CloudSkin

Funded by the European Union

HORIZON EUROPE FRAMEWORK PROGRAMME

# CloudSkin

(grant agreement No 101092646)

## Adaptive virtualization for AI-enabled Cloud-edge Continuum

## D5.1 Design and early prototype of CLOUDSKIN Learning Plane

Due date of deliverable: 30-06-2023
Actual submission date: 30-06-2023

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

# Summary of the document

| | |
|---|---|
| **Document Type** | Other |
| **Dissemination level** | Public |
| **State** | v1.0 |
| **Number of pages** | 38 |
| **WP/Task related to this document** | WP5 / T5.1 |
| **WP/Task responsible** | BSC |
| **Leader** | Josep Ll. Berral (BSC) |
| **Technical Manager** | Andre Martin & Christof Fetzer (TUD) |
| **Quality Manager** | Pablo Gimeno (URV) & Ardhi P.P. Hartono (TUD) |
| **Author(s)** | Josep Ll. Berral (BSC), Anna M. Nestorov (BSC), Ferran Agulló (BSC), Marc Sanchez Artigas (URV) |
| **Partner(s) Contributing** | BSC, URV, TUD |
| **Document ID** | CloudSkin_D5.1_Public.pdf |
| **Abstract** | Design of the architecture and technologies to produce the CLOUDSKIN Learning Plane, and first experiments of components. |
| **Keywords** | Learning Plane, AI-Driven Continuum, Orchestration, Resource Management, Workload Characterization |

# History of changes

| Version | Date | Author | Summary of changes |
|---|---|---|---|
| 0.1 | 25-05-2023 | Josep Ll. Berral | First draft. |
| 0.2 | 29-05-2023 | Anna Maria Nestorov | Description of experiments and prototype Scalable Scheduling. |
| 0.3 | 09-06-2023 | Ferran Agullo | Description of experiments and prototype Characterization and Transformers. |
| 0.4 | 22-06-2023 | Josep Lluis Berral | Architecture and design description. |
| 0.5 | 23-06-2023 | Josep Lluis Berral | Homogenization with Deliverable 2.1 & Publication + Dissemination. |
| 1.0 | 30-06-2023 | Josep Lluis Berral | Final version after revisions. |

## Table of Contents

## List of Abbreviations and Acronyms

| | |
|---|---|
| **AI** | Artificial Intelligence |
| **API** | Application Programming Interface |
| **CC** | Creative Commons |
| **CSV** | Comma-separated values |
| **DAG** | Directed Acyclic Graph |
| **DOI** | Digital Object Identifier |
| **FL** | Federated Learning |
| **HPC** | High-Performance Computing |
| **HPDA** | High-Performance Data Analytics |
| **LP** | Learning Plane |
| **MAPE** | Mean Absolute Percent Error |
| **ML** | Machine Learning |
| **MSE** | Mean Squared Error |
| **PDA** | Predictive Data Analytics |
| **PV** | persistent Volume |
| **QoS** | Quality of Service |
| **RL** | Reinforcement Learning |
| **SL** | Swarm Learning |

# 1 Executive summary

**Deliverable D5.1** "Design and early prototype of CLOUDSKIN Learning Plane" aims at presenting the initial specifications of the Learning Plane's architecture complementing the Architecture Design published in the **Deliverable D2.1**, reviewing the selection of technologies and initially benchmarking system. This deliverable contains the identification of the requirements and the description of the Learning Plane's main components, towards workload analysis and characterization, and how the modelling and prediction methods are integrated with the orchestration components from the Control Plane and the telemetry components from the Data Plane.

Furthermore, this deliverable includes a set of early implementations and prototypes of the methods to be used in the workload characterization and decision making, along with benchmarking and comparison experiments, evaluating the proposed methods and showing usefulness as proof-of-concept for the proposed research, towards for composing the Learning Plane's "smart" elements in the CLOUDSKIN software stack. A set of statistical and machine learning methods is described first, then tested against state-of-art or trivial alternatives, demonstrating the potential of the proposed methods to improve the performance of Edge-Cloud environments.

## 2 Introduction

The current distributed architectures, where resources and workloads must be managed either at local or holistic levels, include the concepts of Control Plane and Data Plane. The Control Plane is responsible for the control loop concerning **monitoring**, **analysing**, **planning** and **executing** policies and actions on the system elements, while the Data Plane is responsible for coordinating the data sourcing, transporting, storing and delivering. In addition to these planes, when "Smart Management" enters the scene, an additional specialised plane is required, the Learning Plane, responsible for managing the obtained knowledge and its application across the existing planes. A direct example would be the management of the available workload behavior models, deciding when to train or update them, publishing them available for the Control Plane to ask for recommendations or predictions on the current applications, sharing them across different local actors across the distributed system as in federated or swarm learning methods, and controlling the access to them from different controllers in the distributed system.

The classic methodology for including artificial intelligence (i.e. machine learning) into the control loop is to collect data from the different actors into a single repository, creating a corpus of examples (i.e. the dataset) to train a model. Then this model is queried by the different system actors (e.g. schedulers in the Control Plane) to make decisions about workload placement, resource provisioning, user prioritisation and other orchestration actions. However, orchestrating a hyper-distributed system like the Edge-Cloud Continuum requires novel methods that do not rely on a centralised system for data collection, storing, processing, modelling and prediction. Managing a huge amount of involved elements (nodes or groups of nodes, regions, devices or groups of devices in the Edge) requires certain levels of autonomy, to avoid overwhelming communication towards a central node, avoid dependency on fully-reliable network connections, and avoid high response-times when querying a remote and most probably overloaded Cloud system. This implies the distribution of modelling and prediction, where each autonomous environment is in charge of retrieving and modelling their data, without renouncing to share such obtained knowledge across peers and overseer nodes.

The proposed **Learning Plane** (LP) covers the management of such distributed modelling, as a layer connected to the **Data Plane** (DP) retrieving data from the system and environment, and to the **Control Plane** (CP) providing recommendation and prediction services, oriented to management and orchestration for either centralised and distributed management agents in the Edge-Cloud Continuum. To accomplish its mission, the LP requires the following capabilities:

- **Model retrieval**: Obtain the generated or updated models from every part of the system, autonomous and general orchestrators, monitors and analytic engines. This retrieval needs to be done over a platform or technology that avoids unneeded centralisation, and allows local models to stay near-data and near-user on the Edge.

- **Model storage**: Provide persistence and availability to the retrieved models, to be reachable from all the distributed and centralised orchestrators. This storage must be again implemented over a distributed platform, where local nodes can keep control and easy access, and far nodes can retrieve without excessive overhead.

- **Model federator**: Allow functions to federate the retrieved models, allowing efficient model selection when choosing among models from different sources or efficient aggregation when generalising models from different sources. The platform allowing such federation operations must allow operations either in the full vertical range of the Continuum, near the model storage or near-data, considering the computing capabilities of the nodes in the vicinity.

- **Model executor**: Provide availability to an execution-ready environment, near-storage or near-data, to use the retrieved models for prediction, recommendation, update, refinement and aggregation, without depending on components (nodes or resources) with low reliability or availability. The technology for orchestrating the orchestrator must be aware of the computing

capabilities across the distributed system, and the placement for data and models, considering efficiency when transporting models and data towards computing resources.

The indicated capabilities must ensure the proper execution of the orchestration functions in terms of policy enforcement (Control Plane), data retrieval and sharing (Data Plane) and model usage and federation (Learning Plane). Such functions can be implemented as **data connectors**, or pipelines of data connectors, where the inputs are retrieved from the Data Plane, and the resulting data are the information needed for decision making on the Control Plane.

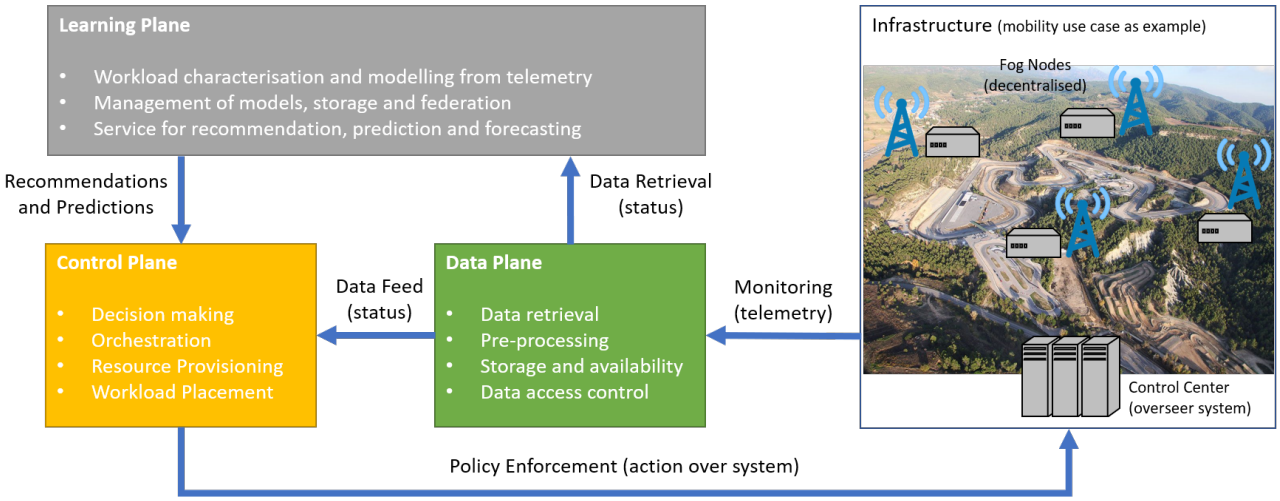



Figure 1: Learning Plane as a provider for Model Management, Model Storage and Provision, System Modelling and Characterisation, and Service for Prediction and Forecasting

To achieve this, CloudSkin is designing the early prototype involving different technologies, algorithms and methods on 1) workload characterization, by collecting monitored data from the system (i.e. resources telemetry and quality of service indicators), modelling it for behavior identification towards prediction or forecasting; 2) model distribution, for model collection and aggregation, towards federated training, storage and usage; and 3) model usage on a distributed environment, in a reliable, available and scalable platform and technology. At the time of this report, we are studying the different frameworks on containerization, serverless runtimes, object storage and active storage, to implement a first version of the prototype on a physical environment. Also, we are benchmarking different methods and plugins for analytics flow management, e.g. in Apache Spark workloads, dedicated to select the configurations and locations for serverless execution and storage environments closer to the data.

A first version of the integrated prototype is being designed using as guide the **mobility use case** (Task 5.4), contemplating a case with a full usage of Data-Control-Learning components in a real physical scenario, with the expectation of being generalised to the other use cases. As the implementation of the prototype is focusing on data connectors deployed over serverless and micro-function environments, the different use cases are also focusing on exposing functionalities and properties in the form of data connectors, to be composed over an agnostic Learning Plane regardless the content of the applications (not to confuse with the characterisation of the workload applications).

## 2.1 Development Progress

The first task on the Learning Plane is an "Initial Design" (T2.1 and T5.1), followed by development stages for WP5 (Tasks 5.1, T5.2 and T5.3) starting with focus on the "Workload Characterization" and "Planning & Provisioning Algorithms". This deliverable focuses on the Initial Design and Workload Characterization & Planning Algorithms. During the following months, planning and integration with the use cases (T5.4, T5.5, T5.6 and T5.7) will start, integrating the T5.1, T5.2 and T5.3 algorithms with the architecture and the technologies provided and developed by all involved CLOUDSKIN

partners. Table 1 shows the stages, along with the corresponding efforts and WP5 tasks, also the sections of this deliverable where progress and experiments are provided.

| | Stage 0 | Stage 1 | Stage 2 | Stage 3 |
|---|---|---|---|---|
| Efforts | Learning Plane Design | Workload Characterization | Instrumentation & Deployment | Integration with Use Cases |
| Ref. Tasks | T2.1, T5.1 | T5.2 | T5.3 | T5.4, T5.5, T5.6, T5.7 |
| Report Detail | Section 3 | Section 4 | Not started yet | Not started yet |
| Experiments | NA | Section 5 | Not started yet | Not started yet |

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

# 3 Early Implementation of the Learning Plane

This section extends "Learning Plane design and integration" from **Deliverable D2.1**. For the fundamental details on the integration of the Learning Plane with respect to the rest of the architecture, refer to that deliverable released in parallel with this deliverable.

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

- **Modelling the System**: This functionality involves the set of algorithms for system modelling and characterization. For this, different methods are being explored and benchmarked, starting from naïve methods (heuristics), through statistical and machine learning methods we have been previously researching and adapting for this kind of scenarios, such as ThetaScan [1] and AI4DL [2], oriented to the detection of statistical properties in the workloads as multi-variate time-series, or even time-series-based neural networks such as Transformers (current study in process of publication). Current efforts are focused on accurate models detecting unexpected and extreme changes on workload behavior with respect to resource demanding, i.e. peaks that could affect quality of service.
- **Model Federation and Storage**: This functionality involves the distribution of the models towards sharing and aggregation (what is "learned"); also the meta-data related to the models, datasets and execution environments for the algorithms fitting and inferring data using the models (the indications on "how to use the model"); and the objects containing the implementation of those algorithms (the model's "engine" application). Different candidates were considered as distributed file-systems and object storage platforms, as an example for the first ones Hadoop HDFS for distributed data with replication [3, 4] and GekkoFS for distributed in-memory volatile data [5] were considered. Such systems are a baseline option without the optimizations required for the current case. Therefore, our preferences are object storage systems, like GEDS [6], providing a distributed storage system with replication and resilience, developed by the IBM partners of CLOUDSKIN.
- **Provisioning Execution**: This functionality involves the execution of prediction and recommendation, and provisioning algorithms with the retrieved data and learned models. Current technologies involve containerization of the fitting and inference engines (e.g. PyTorch, TensorFlow, ScikitLearn, etc.) through Kubernetes and/or K3S on the one hand, and micro-function serverless computing frameworks like Lithops [7] or WebAssembly (WASM) [8] on the other. Actions to be performed include model and meta-data management (e.g., training, prediction and forecasting, model aggregation, model updating), that must be executed near the data, either source or storage.

## 3.1 Architecture and Methods

The general functional schema and integration with the CLOUDSKIN architecture is detailed in **Deliverable D2.1**. Here it is summarized while adding the indications referring to the specific technologies selected.

### 3.1.1 Architecture Design

In Figure 2 we observe the schema for an **inference** query from the Planner component. In the example, an application arrives at the system and it needs to be deployed. The Planner, acting as the *orchestrator*, needs information about how the application will behave to make the correct placement. While a naïve approach would follow the specifications from the user or the records from a previous execution, a "smart" approach will attempt to forecast some *a-priori* information from the application

to place and provision it with less resource waste or more quality of service. Notice that the Planner does not need to make just a single decision on the app, but can be continuously monitoring and making decisions on whether to re-provision or migrate the application.

