TRUSTWORTHY, COGNITIVE AND AI-DRIVEN COLLABORATIVE ASSOCIATIONS OF IOT DEVICES AND EDGE RESOURCES FOR DATA PROCESSING

Grant Agreement no. 101136024

Deliverable D3.2
Software-defined Edge Interconnect and Service Assurance Mechanisms

Programme:	HORIZON-CL4-2023-DATA-01-04
Project number:	101136024
Project acronym:	EMPYREAN
Start/End date:	01/02/2024 – 31/01/2027

Deliverable type:	Report
Related WP:	WP3
Responsible Editor:	NVIDIA
Due date:	30/04/2025
Actual submission date:	30/04/2025

Dissemination level:	Public
Revision:	FINAL

This project has received funding from the European Union's Horizon Europe research and innovation programme under grant agreement No 101136024

Revision History

Date	Editor	Status	Version	Changes
08.01.25	NVIDIA	Draft	0.1	Deliverable ToC
12.03.25	NVIDIA	Draft	0.2	Add initial contributions by NVIDIA, ICCS in sections 4, 6
26.03.25	NVIDIA	Draft	0.3	Add contributions by NVIDIA, NUBIS, RYAX, ICCS in sections 3,4,5,6,7
16.04.25	NVIDIA, ICCS	Draft	0.4	Complete deliverable for internal review
25.04.25	NVIDIA	Draft	0.5	Address review comments
29.04.25	NVIDIA	Final		

Author List

Organization	Author
ICCS	Aristotelis Kretsis, Panagiotis Kokkinos, Fotis Kouzinos, Emmanouel Varvarigos
NVIDIA	Dimitris Syrivelis
RYAX	Pedro Velho, Yuqiang Ma, Michael Mercier, Yiannis Georgiou
NUBIS	Anastassios Nanos, Charalampos Mainas, Georgios Ntoutsos, Ilias Lagomatis, Konstantinos Papazafeiropoulos, Apostolos Giannousas

Internal Reviewers

Javier Martin (IDEKO)

Roberto Gonzalez (NEC)

Abstract: Deliverable D3.2 presents the key outcomes of the activities that took place in the context of Task 3.3 “Software-Defined Edge Interconnect for Distributed Computations and Hardware Acceleration” and Task 3.4 “Autoscaling, Service Assurance and Computing Management” during the first implementation period (M04-M15). These tasks focus on the design and development of critical components within the EMPYREAN platform, including: (i) software-defined interconnection to organize edge devices into logical clusters, offering a unified memory layer; (ii) interoperable hardware acceleration functionality across heterogeneous IoT and edge nodes; (iii) AI-enabled workload autoscaling mechanisms; and (iv) service assurance and low-level computing management mechanisms.

Keywords: Software-Defined Interconnect, Hardware Acceleration, Service Assurance, EMPYREAN Associations, Vertical Pod Autoscaling, Autos-Scaler

Table of Contents

List of Figures

List of Tables

Abbreviations

AI	Artificial Intelligence
AMBA	Advanced Microcontroller Bus Architecture
API	Application Programming Interface
ASGI	Asynchronous Server Gateway Interface
CRD	Custom Resource Definition
CRUD	Create, Read, Update, Delete
D	Deliverable
FIFO	First In, First Out
FPGA	Field-Programmable Gate Array
GenOps	Generic Operations
GI	GPU Instance
GPU	Graphics Processing Unit
HMAC	Hash-based Message Authentication Code
I/O	Input / Output
K3s	Lightweight Kubernetes
K8s	Kubernetes
KPI	Key Performance Indicator
MIG	Multi-Instance GPU
ML	Machine Learning
MPI	Message Passing Interface
MPS	Multi-Process Service
NBI	Northbound Interface
OOM	Out-Of-Memory
QoS	Quality of Service
QP	Queue Pair
RDMA	Remote Direct Memory Access
REST	Representational State Transfer
RPC	Remote Procedure Call
RRR	Remote Ring-buffer Runtime
RTT	Round Trip Time
SBI	Southbound Interface
SDI	Software-Defined Interconnect
SLO	Service Level Objective
SM	Streaming Multiprocessor
SQ	Submit Queue
SQD	Submit Queue Data
TCP/IP	Transmission Control Protocol/Internet Protocol
TLS	Transport Layer Security
TPU	Tensor Processing Unit
VM	Virtual Machine
VPA	Vertical Pod Autoscaler

1 Executive Summary

This technical report presents a comprehensive overview of the software-defined interconnect mechanisms and their seamless integration with hardware acceleration abstractions within the EMPYREAN framework. It explores the design and implementation of these components, their interaction with the broader EMPYREAN architecture. Emphasis is placed on the design principles, functionalities, and exposed public APIs of each system component, offering a detailed technical perspective aligned with the objectives of Task 3.3, "*Software-Defined Edge Interconnect for Distributed Computations and Hardware Acceleration.*"

In addition, the deliverable introduces the initial outcomes of Task 3.4, *"Autoscaling, Service Assurance and Computing Management."* This includes the design and development of an AI-enabled vertical autoscaling mechanism for Kubernetes, which intelligently adjusts resource requests and limits based on workload telemetry. The task further delivers service assurance capabilities for the EMPYREAN platform, including the development of the Analytics Engine and AI-enhanced algorithms. These service assurance mechanisms enable the orchestration services to perform self-adaptive actions such as workload migration and resource reallocation in response to performance degradation or operational issues.

This deliverable builds upon the EMPYREAN reference architecture defined in deliverable D2.3 (M12), towards the provision of the initial release of the platform's software-defined interconnections, hardware acceleration abstractions, service assurance components, and AI-driven workload auto-scaling framework.

The final release of the components and mechanisms developed under Tasks 3.3 and 3.4 will be presented in deliverable D3.3 *"Final report on security, trust, seamless data and computing management"* (M26).

2 Introduction

2.1 Purpose of this document

This document is a comprehensive technical report detailing the enablements and methodologies surrounding software-defined interconnects. It explores the integration strategies with vAccel[1], a cutting-edge acceleration framework, while also delving into the intricate interactions within the broader EMPYREAN stack. These interactions are further contextualized through an examination of relevant service assurance mechanisms, ensuring a robust and reliable operational framework.

The report meticulously outlines each system component, providing an in-depth analysis of their design principles, functionalities, and exposed APIs. These APIs are presented not only as technical interfaces but also as pivotal tools that facilitate seamless communication between components. The document aims to offer readers a clear understanding of how these components are architected to interact harmoniously, emphasizing the rationale behind various design choices. Furthermore, it elaborates on configurable parameters, shedding light on the decisions made during the initial stages of development efforts.

2.2 Document structure

The present deliverable is split into 5 major chapters:

- EMPYREAN Architecture Mapping
- Software-defined Interconnect
- Hardware Acceleration Abstraction
- Service Assurance
- Intelligent Autoscaling and Adaptive Computing management.

2.3 Audience

This document is publicly available and should be of use to anyone interested in the description of the data interconnection, hardware acceleration, intelligent autoscaling, and service assurance aspects of EMPYREAN. Moreover, this document can be also be useful to the general public for obtaining a better understanding of the framework and scope of the EMPYREAN project.

[1] https://vaccel.org

3 EMPYREAN Architecture Mapping

The EMPYREAN architecture was first introduced in deliverable D2.2 "Initial Release of EMPYREAN Architecture" (M07), and later refined in its final version in D2.3 " Final EMPYREAN architecture, use cases analysis and KPIs" (M12). This refinement incorporated key insights gained from the initial implementation phase. D2.3 provides a comprehensive overview of the architecture, detailing the EMPYREAN components, their interfaces, and the supported operational flows.

In this section, we present a concise description of the architecture (Figure 1) to support the discussion of the initial developments in WP3, particularly focusing on dynamic provisioning of high-performance software-defined edge interconnect, seamless provision of hardware acceleration abstractions, intelligent and efficient workload autoscaling mechanisms, and distributed and data-driven service assurance mechanisms within the EMPYREAN.

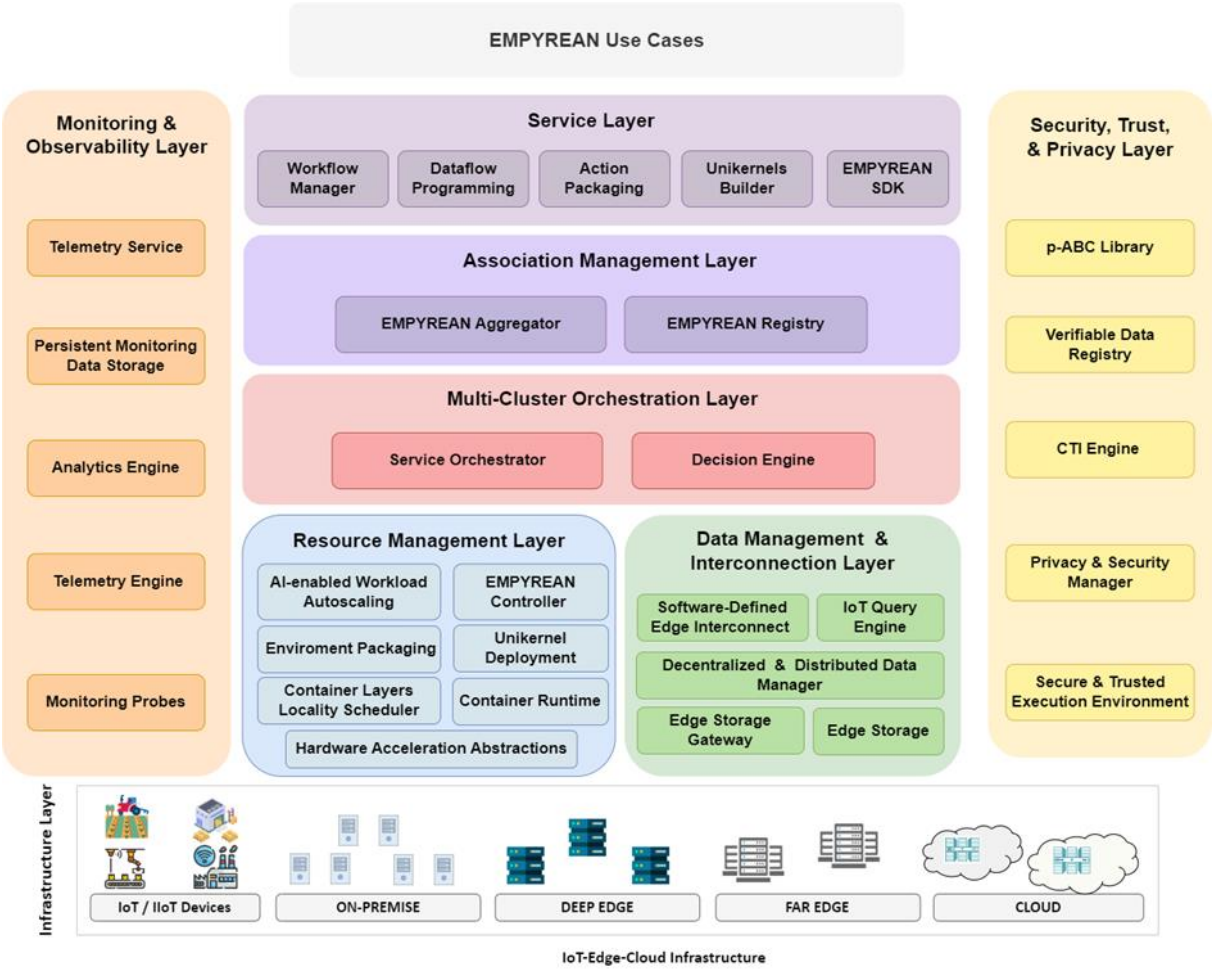

Figure 1: EMPYREAN high-level architecture

The ***Service*** layer facilitates the development of Association-native applications, providing robust support for application-level adaptations, interoperability, elasticity, and scalability across the IoT-edge-cloud continuum. Deliverable D4.1 (M15) provides a detailed description of the design and development of this layer's components.

The ***Association Management Layer*** dynamically manages Associations within the IoT-edge-cloud continuum. Forming resource federations, it enables seamless collaboration, resource sharing, and data distribution across various segments within the continuum. Together with the Multi-Cluster Orchestration Layer, it is central to EMPYREAN's distributed and autonomous management, establishing a resilient Association-based continuum.

The ***Multi-Cluster Orchestration Layer*** handles service orchestration and resource management across EMPYREAN's disaggregated infrastructure. Using autonomous, distributed decision-making mechanisms, it orchestrates dynamic, hyper-distributed applications while enabling self-driven adaptations. Multiple instances of this layer's components provide decentralized operation, optimize resource utilization, and ensure scalability, resiliency, energy efficiency, and high service quality. Deliverable D4.2 (M15) provides a detailed description for designing and developing the Association Management and Multi-Cluster Orchestration layers' components.

The ***Resource Management Layer*** unifies the management of IoT, edge, and cloud platforms under the EMPYREAN platform. It integrates software mechanisms for both platform-level scheduling (e.g., EMPYREAN Controller, AI-enabled Workload Autoscaling) and low-level mechanisms (e.g., Unikernel Deployment). This layer operates within Kubernetes (K8s) or Lightweight Kubernetes (K3s) clusters and offers modularity, simplifying the integration of new hardware and software. The deliverable presents the initial developments for two of its core components.

The *Hardware Acceleration Abstractions* (Section 5) component enables offloading compute-intensive tasks to hardware accelerators on neighbouring nodes. This offloading is performed while ensuring data security and integrity, thereby enhancing performance for resource-heavy workloads without compromising on safety. Moreover, the *AI-enabled Workload Autoscaling* (Section 7) component enhances the Kubernetes orchestrator by incorporating AI/ML techniques for intelligent workload autoscaling. By analysing historical data, this component ensures optimized resource allocation, dynamic application-level adaptations, and efficient utilization of resources, providing a more responsive and adaptive environment for workloads.

The ***Data Management and Interconnection Layer*** ensures dynamic communication and secure data storage between IoT devices and computing resources. Operating at both cluster and Association levels, it provides flexible and scalable data management and seamless integration of IoT, edge, and cloud resources. It also supports distributed operation, facilitating efficient operation in complex, distributed environments.

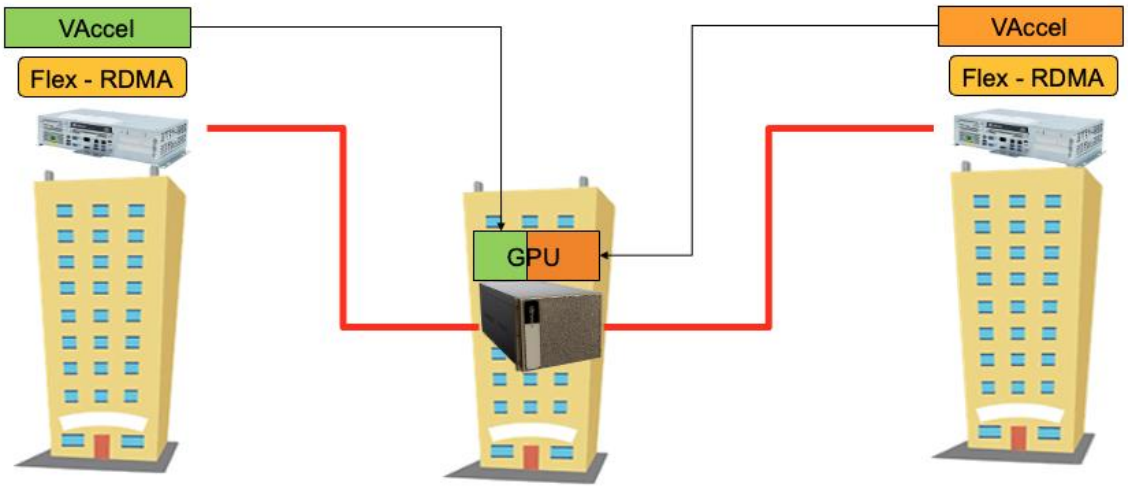

Figure 2: Example deployment scenario within EMPYREAN for software-defined interconnect and hardware acceleration abstractions

The *Software-Defined Edge Interconnect* (Section 4) serves as a low-level communication interface and key enabler within this layer. It delivers high-performance data transport service integrating remote I/O operations into large computational pipelines, such as AI training workflows. By leveraging Remote Direct Memory Access (RDMA), it optimally overlaps computation with network I/O, significantly enhancing the performance of data-intensive tasks across distributed environments and thereby supporting real-time processing and analytics. Figure 2 illustrates how the software-defined interconnect, together with vAccel framework, enables the dynamic deployment of composable infrastructures at the edge. This is achieved by enabling seamless resource sharing from a common pool of hardware acceleration resources across the edge deployment, promoting agility and scalability.

