeding into a single LSTM layer with four memory units, followed by a dense output layer producing one-step-ahead predictions. This small configuration is deliberate, as it balances predictive capability with low computational overhead, making it suitable for real-time inference in the LP.

Training is performed using the Adam optimizer with a mean squared error (MSE) loss function, reflecting the regression nature of the task. The model is trained for 100 epochs with a batch size of 1, iterating over the chronological training slice. After training, predictions are generated for both training and test sets, and evaluation metrics such as Root Mean Squared Error (RMSE) are computed in normalized and original scales (via inverse transformation) to assess accuracy. The trained model is exported as a portable artifact (`nct_lstm_model.keras`) for integration into the Learning Plane inference pipeline.

At inference time, the model supports two rollout modes (see Fig. 18): *open loop*, where actual next values are fed back at each step, and *closed loop*, where the model's own predictions are recursively used as input. The latter enables multi-step forecasting by chaining one-step predictions, which is critical for anticipating workload surges and issuing proactive scaling actions. These forecasts are then mapped to the required number of Pravega segment stores using the performance profile established earlier, ensuring latency Service Level Objectives (SLOs) are maintained during dynamic

workload conditions.

4.4.5 Model Comparison and Evaluation

C1 – Smart CPU & GPU Allocation To evaluate the bin-packing strategy, we compare single-GPU and multi-GPU deployments for Liver Segmentation, the most resource-intensive workload. We focus on three metrics: *GPU utilization*, *workload density* (concurrent streams), and *FPS stability* (compliance with the $\geq$30 FPS SLO). Comprehensive results for Instrument Detection and Phase Detection are reported in Deliverable D5.4.

Figure 19 presents scaling behavior when consolidating multiple Liver Segmentation streams onto a single GPU via time-slicing. FPS scales *non-linearly* and remains at 30$\pm$5 FPS up to 6 concurrent streams, as does GPU utilization: utilization increases from $\approx$18% (1 stream) to$\approx$75% (6 streams), but the per-stream overhead grows disproportionately at higher densities due to time-slicing contention and context-switching costs. Beyond 6 streams, frame rate degradation occurs as the GPU becomes saturated, establishing the practical consolidation limit for heavyweight workloads on a single device.

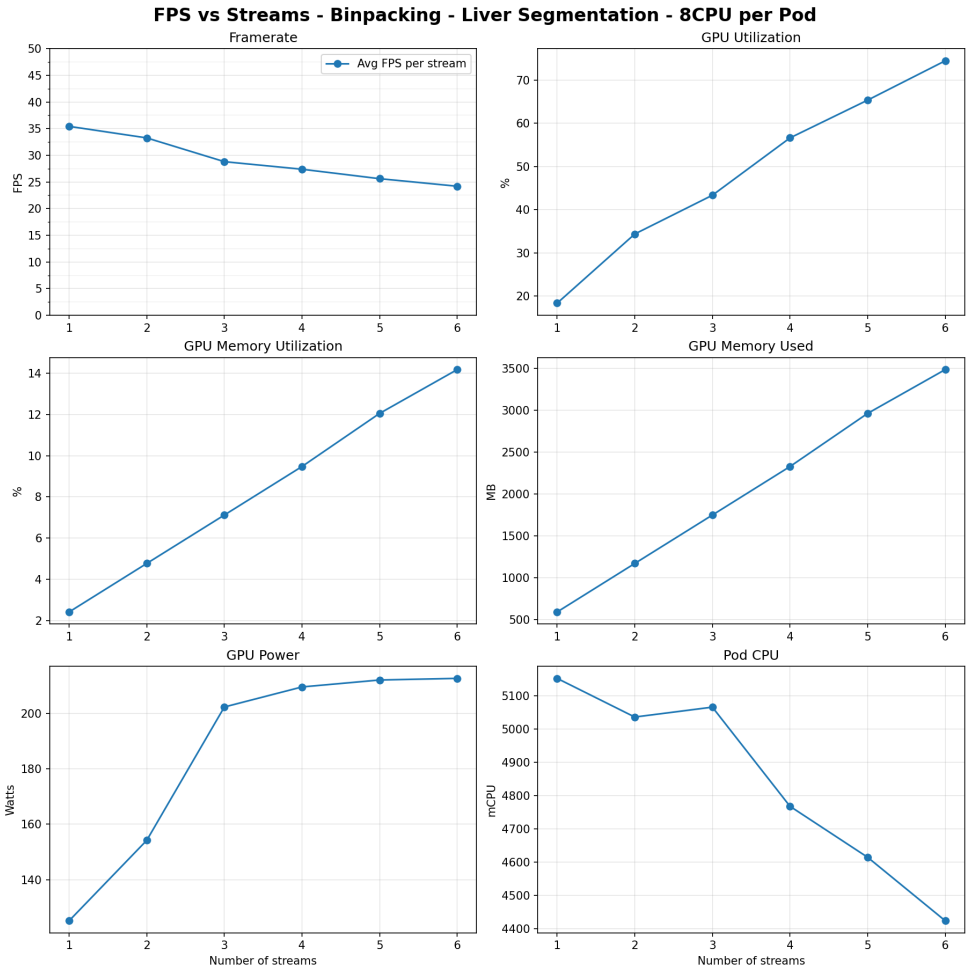


Figure 19: Single-GPU bin-packing scalability for Liver Segmentation: FPS declines gradually when increasing to 6 streams, while GPU utilization also grows non-linearly due to time-slicing overhead.