In this example, after the arrival, the Planner gets some information from the application, e.g. a profile, a trace from a previous execution, or on-line monitoring from the first seconds/minutes of the application execution. Using this information, the Planner asks the Learning Plane for additional information on the application, e.g. its expected demand for resources, its quality of service if placed in a certain node, etc. The "Learning Application" receives the query for predictions/forecasting from the Planner through a service, API or remote procedure call. This Learning Application is a machine learning engine, in charge of loading an already trained model capable of performing such predictions or forecasts, performing inference with the application information, the model and additional data such as meta-data or telemetry from the system. The Learning App returns the prediction or forecasting to the Planner, that uses it along with the Planner's information to make a decision on how to provision and place the application.

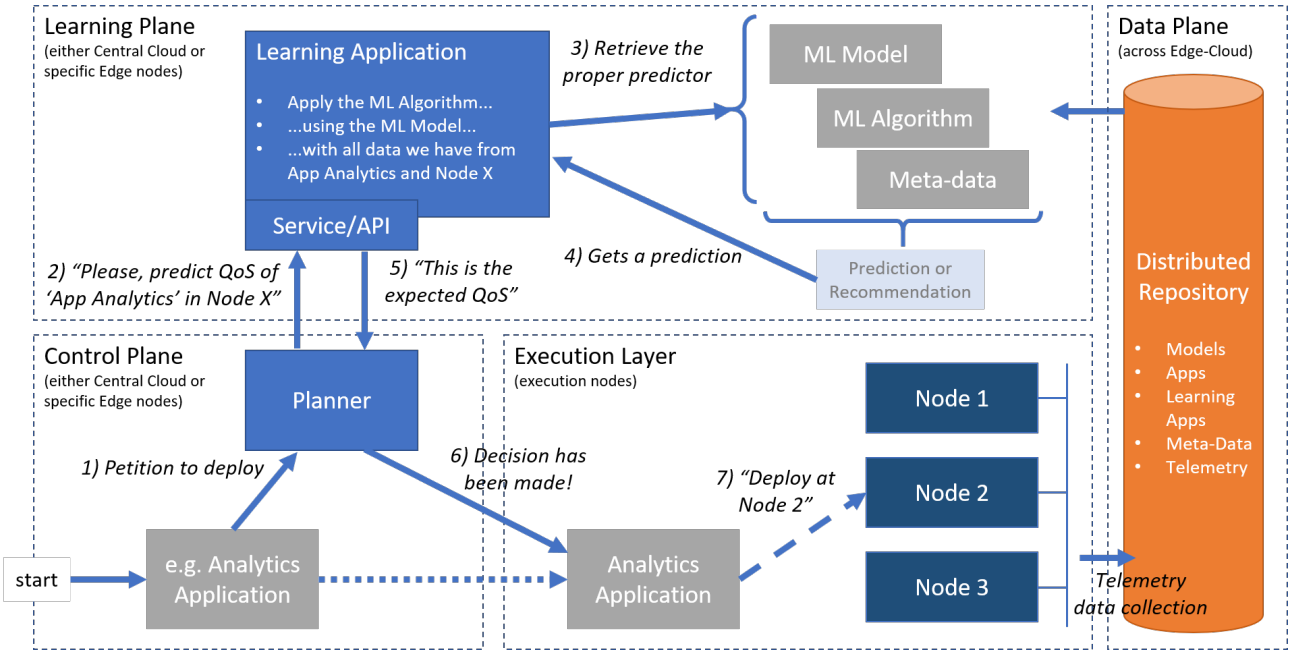



Figure 2: Example of an early functioning schema of the Learning Plane on a query for application placement and provisioning.

Here we describe more in detail the components indicated in Figure 2:

- **Planner**: This is the orchestrator, or the component capable of enforcing the placement and provisioning policies. I.e., this component is the one submitting and deploying an application into a set of resources and machines. This component relies on the Control Plane.
- **Learning Application**: This is an engine (e.g., script or application with PyTorch, TensorFlow, Scikit-Learn, or any other machine learning library) capable of training or loading a model to perform inference on the data received. It contains all the code to receive queries from the Planner (e.g., a service, an API or a remote call procedure), to train or retrieve a machine learning model from the repository of trained models, to perform the inference using the model, the data from the planner, the meta-data and additional data from the system. It returns the results of the prediction, forecasting or recommendation.
- **ML Model**: This is a trained machine learning model, created by the Learning Application (or uploaded by a third party, while compatible with the Learning Application), that receives new

inputs to generate a prediction, forecasting or recommendation. It is stored in a Distributed Repository, to cover situations in which the learning or forecasting are performed in a decentralized infrastructure (e.g., a Cloud-Edge system, a multi-machine data-center, etc.).

- **ML Algorithm**: This is the method to train and infer a machine learning model of a kind (the selected machine learning algorithm). It can contain the description of the modelling, the hyper-parameter selection, or even the code to be included in the engine depending on the chosen learning platform. E.g., if the Learning Application uses PyTorch, the algorithm would contain the neural network architecture plus the hyper-parameters and meta-data such as data formatting, and after passing data through PyTorch using the algorithm information, a trained model would result as the output.
- **Meta-Data**: This is the additional information that both the Learning Plane's and the Control Plane's components require from the system, such as *system architecture*, *system status* (telemetry), *security information & permissions*, etc.
- **Prediction or Recommendation**: This is the output of loading the model and algorithm into the engine in the Learning Application and feeding with new data to be inferred.
- **Distributed Repository**: This repository contains the objects and information required for proper "smart" decision making. It contains the models, applications, learning applications, meta-data and telemetry. As Learning, Control and Data planes can be distributed, such objects must be accessible by all involved components, while leveraging distribution to increase accessibility, replicability and resilience.

In a use case example (e.g. Task 5.4), the *NearbyONE* orchestrator, located in the central Cloud controlling all the Edge nodes, acts as the Planner. When an application needs to be deployed, NearbyONE performs a call to an *oracle* (a Learning Application) deployed in the same central Cloud indicating the details for such application, requiring an estimation for the QoS when placing it in each candidate Edge node. The oracle retrieves from the model repository the ML application designed to model and predict QoS, the trained model and algorithm for that kind of application, and the latest telemetry of the Edge nodes. The oracle executes the application in a node from the central Cloud, to produce the QoS predictions. The oracle returns a list of expected QoSs, one per Edge node, and NearbyONE decides to place the application in the Edge node with higher predicted QoS value. While this example focuses on "mobility", other use cases like in Task 5.6 would consult the oracle about outsourcing AR generation and streams to close or far machines in the operation room, depending on the use of resources and network status, to get proper real-time AR in the operation room.

### 3.1.2 Scalability and Distribution of the Architecture

In a Cloud-Edge scenario, the scalability of a design can rely on a "horizontal" or "vertical" approach. In Cloud systems, applications and workloads can be distributed horizontally among clusters of nodes with similar or specialized purposes. However, in Edge systems, distribution can be vertical between Cloud nodes (up) and Edge nodes (down), where scalability becomes horizontal across the Edge, or vertically distributed between Edge and Cloud. This approach allows to decentralize the architecture, where Cloud elements can be off-loaded to the Edge, scaling across Edge nodes while using the Cloud for specific high-performance tasks. For the CLOUDSKIN project, the Learning Plane will start with a centralized approach, where all control and management operations are in the Cloud, while the aim during the project is to explore ways to off-load such management to the Edge, allowing scalability and autonomy to the different Edge regions.

Therefore, the basic scenario corresponds to a centralized system where management and orchestration relies in a single location (single- or multi-node). For the described examples before, a centralized system must have visibility and control of all the infrastructure, concentrating all decision making on Edge placement and provisioning. The Planner and Learning applications are in a

central Cloud, while the storage can still be distributed for the sake of collecting telemetry from the Edge nodes without pushing all data to the central Cloud. For the early designs of CLOUDSKIN, we are pushing for this approach for its initial simplicity regarding to the proposed use cases.

However, an alternative is a distributed system, where management and decisions are distributed across Edge nodes (or intermediate Fog nodes), as shown in Figure 3. In case of large-scale systems that must maintain certain autonomy, the Control and Learning Plane can be distributed (along with the already distributed Data Plane). For this, each Edge node or group of Edge nodes can coordinate locally and later report or coordinate with the Central Cloud. Each Edge node/group would perform orchestration and forecasting of applications autonomously, leveraging the distributed repository to share models, applications, learning applications, meta-data and telemetry from other nodes, being able to create larger federated schemes on sharing data and models. In this early stage of CLOUDSKIN, we are focusing on a centralized architecture, but without losing sight of distributed architectures, as they will allow off-loading part of the management load from the Cloud to the Edge.

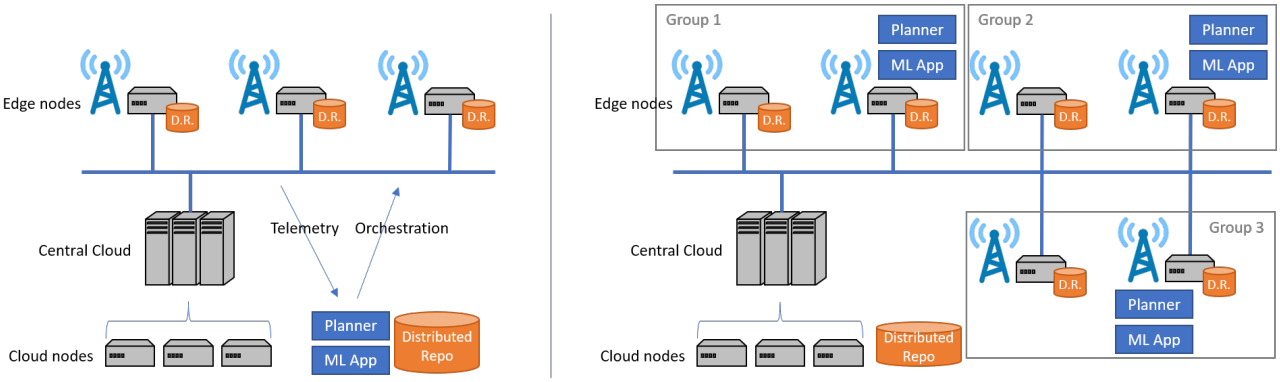



Figure 3: Example of centralized (left) vs. decentralized (right) schemes where the control and learning is placed in the Cloud nodes or in the Edge nodes

### 3.1.3 Training and Updating

In a centralized scenario, all collected data is stored in the (distributed) repository to be retrieved in the Central Cloud. There, machine learning models are trained using the application characterized data plus telemetry and different meta-data, then stored in the model repository in the same storage. However, in a distributed scenario, each Edge node or group of nodes can create their own local models, fitting their models to the specific scenarios and sensors. A distributed learning scenario (or a centralized in which the learning load is off-loaded to the Edge or Fog nodes, to save on networking and HPC resources in the Cloud), allows that every autonomous element generates their own set of models, to be shared using the distributed repository, and then aggregate if needed through Federated Learning techniques to obtain more general models.

Finally, for model updating, part of the research on workload characterization focuses on discovering when models are no longer valid due to a shift on the workload behavior. In that situation, stored telemetry for the most recent executions can be used to update models or fit newer models, to be stored in the distributed repository. Being capable to store different versions will allow retrieving and comparing old and new models for behaviors with recurring (periodic) patterns.

# 4 Methods and Algorithms for Scaling and Provisioning

Here we present the methods for scheduling and scaling applications upon resources, and towards modelling and characterising workloads, as the initial tool-set for making predictions and proposing decisions from the Learning Plane towards the Control Plane.

## 4.1 Scalable Scheduling for HPC/HPDA

The first subset of tools refers to the Scheduling and Resource Policy methods based on *a-posteriori* knowledge, with the objective to establish a base for machine learning models to be deployed over, driving the decisions of the scheduler and resource provisioner. The two approaches to be proposed in the first stage of CloudSkin are *Floki*, adapted from the initial works from A.Nestorov [9], and *Custom Scale-Out*. These both technologies are complementary methods to orchestrate pipelines, deciding how to efficiently communicate distributed applications and functions, and to orchestrate scalable tasks inside an application, deciding how much distributed resources to dedicate to a certain set of tasks in the application.

### 4.1.1 Distributed Workload Interconnection through Floki

Efficient intermediate data sharing between workload tasks represents a key challenge for chained task execution, especially when dealing with data-intensive workloads. The current state-of-practice in cloud environments is to exchange data through shared object storage, e.g., Amazon S3 [10]. In the context of data-intensive workloads, represented as Directed Acyclic Graph (DAG) and characterized by a considerable amount of intermediate data transfers, shared object storage becomes a bottleneck for efficient inter-task communication due to its high-latency access. Recent studies tackle this problem by implementing optimized exchange operators, using multi-tier storage combining slow with fast storage or solely remote in-memory storage, exploiting per-node caches, co-locating tasks on a single container, or handling external storage on long-running Virtual Machines (VMs). However, these methods either use domain-specific optimizations, require two copies of data over the network, are not fully transparent to the user, or break the advantage of fine-grained scaling.

We tackle the intermediate data sharing problem by relying on Floki [9], a system that proactively enables faster point-to-point data sharing exploiting local resources and TCP socket connections. Floki's architecture solves the centralized storage bottleneck by offering direct communication between tasks, where data exchanges are managed on a producer-consumer functions pair level, minimizing data copying over the network. The system's architectures consider five key components: the *data-* and *sync-pipe*, the *client* and *server sockets*, and the *forwarding agent*:

- **Data- and sync-pipe:** The two pipes, exposed on a local persistent volume (PV), allow to exchange data between the user container and the local *forwarding process*. While the *data-pipe* represents the data communication channel, the *sync-pipe* synchronizes Floki with the user container letting the *forwarding agent* to write on the *data-pipe* only when the user container is ready to receive[1].

- **Client and server sockets:** These are the key components in charge of transmitting data between pairs of nodes. We choose to implement TCP sockets guaranteeing features such as error checking, ordered data delivery, and enabling uniquely identified connections between two endpoints, i.e., combining client and server sockets.

- **Forwarding agent**: This component, instantiated into a process in the host namespace, represents Floki's architecture core component. At a high level, the primary purpose of this component is to drive inter-node communication, forwarding data directly from the producer to the consumer function. More precisely, its role is threefold. First, it creates and sets up the required TCP connections. Second, it supplies the necessary input data to the user container. Third, it forwards the data produced by the user container to the following functions in the

[1]Prevents the write operation from receiving a broken pipe signal when the read file descriptor referring to the pipe read end is not opened.