The architecture is complemented by the ***Security, Trust, and Privacy*** and the ***Monitoring and Observability*** layers, which are across the other layers, providing critical functionalities for the overall platform. The former ensures secure access, privacy, and trusted execution across the EMPYREAN platform. Operating at both the cluster and Association levels, it delivers distributed trust services, enables secure and trusted execution environments, and provides controlled data access to guarantee data confidentiality and continuous validation of trust among entities.

The Monitoring and Observability layer integrates real-time monitoring, observability, and service assurance components to provide comprehensive visibility and control over the EMPYREAN platform. The telemetry components are described in deliverable D4.2 (M15). The *Analytics Engine* (Section 6) provides service assurance by using AI-driven analytics on top of monitoring and observability data. This approach ensures that applications perform as intended by dynamically adjusting deployments based on changing conditions and requirements.

4 Software-Defined Interconnect

4.1 Overview

The EMPYREAN Software-Defined Interconnect is a low-level, high-performance communication interface that operates on top of Remote Direct Memory Access (RDMA) verbs. It offers low-latency round-trip times (RTT) and efficient message aggregation through a very simple circular-buffer control plane. The software interface resembles the design of the Linux kernel's ***io_uring***, adopting the same ***Proactor*** pattern for asynchronous operations. Additionally, we have designed a specialized accelerator interface version that disaggregates FIFOs offering an Advanced Microcontroller Bus Architecture (AMBA) AXI-S hardware specification interface, enabling seamless hardware integration.

Value proposition of the EMPYREAN Software-Defined Interconnect (SDI) approach:

1. ***High-Performance Remote I/O***: Significantly improves remote I/O network data path performance over RDMA, enabling scalable performance for data-intensive applications.
2. ***Standardized Interface***: Offers a standard io_uring-like interface at the software level, following the same Proactor-based asynchronous model used in the local Linux kernel *io_uring* interface.
3. ***Fully User-Space Solution***: Operates entirely in user space without requiring kernel-side servers yet achieving similar performance to a kernel-side implementation, which is amenable to use with containers and hypervisors.

The in-band data path functionality of the circular buffer is straight forward and lock-free. Each endpoint manipulates the locally exposed buffer construct by:

- ***Writing data***: updating the head pointer.
- ***Reading data***: updating the tail pointer.

The out-of-band control brings up the described buffer connections across the deployment and manipulates their depth and head/tail update strategies. The mechanism also controls Quality of Service (QoS) across the service deployment. This approach is particularly important especially for latency-sensitive operations, such as vAccel interface disaggregation (Section 5), where jitter can significantly impact performance.

Applications using the disaggregated circular buffer should follow the Proactor pattern (Figure 3), a well-known software design pattern for handling asynchronous events efficiently. This pattern is particularly useful in applications that require concurrent execution of operations without the overhead of multiple threads and are ideal for containerized environments. The pattern simplifies asynchronous application development by integrating the demultiplexing of completion events and the dispatching of their corresponding event handlers.

Key participants in the Proactor pattern:

1. ***Proactive Initiator***: The entity in the application that initiates an asynchronous operation. It registers both a *Completion Handler* and *Completion Dispatcher* with the *Asynchronous Operation Processor*.
2. ***Asynchronous Operation Processor***: It executes asynchronous operations and notifies the *Completion Dispatcher* when an operation completes.
3. ***Completion Handler***: It processes the results of an asynchronous operation. It is notified by the *Asynchronous Operation Processor* when an operation is complete.
4. ***Completion Dispatcher***: It invokes the correct call to the *Completion Handler* based on the execution environment.
5. ***Asynchronous Event Demultiplexer***: This component blocks waiting for events to occur on the *Completion Event Queue* and returns completed events to its caller.
6. ***Completion Event Queue***: It buffers completion events until they are dequeued by the *Asynchronous Event Demultiplexer*.

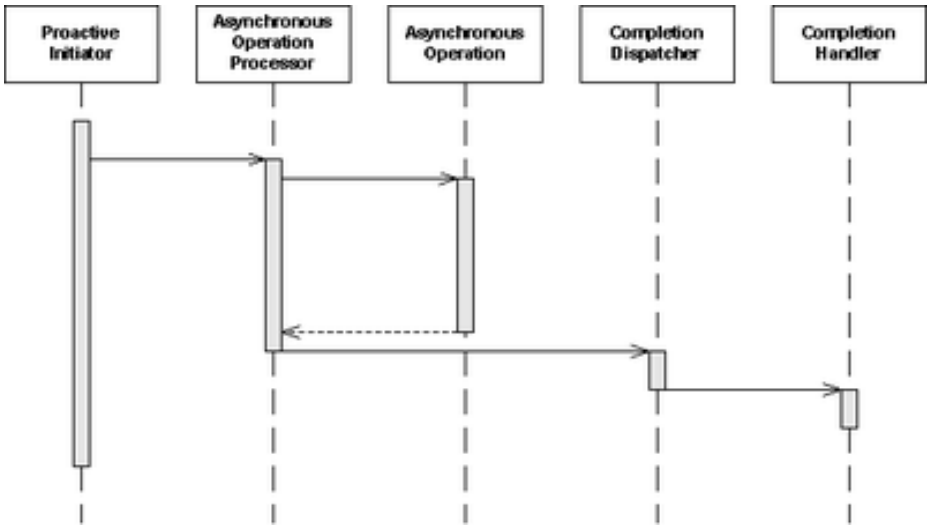

Figure 3: Proactor Software Pattern

4.2 Relation to Project Objectives and KPIs

This component serves the EMPYREAN functional requirement *F_SO.12 "Offload acceleration to nearby devices"* and provides enabler *EN_3 "High-Performance Data Transport Service"*. Moreover, it contributes to achieving the technical KPIs *T4.1 "Increase application-level small-message transfer performance"* and *T4.2 "Improve the RDMA programming efficiency of applications".*

To meet these goals, the deliverable describes the integration of the Software-Defined Interconnect (SID) with the vAccel (Section 5), enabling the offloading of computationally-intensive workloads to nearby acceleration devices. The SDI articulates the RDMA transport capabilities with network facing signalling rates up to 200Gb/sec, providing a high-throughput, low-latency data path. The proposed approach promotes the circular buffer interface concept for RDMA network I/O, directly contributing to the achievement of the KPIs T4.1 and T4.2. The developed mechanism supports transparent remote synchronization and enables batching small messages into larger buffers. As a result, it reduces housekeeping overhead, improves overall throughput, and significantly enhances RDMA programming efficiency, thereby supporting the targeted technical KPIs.

4.3 Architecture and Interfaces

The disaggregated circular buffer communication system is the heart of the software-defined interconnect architecture developed within EMPYREAN. The main concept is depicted in Figure 4. In EMPYREAN, we adopt and extend the well-known circular buffer paradigm to serve as the foundation for a low-level RDMA-based communication library, referred to as libRRR. Next, we provide an overview of the internal design and operation of this system.

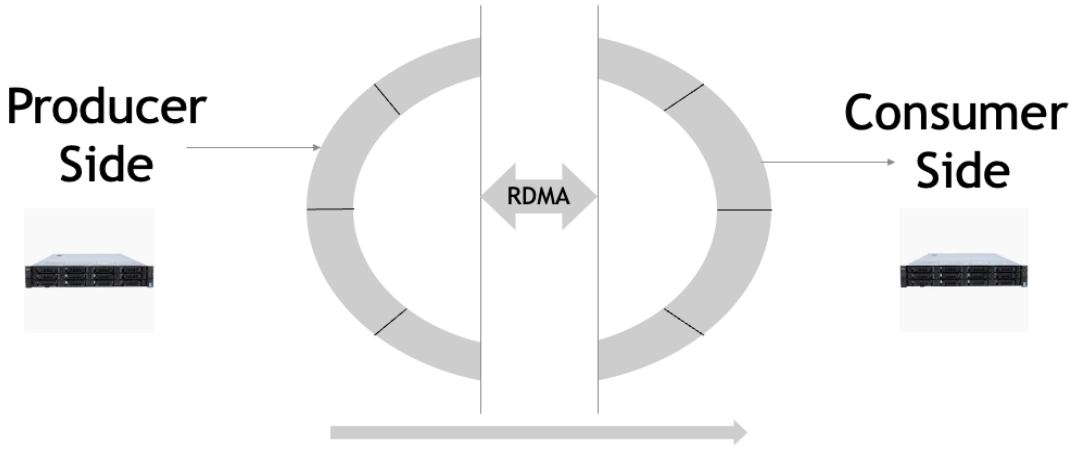

Figure 4: Single-sided RDMA Circular Buffer

Each RDMA circular buffer is single-sided and the baseline supports one producer and one consumer. Multi-producer/multi-consumer access is expected to be handled explicitly by the application that uses the library. The producer pushes data to the buffer and the consumer pulls from it, using the provided API. Notably, the buffer head and tail need to be polled for access, as the library does not provide any other notification mechanism. The applications are not exposed at all to RDMA communication, while form the developer's perspective interaction with the buffer is similar to accessing a local buffer, albeit through the exposed API, abstracting away all RDMA communication complexities.

Each circular buffer instance moves traffic towards one direction. To implement a full-fledged RPC communication, typically there is a requirement of instantiating two buffers per direction: one for control and one for data for request side and another set for response side (Figure 5).

The library offers comprehensive wrappers to implement the described RPC transport model. In this scheme, control and data messages can be transmitted out of order, and there is no one-on-one requirement for control and data buffers. Each control (Ctrl) command received at the destination may consume anywhere from zero data entries up to the whole buffer, a decision that is entirely left to the application.

Figure 5 illustrates the RPC mechanism. The API uses separate circular buffers for control (Submit Queue - SQ) and data (Submit Queue Data - SQD), which operate independently. This separation is crucial because some RPC calls may feature only control operations (e.g., remote read), while others may involve both control and variable-sized data (e.g., remote write), potentially end up consuming several entries of the circular buffer data. The circular buffer entries are of fixed size, defined during initialization, which simplifies memory management and alignment. The same buffer configuration applies to the asynchronous response path, which may or may not need to transfer data back to the requestor. To enable full-duplex RPC, a total of four (4) circular buffers are initiated.

Notably, these buffers are independently synchronized with their remote counterparts through the same RDMA channel. This design decouples the number of RDMA channels from the number of disaggregated circular buffers, allowing multiple RPC channels to be multiplexed over the same RDMA channel. This approach reduces connection context memory pressure. With buffers supporting lock-free operations, application developers can even use independent threads to feed control and data channels on each side, if needed by the application distributed communication. Nevertheless, the circular buffers structure guarantees in-order delivery within each stream.

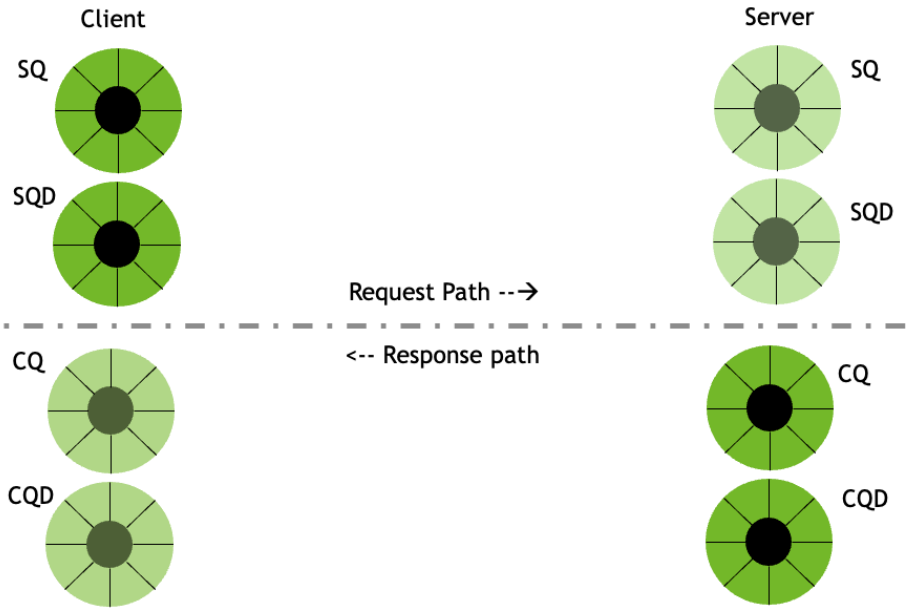

Figure 5: Typical RPC channel disaggregated buffer deployment

Figure 6 depicts the complete RDMA interaction sequence used to synchronize the contents of a local buffer with a remote buffer. This process involves three (3) key steps, all performed exclusively through RDMA memory operations.

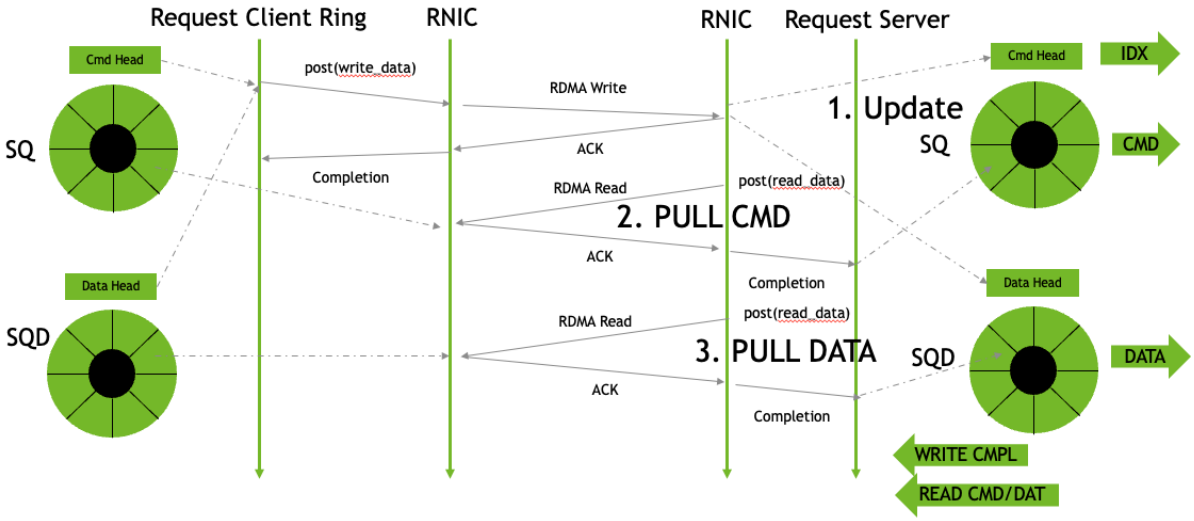

Figure 6: Request Path RDMA Remote Circular Buffer Sync

The first step involves RDMA write operations that update both tail and head entries of the remote circular buffers (i.e., rings). Specifically, the head is updated for the remote ring that is receiving new data, while the tail is updated for the ring in the opposite direction, for which the request initiator has already consumed an entry locally. In most cases, tail updates occur typically implicitly, as part of head updates in the opposite direction. However, explicit tail updates are also supported but are left to the application's responsibility when needed.