To overcome this single-GPU bottleneck, Figure 20 demonstrates multi-GPU distribution where streams are spread across available devices using the first-fit decreasing algorithm. In this configuration, scaling behavior *linearizes*: GPU utilization remains at $\approx$15% for 1–4 streams (each placed on separate GPUs), then increases to $\approx$30% after 4 streams as consolidation begins. Crucially, FPS remains stable at 30–32 FPS across all configurations, and resource consumption (memory, power,

CPU) scales proportionally with stream count. This confirms that the non-linear overhead observed in the single-GPU case stems from excessive time-slice contention rather than fundamental GPU capacity limits. By distributing workloads across multiple GPUs before consolidating, the bin-packing strategy avoids premature saturation while still achieving improved utilization over baseline allocation. This informs one of our bin-packing constraints for each GPU: CPU cores ($C_{max} = 16$).

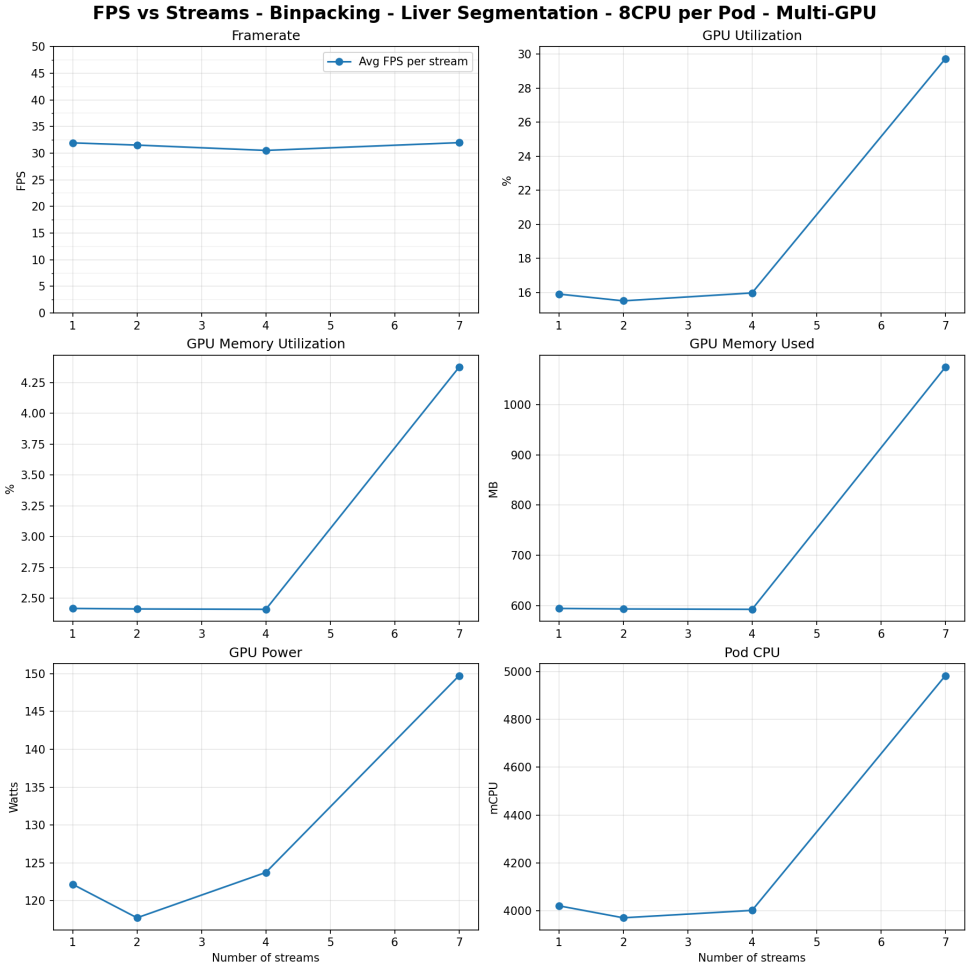


Figure 20: Multi-GPU bin-packing scalability for Liver Segmentation: distributing streams across GPUs linearizes scaling behavior, maintaining stable 30–32 FPS with proportional resource consumption.

These results validate the bin-packing strategy's two-phase approach: (1) distribute workloads across GPUs to avoid single-device contention, then (2) consolidate within the 6-stream time-slice limit per GPU. This achieves up to 7× higher workload density compared to baseline (1 stream per GPU) for Liver Segmentation, with even greater gains for lighter workloads (see Deliverable D5.4).

C2 – Predictive Auto-Scaling of Pravega The evaluation of predictive auto-scaling focused on comparing reactive scaling strategies against the LSTM-based predictive approach for Pravega segment store elasticity under fluctuating workloads derived from NCT operating room traces.

In this section, we summarize the results we obtained from comparing the following auto-scaling strategies for Pravega:

- *Reactive Approach*: 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

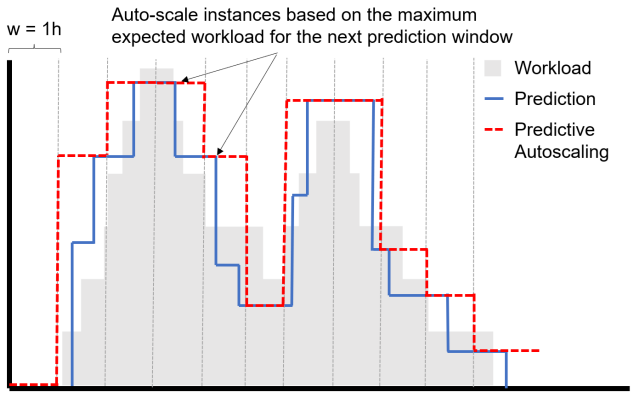


Figure 21: Mean squared error for LSTM predictions in open and closed loop modes.