**Algorithm 1** Floki's forwarding agent algorithm.

```
 1: procedure RECVDATA(sSocket, listObjsToRecv)
 2:     AquireLock(dataPipe)
 3:     for all objName ∈ listObjsToRecv do
 4:         dataSize = RecvAndWriteSize(sSocket, dataPipe)
 5:         RecvAndWriteData(sSocket, dataPipe, dataSize)
 6:     end for
 7:     ReleaseLock(dataPipe)
 8: end procedure
 9: procedure SENDDATA(cSocket, listObjsToSend)
10:     for all objName ∈ listObjsToSend do
11:         if currentObjName == objName then
12:             SendDataSize(cSocket, sizeBuffer)
13:             for k = 1 to ⌈outDataSize/packetSize⌉ do
14:                 SendDataPacket(cSocket, dataBuffer)
15:             end for
16:         end if
17:     end for
18: end procedure
19: procedure FORWARDINGAGENT( )
20:     producers = GetSocketsProducersNames(sSockets)
21:     SendOwnName(cSockets)
22:     WaitReady(syncPipe)
23:     for i = 1 to #sSockets do
24:         thsIn[i] = thread(RECVDATA, sSockets[i], listObjsToRecv)
25:     end for
26:     WaitThreadsEnd(thsIn)
27:     for j = 1 to #cSockets do
28:         thsOut[j] = thread(SENDDATA, cSockets[j], listObjsToSend)
29:     end for
30:     for all outObjName ∈ outObjsNames do
31:         currentObjName = outObjName
32:         outDataSize = ReadDataSize(dataPipe, sizeBuffer)
33:         for k = 1 to ⌈outDataSize/packetSize⌉ do
34:             ReadDataPacket(dataPipe, dataBuffer)
35:         end for
36:     end for
37:     WaitThreadsEnd(thsOut)
38: end procedure
```

chain. Since the *forwarding agent* mainly performs write/read memory buffers operations, we expect its overhead to be negligible. To proactively set up the communication infrastructure for the specific workflow, it internally stores the function name to which it refers and the ordered lists of data object names to receive/send for each of the previous/following functions in the workflow. In addition, it stores the mapping between the following functions in the workflow and the IP address of the nodes to which they have been scheduled to account for the underlying scheduler functions placement. Based on this information, it automatically derives the necessary number of *server* and *client socket* connections, i.e., #*sSockets* and #*cSockets*.

Algorithm 1 shows the pseudo-code of the *forwarding agent*, whose top-function is represented by the FORWARDINGAGENT procedure. Once the #*sSockets server* and #*cSockets client socket* connections are opened and set up, the *forwarding agent* receives producer functions' names getting the correspondence with the *server socket* connections (line 20). Storing for each producer the list of data object names to receive allows the *forwarding agent* to know the number and the names of the data objects transmitted on each *server socket*. Then, the *forwarding agent* sends the related function name on all *client sockets* (line 21). Since the *forwarding agent* starts before the workflow is deployed, to respect the coordination of the pipe operations, the *forwarding agent* waits for the user container ready signal eventing that it is running and ready to read data from the *data-pipe* (line 22). Once received, the *forwarding agent* creates the #*sSockets* input threads (lines 23-25). The input threads alternately write

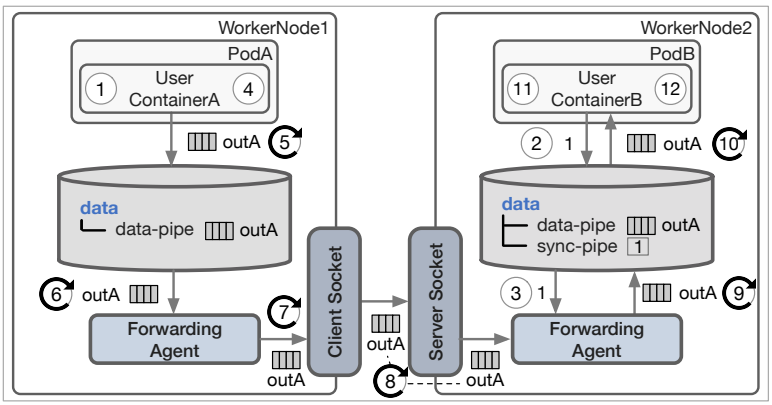



Figure 4: A step-by-step example of Floki, where TaskA and TaskB produces and reads the intermediate data object outA, respectively.

on the *data-pipe*, sending first the data size (line 4) and then the data content read in a packet-based fashion from the corresponding *server socket* (line 5). The input threads' *data-pipe* write operations follow the order declared in the stored list of data objects to receive. To handle *data-pipe* contention, the threads' write operations are synchronized by acquiring (line 2) and releasing (line 7) a lock. When all input threads finish (line 26), the output threads, in charge of sending the user container-produced data on the *client sockets*, are created (lines 27-29). For each produced data object, the *forwarding agent* reads the data object size *outDataSize* (line 24) and the data object content from the *data-pipe* (lines 32-35). To guarantee output threads access to both data object size and content, the *forwarding agent* read operations store them in $sizeBuffer$ and $dataBuffer$ shared memory buffers. Finally, each output thread sends the buffers on the *client socket* (lines 12-15) if the current received data object, i.e., *currentObjName*, belongs to its list of objects to send (line 11).

Figure 4 illustrates how Floki works step-by-step with a simple example of a two functions workflow. The first function reads the workflow input stored in the object storage and creates the intermediate output *outA*. In contrast, the second function reads the intermediate data object *outA* and computes the workflow output *out*, saving it in the object storage. For data transferred in a packet-based fashion, in Figure 4 we highlight the operations performed multiple times with circular arrows. While the first function reads the workflow input from the object storage (step 1), the second function sends the ready signal on the *sync-pipe* to the local *forwarding agent* (step 2), eventing it is up and running and waiting to read data on the local *data-pipe*. During the intermediate data object *OutA* computation (step 4), the *forwarding agent* on the second node reads the ready signal (step 3) from the *sync-pipe* and waits for the first packet on the *server socket*. Once the first function ends to compute the intermediate data object *outA*, it first writes *outA* size packet and then iteratively writes *outA* content in packets on the local *data-pipe* (step 5). The local *forwarding agent* reads the packets from the *data-pipe* (step 6) and sends them to the *client socket* (step 7). On the consumer side, packets are read from the *forwarding agent* (step 8) and written to the local *data-pipe* (step 9). Finally, packets are read from the second function (step 10), which, once received *outA*, computes the workflow output *out* (step 11) and stores it in the object storage (step 12).

### 4.1.2 Custom Scale-Out

Current systems allocate resources at the per-application level, using general heuristics like fair scheduling, shortest-job-first, and simple packing strategies, ignoring completely application characteristics. For instance, by default, the scheduler of Spark [11] allocates the resources in a FIFO manner: the first stage gets priority on all available resources, then the second stage gets priority, etc. Figure 5 shows an example of a computation over 10 executors when relying on Spark's default FIFO policy. As can be noticed, this leads to a high inefficiency: looking to a given stage as a black-box, all resources are allocated even if a given stage does not need all of them. Consider-

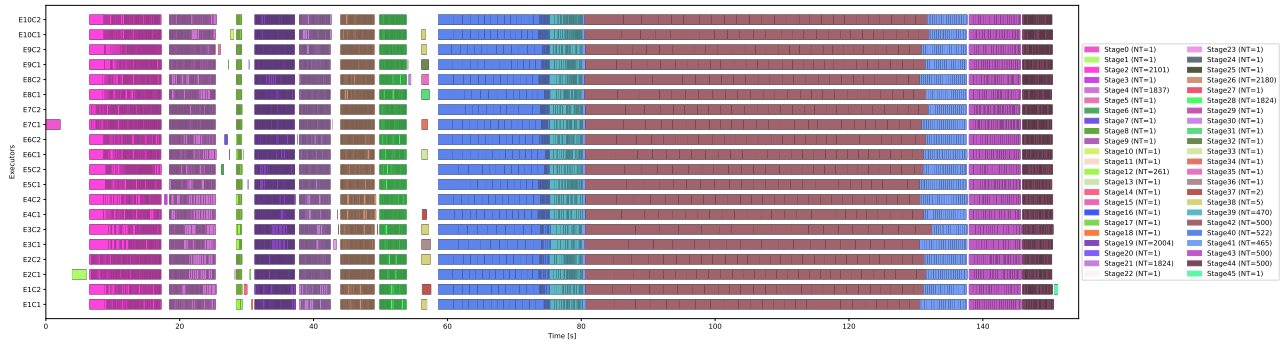



Figure 5: TPD-DS q72 execution on 10 executors (2 cores each) with Spark's default FIFO policy. Stages are identified with different colors, tasks are delimited by vertical bars inside each stage, and white spaces identify time-windows in which executors are free.

ing application-specific characteristics, e.g., stage parallelism levels, systems can efficiently allocate shares of resources to application stages to achieve similar or higher overall performance. Setting the scale-out level at lower granularity, i.e., per-job or per-stage, avoids wasting resources on jobs/stages with little inherent parallelism or running on small input data, leaving more free resources to jobs/stages with large input which can harness additional parallelism. Efficient utilization of resources matters for both the service provider and the user: the provider can save millions of dollars at scale, and the user can benefit from higher performance at the same or even lower cost.

Among others, works like [12] tackles this problem at per job-level by presenting Decima, a system using Reinforcement Learning (RL) and neural networks to learn workload-specific scheduling algorithms without any human instruction beyond the high-level objective of minimizing average job completion time. In particular, it uses existing monitoring information and past workload logs to learn sophisticated scheduling policies automatically. Even though Decima outperforms existing heuristics, reducing the average job completion time of TPC-H [13] query mixes by at least 21%, it does not represent a solution for production environments. More precisely, Decima requires a large number of offline simulated experiments to train its RL algorithms.

In contrast, we target a fully-online solution, requiring a few execution steps to predict a near-optimal per-stage share of resources. Furthermore, to reduce the space of scheduling policies, Decima's authors do not consider stage-level parallelisms. Finally, Decima's high-level objective is to minimize average job completion time. Differently, we believe that a more flexible system is necessary, leaving the optimization strategy to be decided by the user. For instance, the user may decide to either minimize the cost, maximize the performance, or balance the performance/cost ratio.

Current efforts on the CustomScaler are focused on methods for retrieving additional a-priori information on what to expect from a task and stage to conclude given a certain level of scaling. The current Work-in-Progress towards dynamically adjusting the number of executors (and therefore resources) dedicated to the application.

### 4.2 Workload Characterization

The previous mentioned methods for scalable scheduling are based on direct policies, where the system is modelled *by hand* by an expert creating a Utility Function that relies on a-posteriori data for *heuristic* provisioning. The principal contribution of the Learning Plane is to allow "Smart Management" through retrieved knowledge through modelling and prediction (this is, Machine Leaning), first to plan provisioning and scalability ahead through predicted a-priori information, and second to deal with uncertainty from extremely variable workloads. During the first exploration process and proposition of methods for workload forecasting, we propose a set of methods developed at the Barcelona Supercomputing Center, in collaboration with academic and industrial partners involved in Cloud provisioning (e.g., IBM), that can deal with the principal problems that arise from observed

Edge-Cloud workloads. Such methods, capable to identify behaviors on workloads, are being incorporated into the scalable scheduling methods, as additional forecast a-priori data for planing proactively instead of reactively. Here we present the methods, with the considerations for the workloads and infrastructures to deal with in the Edge-Cloud environments.

### 4.2.1 Periodic Behavior Detection - ThetaScan

Current methods for vertical auto-scaling (providing additional resources to a single job or application depending on their latest demand) are well suited for stable traces where increase and decrease of resource demand is progressive or under controlled upper and lower bounds. However, such strategies fall into the reactive set of methods, where forecasting depends on the trend in-course, blind to patterns that the demand may contain, based uniquely in a bounded time window. As a result, such naïve strategies fail to provision applications that follow specific behaviours without constant resource usage, commonly found in CPU and Memory time-series. Therefore, detecting these behaviours and adapting the corresponding provisioning requests is crucial.

One of the principal patterns we pursue to detect towards proper provisioning is **periodicy**, and that's the purpose for the ThetaScan algorithm [1]. We can observe in in Figure 6 how *periodic* behaviors, as patterns repeated in regular time intervals, maintaining a time window without detecting the existence of such behavior nor its period (a task hard to fully automate even with methods to detect periodicity) may cause a catastrophic delay on its assumption for required resources. In the worst-case scenario, shown in the figure, the recommended request is always the opposite of what it should be provided. So, computing requests using solely the statistics of the previous time window presents some weaknesses when provisioning periodic behaviours. The Theta-Scan method is intended for detecting periodicity, leveraging the Theta-Model. Here we introduce our implementation of the Theta-Model based on the Simplified Exponential Smoothing method (SES), also known as Holt-Winters smoothing [14], with a detrending and deseasonalization of the analyzed time series.

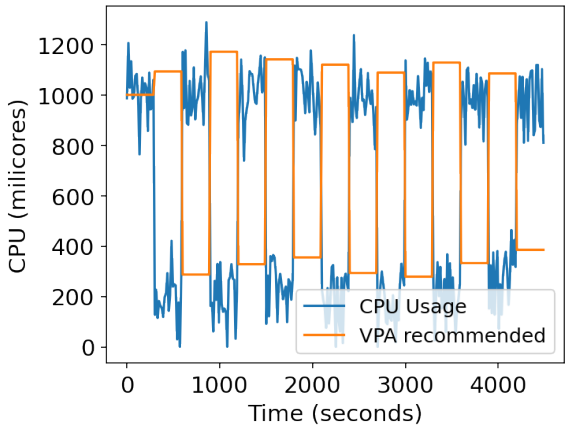



Figure 6: Example of worst-case scenario given a periodic trace, for provisioning resources without detecting or considering the exact behavior period.

The targeted behaviors to be detected in the CPU/Memory consumption are related to the Trend and Seasonality (i.e., periodic patterns). Modeling the time-series will involve a detrending process, where we extract (and keep) the trend of the time-series, separating it from the potential seasonality and drift. Next, the resulting series are deseasonalized, where we extract (and keep) a potential season (periodic pattern) in the series, separating it from the drift. And finally we can model the drift for future prediction. Forecasting using this model requires to extend the trace from SES, then add the found seasonal pattern and the found trend.