The second step is initiated by the receiver, which due to the previous head update, it can determine which part of the remote command (cmd) ring contains new command data that is not yet available locally. Using this information, the receiver performs an RDMA read to fetch all the updated remote command data. At this point batching of remote ring entries in one transfer is achieved, improving efficiency. If the application logic requires examining the command entries before deciding which data entries to fetch, a separate RDMA read is issued to pull in the relevant data ring contents, provided they are available.

In hardware implementations, the circular buffer logic is entirely abstracted from accelerators, which interact through a simple streaming interface.

Moreover, the software-defined control layer provides an interface for managing communication link establishment and buffer configurations. It ensures that:

- Links are only activated when endpoints are authenticated and associated.
- Message transmission may include optional authentication (e.g., HMAC[2]).
- Link-level properties such as jitter and latency are properly configured, as they directly related to circular buffer configurations.

[2] https://datatracker.ietf.org/doc/html/rfc4868

This software-defined control plane takes advantage of the capability of bringing up RDMA Queue Pairs (QPs) out-of-band, thereby decoupling link setup from the application data path. As a result, the application can remain focused purely on data transmission, free from the complexities of RDMA connection management.

4.3.1 Data path software public API

The data path software interface is implemented as a C programming language library that wraps disaggregated circular buffer operations over RDMA. This API handles the instantiation of RDMA communication channels and the associated ring buffers, exposing a typical ring buffer interface that enables efficient and streamlined manipulation of these resources.

Table 1: Data path software API – Available functions in the developed C-language library

rrr_init_rings
int **rrr_init_rings**(int sockfd, struct rdma_connection *res, struct rrr_ring_pair *ringp, char * recv_bufs[], char* send_bufs[],int blocksize,int iodepth, int submit_ctrl_buf_size, int completion_ctrl_buf_size, bool isServer, bool ext_buf_management)
This function is used both on client and server sides and initializes the ring pairs and associates them with an RDMA connection.
rrr_register_buffer_for_transfer
int32_t **rrr_register_buffer_for_transfer**(struct rrr_ring_pair *ringp, struct rdma_connection *res, unsigned char * buffer, uint32_t bfidx, uint64_t size)
This function registers a user application buffer to the ring associating it effectively with a ring entry at index bfidx and registers its address for RDMA transfers
rrr_async_rdma_remote_data_buf_list
int **rrr_async_rdma_remote_data_buf_list**(struct rrr_ring_pair *ringp, struct rdma_connection *res, uint64_t *laddr, uint32_t *lkey, uint64_t *raddr, uint32_t *rkey, uint32_t *len, int entries)
This function registers a receive-side ring buffer that accepts sent data from remote counterpart. The async refers to the asynchronous recycling of the receive buffer.
rrr_ring_get_next_head_cmd_buf
void * **rrr_ring_get_next_head_cmd_buf**(struct rrr_ring_pair *ringp, bool isSubmit)
This function get next head. It can be used either from client or server side
rrr_ring_get_next_tail_cmd_buf
void * **rrr_ring_get_next_tail_cmd_buf**(struct rrr_ring_pair *ringp, struct rdma_connection *res, uint32_t *idx, bool isSubmit);

This function gets next tail it can be used either from client or server side.
rrr_ring_get_next_tail_fixed_data_buf
void * **rrr_ring_get_next_tail_fixed_data_buf**(struct rrr_ring_pair *ringp, struct rdma_connection *res, uint32_t *idx, bool isSubmit);
This function is used to get next tail at the receive side when fixed receive buffers are used by the server for improved performance.
rrr_submit_ring_get_next_tail_free_data
int **rrr_submit_ring_get_next_tail_free_data**(struct rrr_ring_pair *ringp, uint32_t bufnum, uint64_t * localbfs, uint64_t *rembfs);
This function gets next tail data from remote ring side and copies them from remotebfs to localbfs are required updating the tai l in the process.
rrr_ring_unlock_cmd_buf
void **rrr_ring_unlock_cmd_buf**(struct rrr_ring_pair *ringp, uint32_t *idx, bool isSubmit);
Function protects buffer from being reused/retired until application that uses it stops needing it.
rrr_ring_synchronous_unlock_data_buf
void **rrr_ring_synchronous_unlock_data_buf**(struct rrr_ring_pair *ringp, uint32_t *idx, bool isSubmit);
This function protects data buffer contents until application has finished using them.
rrr_register_buffer_for_transfer
int32_t **rrr_register_buffer_for_transfer**(struct rrr_ring_pair *ringp, struct rdma_connection *res, unsigned char * buffer, uint32_t bfidx, uint64_t size);
This function registers buffer memory for RDMA transfers.
rrr_ring_data_async_get_next_head
int **rrr_ring_data_async_get_next_head**(struct rrr_ring_pair *ringp, int bufidx, bool isSubmit);
This function provides asynchronous ring buffer head retrieval
rrr_ring_commit
int **rrr_ring_commit**(struct rrr_ring_pair *ringp, struct rdma_connection *res, bool isSubmit);
This function provides commit / doorbell functionality that initiates remote buffer synching

4.3.2 Data path hardware interface

To support simple hardware sensors and other edge devices, the Software-Defined Interconnect provides a fully hardware-based data path interface built on the AXI-Stream protocol[3]. The circular buffer mechanism described earlier is integrated into the Nvidia FlexDriver[4] system, enabling seamless communication between software and hardware components at the edge.

Table 2: Data path hardware interface

AXI Send Master-Slave Interface
module axi_stream_512 (input wire clk, input wire rst_n, // Master -> Slave output reg [511:0] m_axis_tdata, output reg m_axis_tvalid, input wire m_axis_tready, output reg m_axis_tlast)
This is the verilog module interface that allows 512-bit data to be transmitted in a single cycle. t_data holds the data, t_valid indicates that data are valid, t_ready sets the interface into ready mode and t_last indicates the part of t_data 512-bit word that is valid for read in the current cycle.

4.3.3 Software-Defined Interface

The Restful API described below provides the necessary control path support required by the previous interfaces involved in data forwarding within the EMPYREAN platform. In addition to managing control operations, this control path interface also performs authentication for establishing connections initiated by the user.

This software-defined interface is designed to fulfil three primary roles: (i) define endpoint associations, establishing logical pairs between endpoints intending to utilize EMPYREAN RDMA secure links, (ii) assign desired performance characteristics, which the system will attempt to accommodate, and (iii) manage the actual link establishment, ensuring controller maintain a full overview of the deployment.

[3] https://developer.arm.com/documentation/ihi0051/latest/
[4] https://dl.acm.org/doi/10.1145/3503222.3507776

The API listed below offers the basic control capabilities and is designed to be triggered and orchestrated by EMPYREAN platform control and management plane, seamlessly integrating with the EMPYREAN deployment platforms.

Function: Set User Pair Association and Credentials
Description: Sets a user pair association for link bring up with credentials
URL: /SetPair
HTTP Type: POST
POST JSON Data: { {“Endpoint1”: <Endpoint1 Identifier>”}, {“Endpoint2”: <Endpoint2 Identifier>”}, {“Endpoint2”: <Comms Secret - Optional>”} }
RESPONSE JSON Data: { {“PairID”: <idenfifier>”}}
Success Response (code 200)
Internal Error Codes: code 500 “Endpoints not found”

Function: Set Link Properties
Description: Sets a user pair association for link bring up with credentials
URL: /SetLinkProperties
HTTP Type: POST
POST JSON Data: { {“PairID”: <identifier>”}, {“Jitter”: <Value>”}, {“Latency”: <Value>”} }
Success Response (code 200)
Internal Error Codes: code 500 “Values not in Nanosecond format”
REMARKS: SetPair should be issued before to have a valid pair association id. The values reflect specific configuration of circular buffer that aims to meet requirements. Based on the requested values (e.g. if are unreasonable) or the overall load of the network the requirement might not be met so the global controller should have an overview of the deployment and current activity before configuring.

Function: Bring up Link
Description: Coordinates the link bring up on both sides
URL: /LinkBringUP
HTTP Type: POST
POST JSON Data: {“PairID”: <identifier>”}
Success Response (code 200)
Internal Error Codes: code 500 “Bring up failed”
REMARKS: Function should be called after the previous configuration functions have been issued.

5 Hardware Acceleration Abstractions

5.1 Overview

The vAccel[5] framework enables seamless acceleration workload offloading by abstracting hardware acceleration capabilities and offering a unified API for diverse backend targets. In the context of EMPYREAN, vAccel is integrated to support efficient, low-latency execution of AI/ML workloads across distributed, heterogeneous environments.

A key aspect of this integration involves supporting libRRR, the RDMA-capable, user-level communication library described in Section 4, as a vAccel transport plugin. This integration enables seamless remote AI/ML inference task offloading to hardware-accelerated endpoints across the EMPYREAN IoT-edge-cloud continuum. By embedding the vAccel execution model within libRRR, EMPYREAN introduces a lightweight, low-latency, and high-performance mechanism for invoking vAccel plugins over RDMA channels, ensuring optimal performance and scalability.

5.2 Relation to EMPYREAN Objectives and KPIs

This component is a key enabler for:

- **F_SO.12** – *Offload acceleration to nearby devices*
- **EN_3** – *High-Performance Data Transport Service*

and directly contributes to the achievement of:

- **T4.1** – *Increase application-level small-message transfer performance*
- **T4.2** – *Improve RDMA programming efficiency of applications*

By abstracting the complexity of RDMA operations and integrating with vAccel's plugin-based execution model, this integration offers a transparent and efficient offloading mechanism, particularly suitable for lightweight edge devices. It supports scalable and secure AI inference capabilities across EMPYREAN Associations.

In scenarios such as anomaly detection in smart factories or real-time image processing in surveillance systems and smart agriculture, vAccel enables resource-constrained edge devices to offload processing tasks to nearby accelerators, while maintaining end-to-end TLS encryption and attestation support. Moreover, vAccel integration extends to federated learning workflows, enabling distributed training on multiple edge devices and leveraging accelerated aggregation in more powerful environments (e.g., far edge, cloud) using vAccel-enabled plugins.

[5] https://vaccel.org

5.3 Architecture and Integration Details

The integration of vAccel within EMPYREAN stack adopts a layered, modular architecture, designed to support efficient offloading and execution of AI/ML workloads across distributed systems. Communication is orchestrated through the Remote Ring-buffer Runtime (libRRR) over RDMA, enabling ultra-low latency, zero-copy interactions.

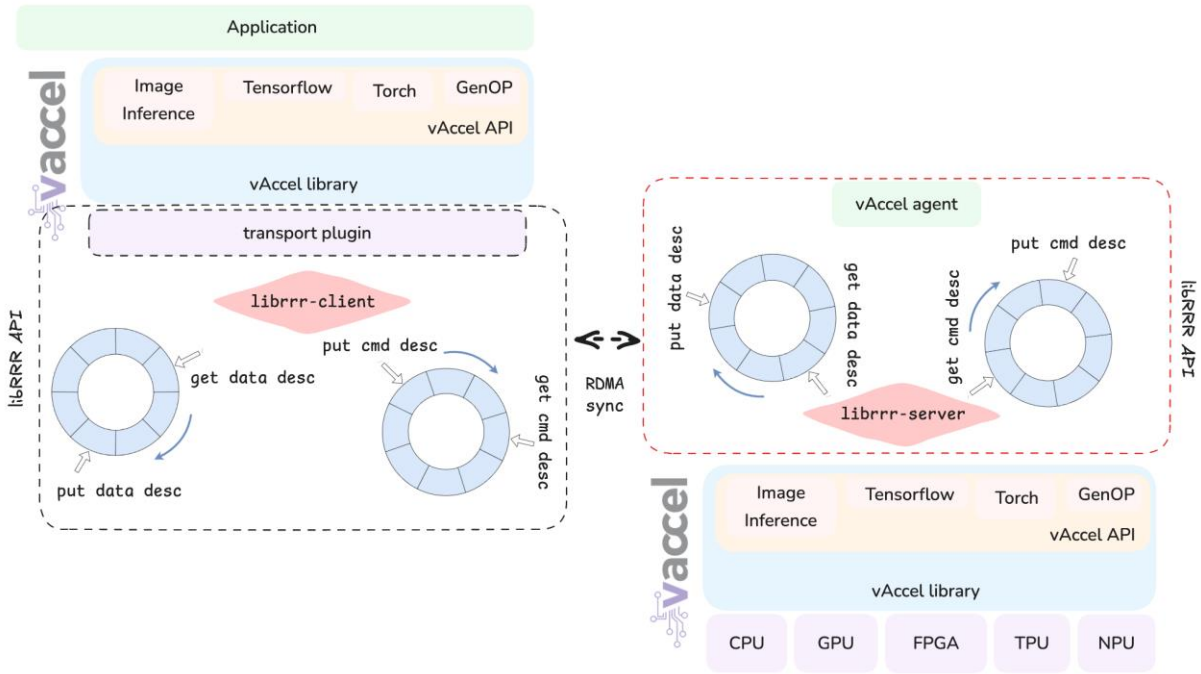

Figure 7: vAccel and libRRR integration

The architecture comprises the following key components (Figure 7):

- ***vAccel Application***: The user application leverages the vAccel API to issue high-level acceleration requests.
- ***vAccel Core Library (libvaccel)***: Acts as the main entry point for applications. It marshals commands and delegates them to the appropriate client backend (plugin).
- ***vAccel Client Backend (vaccel-client-rrr)***: Encodes acceleration requests and transmits them through the libRRR communication layer, leveraging a ring-buffer abstraction for efficient data exchange.
- ***libRRR (Remote Ring-buffer Runtime):*** The high-performance communication runtime mechanism built on top of RDMA transport, facilitating zero-copy, low-latency command and data descriptor exchanges between client and server (detailed in Section 4).
- ***vAccel Server Backend (vaccel-server-rrr):*** Dequeues and interprets incoming requests, dispatching them to the appropriate hardware backend through the vAccel plugin interface.

- ***vAccel Plugin Layer***: Acts as a bridge between the vAccel runtime and the hardware acceleration backends, interfacing with the actual transport layer (libRRR) on one side and supporting hardware devices, such as GPUs, FPGAs, or TPUs on the other side.
- ***Hardware Acceleration Device***: The physical endpoint that executes AI/ML tasks, fully abstracted through the plugin interface for seamless integration and portability.

5.4 Implementation and Integration Points

The integration of vAccel into EMPYREAN platform is implemented through a series of modular development tasks and well-defined interconnection points, carefully aligned with the architectural objectives and system-level constraints of hyper-distributed environments.