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 (simulation only)*: 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 Approach (LSTM)*: With these observations in mind, we use a predictive auto-scaling algorithm as shown in Figure 21. 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.

Predictive auto-scaling results: Next, we summarize the results of the LSTM predictive approach to auto-scaling Pravega instances, comparing simulation-based results (see D5.2) and real experiments (see D5.4). In particular, we draw the following observations:

- *Scaling Events*: Simulation experiments showed that reactive auto-scaling triggered a very high number of instance changes, with the vanilla reactive algorithm incurring *1,310 scaling events*

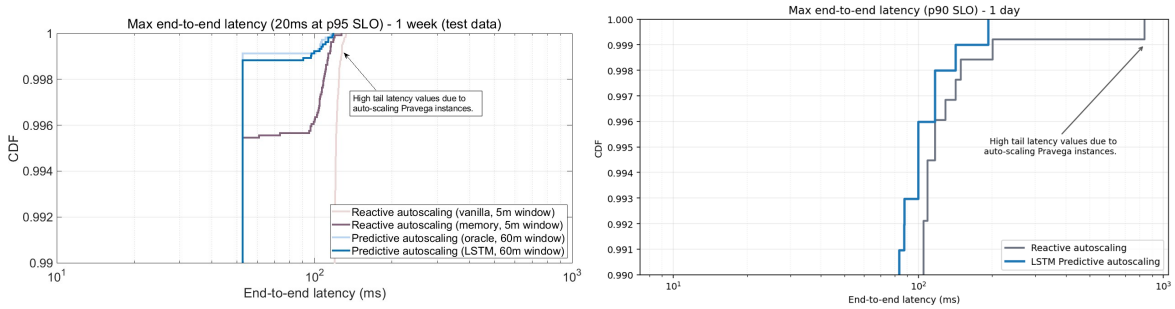


Figure 22: Mean squared error for LSTM predictions in open and closed loop modes.

over one week of NCT workload replay. Adding memory reduced this to 44 events, but the predictive LSTM approach achieved only *12 scaling events*, a *3.6× reduction* compared to reactive-with-memory and more than *100× fewer events* than vanilla reactive scaling. Real experiments confirm this trend: during a full replay of two months of traces at accelerated speed, reactive scaling produced *112 scaling actions*, while predictive scaling reduced this to *16*, representing a *7× improvement in stability*. Fewer scaling events translate into smoother transitions and lower latency spikes for AI video inference.

- *Tail Latency Impact*: In simulation, reactive scaling exhibited severe latency spikes during reconfiguration, with tail latencies reaching $>$ 100ms values at *p87 and p99.56* for vanilla and memory-based reactive methods, respectively. Predictive LSTM reduced these extremes to *p99.89*, nearly eliminating disruptive outliers. Real experiments reinforce these findings: predictive scaling reduced worst-case *p90 latency by nearly* 6× compared to reactive scaling, keeping 99.9% of requests under 150ms, whereas reactive approaches showed heavy tails extending up to 1s. This improvement is critical for latency-sensitive AI video analytics pipelines.

- *Methodology Validation*: The real-world results closely match the simulation-based predictions, validating our methodology. Both studies demonstrate that predictive elasticity significantly reduces oscillatory behavior and tail-latency penalties during scaling events. The consistency between simulated and empirical outcomes confirms that the LSTM-based approach generalizes well from controlled experiments to production-like environments.

Overall, these results highlight the relevance of predictive Pravega auto-scaling in enabling elasticity for fluctuating workloads such as those observed at NCT. By anticipating resource needs and minimizing disruptive scaling actions, the LP ensures stable ingestion and low-latency performance for real-time AI video analytics. This capability is essential for surgical environments where latency spikes can compromise decision-making, and it demonstrates how predictive strategies outperform reactive heuristics in both efficiency and quality of service.

4.5 Usecase: Agriculture

4.5.1 Problem Definition

The practical application of this technology in the agricultural sector faces two main challenges. First, reliable and sufficient data is needed, which requires overcoming resistance and barriers to data sharing, such as the need to control data ownership and usage, differences in units of measurement, and the accessibility of information by external services. Second, for its use to be viable, the ability to process data at different levels of the computational continuum is required. This includes ensuring the feasibility of cloud computing for large volumes of data with predictable and optimized costs.

From a machine-learning perspective, and under the premise that the agronomic workflow is fully specified, we face the problem that **execution times vary significantly depending on the configuration and type of resources, as well as the volume, scope, and frequency of the data sources.**

The main objectives, which at the same time respond to the primary scientific and technological challenges, align with the development of an intelligent optimization mechanism capable of predicting and recommending optimal configurations (memory, CPU, parallelism, parameters) for serverless workflows, minimizing both execution time and cost.

The optimization challenge requires:

- **Predicting the duration of each execution** based on configurable parameters (allocated memory, number of vCPUs, number of partitions, etc.).
- Automatically selecting the configuration that minimizes execution time and associated cost.
- Integrating this optimization into the CloudSkin/PyRun learning plane through the existing instrumentation (Profiler, Prometheus).

4.5.2 Data Collection

Multiple data collections have been used both for carrying out the executions and for the analysis and optimization of the results.

We therefore distinguish between data coming from sensors and geospatial information, on the one hand, and resource consumption, performance, and processing data, on the other.