The following equations detail the SES implementation towards Theta-Model fitting and forecasting. Equation 1 shows the steps for extracting the trend and season patterns. First, the raw trace $r$ is fitted with a Linear (or Polynomial) Regression to extract the trend of the trace, keeping the trend

| | |
|---|---|
| $raw\ trace : r$<br>$trend : tr = a \cdot t + b = fit(r)$ (1)<br>$detrended\ trace : w = r/a - b$ | $expected\ period : p$<br>$seasonality : v = average\_season(w, p)$ (2)<br>$deseason\ trace : x = w/repeat(v, len(w)/len(v))$ |
| $s[0] = s[1] = x[0]$<br>$s[t] = \theta \cdot x[t] + (1-\theta) \cdot s[t-1] = \hat{x}[t+1]$ (3) | $filter : \hat{x}[t+1] = \theta \cdot x[t] + (1-\theta) \cdot \hat{x}[t]$<br>$pattern : v = [v_1, v_2, v_3 \dots v_p]$ (4)<br>$trend : tr = a \cdot t + b$ |
| $forecast : \hat{x}[t+1] = \hat{x}[t]$<br>$reseason : \hat{w}[t+1 \dots t+p] = \hat{x}[t+1 \dots t+p] \cdot v$ (5)<br>$retrend : \hat{r}[t+1 \dots t+p] = \hat{w}[t+1 \dots t+p] \cdot a + b$ | |

Table 2: Equations for the specifically designed implementation of the Theta-Model

(slope $a$ and intercept $b$), and such trend is removed from $r$ to create the detrended trace $w$. Equation 2 shows the deseasonalization. This step requires the only hyper-parameter of the Theta-Model necessary in our scenario, the expected period $p$. $p$ is the time-steps that we expect to be repeated during the time-series. The average season is obtained by splitting the time-series in segments of $p$ (each expected repetition), and averaged element-wise. E.g., a series of length $T$ and an expected period $P$ would have segments $\{1 \dots P,\ P+1 \dots 2P, \dots\}$, and the resulting average season would be a vector of length $P$ with the mean values of $\{1, P+1, 2P+1 \dots\}$, $\{2, P+2, 2P+2 \dots\}$, etc. The average season is kept, and it is removed from all the segments of $w$ creating the deseasonalized trace $x$. Equation 3 shows the Simplified Exponential Smoothing filter. For each time-step, the filter predicts a trade-off between the previous observed real value and the previous expected value, weighted by parameter $\theta$. While $\theta$ could be considered a hyper-parameter, general experimentation in our datasets showed us that a fixed value can be introduced for all runs of the filter (not like $p$, that must be adjusted on each trace or time-series segment). Equation 4 shows the final model, composed by the fitted filter, the seasonal pattern, and the trend. Finally, Equation 5 shows the procedure for forecasting the series using the created Theta-Model. The filter just produces as next-step $t+1$ the expected $t$, for the lack of a-priori information. When forecasting a segment of $n$ values, this value is extended $n$ times. Then the seasonal pattern is introduced again, consecutively and element-wise by the opposite operation when removing it to deseasonalize. And at last, the trend is reintroduced by scaling the slope and offsetting it. Remember that, when applying these transformations over the time-series, one must have into account the offset of time $t$ into the seasonal pattern before reintroducing it (season offset $= t\ mod\ len(v)$).

Through this implementation of the Theta-Model, the Theta-Scan algorithm [1] showing in Algorithm 2 attempts the automated detection of periodicity towards next period provisioning of resources. The objective is to leverage the method to detect periodic behaviors from the Learning Plane, towards provisioning or placing applications and resources in advance.

#### 4.2.2 Behavior Similarity Detection - AI4DL

While ThetaScan is intended to detect periodic patterns in workloads, another approach to detect and discover patterns is to cluster telemetry traces, to be later examined by the system architects (human method) or used for statistical estimation (automatic method). The next technology is AI4DL [2] created in a collaboration between IBM and the Barcelona Supercomputing Center, purposed to characterize Cloud containerization for Deep Learning applications, but we consider its application for any kind of workload to be introduced in the CloudSkin ecosystem for non-periodic behaviors on

**Algorithm 2** The Theta-Scan algorithm

```
1:  procedure THETA-SCAN(training_set, test_set)
2:      best_model, best_period, errors_list = null
3:      best_error = ∞
4:      for i in 2 ... ⌊n_train/2⌋ do
5:          model = fit (data = training_set, period = i)
6:          results = predict (model = model, steps = n_test)
7:          error = evaluate (results, test_set)
8:          if error < best_error then
9:              best_model, best_period = model, i
10:             best_error = error
11:         end if
12:         errors_list = errors_list ∪ error
13:     end for
14:     return (best_model, best_period, errors_list)
15: end procedure
```

applications. The proposed methodology collects container resource usage (i.e., CPU and memory, but also network or disk), creates a model encoding these metrics to capture dynamics over the time dimension (behavior), clusters similar behaviors as unique phases, reduces the whole execution to a sequence of phases, and then estimates the resource requests per each phase. Figure 7 shows an example of real trace with differentiated phases detected by AI4DL.

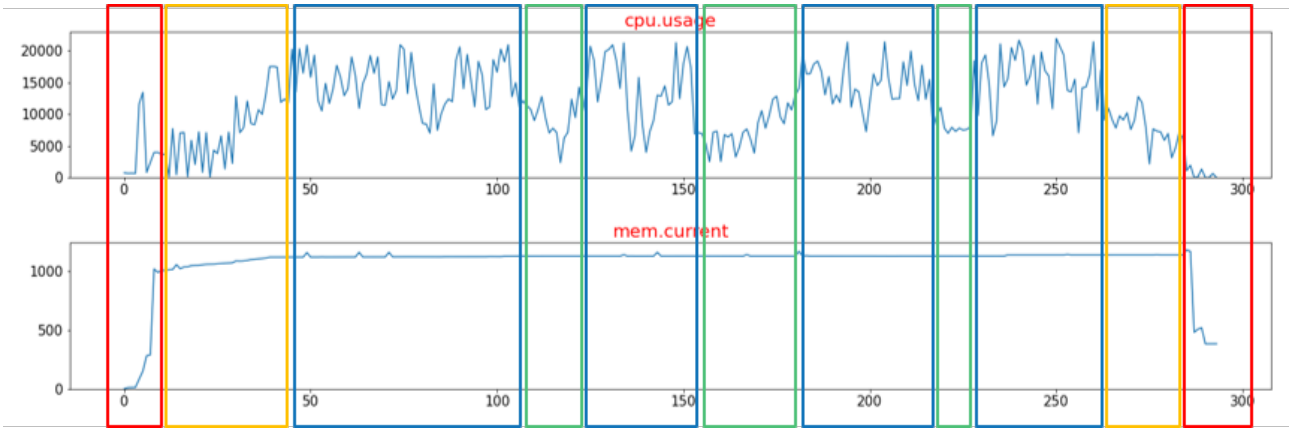



Figure 7: Phase discovery for multi-variate time series on telemetry using AI4DL

The resource usage metrics from the containerized application, including CPU and memory, are firstly encoded using a Conditional Restricted Boltzmann Machine (CRBM) [15]. CRBM is a type of Neural Network, capable of encoding multi-dimensional data into a vector of features, which is ideal for analyzing time-series patterns. Different from hashing functions, CRBMs can produce similar outputs from similar inputs, a property we are exploiting next. CRBM takes container metrics in a sliding time window as input, namely metric at time $t$ plus the history $t-1 \ldots d$, where $d$ represents the *delay* (i.e., a hyper-parameter denoting the size of the time window). The output taken from the CRBM is the encoding from its hidden layer, a vector of $h$ features encoding the input values. Such a vector embeds the instant metrics plus their previous $d$ history steps. Thus, the CRBM can identify behaviors in the time window of $d+1$ steps, where either $d$ (through re-training) or the aggregation time interval of samples is adjustable for long or short training sessions. CRBM at each time step of $1 \ldots T$ produces a $(T-d)$ encoding vector of size $h$, capturing the *behavior* of the resource usage at each time $t$. As CRBM encodes similar behaviors into similar codes, proximity-based clustering

methods, like *k*-means, can be used to group similar behaviors. The proximity-based grouping allows us to discover behavior sequences (phases) along the execution. As the total number of different phases is unknown a-priori, we need a prior analysis to determine *k* as a hyper-parameter [16]. Via passing real-time container metrics through the encoding and the clustering, the phases can be discovered at each time step. Given the metrics from a full execution of a container, we produce the sequence of phases, retrieve the statistics and the type of behavior for each phase, and leverage it to allocate resources. Notice that, as CRBM embeds time in the encoding vector, a phase behavior does not only describe the magnitude of CPU and memory usage but also their dynamics in the time window.

A full life cycle of a containerized application can be encoded as a sequence of phases, representing behavior changes in resource usage. Containers running similar workloads display similar behaviors (e.g., first phases are corresponding to *Memory.load*, next phases to *Intensive.CPU*, last phases to *Memory.unload*). A good representation of phase sequences can indicate what types of behaviors the containers are undergoing, how much resources they need, and in which order they consume resources. The resource usage of different containers may show similar patterns in their phase sequences. E.g., an auto-scaler, resource provisioning or policy enforcement mechanism can leverage behavior recognition, and use the estimated resource usages for a given sequence of phases, to proactively estimate and provision future resource demands.

#### 4.2.3 Characterization using Transformers

As observed in the previous experiments of workload characterization, the resource consumption of a workload is not constant neither stable. Resource usages can have steadiness, variability and abrupt behaviors. In some occasions, the change between different behaviors is abrupt forming what are commonly named spikes, or "burstiness" behavior. Such varied consumption leads Cloud resource providers to demand an allocate an amount of resources is by far over the average usage. This is an easy but inefficient solution for preventing the highest spike of resource consumption to be underneath the requested limit, killing or evicting the container, virtual machine or task before completing the execution. For this reason, the vast majority of resources in common Cloud infrastructures are on-line but underused, wasting energy while costing users and providers their (idle) running costs.

Research in Cloud computing has been working for years to avoid the waste of computing resources, through co-location and consolidation of applications, dealing with the added risks of overwhelming resources and producing a degradation of the provided quality of service. Current approaches focus on co-locating applications with different resource needs in the same computing nodes, avoiding resource competition, or co-locating applications with different demand or tolerance towards lack of resources, allowing low tolerance applications to lend resources to high tolerance ones when required. Such processes are transparent to the user, who only observes the progress of their tasks. However, this approach endures big problems when heavy spikes occur in bursty workloads, as such demand cannot be satisfied when another co-located application is also having a burst or a period of high demand. This impacts the bursty workloads, severely degrading its quality of service. For that reason, we are proposing new methods for predicting the need of resource allocation for workloads, putting special emphasis on spikes of resources usage, attempting to allow the resource manager to anticipate sudden demands.

There are plenty of works in the literature dealing with resource usage forecasting, attempting to predict with high accuracy time series on CPU and memory demand, but the vast majority of proposed models are old methods falling back into regression algorithms or variants like Random Forests or Support Vector Machines. Such methods do not incorporate the last advances in time series forecasting, not even multi-variate series, even less the latest advances of deep learning towards time series. One of such set of methods are the so-called Transformers, proposed by Google in 2017 [17]. Transformers have revolutionized the state of art of natural language processing, moving from recurrent neural networks to convolutional blocks with full-attention layers. Inputs are processed all at the same time, by different parts of the model, receiving between its multiple lay-

ers, all the outputs from their counterparts. These methods have been researched for the past years, showing their advantage over their predecessor models. The current state of art on Transformers include the SCINet model [18], based on convolutional blocks, with popular adaptions like the Informer [19] and YFormer [20], both simplifications of the original architecture with a few extra features for time-series. While the Informer model uses sparse convolutional layers to implement the attention mechanism to reduce drastically the neural network size, the YFormer model merges the Transformer architecture with the well-known U-Net neural network architecture, also allowing it to be used in application domains such as computer vision aside of the already mentioned natural language processing.

The principal target for using Transformers is NOT to forecast a time-series $T_{n+1} \dots T_{n+w}$ from $T_{n-w} \dots T_n$, but to forecast specific behaviors (i.e. spikes and sudden changes on the demand), as we are not interested on knowing the exact consumption but on the changes on its ceiling towards the next provisioning window. And current methods for evaluation, and hence for training and fitting models, are totally oriented to the default forecast matching (find $T_{n+1} \dots T_{n+w}$ with high accuracy), like MSE and MAPE, therefore not useful for Cloud provisioning. Models trained with such metrics put all focus on matching time series more than our real target, not evaluating abrupt spikes on resource demand correctly, nor conforming a proper analysis of the generated model performance towards our main interests. When testing Transformers using such metrics, we observe that they are very fast to compute, but with a heavy drawback: They do not correctly quantify the pronounce and abrupt changes of values, as shown in Figure 8. The three figures produce the same MSE metric but with totally different qualities, being the second one the one that properly predicts the spike in time and amplitude.

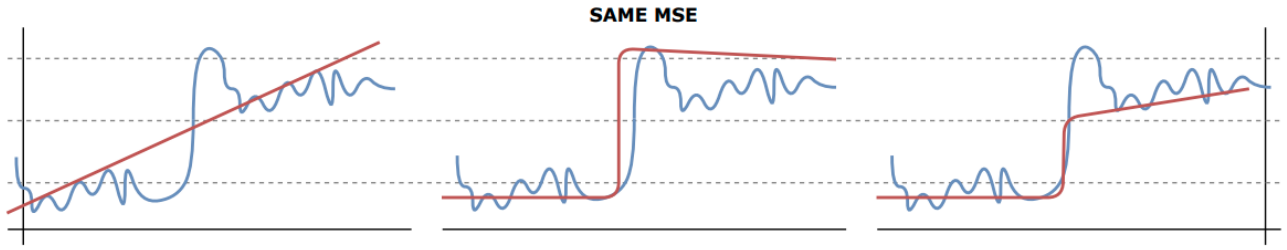



Figure 8: Example of three different predictions (against the ground truth) performing the same value for the MSE metric

As current works do not focus on specific behaviors and evaluation functions tailored for such, we are focusing our research on resource prediction towards a new set of functions that allow attention neural network such as Transformers to learn specific patterns to be discovered or forecasted. In the CLOUDSKIN project, we are progressing in the following directions:

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

This presumably constitutes a very hard problem, and the main work will consist in studying its feasibility, caveats and opportunities. Nevertheless, the model will be able to use the information from repeating patterns of specific traces (either seasonality or periodicity) to create a more accurate

prediction towards our objective of having proper information to decide the amount of provided resources for a workload.

# 5 Early Prototyping and Experiments

Here we introduce the first advances on the development, integration, adaption, benchmarking and testing of the first components selected to build the CLOUDSKIN Edge-Cloud Continuum stack.

## 5.1 Environment and technologies

The methodology employed to build the Learning Plane and its connectors with the rest of the CLOUDSKIN stack (e.g., Control Plane, Data Plane, environment, etc.), involves a first phase of research and the evaluation of specific technologies fulfilling the required functionalities. E.g., which platform of virtualization and containerization should be used, which paradigm of distributed storage is to be used, and which background technology and know-how should be leveraged. I.e., the Data-Centric Computing group, at the Barcelona Supercomputing Center, has been working on different methods and algorithms for workload scheduling, also characterization, that needed evaluation towards CLOUDSKIN, to decide how and where to integrate them as a "decision making" module for the Learning Plane. The same with the Cloud stack from Nearby Computing, analyzing how the integration of the NearbyONE orchestration tool must be done given the proposed use cases. For this, different studies, early implementations and benchmarking tests have been performed, to ensure their proper integration into CLOUDSKIN and the Learning Plane.

For an early testing and prototyping, the initial implementation of the components for scheduling and scaling (Floki and CustomScale), plus workload prediction through Transformers, have been implemented and tested as a first prototype. Components to be integrated, such as the already existing methods ThetaScan and AI4DL, have been kept as a baseline for comparison for future experiments. For this deliverable we have skipped the evaluation of both technologies, as such has been previously done before CLOUDSKIN, and in this occasion we are just studying both methods to be leveraged or used for comparison. Figure 9 shows the initial architecture with a mapping of the technologies proposed to implement each part. While some are marked as "completed" (meaning that the study of their capabilities is complete, the integration is possible, or that a prototype implementation has been validated), some are work-in-progress, evaluating the corresponding methods. This is the case for the workload models and recommenders, and the data connectors for the Learning + Control Plane. Also, other technologies are being studied to cover the distribution and storage needs for models and Data Analytics functionalities.