Key integration steps include:

1. ***Development of vaccel-client-rrr and vaccel-server-rrr:*** These components constitute the client-server communication model for vAccel over the Remote Ring-buffer Runtime (RRR). The client component serializes commands and associated arguments into buffer-length structures, while the server deserializes, validates, and dispatches them for execution. This modular separation ensures flexibility in deployment, e.g., placing clients on IoT devices and servers on edge or cloud nodes.
2. ***Design and Implementation of libRRR Protocol***: The Remote Ring-buffer Runtime (libRRR) layer provides a zero-copy, lock-free communication mechanism over RDMA. It abstracts the complexities of RDMA interactions while exposing efficient enqueue/dequeue operations that are critical for achieving high throughput and low latency. The libRRR component is described in detail in Section 4.
3. ***Deployment of RDMA Transport Infrastructure:*** High-speed RDMA-based interconnects (e.g., RoCE, Infiniband) are used to transport ring-buffer payloads, minimizing latency associated with conventional TCP/IP stack. This is particularly essential for supporting real-time, latency-sensitive AI workloads in distributed edge-cloud execution paths.
4. ***Implementation of Generic Operations (GenOps)***: To support frameworks like TensorFlow[6] and PyTorch[7], high-level tensor operations are abstracted as GenOps, which are routed through the plugin layer. This approach enables device- and framework- agnostic acceleration, making it easier to plug in various backend hardware types seamlessly.
5. ***Resource Containerization via vAccel-resource***: A lightweight vAccel-resource entity is introduced to encapsulate hardware contexts and metadata, such as memory mappings, session states, and device capabilities. This facilitates orchestration and

[6] https://www.tensorflow.org
[7] https://pytorch.org

resource isolation in multi-tenant and containerized deployment environments within EMPYREAN.

6. ***Feedback Loop to EMPYREAN Orchestrator***: The vAccel server continuously exports runtime metrics, such as latency, throughput, queue depth, which are consumed by EMPYREAN's orchestration and deployment mechanisms. This enables dynamic offloading decisions and real-time optimization of distributed workloads based on system state and workload demands.

As a foundational enabler of hardware-accelerated execution within the EMPYREAN control and management plane, the vAccel framework, tightly integrated with the libRRR communication layer, plays a critical role in delivering high-performance, low-latency AI/ML capabilities across all project use cases. Rather than operating as an isolated component, vAccel is deeply embedded within the EMPYREAN compute and resource orchestration stack. It seamlessly interoperates with the key platform components such as the EMPYREAN Aggregator, Service Orchestrator, EMPYREAN Controller, and Telemetry Service.

By leveraging libRRR's ring-buffer-based transport over RDMA, vAccel enables efficient offloading of compute-intensive workloads from constrained edge devices to remote hardware accelerators (e.g., GPUs, FPGAs). This remote execution model dramatically reduces inference latency, preserves energy on edge nodes, and ensures timely AI operations — including inference, pattern recognition, and predictive analytics— even when local compute resources (e.g., IoT devices, on-premise edge) are limited.

Moreover, the integration directly supports dynamic optimization strategies driven by the telemetry and service assurance mechanisms (e.g., Analytics Engine). As the system processes telemetry data and detects evolving runtime conditions, vAccel's abstraction layer facilitates the rapid reallocation or offloading of workloads to the most suitable acceleration resources, whether local or remote.

This dynamic capability aligns tightly with EMPYREAN's mission and goals enabling:

- ***Autonomous and resilient workload adaptation***, especially under varying edge/cloud resource constraints.
- ***Real-time AI inference*** in mission-critical edge deployments, such as industrial automation or autonomous mobility scenarios.
- ***Scalable orchestration of heterogeneous resources***, ensuring uniform access to acceleration across diverse hardware environments.
- ***Energy-aware, latency-optimized execution***, supporting green computing goals while maintaining strict performance guarantees.

Across all EMPYREAN use cases, the integrated vAccel and libRRR stack enhances the platform's ability to balance workloads intelligently, maintain low-latency responsiveness under load, and deliver inference-as-a-service in a hyper-distributed environment. Its deep coupling with the control and management plane makes it a key enabler for fulfilling the project's performance, scalability, adaptability, and operational efficiency technical KPIs.

6 Service Assurance

6.1 Overview

EMPYREAN architecture integrates distributed service assurance mechanisms for the self-driven adaptability of the IoT-edge-cloud continuum through multiple instances of Analytics Engine that utilize real-time telemetry data. The Analytics Engines are part of the Monitoring and Observability layer, enabling EMPYREAN Aggregators to continuously monitor and predict probable performance and security issues in Associations, allowing for prompt response to anomalies and ensuring efficient resource utilization. Unlike deployment and orchestration operations, which are generally executed per-request, service assurance operations are executed in automated closed-loops to ensure applications perform as intended by dynamically adjusting deployments and Association configuration based on real-time analytics and telemetry data.

These engines employ continuous analysis techniques—such as machine learning, machine reasoning, swarm intelligence, and robust adaptive optimization—to drive orchestration mechanisms to (i) adapt resources within the Associations, (ii) provide dynamic load balancing of processing workloads, and data within and across Associations, (iii) migrate workloads to optimize energy efficiency, (iv) detect and categorize abnormal situations in applications/resources, and (v) mitigate resource fragmentation and connectivity issues. These capabilities ensure that applications perform as intended while proactively or reactively triggering necessary re-optimizations to provide optimal performance, reliability, and efficiency across the complex and dynamic Association-based IoT-edge-cloud continuum.

By implementing these data-driven mechanisms, the EMPYREAN platform can achieve robust anomaly mitigation, adaptability, and self-driven recovery, ensuring resilient and efficient operations in the face of unforeseen issues across the infrastructure.

6.2 Relation to EMPYREAN Objectives and KPIs

The Analytics Engine is one of EMPYREAN's enabling technologies that support the autonomous operation and self-driven adaptability across the Association-based continuum. To this end, the Analytics Engine contributes to the achievement of the following key objectives and technical KPIs:

- *T1.2 - Increase reliability in the edge*: Anomaly detection and failure prediction using AI/ML within the Analytics Engine increases reliability by anticipating and pre-empting failures at the edge. The notifications provide reliable asynchronous communication for proactive response to performance issues.
- *T1.4 - Provide low and predictable latency for hyper-distributed applications*: Integration of time-series databases and publish/subscribe mechanisms facilitates low-latency data ingestion and real-time event propagation. Moreover, the modular

design enables localized execution of analytics, allowing edge nodes to quickly react to changes in their local environment.

- *T2.3 - React fast to rapid changes in computational and data demands to maximize the number of demands served*: It provides continuous learning and inference to detect resource saturation or performance degradation, triggering rapid re-optimization actions. Dynamic reconfiguration and orchestration are triggered based on telemetry data, ensuring responsiveness to changes in demand and resources.
- *T2.5 - Increase the robustness of the algorithms, ensuring consistent performance even under uncertain or noisy conditions*: By employing robust adaptive optimization and swarm intelligence, it provides adaptability to uncertainties and non-deterministic behaviours in edge environments. The feature extraction, data normalization, and filtering improve input quality, reducing the impact of noisy telemetry.

6.3 Architecture

A key requirement for designing the Analytics Engine is its scalability, both in terms of integrating diverse data sources and executing AI/ML-based algorithms. Additionally, the Analytics Engine must efficiently handle data ingestion from all telemetry and monitoring resources across infrastructure segments, enabling the seamless combination of data from infrastructure, through Associations, to deployed services and applications. The ability to merge application and infrastructure-based metrics is closely linked to the AI/ML functions of the EMPYREAN distributed control and management plane. Facilitating service assurance mechanisms to aggregate and utilize data from various infrastructure layers and Associations is essential for implementing more intelligent self-adaptation and self-optimization mechanisms.

The design of the Analytics Engine follows a modular and scalable microservices-based approach, providing flexibility for integrating multiple data sources and executing diverse data-driven algorithms. It comprises four primary services: the *Access Interface*, *Data Connector*, *Data Manager,* and *Event Detection Engine*. Figure 8 illustrates the architecture of the EMPYREAN Analytics Engine, highlighting its key components and their interactions with other EMPYREAN services.

The *Access Interface* enables bidirectional communication to exchange commands, information, and notifications among the Analytic Engine instances and other services within the distributed EMPYREAN control and management plane. The design includes two distinct interfaces, each designed to meet specific interaction requirements:

- ***RESTful API*:** It provides a stateless, synchronous interface for executing control operations. It is designed to handle standard CRUD operations (Create, Read, Update, Delete) via standard HTTP methods (POST, GET, PUT, DELETE). The RESTful API is ideal for tasks requiring instant feedback or control over analytics operations. The *API Server* component implement this interface.

- ***Asynchronous interface*:** It is provided by the *Notification Manager* component and is built on the Eclipse Zenoh[8], supporting flexible, asynchronous, and persistent communication using its publish/subscribe and query mechanisms. It allows the Analytics Engine to send real-time notifications and event updates to the service orchestration mechanisms and platform web-based dashboard.

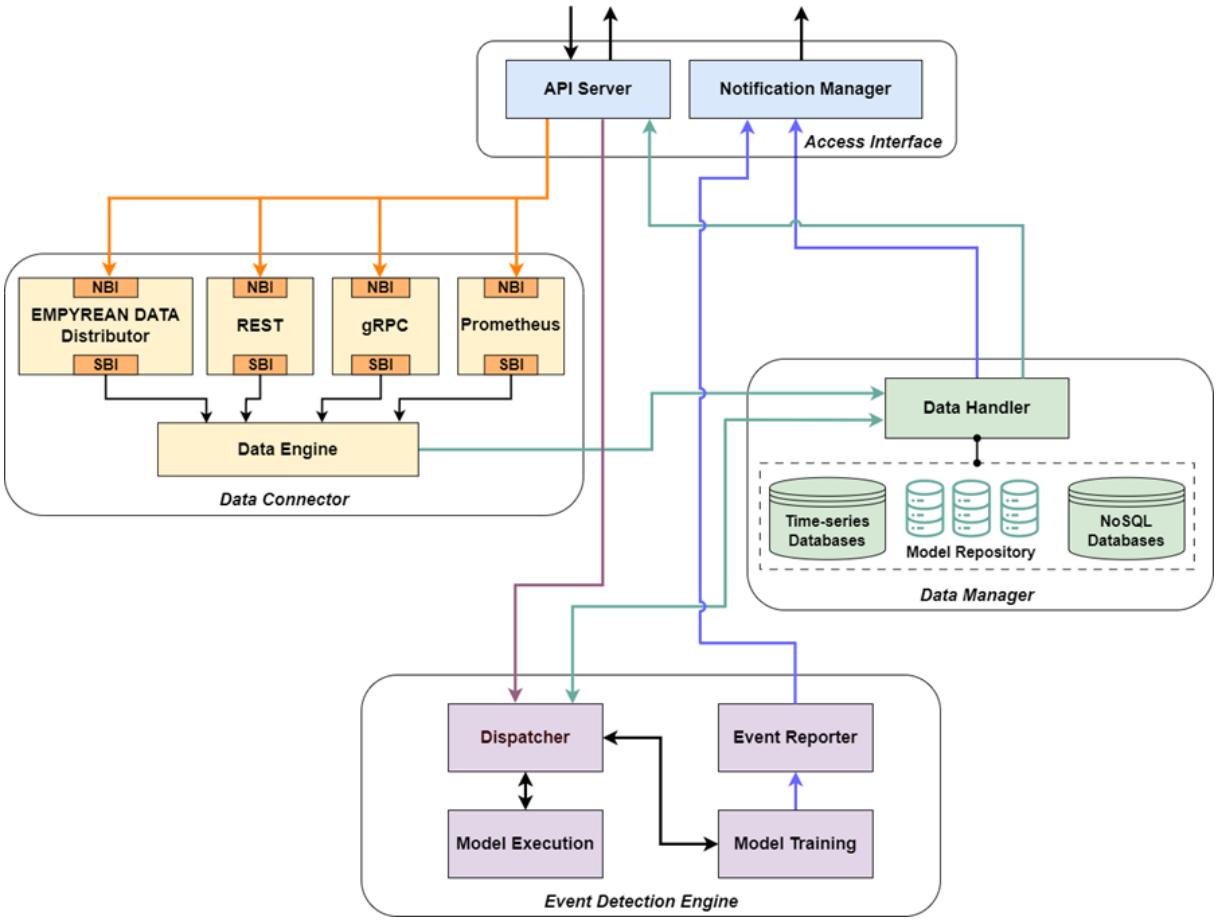

Figure 8: Analytics Engine architecture and core components

The *Data Connector* service manages the collection of raw monitoring and streaming telemetry data from various sources within the Monitoring and Observability layer. It functions as pre-processing element, performing tasks such as data filtering and normalization, before forwarding the processed data to the *Data Manager service.* The design supports multiple data ingestion methods (i.e., pull and push) and accommodates diverse types of monitored data (i.e., metrics, events, streams) through various protocols.

Each data collection mechanism is implemented as a custom plug-in. The initial design supports: (i) a REST client for managing periodic and on-demand monitoring data collection through EMPYREAN's telemetry service default interface, (ii) a gRPC module for handling streaming telemetry data, (iii) an agent customized to interact with the decentralized EMPYREAN data distributor service to ensure seamless data sharing within the EMPYREAN

[8] Eclipse Zenoh: https://zenoh.io

platform, and (iv) a Prometheus client for collecting and managing monitoring data in the OpenMetrics[9] format.

A common design approach has been adopted for these plug-ins, where each implements a standardized Northbound Interface (NBI) and Southbound Interface (SBI). The NBI provides a unified interface for receiving configuration instructions from the Access Interface service, while the SBI forwards the collected data to the Data Engine component.

Pre-processing is implemented as a separate component, the *Data Engine*, which formats as well as augments monitoring and telemetry for predictive model creation. It ensures data normalization for consistency across different telemetry formats, performs feature extraction and transformation to prepare data for advanced analytics, and applies filtering and aggregation to reduce noise and enhance meaningful insights before storage and analysis.

The *Data Manager* service is responsible for managing data storage and facilitating data exchange between internal and external components. It provides local storage of processed data, trained models, and analysis results. The EMPYREAN Edge Storage component provides the repository of trained models, which enables collected data and trained models to be encrypted and stored in a distributed manner. This approach enhances security, fault tolerance, and accessibility, ensuring that data remains protected while being readily available to the Event Detection Engine for analysis, detection, and machine reasoning tasks.

Additionally, the Data Manager incorporates various database technologies to handle both structured and unstructured data, including NoSQL and time-series databases. NoSQL databases store unstructured or semi-structured data such as event logs, metadata, analysis results. and post-analysis content. Time-series databases handle the continuous streams of monitoring data, telemetry, and event logs from the Data Connector service, which later fuels the operations of the Event Detection Engine. By combining edge storage, NoSQL, and time-series databases, the Data Manager ensures efficient data processing, storage, and retrieval, supporting the EMPYREAN platform's mission of real-time event detection and intelligence.

The *Data Handler* component is a RESTful controller that provides a standardized interface for both Analytics Engine components and external EMPYREAN services to access historical telemetry data and interact with the other parts of the Data Manager service. It exposes RESTful methods that simplify data retrieval and management across diverse databases and storage resources, ensuring a unified and efficient data access layer. These methods abstract the complexities of data manipulation, allowing for seamless retrieval of telemetry data based on specified parameters while supporting filtering and query operations. Additionally, the Data Handler ensures that stored information is properly updated and maintained. For secure and efficient storage and retrieval, the component interacts with EMPYREAN's edge storage resources, handling both data and trained models through their standardized S3-compatible APIs.