The following data collections were employed:

- Agricultural use-case dataset: CloudSkin climatic layers (temperature, humidity, irrigation) obtained from the various datasets incorporated into the dataspace.
- Digital terrain models (MDT05) from the IGN (Instituto Geográfico Nacional) for the Region of Murcia, in GeoTIFF format.
- Execution environment metadata from Lithops (via JobRunner/DynamoDB): configuration and execution identifiers.
- Metrics from the profiler sent to Prometheus/AWS Managed Prometheus: time-series data on execution time, CPU usage, and memory usage.
- Results from approximately 150 executions of the evapotranspiration pipeline, varying memory, number of vCPUs, number of data segments, and other parameters.

4.5.3 Technologies and Core Components

The CloudSkin project architecture integrates a set of complementary technologies and core components that enable efficient, reliable, and scalable monitoring across both open-source and multi-tenant execution environments. At a high level, the monitoring stack combines lightweight system introspection, metrics collection and storage, resilient communications, visual analytics, and persistent metadata storage to support performance analysis, bottleneck detection, and continuous optimization.

The main technologies used are the following:

- **psutil**: Provides detailed system metrics (CPU, memory, disk, and network) at process and thread level, enabling fine-grained resource profiling per execution.
- **Prometheus / AWS Managed Prometheus**: Collect, store, and query time-series performance metrics, supporting both local deployments and managed, multi-tenant operation.
- **Tenacity**: Implements robust retry policies to increase resilience when metric extraction or submission fails due to transient errors.
- **Grafana**: Supports interactive visualization in open-source environments through configurable dashboards and panels.

- **Vue.js + Apache ECharts**: Used to build integrated dashboards tailored to multi-tenant environments within the PyRun platform.
- **NoSQL databases**: Persist execution metadata and summaries for querying, filtering, auditing, and historical analysis, while avoiding excessive metric label cardinality.

Monitoring is organised into modular building blocks designed for extensibility and clear separation of responsibilities. The data flow begins when a user execution is initialised and continues until metrics and summaries are safely persisted:

- **Handler**: Acts as the orchestrator that initialises each user execution and launches both the *JobRunner* and the *Profiler*.
- **JobRunner**: Executes the user code and records key high-level attributes such as total duration, result size, and overall resource usage.
- **Profiler**: Runs in parallel on each worker and recursively monitors the resource consumption of processes and threads at configurable sampling intervals.
- **Transmission module**: Encapsulates the logic for dispatching, grouping, or aggregating metrics through the appropriate API, depending on the execution environment:
  - In the open-source Lithops environment, metrics are sent via an Open Source API to a local Prometheus server.
  - In the PyRun (multi-tenant) environment, metrics are securely sent via a PyRun API to AWS Managed Prometheus, typically using grouped or pre-aggregated submissions to reduce request frequency and operational overhead.
- **Historical storage**: Persists execution summaries in a database to support exploration and auditing while preventing high label cardinality in the metrics backend.

Metric management and visualisation are provided through two supported modalities, reflecting the requirements of open-source and multi-tenant operation. In the open-source environment, metrics are dispatched directly to a local Prometheus server and visualised through Grafana dashboards, with label filtering strategies applied to reduce cardinality and maintain query performance. In the PyRun platform, monitoring relies on AWS Managed Prometheus with IAM-based authentication and namespace-based metric segregation. To improve efficiency at scale, metrics are commonly pre-aggregated before ingestion, which optimises downstream queries and reduces operational costs. Visualisation in PyRun is integrated into the platform through dashboards built with Vue.js and Apache ECharts, typically including CPU, memory, disk, and network charts, as well as execution-based time series; representative views may also include Gantt-style timelines of function executions across pipeline stages alongside per-execution resource curves.

PyRun further provides specialised APIs and leverages managed services to ensure performance and availability in multi-tenant environments. These components collectively handle metric ingestion, job registration, and large-scale telemetry:

- **Open Source API**: Sends metrics to a local Prometheus server, suitable for moderate workloads and environments without strict availability constraints.
- **PyRun API (grouped metrics)**: Aggregates metrics to reduce submission frequency, lowers network overhead, and uses IAM authentication for secure, scalable transfer to AWS Managed Prometheus.
- **PyRun API (job logging)**: Stores execution metadata in DynamoDB, including unique identifiers, execution timestamps, configuration parameters, and final status, enabling efficient querying, filtering, and auditing of historical runs.

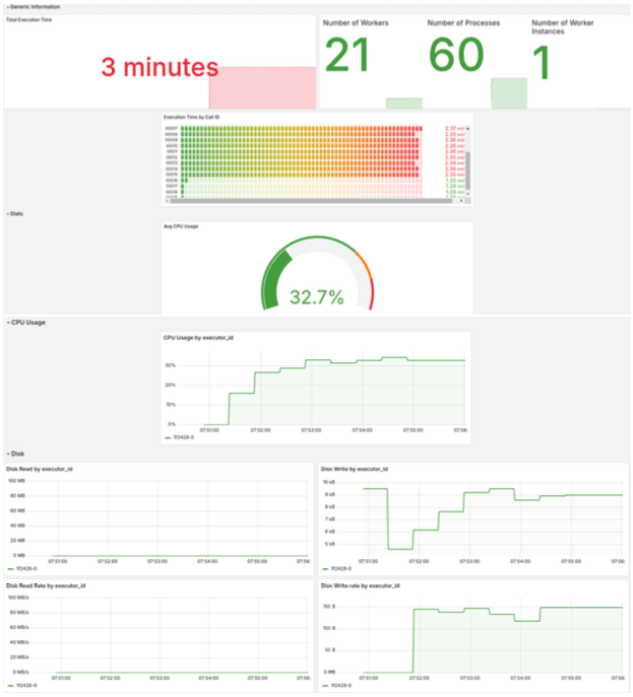


Figure 23: (top) Fragment of the generic Grafana dashboard. (bottom) Resource usage by execution id in Grafana.