The following sections present the evaluation and benchmarking results of the novel technologies here introduced (i.e. Floki, CustomScale and Transformers). Next steps will include the integration of the characterization models (AI-based modules) with the schedulers (recommenders), also on a first design and implementation of the data connectors upon NearbyONE.

## 5.2 Scalable Scheduling Policies

Here we describe the set-up and the different experiments on the prototypes and initial approaches for those methods and algorithms tested for building the fundamentals of the Learning Plane. This involves the benchmarking of the Floki Containerized Pipeline Communications mechanism and the Custom Scale-Out mechanism.

### 5.2.1 Infrastructure and Software Stack

In order to test the different methods and policies for scalability and scheduling, an on-premise environment has been set up, letting us to benchmark a neutral environment far from specific commercial technologies, provided by cloud vendors' specific environments. The provided set-up at the Barcelona Supercomputing Center premises, is composed as follows:

- Virtualized Cluster of Kubernetes
  - Benchmarking experiments are executed on a virtualized Kubernetes cluster composed of one master, representing the Kubernetes control-plane (not to be confused with the CLOUDSKIN Control Plane across the continuum), and 7 worker nodes on which tasks are deployed.

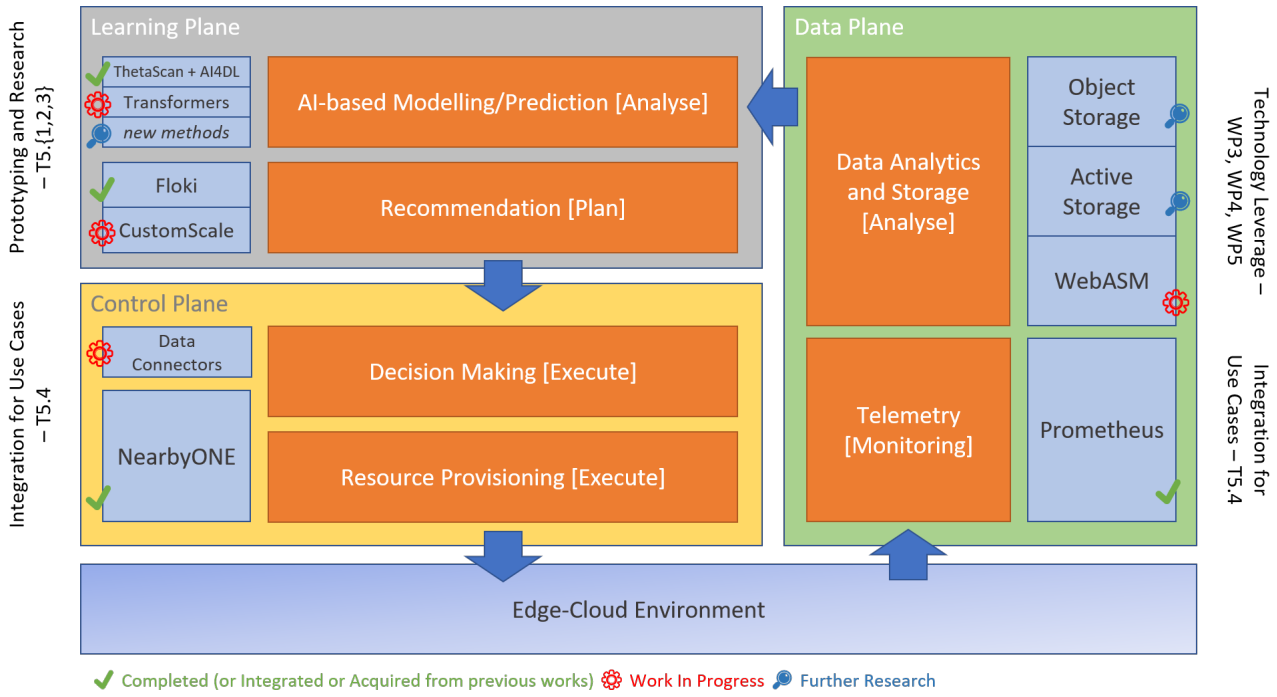



Figure 9: Map of technologies being explored, integrated and completed

  - The master runs in a Linux-based VM with 32GB of memory and 16 virtual cores, while the workers run in a Linux-based VM with 128GB of memory and 16 virtual cores.
  - The VMs are synchronized in the millisecond range.

- MinIO server for HPC object storage
  - A MinIO server, a widely used high-performance object storage, outside the Kubernetes cluster but inside the infrastructure, is used as shared storage.
  - The MinIO server runs bare-metal on a node featuring an Intel®Xeon E5-2620 CPU running at 2.00GHz, interfacing with two 1.6TB Intel®DC P3608 SSDs through NVMe.
  - In-memory key-value stores such as Redis and Memcached have been discarded, since they break one of the serverless advantages by requiring users to select instance types in terms of network, compute, and memory resources to satisfy their application requirements.

- Physical on-premise infrastructure
  - The Kubernetes cluster is mapped on 8 physical nodes residing in the same rack and featuring either an Intel®Xeon Silver 4114 CPU running at 2.20GHz or an Intel®Xeon E5-2630 v4 CPU running at 2.20GHz.
  - Physical nodes are connected through a 10Gbps Brocade VDX6740 network switch.

### 5.2.2 Floki Benchmarking and Performance

To evaluate Floki as a potential in-memory communication mechanism for the components in the Learning Plane, also for applications running on the environment, we analyze the impact of different pipe and socket buffer sizes on data communication latency, evaluating Floki in terms of performance and resource usage impact.

**Pipe and Socket Bugger Sizes Analysis:** In this analysis, we want to analyze the impact of different buffer sizes on fixed-size data communication over pipes and sockets and find the optimal buffer

sizes. The experiments are based on two types of functions: a producer function writing data in packets on a channel and a consumer function reading data in packets from a channel. The evaluation considers different buffer sizes, ranging from 1 system page, i.e., 4KB, to 128 system pages, i.e., 512KB, with the kernel imposed constraint of a power-of-two increment.

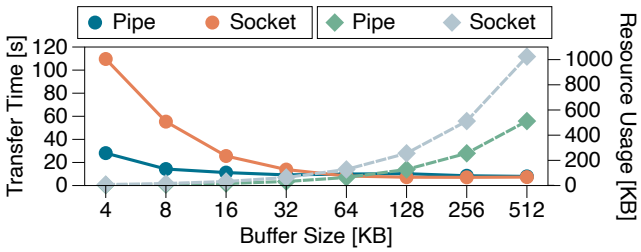



Figure 10: Average transfer time (circles) and resource usage (diamonds) for 16GB data with pipe and socket buffer sizes ranging from 1 to 128 pages (4KB to 512KB).

The lower range limit, i.e., 1 system page, represents the size for which the kernel guarantees pipe writes operations to be atomic. Within Floki, a *data-pipe* is of exclusive use of a single producer at a time; thus, the pipe buffer size can be increased without affecting the atomicity of write operations. Figure 10 shows the average transfer times of a 16GB data object with different pipe and socket buffer sizes, and the related resource usage. Note that the socket resource usage is always twice the corresponding buffer size: TCP allocates twice the requested buffer size and uses the extra space for administrative purposes and internal kernel structures. While the difference between the pipe and socket transfer times is significant for small buffer sizes, the transfer times are comparable for big buffer sizes. In particular, as highlighted in the figure, 16 system pages buffer size, i.e., 64KB, represents the optimal size, reducing and balancing the pipe and socket transfer times. Increasing the buffer size would provide comparable transfer times while using more resources. Therefore, the following experiments and evaluations consider a buffer size of 16 system pages.

**Performance Evaluation:** We conduct a series of experiments to evaluate the performance of Floki in terms of end-to-end times. Targeting data-intensive workloads, the evaluation considers data sizes ranging from 1MB to 16GB with a 2× increment. To measure the end-to-end times, we register the timestamp before each producer function starts to write data and the timestamp after each consumer function finishes reading data. Thus, given $N$ producers and $K$ consumers functions with $TS_{p_i}$ and $TS_{c_j}$ as their timestamps, we derive the end-to-end time $T_{E2E}$ as:

$$T_{E2E} = max(TS_{c_1}, .., TS_{c_K}) - min(TS_{p_1}, .., TS_{p_N}) \tag{6}$$

As Equation 6 shows, the end-to-end time $T_{E2E}$ accounts for possible not fully concurrent operations by considering the minimum of the producers timestamps $TS_{p_i}$ and the maximum of the consumers timestamps $TS_{c_j}$.

Figure 11 shows Floki end-to-end time speedups over the object storage solution baseline (horizontal constant solid line). Floki always significantly outperforms the object storage solution in all the analyzed patterns. Contrarily to a naïve solution relying on shared object storage, producers and consumers functions are deployed and run concurrently. The benefits of the volatile data share are more visible with small data sizes, i.e., from 1MB to 256MB, for which a higher performance increase is obtained. It is worth noting that, since data objects are read sequentially from the consumer functions, with multiple producers, functions using Floki gain smaller performance than those achieved with a single producer. Floki reduces the end-to-end time up to: 74.95× in the one-to-one pattern; 25.34×, 15.83×, and 24.83× in the fan-out pattern; 10.11×, 10.18×, and 7.49× in the fan-in pattern; 9.99× and 8.11× in the all-to-all pattern. Overall, considering the impact of Floki in terms of end-to-end time, the most significant time-savings are reached with a data size of 16GB, featuring the largest

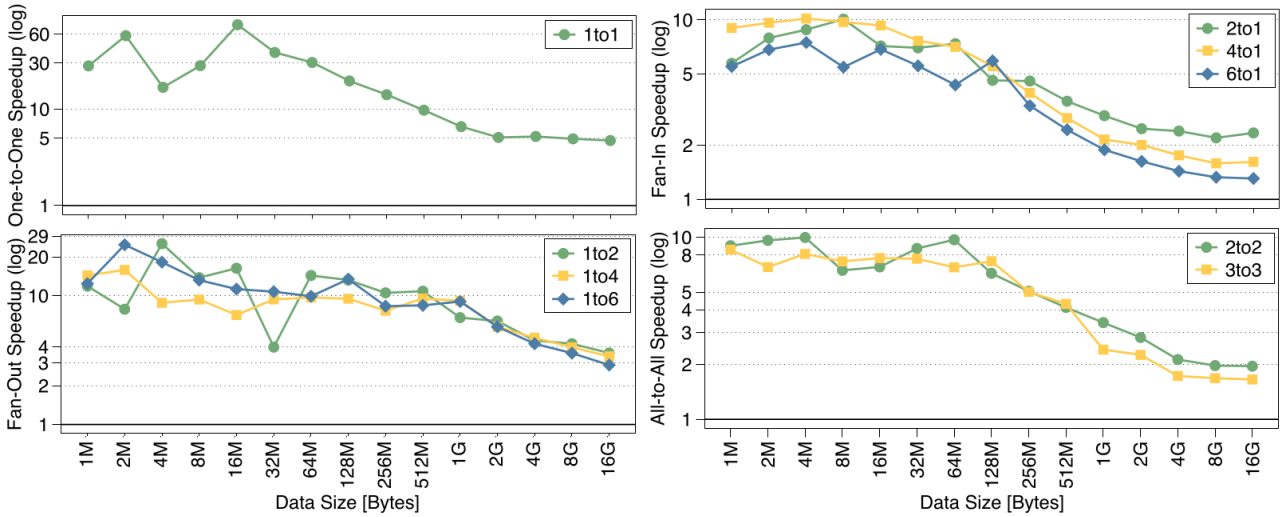



Figure 11: Floki's end-to-end speedups over the object storage solution in the analyzed distributed systems patterns.

data transfer latency. In particular, the higher time reductions are achieved in the 1to6 pattern, where communication latency are reduced from 753$s$ to 260$s$ on average. In other words, Floki allows saving 8.22 minutes on data sharing latency over the object storage baseline requiring 12.55 minutes.

**Resource Usage Evaluation:** We want to estimate and compare the resource usage of Floki to the object storage solution, representing the baseline, in the four considered patterns, i.e., one-to-one, fan-out, fan-in, and all-to-all. To assess the gap of resource requirements between varying data sizes, we choose two extreme cases just for comparison, i.e., 1MB and 16GB. In the following, $DS$ represents the data size, $T$ the total amount of functions composing the specific pattern, $P$ and $C$ the number of producers and consumers functions, and $PBD$ and $SBD$ the pipe and socket local buffer sizes (i.e., 64KB and 128KB), accordingly. The object storage resource usage estimation does not consider the necessary internal buffers to write and read the data object/file since their sizes are negligible compared with the analyzed data sizes. Being the object storage shared among the different functions, we derive the related resource usage $RU_{ObjStorage}$ as:

$$RU_{ObjStorage} = (P * DS)_D \tag{7}$$

The required disk space is proportional to the number of producers functions $P$. Therefore, from a resource usage perspective, there is no difference among the patterns composed of the same number of producers and a different number of consumers. For instance, the fan-in pattern with three producers and one consumer would require the same disk space as the all-to-all pattern with three producers and three consumers. Differently, Floki represents a significantly less expensive resource usage solution. Considering only the memory space needed to hold the local buffers to perform the pipe and socket operations, we derived Floki resources usage $RU_{Floki}$ as:

$$RU_{Floki} = (T * PBD + 2 * P * C * SBD)_M \tag{8}$$

The evaluation process of Floki resource usage for the presented analysis is as follows:

- By applying Equations 7 and 8, the resource-saving is evaluated by dividing the resource usage of the object store baseline for the Floki resource usage. For example, when sharing 1MB, Floki saves $\frac{1MB}{384KB} = 2.67\times$ of resources compared to the baseline.
- Floki always demands a significantly lower amount of resources compared to the object storage solution. More precisely, each function composing the workflow only requires 64KB of memory

for the pipe buffer and 128KB for each client/server socket buffer, allowing resource-saving to scale linearly with the data size increase. For example, considering the simple one-to-one pattern, the 16GB saving differs from the 1MB saving by a factor of $16,384\times$, equivalent to the difference between the two data sizes.

Overall, Floki achieves up to $50,738\times$ of resource-saving, translating into a memory allocation of roughly 2.6MB instead of an object storage allocation of 96GB.

Floki is at its first stage of maturity, with the aim of develop it around the requirements, issues and perks observed in the CloudSkin project and the corresponding use cases. We are currently working to fully integrate it in Kubernetes-based workflow frameworks to enable high-performance intermediate data exchange, and serve not only the CloudSkin stack but other potential OpenSource Software stacks for Cloud Continuum management that can benefit from this technology, or the combined technology of Floki and the CloudSkin stack. The presented and ongoing results are to be submitted as a scientific contribution to one of the primary conferences in the field, i.e., the ACM International Conference on Management of Data (SIGMOD) in July 2023.