[9] OpenMetrics: https://github.com/OpenObservability/OpenMetrics

The *Event Detection Engine* service implements the core functionality of EMPYREAN's distributed service assurance framework, leveraging real-time telemetry data and machine reasoning techniques to ensure system reliability. It enables the integration and execution of data-driven algorithms that safeguard the performance and availability of deployed applications and Associations. By incorporating AI/ML-based mechanisms, the Event Detection Engine builds knowledge and intelligence to sense (detect what is happening), discern (interpret detected events), and infer (understand implications) over an infinite time horizon control loop. Its key functions include (i) AI/ML-driven anomaly detection, failure prediction, and resource optimization, (ii) correlation of infrastructure and application-level metrics to ensure consistent performance, and (iii) triggering adaptive control mechanisms to dynamically respond to detected events.

The Event Detection Engine consists of four main components: the *Dispatcher*, *Model Training*, *Model Execution*, and *Event Reporter*. The Dispatcher handles interactions with the Access Interface and Data Manager services as well as it oversees the operation of the internal components. It also coordinates the execution of inference, analysis, and training operations. The Model Training supports the selection, configuration, and optimization of AI/ML-based predictive and analytical models. The implementation will facilitate the definition and integration of user-defined detection models within the Analytics Engine, provided they align with the adopted processing pipeline and APIs. The Model Execution manages the instantiation and execution of available detection and analysis methods. Depending on the available telemetry information, these mechanisms will operate at different timescales. They will autonomously drive the orchestration mechanisms in re-optimizations and adaptations within an Association. All detected anomalies and performance issues are forwarded to the Event Reporter, which then automatically delivered to the appropriate internal components (e.g., Data Manager) and EMPYREAN orchestration and management services (e.g., EMPYREAN Aggregator, Service Orchestrator) to trigger any required remediation actions.

By combining real-time monitoring, AI-driven insights, and adaptive control, the Analytics Engine (i) ensures fast reaction to rapid changes in computational and data demands to maximize the number of served demands, (ii) increase robustness, ensuring consistent performance even under uncertain or noisy conditions, (iii) maintain optimal performance by quickly identifying and resolving anomalies, and (iv) learns from past anomalies and recovery actions to improve future responses.

6.4 Implementation

During the reporting period, we focused on implementing the Access Interface, Data Connector, and Data Manager services. Additionally, we developed the core logic of the Event Detection Engine, with its full implementation scheduled for the second iteration of the implementation plane (M16-M26). The overall design of the Analytics Engine, along with the initial implementation of these services, ensures a robust, adaptable, and cloud-native application. Bellow, we provide an overview of the functionalities developed during the first iteration of the implementation plan.

The Access Interface components are implemented in Python, with the API Server built using the FastAPI[10] web framework. FastAPI offers high performance, automatic data validation, and ease of use, making it an excellent choice for RESTful APIs and microservices. It is built on Asynchronous Server Gateway Interface (ASGI)[11], using Starlette[12] for async support and Pydantic[13] for automatic validation of request and response data. Figure 9 shows the methods exposed by the Access Interface's RESTful API.

[10] FastAPI framework: https://fastapi.tiangolo.com
[11] ASGI: https://asgi.readthedocs.io/en/latest/
[12] Starlette: https://www.starlette.io
[13] Pydantic: https://docs.pydantic.dev/latest/

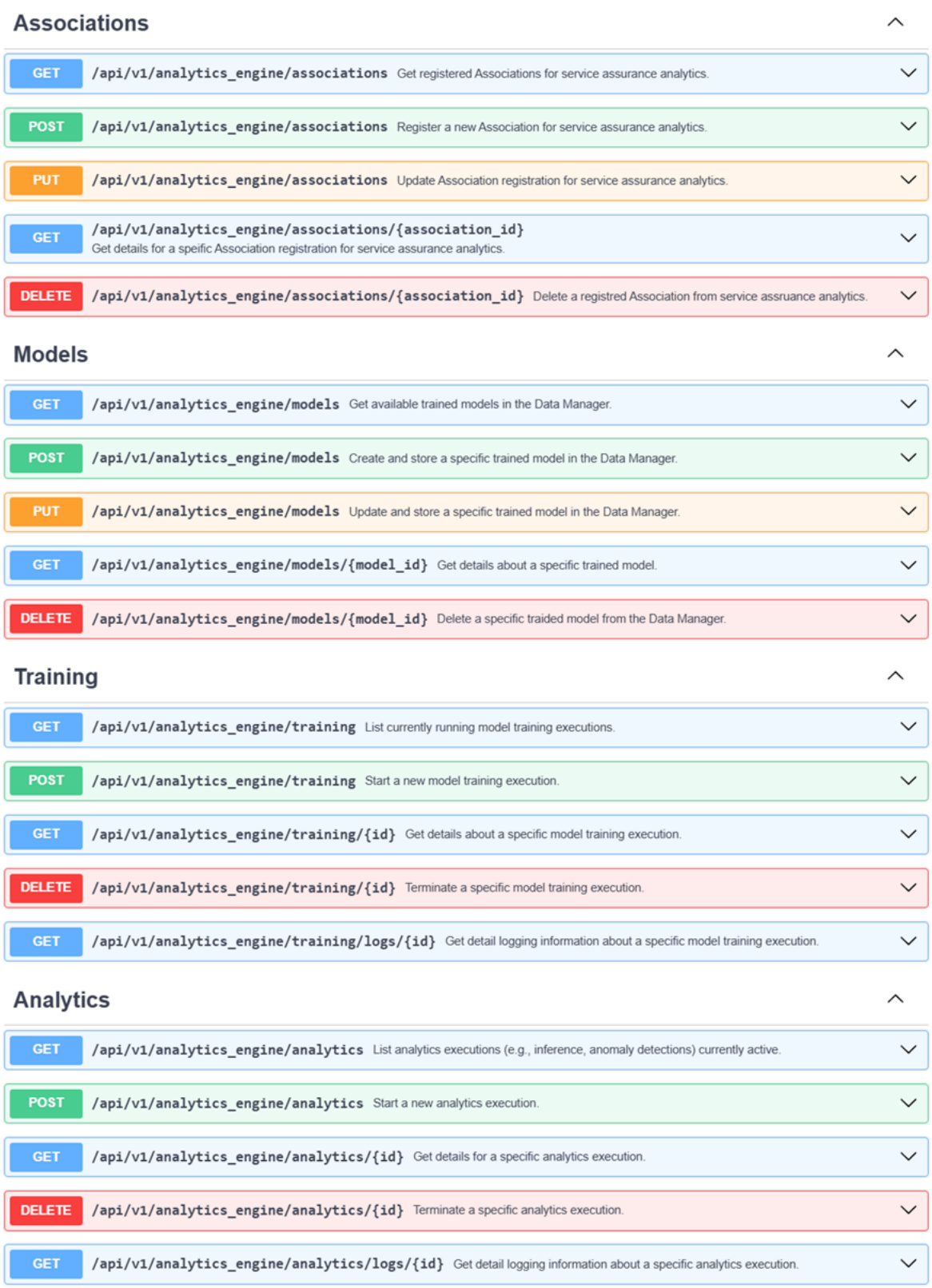

Figure 9: Analytics Engine – Access Interface RESTful API

The Notification Engine is built using the Eclipse Zenoh[14] and the PyQt[15] framework. For the asynchronous interface and communication, the implementation leverages the Zenoh key expressions, the topics, to implement efficient and scalable publish-subscribe interactions, enabling real-time messaging across distributed systems. In this setup, messages published by components such as the Data Manager or Event Detection Engine are sent to designated topics. These topics then broadcast all received messages to subscribed clients (e.g., EMPYREAN Aggregator, UI, CLI), ensuring that each component receives the same set of notifications simultaneously.

The Notification Engine posts messages in JSON format, following a predefined structure:

- ***analytics_engine_uuid***: (*string*): Unique identifier for the Analytics Engine instance.
- ***event*** (*string*): Unique identifier for the event.
- ***message*** (*object*): Collection of event-related parameters containing the necessary information.
- ***timestamp*** *(integer):* The timestamp of the event.

Table 3 outlines the detailed structure of notification messages, describing the expected elements and their roles in the communication process. This structured approach ensures consistent message delivery across the EMPYREAN platform, improving responsiveness and facilitating real-time control and management operations.

Table 3: Analytics Engine - Notification Engine messages

Event Identifier	Description	Parameters
NODE_UP	A new worker node is detected to a specific cluster, triggered on joining a cluster or an existing one becomes online again.	***node_id*** (*integer*): worker node unique identifier ***cluster_id:*** *(integer)*: K8s/K3s cluster unique identifier ***asociation_id***: (*integer*): EMPYREAN Association unique identifier
NODE_FAILED	Worker node does not operate properly, unable to serve workloads.	***node_id*** *(integer):* worker node unique identifier ***cluster_id*** *(integer):* K8s/K3s cluster unique identifier ***association_id***: (*integer*): EMPYREAN Association unique identifier ***message*** *(string):* event related information

[14] Eclipse Zenoh: https://zenoh.io

[15] PyQt: https://riverbankcomputing.com/software/pyqt/intro

NODE_DOWN	A worker node is detected unavailable, triggered on leaving a cluster or becoming offline.	***node_id*** *(integer)*: worker node unique identifier. ***cluster_id*** *(integer):* K8s/K3s cluster unique identifier. ***association_id***: (*integer*): EMPYREAN Association unique identifier.
NODE_STRESSED	A worker node is stressed with high load for a significant amount of time.	***node_id*** *(integer)*: worker node unique identifier. ***cluster_id*** *(integer):* K8s/K3s cluster unique identifier. ***association_id***: (*integer*): EMPYREAN Association unique identifier. ***operational_status*** *(object):* affected operational parameters.
ASSOCIATION_STRESSED	An Association is stressed with a high load for a significant amount of time.	***association_id***: (*integer*): EMPYREAN Association unique identifier. ***operational_status*** *(object):* affected operational parameters.
DEPLOYMENT_FAILED	Deployed application is failed, triggered when at least one microservice is not running properly.	***deployment_id*** *(integer)*: Application deployment unique identifier. ***affected_microservices*** *(array):* List of affected microservices identifier ***feedback*** *(object)*: Information for the detected issue
DEPLOYMENT_QOS	Quality of service is not the expected for at least one microservice	***deployment_id*** *(integer)*: Application deployment unique identifier. ***affected_microservices*** *(array):* List of affected microservices identifier ***feedback*** *(object)*: Information for the detected issue
DEPLOYMENT_MIGRATION	Suggestion for migrating a deployment due to detected issues.	***deployment_id*** *(integer)*: Application deployment unique identifier. ***feedback*** *(object)*: Information for the detected issue

The implementation of the Data Connector service follows a modular approach to ensure flexibility, scalability, and interoperability with the various telemetry sources. It is built using a combination of open-source frameworks and technologies to facilitate seamless data ingestion. Each connector component is implemented as a custom plug-in, supporting different data ingestion methods, while exposing a common RESTful northbound interface (NBI). Figure 10 presents the initial version of the implemented NBI.

The Southbound Interface (SBI) delivers collected telemetry data to the Data Engine component, ensuring compatibility with storage and analysis services. To achieve this, SBI utilizes Pandas[16] DataFrames as the primary data structure, offering several key advantages, such as, (i) efficient data handling as DataFrames provide a structured representation of telemetry data, enabling easier processing, filtering, and manipulation, (ii) interoperability

[16] https://pandas.pydata.org

with popular analysis frameworks (e.g., NumPy, SciPy, Scikit-learn), ensuring smooth data analysis and transformation workflows, and (iii) performance optimization, as Pandas is optimized for high-performance operations on structured data, such as vectorized computations and parallel processing.

Next, the Data Engine enhances the quality of collected data before storage and analysis by offering data normalization, feature extraction and transformation, filtering and aggregation and processed data forwarding to the Data Manager service for storage and retrieval.

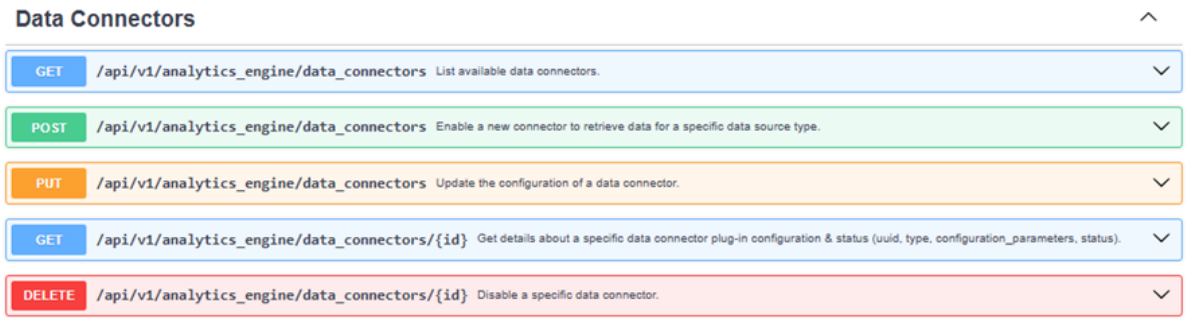

Figure 10: Analytics Engine – Data Connector plug-ins RESTful northbound interface

The Data Manager integrates multiple database technologies to efficiently handle diverse data types, ensuring optimal performance across different workloads. To store event logs, metadata generated by the Event Detection Engine, post-analysis reports, historical insights, and processed data results from analytics pipelines, the Data Manager utilizes MongoDB[17], a document-based NoSQL database for unstructured and semi-structured data. MongoDB was selected as it offers flexibility in storing dynamic data structures as well as indexing and aggregation capabilities for supporting fast querying and data retrieval. For managing continuous telemetry streams, the Data Manager integrates InfluxDB[18], a high-performance time-series database that provides (i) high-frequency data ingestion from the Data Connector service, (ii) efficient querying of historical monitoring data for predictive analytics, and (iii) retention policies to manage the storage lifecycle of telemetry logs. InfluxDB's built-in query language enables complex filtering, transformations, and real-time analytics, making it well-suited for event-driven intelligence in the EMPYREAN platform.

In addition, to ensure local storage and edge-based data management, the EMPYREAN Edge Storage component is leveraged (for further details, see D3.1 (M15)). This approach provides a S3-compatible, secure, fault-tolerant, and distributed object storage system that offers (i) secure encryption for stored data and trained models, (ii) scalability to handle large volumes of telemetry and analytical data, and (iii) seamless integration with the EMPYREAN platform via standard S3 APIs.

[17] https://www.mongodb.com
[18] https://www.influxdata.com

Moreover, the Data Handler component is implemented in Python using also well-known frameworks and libraries such as FastAPI, PyQt, PyMongo[19], Boto3[20], and InfluxDB 3.0 client[21]. It facilitates interactions with the integrated database and storage resources. The Data Handler exposes RESTful methods that allow Data Connector components to efficiently populate the data stores and the Event Detection Engine components to retrieve historical telemetry data. The available RESTful methods are shown in Figure 11.

Figure 11: Analytics Engine – Data Manager RESTful API

6.5 Relation to Use Cases

As an integral part of the EMPYREAN control and management plane, the Analytics Engine functions as an intelligence layer that empowers all project use cases with enhanced situational awareness, adaptive optimization, and context-aware decision-making. Rather than operating in isolation, the Analytics Engine is tightly integrated with core components such as the EMPYREAN Aggregator and Service Orchestrator, ensuring seamless access to its capabilities across the distributed edge-cloud infrastructure.