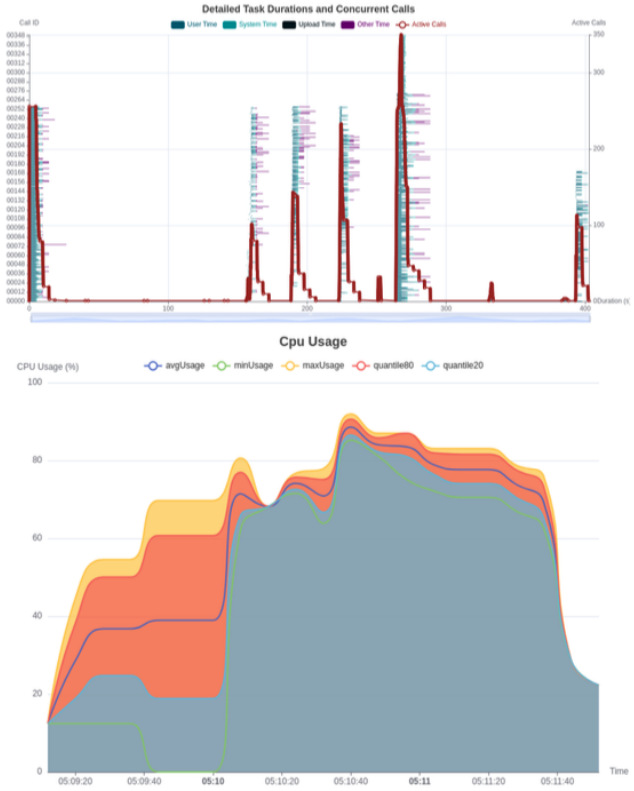


Figure 24: (top) Gantt chart showing function executions across pipeline stages in PyRun. (bottom) CPU usage per execution in PyRun.

From a scalability perspective, the design with independent Profilers per worker scales linearly with the number of workers. However, at high concurrency, mass metric submission to self-managed Prometheus can become a bottleneck; this motivates the use of AWS Managed Prometheus in PyRun, which provides automatic scalability for large metric volumes, high availability through replication and fault tolerance, simplified operations by reducing maintenance overhead, and consistent query responsiveness under heavy load.

To ensure reliability and performance, the implementation includes systematic testing and validation practices:

- **Load testing**: Simulates high concurrency to evaluate monitoring overhead and tune sampling frequency and aggregation strategies.
- **Metric validation**: Cross-checks internally collected measurements with external systems to verify consistency and correctness.
- **Error handling**: Applies automatic retries via Tenacity for transient failures during metric extraction or submission, improving robustness end to end.

4.5.4 Model Training

- Objective and Scope

  The goal of intelligent optimization is to **predict and recommend optimal configurations (memory, CPU, parallelism, parameters) for serverless pipelines, minimizing both execution time and cost.** We will describe the general Learning Plane methodology that is applied specifically to our agricultural evapotranspiration pipeline on Lithops.

- Execution Data Collection and Storage

  Data sources include metadata collected from the Lithops runtime through the JobRunner and DynamoDB, which provide configuration details and execution identifiers, as well as profiler metrics delivered to Prometheus or AWS Managed Prometheus, including execution time and time-series measurements of CPU and memory usage. The agricultural use case dataset comprises approximately 150 runs of the evapotranspiration pipeline, where key parameters such as memory allocation, the number of vCPUs, the number of data slices, and other configuration settings were systematically varied.

- Feature Engineering

  In order to capture the key relationships between configuration parameters and execution time, we transform raw inputs into a more expressive set of features:

- Description of the Agricultural Pipeline

  The agricultural pipeline processes climatic and topographic data to compute daily evapotranspiration and estimate crop water consumption on Lithops. The system ingests two main categories of input data: firstly, the CloudSkin climatic layers, which include temperature, humidity, wind speed and radiation; and secondly, the Digital Terrain Models (MDT05) provided by the IGN for the Region of Murcia, supplied in GeoTIFF format.

  Once these datasets are available, each GeoTIFF file is partitioned using DataPlug into smaller georeferenced slices. These slices are then stored in S3 and their locations are registered in DynamoDB, enabling efficient indexing and retrieval.

  Subsequently, Lithops distributes the individual slices among a set of workers, so that each worker processes a specific subset of the territory in parallel. Within every assigned block, the climatic variables are spatially interpolated in order to guarantee spatial coherence and continuity across the study area.

Raw features	
num_files	Number of input files processed
splits	Number of slices (partitions) used for parallel processing
input_size_gb	Total input data size (GB)
runtime_memory_mb	Memory allocated for the function (MB)
ephemeral_storage_mb	Temporary storage allocated (MB)
worker_processes	Number of worker processes per function invocation
invoke_pool_threads	Number of threads in the invocation pool
vcpus	Number of virtual CPUs allocated
Derived features	
memory_per_slice	runtime_memory_mb / splits
vcpus_per_slice	vcpus / splits
slice_size_gb	input_size_gb / splits
cpu_util_per_gb	mean CPU % / input_size_gb
mem_util_per_gb	mean memory MB / input_size_gb
Interaction terms	
memory_mb * vcpus	runtime_memory_mb $\times$ vcpus
memory_per_slice * slice_size_gb	memory_per_slice $\times$ slice_size_gb
Preprocessing	
Scaling/Normalization	Standardize continuous variables (zero mean, unit variance)
Log transform	Apply $\log(1 + x)$ to skewed metrics (e.g., duration s, input size gb)
Categorical encoding	One-hot encode discrete pipeline parameters (e.g., interpolation method)

Table 16: Feature engineering summary

After interpolation, the Penman–Monteith equation is applied at pixel level to calculate daily evapotranspiration. Finally, evapotranspiration values expressed in millimetres per day are converted into water volume in cubic metres per hectare. These results are assembled into an output raster, which provides a spatially explicit estimation of crop water usage.