### 5.2.3 Custom Scale-Out

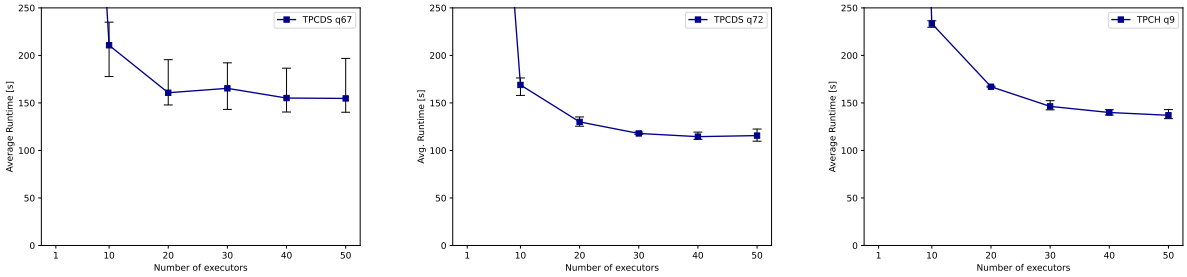
To evaluate the performance of the proposed per-stage scale-out mechanism (here using Apache Spark as platform) in terms of application run-time and cost, we first naïvely estimate the per-stage optimal number of executors. The per-stage optimal scale-out levels are estimated in a naïve manner based on the per-stage average run-times, considering both run-time and cost (assuming an executor cost of one unit per second). More specifically, the estimation process starts with one executor assigned to each stage, then iteratively increases the number of executors of the stage, which returns a higher run-time decrease. The estimation process continues until there is no stage for which increasing the number of executors returns a performance increase greater than the associated cost increase. Once the estimation process finishes, it returns the per-stage optimal scale-out levels. For example, the estimation process for TPD-DS q72 query (a benchmarking query executing a representative data-crossing analytics) returns the following set of a per-stage optimal number of executors:

$$\begin{aligned} scale\_out\_levels = [&1,1,10,1,10,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,10,1,10,1,1,\\ &1,1,10,1,10,1,1,1,1,1,1,1,1,1,1,1,20,10,10,20,10,10,1] \end{aligned} \tag{9}$$

Thanks to this simple estimation, we compare Vanilla Spark with static and dynamic allocation of executors with Spark featuring per-stage scale-out. In particular, we first analyze Vanilla Spark's performance with static allocation when scaling out the number of executors both at the application- and stage-level. Then, we evaluate Spark with per-stage scale-out enabled in terms of performance and cost impact, and compare it with Vanilla Spark with static and dynamic allocation of executors. Cold start latencies are not considered in the following evaluation.

**Scalability Analysis with Vanilla Spark:** During this scalability analysis, and in the following per-stage scale-out performance analysis, we considered two industry-standard benchmarks, i.e., TPC-DS [21] and TPC-H [13], for measuring the performance of decision support solutions including, but not limited to, Big Data systems, such as Apache Spark. We conduct a series of experiments to analyze the two benchmarks' performance when scaling the number of allocated executors from 1 to 50 with an increase of 10. Executors are configured identically and allocated statically when the application starts. The analysis considers three of the most time-consuming queries, i.e., TPC-DS q67, TPC-DS q72, and TPC-H q9, and reports average results computed with a scaling factor of 100 over 5 sequential runs for reliability.

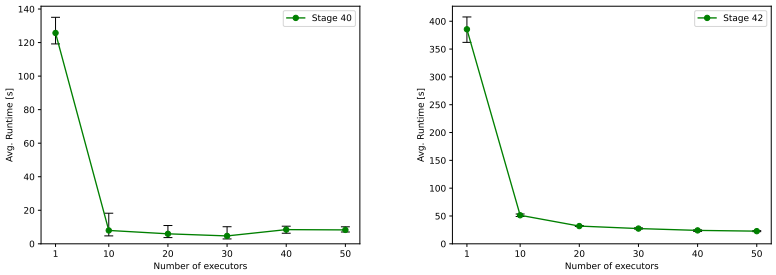
Figure 12 shows the considered queries' average run-time when scaling the number of executors. It can be noted that scaling out does not result in a linear run-time reduction. Different from what we expected, the considered queries' performance curves are quite similar. Assuming an executor cost of one unit per second of allocation and considering the ratio between performance-cost ratio, the optimal number of executors at application-level of the three considered queries are 10, 10, and 20,




(a) (b) (c)

Figure 12: (a) TPC-DS q67, (b) TPC-DS q72, and (c) TPC-H q9 query average run-times

respectively. We perform the same analysis at the stage-level to investigate if different stages show a different optimal number of executors. When analyzing the most time-consuming stages, similar trends in the performance curves are found. Figure 13 reports two TPC-DS q72 stages run-times. While the stage run-time is highly reduced when moving from 1 to 10 executors, scaling to a higher number of executors has a lower impact in terms of performance gain. Furthermore, by analyzing all stages, we found that all of them show an optimal number of executors equal to the one found in the application-level analysis, i.e., 10 executors.




(a) (b)

Figure 13: TPC-DS q72 (a) Stage40 and (b) Stage42 average run-times

Although these results do not confirm our expectations, i.e., different per-stage optimal number of executors, we perform further performance evaluations driven by the following motivations. First, limiting the number of executors for stages with little inherent parallelism or running on small input data can increase the utilization of resources. For example, TPD-DS q72 query first and second stages are characterized by one task. When the query is run with Vanilla Spark on a cluster of 10 executors, all 10 executors are allocated to each of the two stages even though only one executor is necessary, as depicted in Figure 5. Resource utilization can be highly improved by setting the number of executors to 1 for these two stages, leaving more resources free for other computations. Furthermore, benefits from JIT and data caching can be achieved if we constrain Spark to use the same executor for the two subsequent stages. Second, stages with stragglers do not benefit when scaling out resources beyond the number of stragglers. An example of a stage with stragglers is Stage36 of TPC-DS q67 query. As shown in Figure 14, it has overall 500 tasks, 10 of them with a high completion time as compared to the average time taken by other tasks belonging to the stage.

**Performance Evaluation with Per-Stage Scale-Out:** To evaluate the performance of the proposed per-stage scale-out in terms of application run-time and cost, we first naïvely estimate the per-stage optimal number of executors with the estimation process previously presented. Thanks to this simple

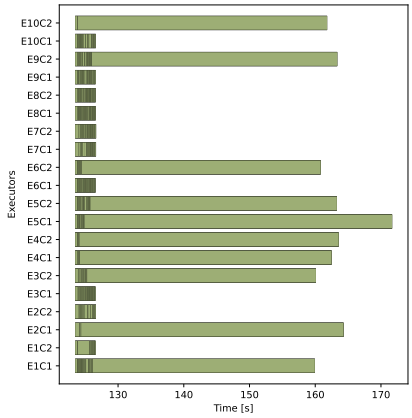



Figure 14: Stage36 of TPD-DS q72 query execution on 10 executors (2 cores each)

estimation, we compare Vanilla Spark with static and dynamic allocation of executors with Spark featuring per-stage scale-out. Figure 15 reports the performance and costs of the three analyzed queries, i.e., TPC-DS q67, TPC-DS q72, and TPC-H q9 queries. As can be noted, in all cases, the executions with per-stage scale-out (i.e., Custom Allocation) always is either faster or cheaper than Vanilla Spark executions. In the scalability analysis previously reported, we found that the optimal number of executors per-application level for the three considered queries is 10, 10, and 20, respectively. In all the cases, Spark with per-stage scale-out achieves higher performance at a similar cost when compared to the per-application optimal number of executors. More precisely, it achieves 1.37×, 1.29×, and 1.75× of performance gain for TPC-DS q67, TPC-DS q72, and TPC-H q9 queries, accordingly.

To increase the applicability of the work, we are currently working towards refining and adding the cold start latency in the analytical model computing the optimal scale-out level. Furthermore, we aim to add a module using acquired knowledge from historical data to speed-up the optimal scale-out level search. This will allow to start the search from a value which is potentially more near to the optimal scale-out level. Once this technology is finished, the Learning Plane is in charge of integrating this policy-making methods into the recommendation system, also complement through the workload characterization mechanism. Current decisions leverage the *a-posteriori* information (e.g. measured average of tasks), while the final objective is to leverage *a-priori* information, such as "expected task length" or "expected quality of service". Next steps consider incorporating the Workload Characterization methods (e.g. ThetaScan, AI4DL, and novel methods such as transformers for time-series in the next subsection) into the mechanisms to decide a resource provision and application placement. The presented and ongoing results are to be submitted as a scientific contribution to one of the primary conferences in the field, i.e., the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) in August 2023.

### 5.3 Characterization using Transformers

In order to evaluate the novel Transformer methods, by using different evaluation functions driving the learning process towards our utility goals, we performed a set of experiments comparing different approaches against a baseline benchmarking using the Autopilot framework for resource provisioning and auto-scaling from Google [22], currently in production at Google Cloud services.

Autopilot automatically performs vertical auto-scaling, computing on-line resource demand predictions for each running workload. Their method uses two forecasting strategies, and the user can choose which one to enforce. Either one or the other do not use cross-job information, and only use information from "lag" values, also are designed for long running workloads, something that

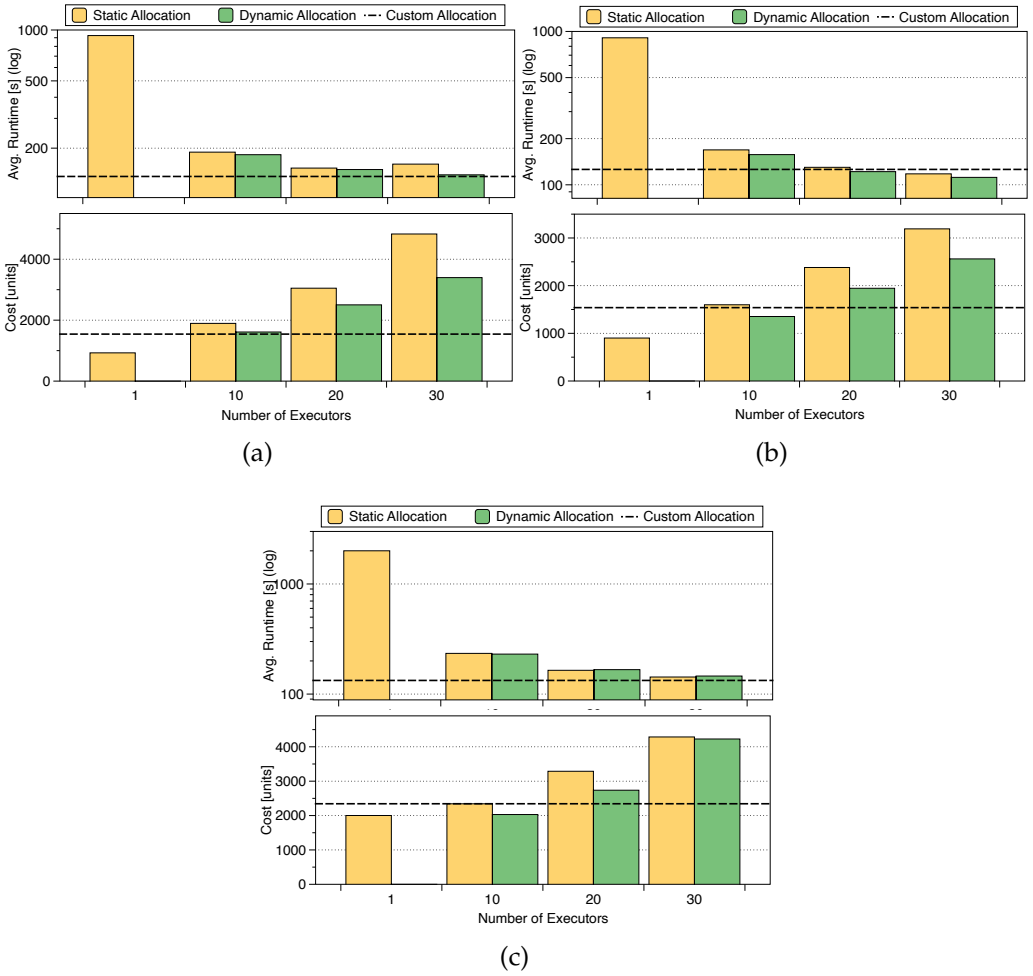



Figure 15: (a) TPC-DS q67, (b) TPC-DS q72, and (c) TPC-H q9 Vanilla Spark and Spark featuring per-stage scale-out run-times and costs

makes it poorly suitable for sudden changes. The first strategy follows a simple method selecting a statistical feature to be used directly as the future prediction value, specifically the maximum, average or percentile of the lag value. The selection is done depending on the workload tolerance to resource demand changes and tuned manually through practical experimentation during years, therefore specifically fitted for specific workloads. also the final prediction is increased by a 10-15% safe margin. The second strategy uses a set of machine learning models, picking a specific model from the set by past hits and misses. The system penalizes over and under-provisioning, and an exceeding change in usage limits.

In order to progress on novel methods to be used in the Learning Plane for provisioning prediction and recommendation, we implement the Transformers neural network architecture, creating a catalog of new evaluation metrics following our interests in deciding the ceiling for resource provisioning given an application:

- *over-prov*: Quantifies the total resources that were allocated over the real usage.
- *under-prov*: Quantifies the total resources that were consumed and were not properly allocated.
- *spikes-prec*: Provides annotations about detected spikes in the trace for allocated resources that do NOT have any correlation with the real-usage spikes found in the trace.
- *spikes-rec*: Provides annotations about the real spikes correctly detected before they happen.

- *spikes-rec-under*: Quantifies the number of spikes that are underneath the forecasting, without taking into account the distance between the prediction and the spike amplitude.