It processes both real-time and historical telemetry data, applying advanced AI/ML-driven predictive analytics and event detection algorithms to support a wide range of operational objectives. This enables the system to proactively respond to dynamic conditions, resource fluctuations, and performance anomalies, directly contributing to the achievement of each use case's KPIs. The Analytics Engine's value is particularly evident across the project's use cases, where it ensures high reliability and low-latency responsiveness in mission-critical edge scenarios. Additionally, its inference and optimization capabilities support the autonomous adaptability of workloads, enabling intelligent workload balancing in response to evolving demands and variations in edge and cloud resource availability.

[19] https://github.com/mongodb/mongo-python-driver
[20] https://github.com/boto/boto3
[21] https://github.com/InfluxCommunity/influxdb3-python

7 Intelligent Autoscaling and Adaptive Computing Management

7.1 Overview

This work focuses on applying AI/ML techniques to enable vertical auto-scaling within a Kubernetes cluster within the edge-cloud continuum. Specifically, the objective is to develop an ML-based vertical auto-scaler, named VPA-pilot, that leverages collected monitoring data to recommend suitable container size for workloads. By tailoring resource allocations more precisely, this approach enhances container bin-packing efficiency on worker nodes, reducing the overall number of active nodes. In cloud environments, this results in lower execution costs, while in on-premise deployments, it can also allow powering off resources, leading to improved energy efficiency and further cost savings.

In the edge-cloud continuum, besides traditional workloads that use CPU and RAM resources, the emerging hyper-distributed AI applications also require GPU resources. However, unlike CPU and memory, dynamic GPU fractioning remains an immature and evolving area within Kubernetes. Therefore, a key focus of this work is to research state-of-the-art methods for dynamic GPU fractioning, and to explore how vertical auto-scaling techniques can be extended to support GPU workloads in hyper-distributed environments.

7.2 Background and Challenges

Kubernetes is the de-facto industry standard for cloud infrastructure resource management and orchestration, and it has also been adopted as the main low-level orchestration software for the edge-cloud continuum. In a Kubernetes cluster, a large number of workloads of different applications are running on a cluster of computers called *nodes*. A workload does not exclusively occupy a node but runs with others together on a node. A workload is hosted in a corresponding *container*, which keeps the workload isolated from others on the same host computer(node). Containers work like VMs but with different mechanisms and much less overhead. When we deploy an application in a Kubernetes cluster, we create a *Deployment* object representing the application deployed in the cluster. Besides, instead of deploying individual containers, we deploy groups of co-located containers - so-called *pods*. A pod is a group of one or more closely related containers that run together on the same worker node and need to share certain Linux namespaces.

To bring intelligence to the low-level orchestrator in the edge-cloud continuum, we enable the autonomous and adaptive workload auto-scaling on the low-level Kubernetes platforms. A common workload auto-scaling technique is horizontal auto-scaling, which already exists in Kubernetes and allows applications to decrease or increase the number of replicas. This is a powerful feature that enables the system to automatically adapt its resource allocation based on real traffic. However, if the limits are not set correctly, the average utilisation might grow

the application in a non-optimal way. Instead, we could keep more resources powered down and gain a lot in the system's energy consumption. Hence, another technique to address the adaptation of workload is *vertical auto scaling*, which enables the automated setting of limits for each replica.

The workload runs inside a container located in a pod, with the assumption that each pod contains exactly one container. A deployment manages several homogeneous pods (i.e., related containers). The goal is to implement a ***vertical auto-scaler*** that collects historical resource usage data, predicts a suitable container size, and applies this prediction across all containers in the deployment. The auto-scaled resources in this work are CPU, RAM, and GPU.

The vertical auto-scaling problem in this context has three key aspects:

- ***Auto-scaler Input:*** The input consists of stable, real-time collected and aggregated telemetry data from the Kubernetes cluster. We specifically focus on historical actual usage metrics for CPU, RAM, and GPU, referred to as *workload usage*. Other potential inputs, such as container size history or workload scheduling information, are intentionally excluded.
- ***Auto-scaler Prediction Algorithm:*** The core ML-based algorithm that predicts future resource usage and estimates appropriate container request and limit values. The prediction should not exceed the (future) workload usage too much, as overestimations can lead to resource waste. It should also not be underestimated as this may disrupt workload execution and even cause service level objective (SLO) violations. The aim is for the ML algorithm to outperform traditional rule-based algorithms, such as threshold- or heuristic-based auto-scalers.
- ***Auto-scaler Output:*** The output includes both the predicted resource requests and limits, given by the core prediction algorithm, as well as the mechanism to gracefully apply them across all containers in the deployment. Specifically, we need to properly configure the container resource request and limit settings according to the algorithm's prediction, followed by resource fractioning and assignment of the appropriate fractioned slice to each container. Two main challenges arise: (i) GPU resources fractioning, which lacks native Kubernetes support (unlike CPU and RAM) and (ii) ensuring performance stability across the deployment when applying auto-scaling on containers.

The theoretical challenge in vertical auto-scaling is making accurate predictions for future container size predictions under a non-clairvoyant situation – without access to future workload usage patterns. As such, the auto-scaler must behave as an *online* algorithm, making real-time decisions with limited information. Although ML techniques can help to mitigate this challenge, their effective application in an online problem setting remains an issue and is the key focus of this work. To this end, the performance of the proposed auto-scaling algorithm will be thoroughly evaluated through a variety of experiments and benchmarks, demonstrating its effectiveness across CPU, RAM, and GPU scaling scenarios.

The practical challenge addressed in this work is the actual implementation of an auto-scaler in the Kubernetes cluster, with the long-term goal of deploying it in a production environment across the edge-cloud continuum. This requires the auto-scaler to adapt and optimise CPU and memory consumption based on different edge or cloud devices. It also requires addressing differences between the theoretical model and real environment, handling cases that are not well considered in the theories (e.g. dealing with Out-Of-Memory Kills), and having enough robustness to handle diverse workloads and the edge-cloud environment.

The first version of our efforts focusses on a vertical auto-scaler inspired by Rzadca's Autopilot algorithm[22]. We introduce several theoretical refinements that include alignments for improving both algorithm's accuracy and efficiency, including as well as a RAM post-processor specifically designed to successfully addresses the Out-Of-Memory (OOM) kills, which were not adequately considered in the original work.

Building on this initial work, we then implemented VPA-pilot that is an actual auto-scaler built as service upon Kubernetes and based on an open-source framework. Special attention was given to edge-cloud adaptability by optimizing the CPU and RAM consumption of the Autopilot ML implementation, based on complexity analysis. The implementation demonstrates low overhead, even when running with high scale and precision (e.g., 20000 sub-models). Algorithm's resource usage remains minimal (12.441 millicore CPU and 17.898MB RAM), which is not even dominate the consumption of the auto-scaler framework's system logic.

To further enhance performance, we developed a long-term auto-scaler simulator along with appropriate methodologies to tune the VPA-pilot's hyper-parameters. The hyper-parameters tuning problem is modelled as an Operations Research (OR) program. Using combinations of sampling- and manual-based methods, we solved the OR program. The dominating set of hyper-parameters is successfully found on CPU resources. However, the RAM tuning presented suboptimal performance, suggesting the need for future improvements on the hyper-parameters tuning model, which are also presented.

Recognizing the growing importance of GPU acceleration in edge-cloud applications, we extend our methodology to include dynamic GPU fractioning and vertical auto-scaling using Multi-Instance GPU (MIG) technology. By leveraging our ML-based approach, we demonstrate a practical pathway for enabling GPU-aware vertical scaling in Kubernetes environments that currently lack native support for dynamic GPU partitioning.

[22] Rzadca K, Findeisen P, Swiderski J, Zych P, Broniek P, Kusmierek J, Nowak P, Strack B, Witusowski P, Hand S, Wilkes J. Autopilot: workload autoscaling at google. InProceedings of the Fifteenth European Conference on Computer Systems 2020 Apr 15 (pp. 1-16).

7.2.1 GPU Fractioning

The purpose of fractioning is to divide the entire resources into multiple slices and allocate them to each container respectively. Good fractioning technologies should be able to limit resources for each container, and should have good isolation between different containers.

This work concerns CPU, RAM and GPU resources in Kubernetes nodes. For CPU and RAM, there are mature technologies such as Cgroups. However, Kubernetes has no native support for GPU. There is also no technology as mature and dominating as Cgroups. This section investigates 3 available GPU fractioning technologies on NVIDIA GPUs. Then compare them and choose one for GPU vertical auto-scaling in this work.

GPU has its own computing and memory resources. Here we consider the combination of *Streaming Multiprocessor (SM)* and *Global Memory (GPU Memory)*, which is similar to the CPU/RAM combination. SM is a fundamental computing component of NVIDIA GPUs that executes instructions in parallel. As for memory, GPUs have a (more complicated) hierarchical architecture like ordinary memory: L1/L2/constant cache, shared/local/global/texture and constant memory. Among these, the global memory is similar to the ordinary "memory" concept. So, for convenience, we use GPU Memory in this work to refer to the global memory.

7.2.2 Time-Slicing GPU

To access and configure GPU resources in Kubernetes, NVIDIA proposed the NVIDIA GPU Operator, which is a set of components installed in a Kubernetes cluster. Among these components, there is one called NVIDIA Kubernetes Device Plugin. This plugin implements the Time-Slicing GPU feature.

Time-Slicing GPU enables the system manager to define a set of replicas for a GPU. Each replica can be used independently by a container to run the workload. Internally, Time-Slicing is used to multiplex workloads from replicas of the same underlying GPU. Time-Slicing allows each workload to use the entire SMs of a GPU in turn, like the context switch of CPUs. The GPU Memory is split and assigned to each workload, but without any memory and fault isolation.

Time-Slicing is a bad fractioning method for auto-scaling. First, auto-scaling aims to improve resource utilization when there are multiple workloads, each of which is small compared to the entire resource. However, with Time-Slicing, each small workload still uses the whole GPU during its round. The wasting of SMs is not relieved at all (i.e., we do not simply want parallelism here). Second, isolation is important for Kubernetes containers. So, no isolation of GPU Memory in Time-Slicing is unacceptable.

7.2.3 Multi-Instance GPU (MIG)

Multi-Instance GPU (MIG) is a new feature proposed on NVIDIA GPUs starting from the Ampere architecture. MIG allows GPUs to be securely partitioned into up to 7 separate *GPU Instances (GIs)*, then assigning each GI to different workloads. Different from Time-Slicing, each GI owns a certain part of the resources (SM and GPU Memory) of the entire GPU spatially. The SMs and GPU Memory between different GIs are completely isolated and their resource usages are strictly limited, as if the workloads are running on different GPUs. In Kubernetes cluster, the NVIDIA Kubernetes Device Plugin also implemented MIG support. With this Kubernetes support, we can read the MIG information of the GPUs in the cluster, configure MIG, and assign the GIs to different containers.

Strategy	Profile	Amount of GIs	Fractioning illustration on NVIDIA H100 GPU In each row, the SM and GPU memory with the same color belongs to the same GI SM (7g total on a GPU) / GPU Memory (80GB total on a GPU)	Comment
single	1g.10GB	7	1g 1g 1g 1g 1g 1g 1g | 10GB 10GB 10GB 10GB 10GB 10GB 10GB	10GB wasted
single	1g.20GB	4	1g 1g 1g 1g | 20GB 20GB 20GB 20GB	3g wasted
single	2g.20GB	3	2g 2g 2g | 20GB 20GB 20GB	1g and 20GB wasted
single	3g.40GB	2	3g 3g | 40GB 40GB	1g wasted
single	4g.40GB	1	4g | 40GB	3g and 40GB wasted
single	7g.80GB	1	7g | 80GB	
mixed	all-balanced	3	1g 1g 2g 3g | 10GB 10GB 20GB 40GB	4GIs: 1g.10GB * 2 + 2g.20GB + 3g.40GB

Figure 12: Illustration of MIG Strategies, MIG Profiles and GIs on NVIDIA H100 GPU

For vertical auto-scaling, MIG perfectly satisfies the requirements of limiting and isolating resources. However, MIG has unignorable drawbacks in flexibility.

The sizes of GIs are not arbitrary. We cannot request an arbitrary part of the GPU as we did on CPU or RAM. With MIG, the SMs on the GPU are grouped into 7 *compute slices* of the same size. The entire GPU memory is also divided into 8 *memory slices* of the same size. These compute slices and memory slices are grouped into several *GPU Instances (GIs)*. Due to technical limits at the GPU level, the grouping of these slices is not arbitrary (less than 10 valid grouping methods for each GPU architecture)[23]. In this way above, a GPU is finally fractioned into a combination of GIs. Each valid combination is called a *MIG Profile*. The size of a GI is measured by the number of compute slices and memory slices it owns, denoted as *A*g.*B*GB if the GI owns *A* compute slices and *B* GB of memory slices.

[23] Nvidia multi-instance GPU user guide. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html

In Kubernetes cluster, the MIG support allows us to configure a MIG profile for each GPU device independently. In Kubernetes MIG support, the MIG Profiles are categorized into 2 *MIG Strategies*: *Single Strategy* if each GI in the MIG profile have the same size, or *Mixed Strategy* if they have different size. To configure MIG for a GPU in the cluster, we need first to choose the MIG Strategy, then choose a MIG Profile that belongs to it. After configuring, we can assign a GI to a container by labelling it in the manifest YAML file.

As a particular example, Figure 12 shows the valid MIG Strategies, MIG Profiles, and their combination of GIs on a Kubernetes nodes with NVIDIA H100 GPU with 80 GB GPU Memory.

Changing a container's GI size is not fully dynamic. In Kubernetes cluster with MIG fractioned GPU, switching the MIG Profile and MIG Strategy requires additional time overhead. In our experiment on a Scaleway Kubernetes node with one NVIDIA H100 GPU, switching MIG Profile takes 36-39 seconds and switching MIG Strategy takes 33-36 seconds. If considering the pod recreating time, the total time overhead is up to 70-110 seconds. During this period the pod cannot provide service, which may cause SLO violation problems in auto-scaling depending on how to deal with this overhead. Moreover, we need to ensure that all containers using this GPU are stopped before switching MIG Profile or MIG Strategy, otherwise the switching will fail. This is also a limit in auto-scaling.

We have 3 ways to change the size of a container's current GI: 1. Reallocating another GI in a fixed MIG Profile under Mixed Strategy. 2. Switching the MIG Profile under Single Strategy. 3. Switching MIG Strategy between Single and Mixed. The second and third ways both have the switching overhead. Only the first way can avoid this. However, in the vertical auto-scaling context, this first way has other problems with resource utilization. This will be discussed in follow-up subsection.

7.2.4 GPU Multi-Process Service (MPS)

Multi-Process Service (MPS) is a CUDA API binary-compatible runtime implementation, allowing multiple CUDA kernels to be concurrently on the same GPU [14]. MPS is a client server architecture. Each user owns a single MPS client attached to the user's CPU process. This MPS client submits the task and its CUDA context to the MPS server. The MPS server combines the contexts of the received client tasks and lets them run on the GPU as a single ap plication, to reach higher GPU utilization. Starting from Volta architecture, the MPS server is no longer a separate component. Its functionality is taken by GPU hardware and MPS clients.