- Predictive Modeling

  For this work, the **XGBoost algorithm** was used in regression mode, specifically selected for its strong performance and high accuracy when working with structured data. This model is particularly well suited to problems where the features are clearly defined and organised in tabular form, as is the case with our pipeline runs.

  The configuration of the model was not done with fixed values but through a hyperparameter optimisation process based on Bayesian search using the Optuna library. In this process, different values were explored for key parameters such as the maximum tree depth (between 3 and 10), the learning rate (between 0.01 and 0.3), and the number of estimators (between 50 and 500). In addition, other hyperparameters related to data and feature sampling, such as subsample and colsample_bytree, were tuned, as well as the L1 and L2 regularisation terms, with the aim of improving the generalisation of the model and avoiding overfitting.

  To evaluate the performance of the model, 5-fold cross-validation was applied to a dataset comprising 150 runs. In this evaluation, the model achieved a mean absolute error (MAE) of 12.5 seconds and a coefficient of determination R^2 of 0.92 on the validation folds, which indicates that it explains most of the variability in the actual execution times. As a comparative reference, benchmarking was carried out against other common models for this type of problem, such as Random Forest and LightGBM. **In these comparisons, XGBoost obtained between 10% and 15% lower MAE, which supports its selection as the final model.**

  Once the model has been trained and validated, its operational use is to support decision-making on the configuration of new pipeline executions. For each new request, different candidate configurations are generated, for example combinations of memory, number of vCPUs and partitions. The model predicts the estimated execution time for each of these configurations and, based on these predictions, the configuration with the shortest estimated duration is selected, thus optimising compute time before actually launching the execution.

- Comparison and Evaluation of Configuration

  For each new run, the Learning Plane:

  1. Generates candidate configurations by varying memory, vCPUs, and number of partitions.

  2. Uses the XGBoost model to estimate the duration of each candidate.

  3. Selects the configuration with the lowest predicted time and returns it for Lithops to apply in the next run.

- DataCockpit: Data Import Panel for the Agricultural Pipeline

  DataCockpit is an interactive, web-based panel designed specifically for the agricultural use case. It acts as a central interface for importing the necessary input datasets, preprocessing them, and launching the water-consumption pipeline with AI-driven optimisation. By bringing these steps together in one place, DataCockpit streamlines the user workflow: it provides a single panel to upload all required data, automatically performs the preprocessing needed to match the pipeline's expected format, integrates a predictive optimisation model to suggest the most suitable execution setup, and then starts the pipeline on Lithops using the optimised parameters.

  Within the panel, the user selects the relevant climatic inputs, including CloudSkin layers such as temperature, humidity, wind speed, and radiation, as well as topographic inputs in the form

of MDT05 GeoTIFF files from IGN. Once these inputs are chosen, DataCockpit invokes DataPlug in the background to partition each GeoTIFF into georeferenced slices, store those slices in S3 while registering their references in DynamoDB, and distribute the slices to individual Lithops workers so they can be processed in parallel.

After the import step is complete, the user can trigger the Learning Plane's predictive model with a single click. DataCockpit then requests runtime estimates for a range of candidate configurations, recommends the configuration that minimises both time and cost, automatically applies the suggested memory, vCPU, and partitioning parameters, and launches the pipeline execution on Lithops.

- Overview of DataPlug DataPlug is a serverless library designed to partition large scientific data files stored in object storage efficiently, without altering the original data. It supports DataCockpit's data import process by slicing GeoTIFF files into manageable chunks. DataPlug now includes support for Cloud Optimised GeoTIFF (COG), which enables direct partitioning of cloud-optimised terrain models and other high-resolution geospatial datasets. Expert users have validated DataPlug's partitioning workflow in production-like environments, confirming that the resulting slices preserve geospatial precision and meet the performance requirements needed for distributed processing.

  The documentation has also been extended to explain several approaches for configuring access to data stored in S3, including the use of environment variables such as AWS Access Key ID and AWS Secret Access Key, AWS CLI profiles defined in the shared credentials file, IAM roles provided through EC2 instances or ECS/EKS task profiles, and automatic integration within managed runtimes such as PyRun. By combining robust partitioning with seamless integration into DataCockpit, DataPlug enables a fully automated end-to-end workflow that preserves geospatial accuracy and promotes optimal resource utilisation when running the agricultural pipeline.

4.5.5 Model Comparison and Evaluation

- Results, Savings, and Return on Investment Execution time is reduced by up to 79.9% compared with sub-optimal Design Space Analysis configurations. The mean absolute error is 75.3% lower than when using the historical mean. The cost per execution is 30% lower, equivalent to USD 0.069 per run, which represents 30% of a baseline cost of USD 0.23.

  The initial training cost is USD 38.75 (approximately 150 runs at USD 0.23 each). With a per-run saving of USD 0.069, the break-even point is reached after about 562 runs, which at a rate of 10 runs per day corresponds to roughly 56 calendar days. Projected cumulative savings grow linearly, reaching the break-even point of USD 38.75 after approximately two months and exceeding USD 500 by month 24.