The first four proposed metrics give enough and significant information about the characteristics of the forecasted resource demand. We know if our algorithm tends to over-provision or under-provision and in what magnitude, and we are aware of the real performance in the detection of spikes of resource usage. Table 3 shows the the comparison among our transformers with different evaluation methods against the Autopilot methodology, using the mentioned open datasets from Google and Alibaba services.

| dataset | model | over-prov ↓ | under-prov ↓ | spikes-prec ↑ | spikes-rec ↑ | spikes-rec-under ↑ |
|---|---|---|---|---|---|---|
| Alibaba | SCINet | 0.8267 | 0.0365 | 0.1887 | 0.2147 | 0.3658 |
| | Informer | 0.6282 | 0.0346 | **0.3470** | **0.2677** | 0.2867 |
| | WRecomPeak | 8.6559 | 0.0192 | 0.2000 | 0.0861 | 0.8610 |
| | WRecomW | **0.2507** | 0.8669 | 0.0000 | 0.0365 | 0.0630 |
| | WRecomPerc | 1.9116 | 0.1473 | 0.0191 | 0.0861 | 0.3499 |
| | MLRecom | 13.5658 | **0.0088** | 0.1788 | 0.0561 | **0.9426** |
| Google | SCINet | 1.2805 | 0.0354 | 0.1197 | **0.3199** | 0.6464 |
| | Informer | 1.3909 | 0.0299 | **0.2192** | 0.2705 | 0.6924 |
| | WRecomPeak | 4.2739 | 0.0089 | 0.1483 | 0.0799 | 0.9227 |
| | WRecomW | **0.1726** | 4.6287 | 0.0000 | 0.0398 | 0.0807 |
| | WRecomPerc | 6.5960 | 0.0308 | 0.0565 | 0.0656 | 0.8405 |
| | MLRecom | 7.3026 | **0.0027** | 0.1086 | 0.0332 | **0.9771** |

Table 3: Comparison between the proposed approach (SCINet and Informer) and the Autopilot framework with its different strategies (Peak, Weighted, Percentile and ML-based Recommendations). Highlighted values correspond to the best value for each metric

We can observe how the Autopilot forecasting results are poor in regards to our approach, through the low precision and recall obtained metrics. The reason relies in the fact that the Autopilot's peak strategy is handcrafted towards resorting to over-provisioning in order to contain all the potential spikes underneath the allocation of resources. The strategy's objective is not to predict spikes, but to guide a long-term executions ignoring urges in demand. Figures 16 and 17 clearly show the predictions for the machine learning and peak window recommenders. The behavior is absolutely reactive, as the spikes are not predicted a-priori, but once a spike occurs the predictions adapt to accommodate possible future spikes with similar amplitudes. This explains the huge over-provisioning and high values for the *spikes-rec-under* metric, as the strategy is totally conservative. On the other hand, the weighted and percentile window recommenders have a different behaviour, not very suitable for ours or Google objectives. They predict the usage average, skipping the true purpose of understanding the resources demand ceiling, and not usable for forecasting spikes beforehand, neither to over-provision towards covering demand excesses. Figures 18 and 19 show the detail for both recommenders, using the same previous time-series examples.

Figures 20 and 21 show some of the best results when applying our novel methodology, using the Alibaba dataset. As the results are not perfect, as there are undetected spikes and areas with highly undesired over-provisioning, there results are promising towards research and improvement. Next steps focus on a more extensive and longer hyper-parameter search to be performed on all the models. Current works had a limited extent as a first approach and proof-of-concept, also with the creation of the baseline approach (Google's Autopilot implementations are not public, and development from scratch from the original scientific papers has been required). This first exploration of hyper-parameters has targeted those models taking a limited (and feasible) amount of time, in

order to explore the potential of the forecasting method. The performance of the obtained models can be improved by spending more time in training and varying interval ranges. Nevertheless, with the preliminar exploration of this early implementation and modelling, it is remarkable the amount of spikes that could be forecasted in amplitude and time. Is has to be researched which amount of spikes are result of "noise" and application misbehavior, becoming random events that cannot be predicted. However, our approach is capable of doing this task with good precision and recall.

We conclude this set of experiments by indicating the difficulty of comparing results from the different methods, as those have different goals, also in certain situations could be both leveraged to adjust to trends while checking for spikes. However, there is potential on the Transformers approach as a burstiness detector.

### 5.4 Publication of Prototyping Components and Methods

The current status of the presented prototype components is of "publication pending", as either the latest evaluations and benchmarking for Floki and Custom Scale-Out, also the Transformer-based predictor, have been submitted for publication to two top conferences and a top scientific journal. The publication plan includes opening the code as Open Source Software in a GIT repository for each (in current debate to use a private service such as GitHub or use a partner-provided service such as GitLab from BSC services), opening the data generated as Open Data (in current debate to use an public repository such as Open-Data Europe or host in a partner-provided service such as BSC public data repositories). Also, the specific results for Floki and CustomScale will be push-requested towards integration into the Kubernetes and Spark official repositories, as a community contribution to the open source platforms.

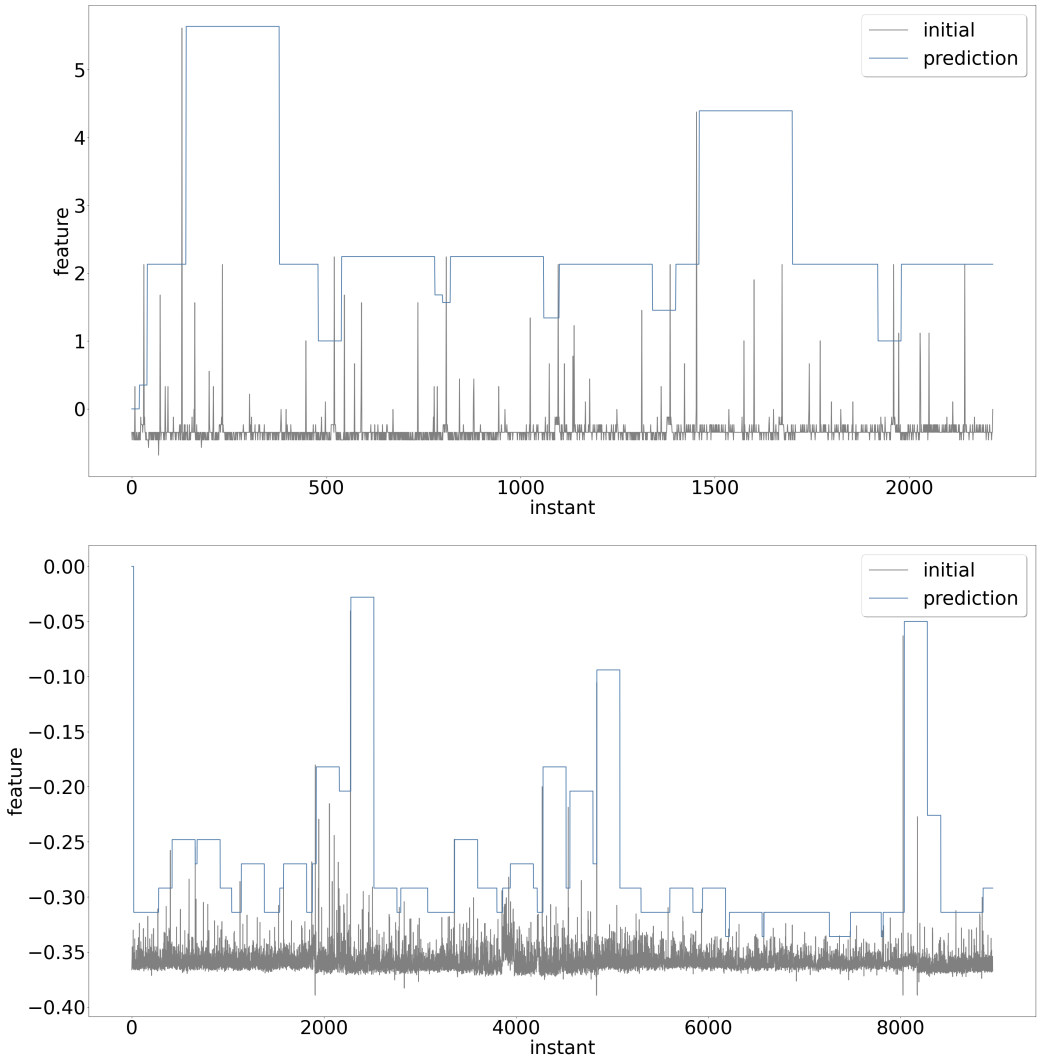



Figure 16: Autopilot predictions with the peak window recommender on two time series from the Alibaba and Google dataset

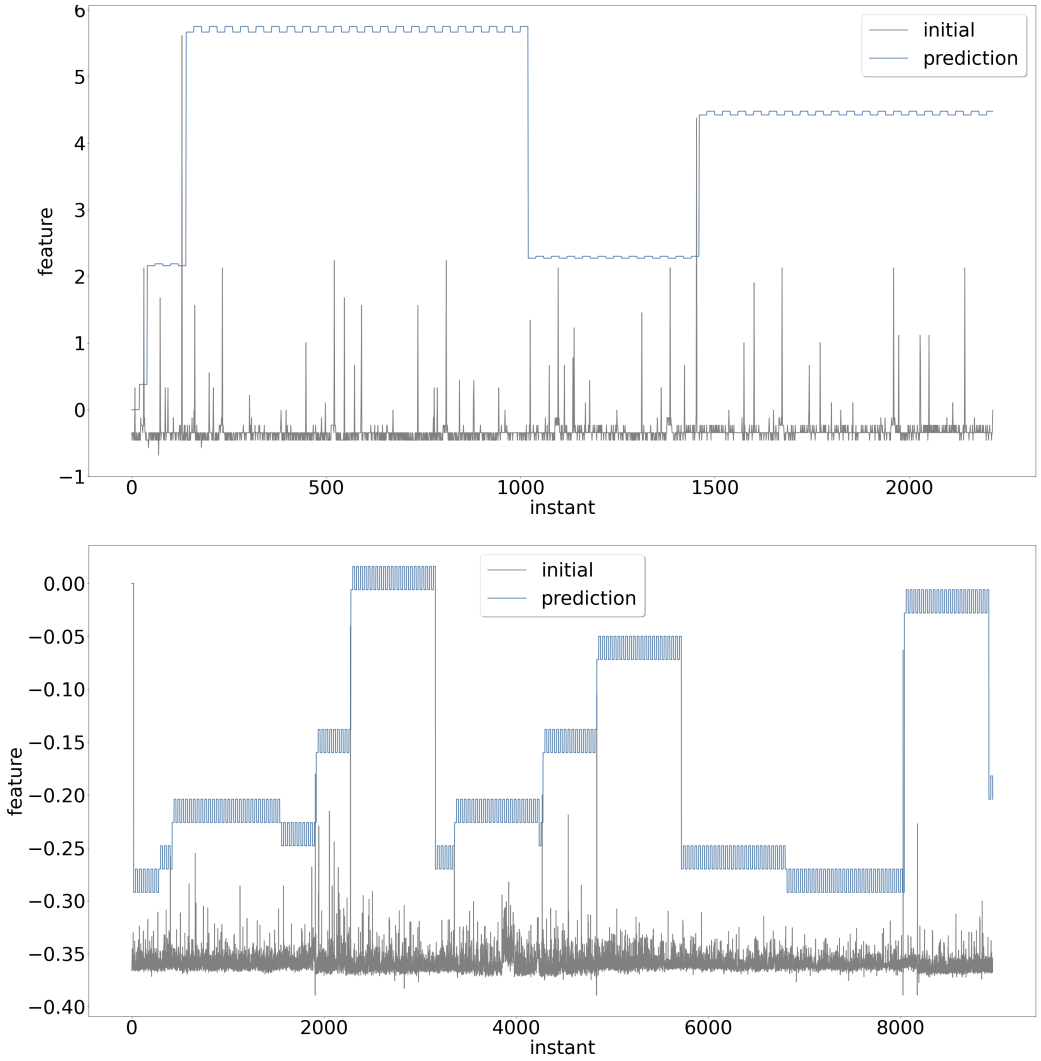



Figure 17: Autopilot predictions with the machine learning recommender on two time-series from the Alibaba dataset

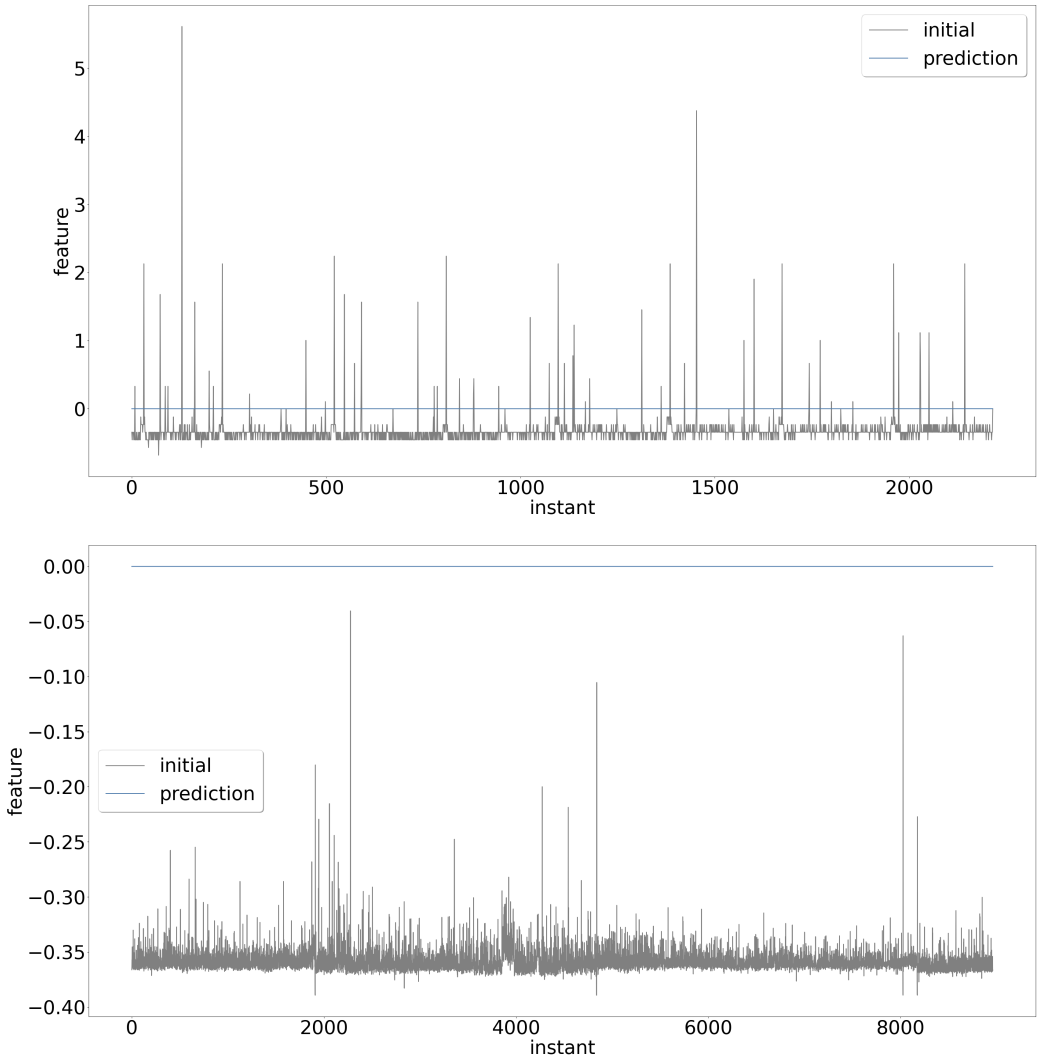



Figure 18: Autopilot predictions with the percentile window recommender on two time-series from the Alibaba and Google dataset