MPS is designed mainly for multi-process collaboration programs like Message Passing Interface (MPI). However, MPS can also be used in Kubernetes to fraction GPU. A third-party fork of the NVIDIA Kubernetes Device Plugin implemented MPS support in Kubernetes. After replacing the official Kubernetes Device Plugin to this fork in the cluster, we can configure MPS and assign fractioned GPU slices to containers by manifest YAML files. Thanks to MPS server's functionality, MPS can fraction the SMs spatially, different from context-switching in Time-Slicing. We can arbitrarily create the limit on maximum SM usage for each container(task).

Regarding GPU Memory, we can also arbitrarily limit each container(task)'s allocatable GPU Memory size. Each container has isolated address space. However, the GPU Memory access is not fully isolated. An out-of-range write in a CUDA Kernel can modify the CUDA accessible memory state of another process. In the experiment on Kubernetes node with NVIDIA H100 GPU, a container knows the existing workloads in other containers, and also the GPU Memory size of other containers. The fault isolation is also bad. A fatal GPU fault generated by an MPS client process will be shared with some of the other clients on the same GPU. These bad isolations are unacceptable in vertical auto-scaling.

7.2.5 Conclusions on GPUs Fractioning

Regarding the choice of this work, as presented previously, our goal is to guarantee a stable auto-scaler implementation rather than chasing for top performance. We first discard Time-Slicing GPU, as discussed in comments. Then as for MPS, we appreciate its flexibility to arbitrarily slice GPU, as how Cgroups manages CPU and RAM. However, poor isolation is unacceptable for a Kubernetes implementation. Although there are safe isolation methods based on CUDA logic or practical analysis, they are not open-source. Studying and implementing these novel techniques from scratch is not a stable choice. Therefore, we discard MPS as our GPU fractioning implementation.

MIG is a very safe and stable choice for our goal. To fraction GPU using MIG, we need to choose a specific way to change the container's GI size:

Using Mixed Strategy is the most flexible, as we only need to reallocate another GI under the same MIG Profile. However, if the sizes of workloads are not uniformly distributed, Mixed Strategy can lead to severe resource waste. For example, if each NVIDIA H100 GPU in the cluster is configured as the Mixed Strategy in Figure 12, and all workloads are 1g.10GB in size. Then the 2g.20GB and 3g.40GB GIs will all remain unused. This will waste 71.4% of the SMs and 75% GPU memory. Unfortunately, we cannot guarantee the uniform distribution of workloads. Thus, the Mixed Strategy has to be aborted.

Alternatively, by using the Single Strategy for all GPUs in the cluster while dynamically configuring the MIG profile on each GPU, we can adapt to any distribution of workload sizes and avoid resource waste. For example, when there are more 1g.10GB workloads, we can configure more free H100 GPUs as 1g.10GB MIG Profile. Regarding the Profile switching overhead (up to 70-110s), we can reduce the frequency of switching to lower the total overhead. Besides, the auto-scaler will be integrated into the Ryax platform[24], which provides higher-level scheduling information. We can schedule tasks and prepare for the switching in advance, which can mitigate the service interruption caused by switching. Overall, we choose MIG technology with dynamically configured MIG Profiles under Single Strategy for GPU fractioning in this work.

[24] https://github.com/RyaxTech/ryax-engine

7.3 Theoretical Aspects

This section presents the main theoretical contributions of our vertical auto-scaler in this work. The contributions include refining the auto-scaler's core recommendation algorithm, and designing the dynamic GPU fractioning algorithm on the auto-scaler output side.

About the core recommendation algorithm this work chooses the Autopilot ML algorithm to implement and refine. In this section, we first briefly recall the contributions of the Autopilot paper, including the ML model that we will refine, and a rule-based algorithm that will be used as a baseline in performance evaluation. Then, we present the refined ML model in detail which is named *VPA-pilot*.

About the dynamic GPU fractioning algorithm as the auto-scaler's output, this work chooses MIG with dynamically configured Profiles under Single Strategy. In the follow-up sections we will present the theoretical details of our dynamic GPU fractioning algorithm based on the MIG and VPA-pilot model.

7.3.1 Recall the autopilot paper

Rzadca's Autopilot paper presents an ML and a rule-based vertical auto-scaling algorithm that inputs CPU and RAM usage data and outputs the corresponding resource limit.

7.3.1.1 Autopilot ML

This subsection recalls the Autopilot ML algorithm(model). In Rzadca's original work, some descriptions and formulas are unclear. So, we provide some additions and modifications, e.g. Equation 3.1 and 3.2 in the model's input, Equation 3.6, 3.7 and 3.8 in the model selector.

In Autopilot ML, the workloads are called *tasks*. Several concurrent tasks are grouped as a *job*. Autopilot ML takes task-level usage data and outputs resource limit recommendations at the job level using an ML algorithm.

We first model the input of Autopilot ML. The resource usage data of task i at time τ is denoted as $u_i[\tau]$. For each resource of each task, the frequency of $u_i[\tau]$ is 1 per second. For CPU, $u_i[\tau]$ is measured in cores, denoted as $u_i^{CPU}[\tau]$. For RAM, $u_i[\tau]$, is measured in bytes and denoted as $u_i^{RAM}[\tau]$.

Then for each task, every 5 minutes we aggregate per resource type $u_i[\tau]$ into one histogram $s_i[t]$, where t is the 5-minute window and $\tau \in t$. A histogram $s_i[t]$ has K buckets, and the upper bound of each bucket is $b[k]$, $k \in \{1...K\}$. The histogram structure is identical for each resource type but varies between different resource types, i.e. K^{CPU} buckets and $b^{CPU}[k]$, $k \in \{1...K^{CPU}\}$ upper bounds for a CPU histogram, K^{RAM} buckets and $b^{RAM}[k]$, $k \in \{1...K^{RAM}\}$ upper bounds for a RAM histogram.

The CPU histogram is aggregated as Equation 3.1 below. Each bucket contains the number of usage data points that fall into this bucket.

$$s_i^{CPU}[t][k] = |\{u_i[\tau] : \tau \in t \wedge b[k-1] \leq u_i[\tau] \leq b[k]\}| \quad (3.1)$$

The RAM histogram only records the peak usage, as equation 3.2 below, because we usually want to provision for close to the peak RAM usage.

$$s_i^{RAM}[t][k] = \begin{cases} 1 & if\ b[k-1] \leq max\{u_i[\tau] \mid \tau \in t\} < b[k] \\ 0 & otherwise \end{cases} \quad (3.2)$$

Right after each aggregation, the task-level histograms are merged into job-level ones:

$$s[t][k] = \sum_i s_i[t][k] \quad (3.3)$$

This merged histogram serves as the actual input of the Autopilot ML model. Now we describe the core model of Autopilot ML. As mentioned in the state of the art described in D2.1 section 3.1.2.2.3, Autopilot ML is a Hierarchical (HMS) model containing multiple sub-models and a global model selector. In Autopilot ML, each sub-model *m* is an *argmin* function that outputs a resource limit value $L_m[t]$ at aggregation window t given the historical usage $s[t]$, parameterized by a decay rate d_m and a safety margin M_m.

In detail, inside each sub-model, every possible limit value *L* is evaluated. An overrun cost and an underrun cost that counts the number of data points in buckets above/below the limit L are calculated as Equation 3.4 below.

$$o(L)[t] = (1 - d_m)(o(L)[t-1]) + d_m\left(\sum_{j:b[j]>L} s[t][j]\right)$$
$$u(L)[t] = (1 - d_m)(u(L)[t-1]) + d_m\left(\sum_{j:b[j]>L} s[t][j]\right) \quad (3.4)$$

Then the sub-model chooses an L value that minimizes a function consisting of overrun and underrun cost above, and a limit switching cost, as shown in Equation 3.5 below. The limit switching cost aims to avoid frequent limit changing, because in auto-scaler implementation, each changing causes an eviction and corresponding workload restarting. Finally, the sub-model specific safety margin Mm is added to the argmin function output.

$$L'_m[t] = \arg min_L \left(w_o o(L)[t] + w_u u(L)[t] + w_{\Delta L}\Delta(L, L'_m[t-1])\right)$$
$$L_m[t] = L'_m[t] + M_m \quad (3.5)$$

where $\Delta(x, y) = 1$ if x ≠ y and 0 otherwise. $w_o, w_u, w_{\Delta L}$ are hyper-parameters representing the weights of each cost.

The global model selector holds a cost function for each sub-model, and an argmin function that dynamically selects the best sub-model based on their cost functions. This per sub-model cost function is shown in Equation 3.6 below.

$$c_m[t] = d(w_o o_m(L_m[t],t) + w_u u_m(L_m[t],t) + w_{\Delta L}\Delta(L_m[t],L_m[t-1])) +$$

$$(1-d)c_m[t-1] \quad (3.6)$$

where $o_m(L_m[t],t) = \sum_{j:b[j]>Lm[t]} s[t][j]$ and $u_m(L_m[t],t) = \sum_{j:b[j]<Lm[t]} s[t][j]$. w_o, w_u, $w_{\Delta L}$ are the same hyper-parameters as Equation 3.5, which are consistent across all sub-models. *d* is another hyper-parameter representing the decaying weight of the history cost.

This cost function is built based on the assumption that both the recent past workload usage and the scaling method performance are likely to represent the near future: hence using recent past statistics to represent the near future. So here the overrun/underrun cost of each sub-model's prediction is evaluated on current usage data.

Then the best sub-model *m[t]* at the current aggregation window t is selected by:

$$m[t] = \arg min_m \left(c_m[t] + w_{\Delta m}\Delta(m[t-1],m) + w_{\Delta L}\Delta(L[t-1],L_m[t])\right) \quad (3.7)$$

Where $w_{\Delta m}$ is another hyper-parameter that weighs how much we should avoid frequent sub model changing: frequently switching sub-models causes more SLO violations.

Finally, the limit value given by the best sub-model is used as the output of the entire Autopilot ML auto-scaler:

$$L[t] = L_{m[t]}[t] \quad (3.8)$$

7.3.1.2 Autopilot rule-based: a baseline

This subsection recalls the rule-based auto-scaler in Rzadca's work, which will be used as a baseline in the following sections. This auto-scaler gives resource limit recommendations based on statistics on the same input histogram of Autopilot ML, i.e. s[t][k] in Equation 3.3.

For CPU resources, a raw recommendation is calculated by the 90%ile of an adjusted usage histogram. The histogram is defined as:

$$h[t][k] = b[k] \cdot \sum_{\tau=0}^{\infty} w[t] \cdot s[t-\tau][k] \quad (3.9)$$

where *w[τ]* is a decaying weight at time *τ*, defined as:

$$w[\tau] = 2^{\frac{\tau}{t_{1/2}}} \quad (3.10)$$

where $t_{1/2}$ is a config parameter representing the decaying half-life.

The raw recommendation value at current time *t* is: $S_{p90}[t] = P_{90}(h[t])$.

For RAM resources, we use the max of recent input samples as the raw:

$$S_{max}[t] = max_{\tau\in\{t-(N-1)\ldots t\}} \{b[j]: s[\tau][j] > 0\} \quad (3.11)$$

Finally, a 10-15 safety margin is added to the raw recommendations above, and we use the maximum margined recommendation value over the last hour as the final output of this rule-based auto-scaler.

7.3.2 VPA-pilot

In the context of EMPYREAN, we improved the Autopilot ML described previously to propose VPA-pilot as our vertical auto-scaler's core recommendation algorithm. VPA-pilot takes the resources usage of the workloads running in each container of the deployment as input, and predicts the corresponding resource request and limit for these containers. This section presents the theoretical model of VPA-pilot.

VPA-pilot includes two major enhancements. The first is aligning the recommended request values with bucket bounds thus calculating on array indexes (i.e. integers instead of floats), to reduce computation and increase accuracy. The second is introducing a post-processor for RAM resources to address the Out-of-Memory Kill (OOM Kill) issue that was not well handled in Autopilot paper. The following subsections detail the process of the whole model while describing these 2 enhancements respectively.

In addition to major enhancements, this work also performs several minor enhancements, adaptations to our requirements, and clarifications of Rzadca's Autopilot work. These will not be presented in dedicated subsections but within the following 2 major enhancements subsections as they arise.

7.3.2.1 Alignment: more lightweight, efficient, and accurate

Although Autopilot ML is well designed, from the implementation perspective, Autopilot ML still has drawbacks on its efficiency and accuracy:

- ***Floating-point errors***: The function $\Delta(x, y)$ is frequently applied to floating-point numbers, e.g. in Equation 3.5, 3.6, 3.7, the floating-point limit values L serve as inputs to $\Delta(x, y)$. This causes frequent equality comparisons between floating-point numbers. Due to floating-point errors, directly comparing the equality of floating-point numbers is inaccurate. An alternative is to consider the floating-point numbers equal if their difference is less than a small threshold ε. However, if the model frequently calls this alternative, the additional statements involved can be a bottleneck, significantly impacting the model's efficiency (i.e. speed, CPU usage). Therefore, we are considering whether we can replace floating-point values with integer values while keeping the same mathematical meaning.
- ***Redundancy on the limit L variable:*** From Equations 3.4 and 3.5, we know that the optimization variable L influences the sub-model's output $L_m[t]$ only through its comparison with the bucket bounds $b[k]$, $k \in \{1...K\}$, which are finite number of elements. For the infinite amount of possible real numbers $L : b[k] < L < b[k+1]$, the optimal value of the argmin function in Equation 3.5 is the same. Thus, although evaluating L as every possible real-number value seems to increase the model's

precision, the $L : b[k] < L < b[k+1]$ values are completely unuseful, so this precision increase is actually not realized. Therefore, we can just align L to the bound of buckets, i.e. $L \in \{b[k]\}, k \in \{1...K\}$.

- ***Invalid penalties on switching limits***: In Equations 3.5, 3.6 and 3.7, the penalties of switching limits are all calculated by $\Delta(x, y)$ with 2 real numbers. In the infinite real numbers set, achieving equality between two elements is difficult, even allowing for a threshold ε. Therefore, these penalties all become invalid, which leads to frequent switching of recommended limit value, and finally causes SLO violations. In Equation 3.5, by aligning L to bucket bounds as described above, the penalty $\Delta(L, L'_m[t-1])$ becomes valid again, and the value $L'_m[t]$ is also aligned to the bucket bounds. Then to ensure the penalties in Equations 3.6 and 3.7 are valid, any $L_m[t]$ must also be aligned to bucket bounds. Known that $L'_m[t]$ is already aligned, aligning $L_m[t]$ requires: (1) the histogram should be linear, i.e. $b[k] = k \cdot b[1]$, and (2) the safety margin of each sub model M_m should align with bucket bounds, i.e. $M_m = b[m_m],\ m_m \in \{1...K\}$. While these requirements seem to cause a precision loss, this is necessary for the model to function properly. Moreover, if the number of buckets is large enough, this precision loss is negligible.

Based on the analysis above, we use linear histogram (i.e. $b[k] = k \cdot b[1],\ k \in \{1...K\}$), and align L and M_m to histogram bucket bounds $b[k]$ in VPA-pilot's design. We also noticed that before the global model selector outputs the final recommendation, all limit values (L, $L_m[t]$, etc.) are only used to compare with other limits or the bucket bounds. Besides, the linear histogram has monotonic increasing property, i.e. for any $k_1 < k_2$, $b[k_1] < b[k_2]$. Therefore, before the final output, we can all use the index of the bucket bounds as the unit and perform integer calculations and comparisons, instead of calculating the exact floating-point values at the very beginning and then performing floating-point comparisons. This perfectly solves the floating-point errors. During implementation, this can also save the model's RAM consumption, and some CPU on specific architectures.