- Conclusions

  This project has demonstrated the feasibility of executing and optimising the Water Consumption pipeline in serverless environments by leveraging open data from CloudSkin and the Spanish National Geographic Institute (IGN), and by using modern technologies for data partitioning, orchestration, and machine learning. Pipeline monitoring was implemented through instrumentation with tools such as psutil, Prometheus, and Grafana, enabling real-time collection of detailed execution metrics and supporting the identification of bottlenecks as well as continuous performance improvement. **Optimisation through the Learning Plane was achieved with a predictive XGBoost model, tuned with Optuna, which estimates pipeline duration from configuration features and recommends the most suitable resource settings, thereby reducing execution time and cost without manual intervention.** The integration of DataPlug, which efficiently partitions digital terrain models, with DataCockpit, which provides an interactive interface to import data and launch the pipeline using the AI-recommended configuration, has simplified experimentation and improved usability for scientific users. In addition, a

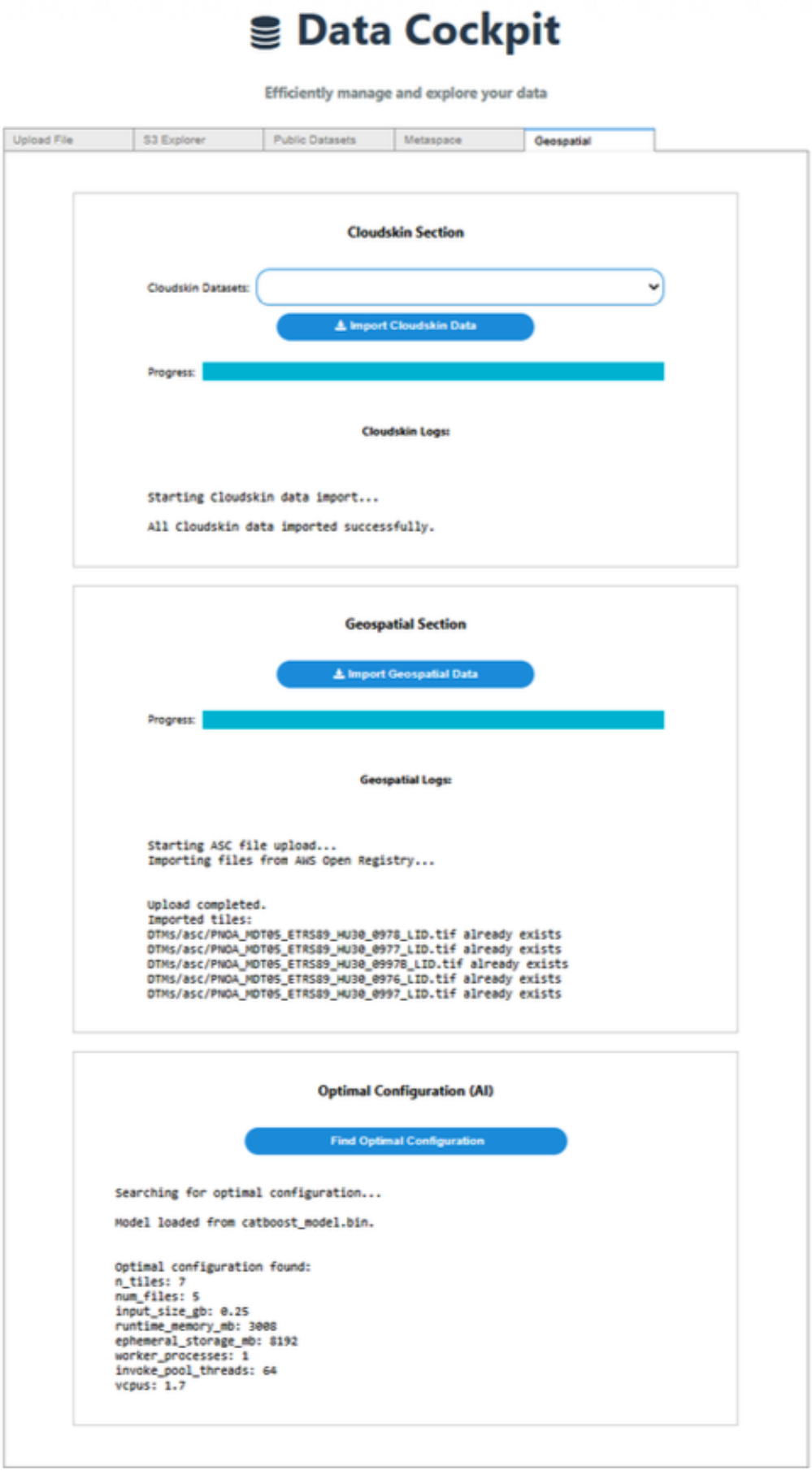


Figure 25: DataCockpit panel for importing CloudSkin and GeoTIFF data, partitioning via DataPlug, and executing the agricultural water consumption pipeline with AI-optimized parameters.

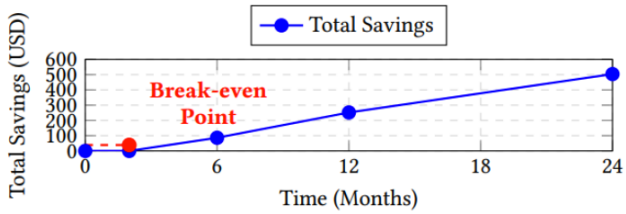


Figure 26: Projected cumulative savings assuming 10 runs/day. Break-even at approximately two months.

reproducible and scalable workflow was enabled through Lithops and managed services such as AWS S3, IAM, and DynamoDB, allowing PyRun to execute the pipeline at scale without dedicated infrastructure while adapting to different configurations and data volumes.