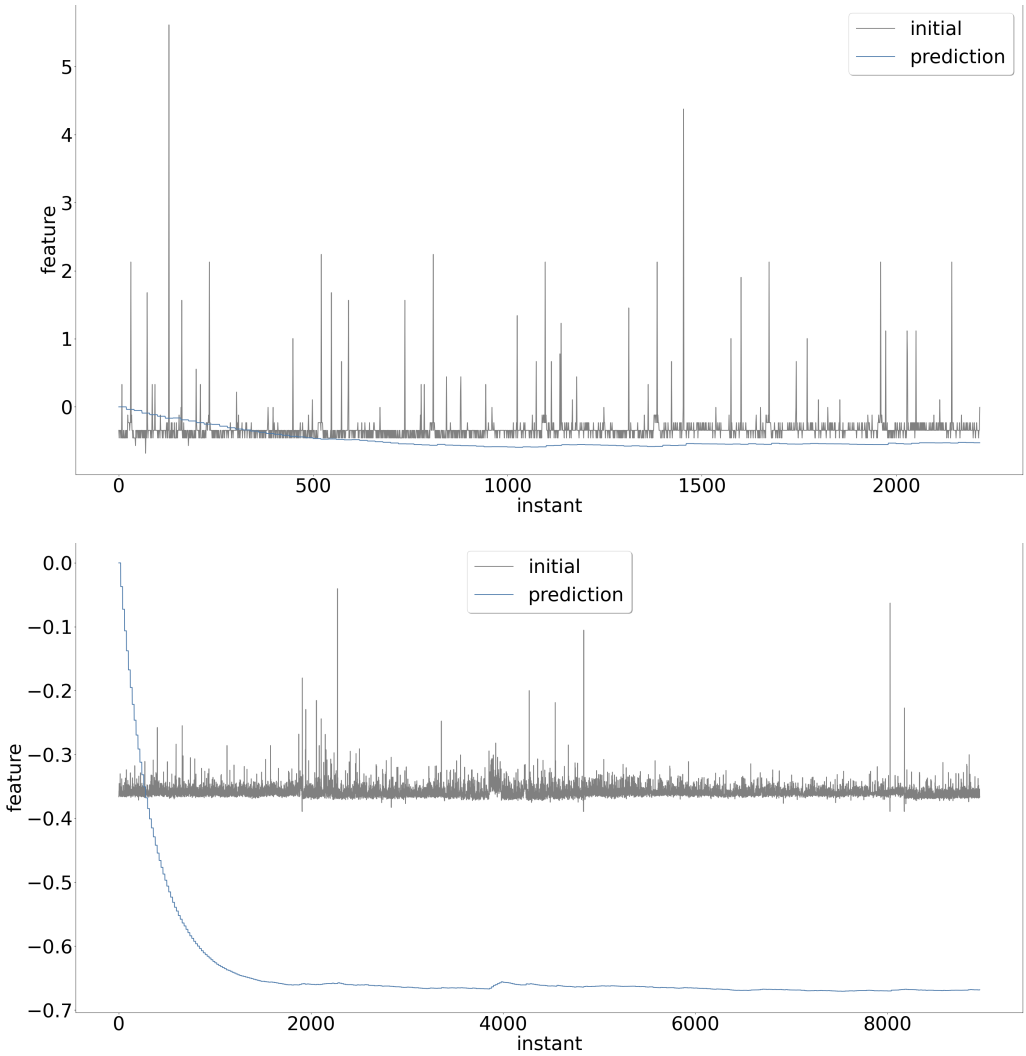



Figure 19: Autopilot predictions with the weighed window recommender on two time-series from the Alibaba and Google dataset

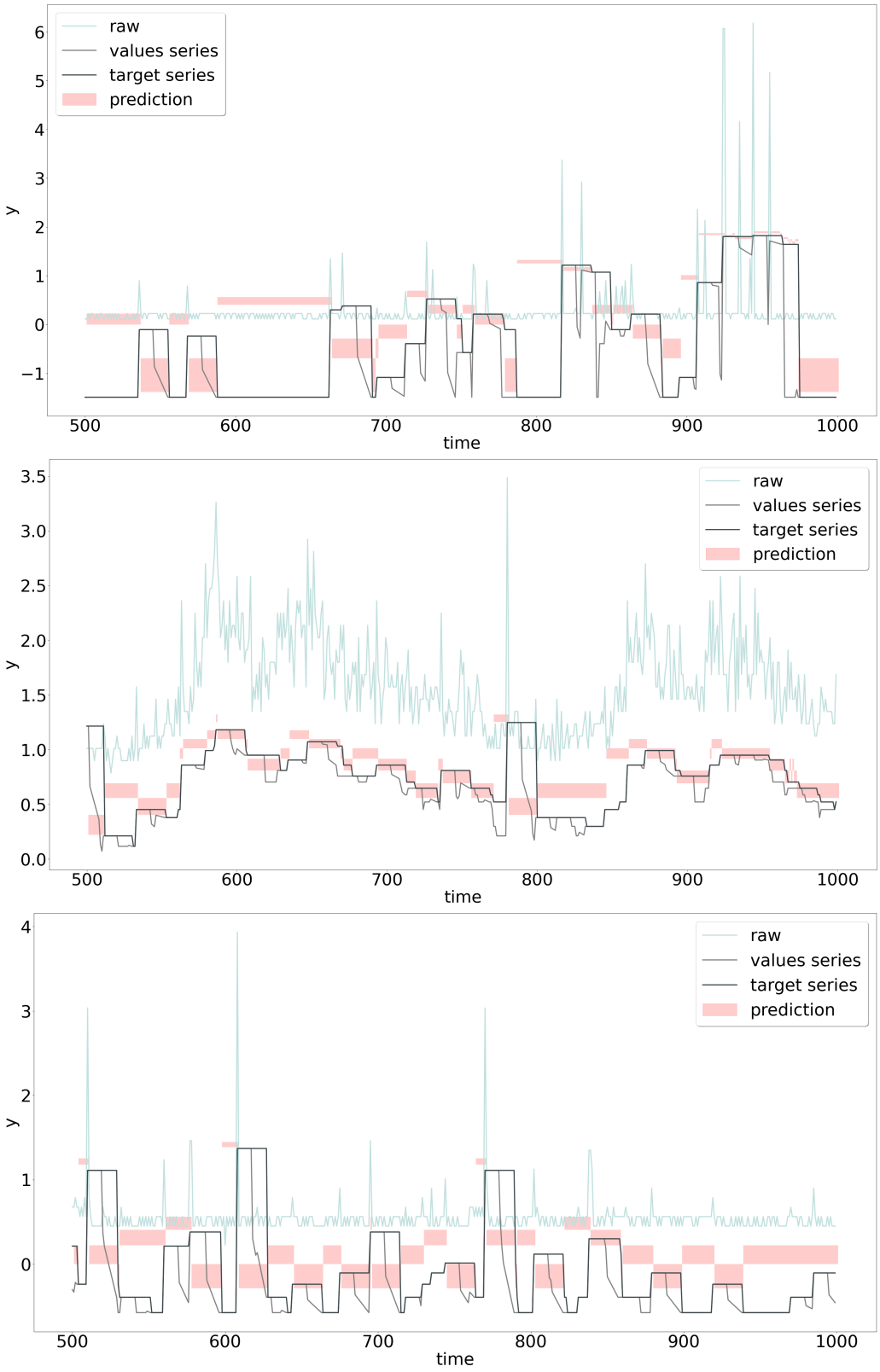



Figure 20: Series pre-processing experiments on the Alibaba dataset. Examples of predictions for three time-series. The *target* series are pre-processed with the *past-max* filter. The results are extracted from the TemporalInception model

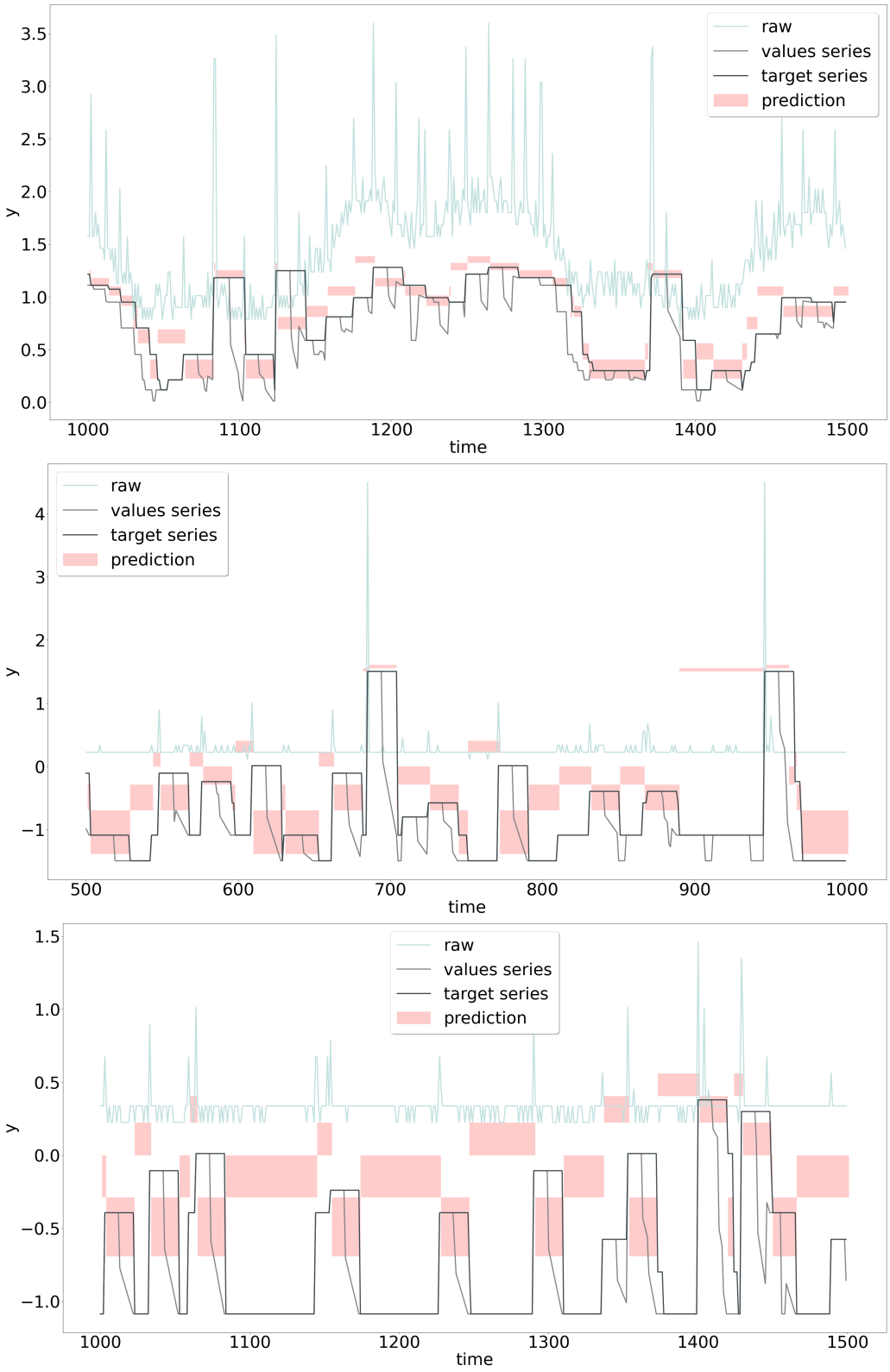



Figure 21: Series preprocessing experiments on the Alibaba dataset. Examples of predictions for three time series. The *target* series are pre-processed with the *past-max* filter. The results are extracted from the TemporalInception model

## 6 Conclusions

This deliverable presents the initial design of the CLOUDSKIN Learning Plane, considering the first studies and tests on methods for workload characterization and decision making on workload scheduling. Some of the contents of this deliverable expand the ones in *Deliverable D2.1* referring to the architecture and integration of CLOUDSKIN components, here explaining deeper the components in the Learning Plane. The methods, algorithms and architectures described in this deliverable are intended to explore and innovate with potential novel and state-of-art technologies (AI and ML) to be leveraged on "smart" management of Cloud-Edge systems in the CLOUDSKIN project. More specifically, it is intended to research on methods that will be integrated with the proposed technologies from the technical work packages of CLOUDSKIN in regards to Execution Environments (Containerization and WASM) and Storage (Distributed Object Storaging). The proposed architecture focuses on an initial centralized management approach, but contemplating the off-loading of management across the Cloud-Edge Continuum.

In this deliverable, we provided the current efforts and preliminary results on:

- The components forming the Learning Plane in the CLOUDSKIN architecture, along with their characteristics and details for the integration in the management engine. Here we detailed the initially designed pipelines for the Learning, Control and Data planes in a placement and provisioning scenario.
- An initial set of methods for workload characterization and resource planning, setting up a baseline for research and innovation. Such methods are to be updated and iterated along the project, while complemented by new methods to be incorporated.
- An initial evaluation of those methods, showing the progress in workload characterization and scheduling. Current results are a study to evaluate their suitability for the current cases, and might be updated with the preliminary integration of new developed scenarios from the CLOUDSKIN partners.

Definitive results of the current evaluations and experiments of the presented methods are pending validation and publication, and will be added to the next update for the Learning Plane reporting.

## References

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

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

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

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

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

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

[7] J. Sampé, M. Sánchez-Artigas, G. Vernik, I. Yehekzel, and P. García-López, "Outsourcing data processing jobs with lithops," IEEE Transactions on Cloud Computing, vol. 11, no. 1, pp. 1026–1037, 2023.

[8] "WebAssembly." https://webassembly.org.

[9] A. M. Nestorov, J. L. Berral, C. Misale, C. Wang, D. Carrera, and A. Youssef, "Floki: A proactive data forwarding system for direct inter-function communication for serverless workflows," in Proceedings of the Eighth International Workshop on Container Technologies and Container Clouds, WoC '22, (New York, NY, USA), p. 13–18, Association for Computing Machinery, 2022.

[10] AWS Simple Cloud Storage (S3). https://aws.amazon.com/s3.

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

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

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

[14] C. C. Holt, "Forecasting seasonals and trends by exponentially weighted moving averages," International Journal of Forecasting, vol. 20, no. 1, pp. 5–10, 2004.

[15] G. W. Taylor, G. E. Hinton, S. T. Roweis, P. B. Schölkopf, and T. H. J. C. Platt, "Modeling human motion using binary latent variables," Advances in Neural Information Processing Systems, vol. 19, pp. 1345–1352, 2007.

[16] D. B. Prats, J. L. Berral, and D. Carrera, "Automatic generation of workload profiles using unsupervised learning pipelines," IEEE Transactions on Network and Service Management, vol. 15, pp. 142–155, March 2018.

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

[18] M. Liu, A. Zeng, Z. Xu, Q. Lai, and Q. Xu, "Time series is a special sequence: Forecasting with sample convolution and interaction," 2021.

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

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

[21] TPC-DS Benchmark. https://www.tpc.org/tpcds/.

[22] K. Rzadca, P. Findeisen, J. Świderski, P. Zych, P. Broniek, J. Kusmierek, P. K. Nowak, B. Strack, P. Witusowski, S. Hand, and J. Wilkes, "Autopilot: Workload autoscaling at google scale," in Proceedings of the Fifteenth European Conference on Computer Systems, 2020.