Taken together, these results validate how the developed technology supports computing on the continuum, enabling advanced agricultural data analysis as a service. By dynamically selecting the most efficient execution configuration and exploiting elastic, serverless resources, the approach increases processing speed while bounding and reducing computation costs, and it consequently lowers the carbon-footprint impact associated with running large-scale data-processing workloads.

5 Conclusions

The Learning Plane provides a practical framework and a list of models at the orchestration layer of the CloudSkin platform. This deliverable demonstrates how AI is integrated into and helps with real-world use cases.

References

[1] 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.

[2] G. Bravo-Rocca, P. Liu, J. Guitart, A. Dholakia, D. Ellison, J. Falkanger, and M. Hodak, "Scanflow: A multi-graph framework for machine learning workflow management, supervision, and debugging," Expert Systems with Applications, vol. 202, p. 117232, 2022.

[3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in Advances in Neural Information Processing Systems (I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), vol. 30, Curran Associates, Inc., 2017.

[4] A. Zeng, M. Chen, L. Zhang, and Q. Xu, "Are Transformers Effective for Time Series Forecasting?," arXiv, May 2022.

[5] S.-A. Chen, C.-L. Li, S. O. Arik, N. C. Yoder, and T. Pfister, "TSMixer: An all-MLP architecture for time series forecast-ing," Transactions on Machine Learning Research, 2023.

[6] V. Ekambaram, A. Jati, N. Nguyen, P. Sinthong, and J. Kalagnanam, "Tsmixer: Lightweight mlp-mixer model for multivariate time series forecasting," in Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD '23, (New York, NY, USA), p. 459–469, Association for Computing Machinery, 2023.

[7] A. Gu, I. Johnson, K. Goel, K. K. Saab, T. Dao, A. Rudra, and C. Re, "Combining recurrent, convolutional, and continuous-time models with linear state space layers," in Advances in Neural Information Processing Systems (A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, eds.), 2021.

[8] L. Graf, T. Ortner, S. Woźniak, and A. Pantazi, "FlowState: Sampling Rate Invariant Time Series Forecasting," arXiv, Aug. 2025.

[9] L. Graf, T. Ortner, S. Woźniak, and A. Pantazi, "Flowstate: Sampling-rate invariant time series foundation model with dynamic forecasting horizons," in Recent Advances in Time Series Foundation Models Have We Reached the 'BERT Moment'?, 2025.

[10] J. T. Smith, A. Warrington, and S. Linderman, "Simplified state space layers for sequence modeling," in The Eleventh International Conference on Learning Representations, 2023.

[11] A. Gu, I. Johnson, A. Timalsina, A. Rudra, and C. Re, "How to train your HIPPO: State space models with generalized orthogonal basis projections," in International Conference on Learning Representations, 2023.

[12] A. Gu, T. Dao, S. Ermon, A. Rudra, and C. Ré, "Hippo: Recurrent memory with optimal polynomial projections," Advances in neural information processing systems, vol. 33, pp. 1474–1487, 2020.

[13] A. F. Ansari, L. Stella, C. Turkmen, X. Zhang, P. Mercado, H. Shen, O. Shchur, S. S. Rangapuram, S. P. Arango, S. Kapoor, et al., "Chronos: Learning the language of time series," arXiv preprint arXiv:2403.07815, 2024.

[14] A. Auer, P. Podest, D. Klotz, S. Böck, G. Klambauer, and S. Hochreiter, "Tirex: Zero-shot forecasting across long and short horizons," in 1st ICML Workshop on Foundation Models for Structured Data, 2025.

[15] H. Wu, J. Xu, J. Wang, and M. Long, "Autoformer: Decomposition transformers with auto-correlation for long-term series forecasting," in Advances in Neural Information Processing Systems (M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, eds.), vol. 34, pp. 22419–22430, Curran Associates, Inc., 2021.

[16] T. Zhou, Z. Ma, Q. Wen, X. Wang, L. Sun, and R. Jin, "FEDformer: Frequency enhanced decomposed transformer for long-term series forecasting," in Proceedings of the 39th International Conference on Machine Learning (K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu, and S. Sabato, eds.), vol. 162 of Proceedings of Machine Learning Research, pp. 27268–27286, PMLR, 17–23 Jul 2022.

[17] 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," in Proceedings of the AAAI conference on artificial intelligence, vol. 35, pp. 11106–11115, 2021.

[18] H. Wu, T. Hu, Y. Liu, H. Zhou, J. Wang, and M. Long, "Timesnet: Temporal 2d-variation modeling for general time series analysis," arXiv preprint arXiv:2210.02186, 2022.

[19] S. Hochreiter and J. Schmidhuber, "Long short-term memory," Neural Computation, vol. 9, no. 8, pp. 1735–1780, 1997.

[20] BSC, "Marenostrum 5: Technical information." https://www.bsc.es/marenostrum/marenostrum-5, 2025.

[21] A. W. Services, "Understanding the lambda execution environment lifecycle," 2025.

[22] K. Ovchinnikova, V. Kovalev, L. Stuart, and T. Alexandrov, "Offsampleai: artificial intelligence approach to recognize off-sample mass spectrometry images," BMC bioinformatics., vol. 21, no. 1, 2020-12.

[23] G. Dósa, "The tight bound of first fit decreasing bin-packing algorithm is ffd (i) 11/9 opt (i)+ 6/9," in International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies, pp. 1–11, Springer, 2007.