Based on these designs, we now present the VPA-pilot's theoretical model for auto-scaling CPU and RAM resources (GPU will be present together with MIG in following section). On the input side, the *linear* histogram $s[t][k]$ in Equation 3.3 is used as the model's input at the aggregation window t. Its aggregation methods are the same as Equations 3.1 and 3.2. Because in our problem context we have the deployment and containers(pods) instead of the job and tasks, so in these equations, $s_i[t][k]$ represents the histogram on the container's level, and $s[t][k]$ is the histogram on the deployment's level. The bucket amount K and per bucket size $b[1]$ in the linear histogram are set as follows: We use 400 buckets for CPU resource (K^{CPU} = 400), and 500 buckets for RAM (K^{CPU} = 500), because the number of RAM is larger and requires higher precision for effective fractioning. Assuming a node's total amount of current scaled resource is B and the number of pods(containers) in the deployment is C, then the size of each linear histogram bucket $b[1] = \frac{B}{K \cdot C}$ this is because containers in the deployment exist concurrently, each container uses no more than B/C resources.

Different from Rzadca's Autopilot work, VPA-pilot recommends the containers' resource requests instead of limits. Thus, in the sub-models of VPA-pilot, we need to evaluate every possible request value $R \in \{b[k]\}$, $k \in \{1...K\}$. Then based on the previous paragraph's discussion, we set r as the index of the bucket whose bound is aligned by R, i.e. $R = b[r]$, $r \in \{1...K\}$. We iterate on all possible r values instead of R.

The sub-models of VPA-pilot are parameterized by the decay rate $d_m \in [0,1]$ and the index m_m of a bucket whose bound is aligned by the safety margin M_m, i.e. $M_m = b[m_m]$ as discussed in "invalid penalties on switching limits" above. To ensure that the sub-models cover all possible scenarios as thoroughly and representatively as possible, we uniformly sample N_{dm} values for d_m from its interval [0,1], and select m_m from N_{mm} possible integers starting from 0: $\{0,...,N_{mm}-1\}$. By pairing each d_m and each m_m, we have $N_{dm} \cdot N_{mm}$ sub-models in VPA-pilot.

In each sub-model, inspired by Equation 3.4, we calculate the overrun and underrun cost by:

$$o(r)[t] = (1 - d_m)(o(r)[t-1]) + d_m\left(\sum_{j:j>r} s[t][j]\right)$$
$$u(r)[t] = (1 - d_m)(u(r)[t-1]) + d_m\left(\sum_{j:j<r} s[t][j]\right) \quad (3.12)$$

Then each sub-model outputs the index $r_m[t]$ of a bucket whose bound is aligned with the recommended resource request $R_m[t]$, i.e. $R_m[t] = b[r_m[t]]$. Inspired by Equation 3.5, the sub model's recommendation (bucket) index is calculated by:

$$r'_m[t] = \arg min_r \left(w_o o(r)[t] + w_u u(r)[t] + w_{\Delta R}\Delta(r, r'_m[t-1])\right)$$
$$r_m[t] = \min(r'_m[t] + m_m, K) \quad (3.13)$$

where $\Delta(x, y) = 1$ if $x \neq y$ and 0 otherwise. $w_o, w_u, w_{\Delta R}$ are hyper-parameters. The min function prevents the recommendation index from exceeding the histogram's upper bound so that the real resource request value can be retrieved using this index in the future.

The global model selector of VPA-pilot maintains a cost function for each sub model. This function $c_m[t]$ is inspired by Equation 3.6 while using the comparison between bucket indexes instead of limit values:

$$c_m[t] = d\left(w_o\left(\sum_{j:j>r_m[t]} s[t][j]\right)\right) + w_u\left(\sum_{j:j<r_m[t]} s[t][j]\right) + w_{\Delta R}\Delta(r_m[t], r_m[t-1]) + (1-d)c_m[t-1] \quad (3.14)$$

where d is another hyper-parameter.

Then the global model selector iterates all the sub-models m and chooses the best sub-model $m[t]$ at aggregation window t according to this argmin function (inspired by Equation 3.7):

$$m[t] = \arg min_m \left(c_m[t] + w_{\Delta m}\Delta(m[t-1], m) + w_{\Delta R}\Delta(r[t-1], r_m[t])\right) \quad (3.15)$$

where $w_{\Delta m}$ is another hyper-parameter.

Finally, the global model selector takes the recommended bucket index $r[t]$ given by the selected sub-model, and calculates its corresponding bucket bound value as the resource request recommendation $R[t]$ of the whole algorithm:

$$r[t] = r_{m[t]}[t]$$
$$R[t] = b[r[t]] \quad (3.16)$$

7.3.2.2 Resource Limit and the RAM Post-Processor for OOM Kill

In the Kubernetes cluster, a container has both request and limit values for each of its CPU and RAM resources. Previously, we enabled the VPA-pilot algorithm to recommend the request values. Now we decide the resource limit values for CPU and RAM by discussing the underlying mechanisms of request and limit values in Kubernetes.

CPU limits

The CPU request is implemented using the *cpu_shares* field in Cgroups[25]. If all containers use CPU simultaneously, the CPU time is allocated to each container based on the proportion of their *cpu_shares* values. When some containers are idle, the

CPU time is allocated proportionally among the remaining active containers. This mechanism prevents the waste of CPU time while ensuring the lower bound of CPU time for each container. The CPU request is also a basis for Kubernetes to schedule each pod placed on which node, instead of the CPU limit. Therefore, the request is more important than the limit for the workload bin-packing performance of our auto-scaler.

The CPU limit is implemented using the ratio of fields *cpu_quota* to *cpu_period* in Cgroups. Within the time of *cpu_period*, if a container uses more CPU time than its *cpu_quota* value, it will enter the *CPU throttling* state. During CPU throttling, the container cannot use CPU until the next *cpu_period*. This mechanism effectively limits the upper bound of CPU usage for containers but can cause significant execution delays of the workload inside the container.

In our context, the CPU request value is properly set by the VPA-pilot algorithm. In this case, setting the CPU limit brings no advantage but only the CPU throttling drawbacks. This is because the sharing mechanism behind the CPU request and the node's Linux kernel already can efficiently handle CPU bursts without too much work load delay, e.g. the CPU Burst feature in Linux kernel from 5.14. Even if the user workloads' CPU bursts are excessive, causing insufficient CPU time for the system's daemon services, we can move the daemon services to another node, or employ additional recovery mechanisms to temporarily set limits to prevent system crashes.

Overall, we decided to set no CPU limit in VPA-pilot's output.

[25] The fields mentioned here all refer to Cgroups V1. Our experimental environment Scaleway Kubernetes cluster uses a new version: Cgroups V2. The fields in Cgroups V2 are more complex than in Cgroups V1, but the underlying mechanisms are very similar.

RAM limits

The RAM request is mainly used as an indication for the Kubernetes scheduler to determine each pod placed on which node. There are no mechanisms related to the RAM request in Cgroups (V1), so Kubernetes does not intervene in the container's actual RAM usage if only the RAM request is set.

The RAM limit is implemented by the *limit_in_bytes* field in Cgroups. If the container tries to allocate more memory than the RAM limit, the Linux kernel out-of-memory subsystem is activated, to intervene by stopping one of the processes in the container that tried to allocate memory. This is called *Out-Of-Memory Kill (OOM Kill)*. After this, an OOM Killed event is raised and the pod is evicted.

In our auto-scaling context, if we do not set the RAM limit or set it higher than the RAM request, a container might use more RAM resources than Kubernetes is aware of. If many containers do this at the same time, the RAM of the node's Linux system will be exhausted before Kubernetes realizes that the node's available RAM is full and prevents it. This can lead to hidden system crashes in the cluster that are difficult to diagnose from Kubernetes. Therefore, we decided to set the RAM limit equal to the RAM request in VPA-pilot's output.

In the vertical auto-scaling problem, the consequences of the container's CPU or RAM usage exceeding the corresponding limit are entirely different. We assume that the workload in the container is deterministic (i.e. denote the state of the workload at time t as $\sigma(t)$, if $\sigma(t')$ executed successfully but $\sigma(t'+1)$ failed, the workload will retry exactly with state $\sigma(t'+1)$ rather than any other state). If the CPU usage exceeds the limit, the workload enters CPU throttling but the container remains running. In this case, the auto-scaler can still collect this extremely high CPU usage data and use this data to enhance future container size. However, if the RAM usage exceeds the limit, an OOM Kill will be triggered and the container (corresponding pod) will be evicted. In this case, the auto-scaler cannot obtain the corresponding extremely high RAM usage and will recreate the pod with the original size. Then the deterministic workload will use RAM exceeding the limit again. Thus, a dead loop is formed here. The left of Figure 13 illustrates this RAM exceeding dead loop.

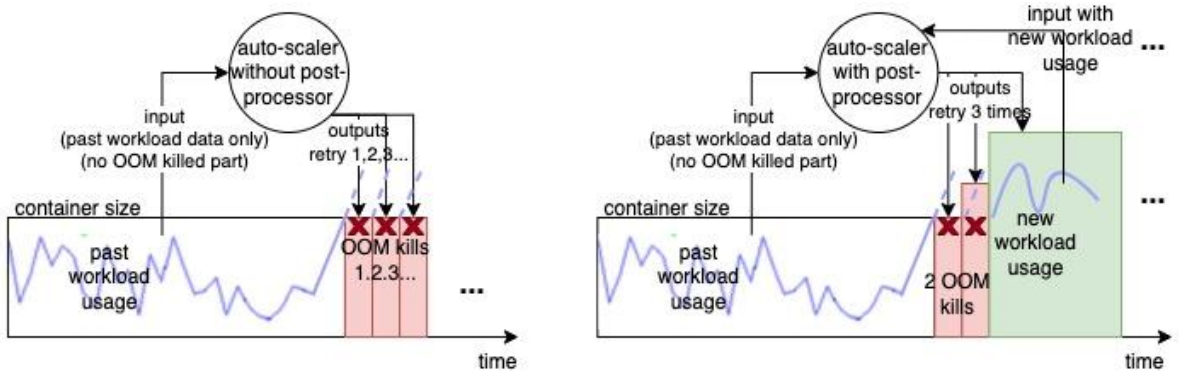

Figure 13: Illustration of the auto-scaler's behaviour under a deterministic workload, without(left) or with(right) the RAM post-processor. The purple dashed line means the workload fails to execute at that time. The red box with an 'X' indicates the container is immediately OOM Killed. The green box indicates the container is successfully created and is running.

In the RAM exceeding situation with a deterministic workload, any auto-scaling algorithm that only relies on workload usage as input can do nothing, including the VPA-pilot algorithm described previously. Therefore, VPA-pilot requires a RAM post processor that is triggered by OOM Kill events and outputs the new request and limit values after considering the OOM Kill event.

Now we describe the RAM post-processor designed in this work. The RAM post-processor contains a configuration parameter *b,b* > 1 representing the bump-up ratio, and maintains a state value $PR[t]_i$ represents the post-processed request value after considering the i^{th} OOM Kill event during the current 5-minute aggregation window t. At the beginning of each aggregation window t, this value is (re-)initialized as the current request recommendation given by the algorithm in Alignment subsection:

$$PR[t]_0 = R[t] \quad (3.17)$$

Then, if an i^{th} OOM kill event occurs within this aggregation window, the post-processor calculates the new state value as follows:

$$PR[t]_i = PR[t]_{i-1} \cdot b \quad (3.18)$$

This state value $PR[t]_i$ is also the output of the RAM post-processor in real-time. The right of Figure 13 illustrates how this RAM post-processor handles the RAM exceeding situation with a deterministic workload. When the coming deterministic workload usage $u[t]$ is extremely high, after j consecutive restarts (OOM Kills), from Equations 3.17 and 3.18 we derive the current RAM request value $PR[t]_j = R[t] \cdot b^j$.

Therefore, after $\left\lceil log_b(\frac{u[t]}{R[t]}) \right\rceil$ restarts, the container will have a suitable RAM request (and limit) to let the workload run without OOM Kill. Then this high-usage data can be successfully passed into the Refined Autopilot ML algorithm, to get its suitable recommendation as it does for CPU resources. After this, the post-processor finishes handling this OOM Kill, and then reset at the beginning of the next aggregation window.

The above derivation demonstrates the performance of this post-processor under the most challenging and representative deterministic workload scenario. This proves that the RAM post-processor is sufficient to handle any OOM Kill issues.

Finally, combining the post-processor with previous discussions about resource limits, we summarize the output of the entire VPA-pilot algorithm on the container's CPU and RAM resources:

- For CPU resources, the recommended container's request at aggregation window *t* is $R[t]$, and no limit is set to the container.
- For RAM resources, if there are already *i* OOM Kills in the current aggregation window *t*, then both the recommended container's request and limit values are $PR[t]_i$.

7.3.2.3 GPU auto-scaling with VPA-pilot

In this section, we present the GPU resource vertical auto-scaling model, based on VPA-pilot and MIG technology.

We start by modelling the dynamic GPU fractioning method to decide the object to be managed by the auto-scaler. As discussed previously, we fraction the GPU using MIG technology with dynamically configured MIG Profiles under Single Strategy. Therefore, in our model, a GPU can dynamically adopt one of the *P* pre-defined MIG Profiles. In each MIG profile, a GPU is divided into n_p^{GI} GIs with the same size. We model a fractioned GPU using an element in an *(increasingly) ordered* set of Profiles, where each Profile is denoted with a key-value pair: The key is the GI type, and the value is the number of GIs in the Profile.

$$GPU \in Profiles = \{ GI_p, n_p^{GI} \}, p \in \{1 \dots P\} \quad (3.19)$$

A GI type is further denoted as a combination of its SM and GPU Memory size:

$$GI_p = A_p g.B_p GB,\ p \in \{1 \dots P\} \quad (3.20)$$

Notice that the Profiles set is a pre-defined constant based on analysis of the specific GPU's MIG configuration, and the elements (K-V pairs) in the set are *indexed in increasing order* of their GI's SM and GPU Memory size. For example, on the NVIDIA H100 80G GPU illustrated in Figure 2.6, there are 6 different MIG Profiles under Single Strategy. Among these, the 4g.40GB one is strictly worse than the 7g.80GB one because it wastes more space to get the same number of fractions, so we discard the 4g.40GB GI.

Therefore, we set the valid Profiles number *P* = 5, and the sorted elements in the Profiles set are:

$$(GI_1 = 1g.10GB,\ n^{GI}{}_1 = 7),\ (GI_2 = 1g.20GB,\ n^{GI}{}_2 = 4),\ (GI_3 = 2g.20GB,\ n^{GI}{}_3 = 3),$$

$$(GI_4 = 3g.40GB,\ n^{GI}{}_4 = 2),\ (GI_5 = 7g.80GB,\ n^{GI}{}_5 = 1)$$

Based on the dynamic GPU fractioning settings above, we design the GPU vertical auto-scaling model that adopts VPA-pilot, taking the workload's GPU usage as input to recommend a suitable GI type for the container. The structure of our designed GPU auto-scaler is shown in Figure 14. The GPU usage data of SM and GPU Memory resources are processed separately by 2 VPA-pilot models without post-processor (the part in subsection 7.3.2.1 that outputs $R[t]$, without post-processing and limit-setting parts in subsection 7.3.2.2), to get SM and GPU Memory raw recommendation values. Then, these 2 raw recommendations enter the combiner to get a suitable GI type that can accommodate the recommended SM and GPU Memory size. GPU Memory OOM events are processed finally to get the final GI type recommendation. In the next paragraphs, we formally model the details of the GPU auto-scaler.

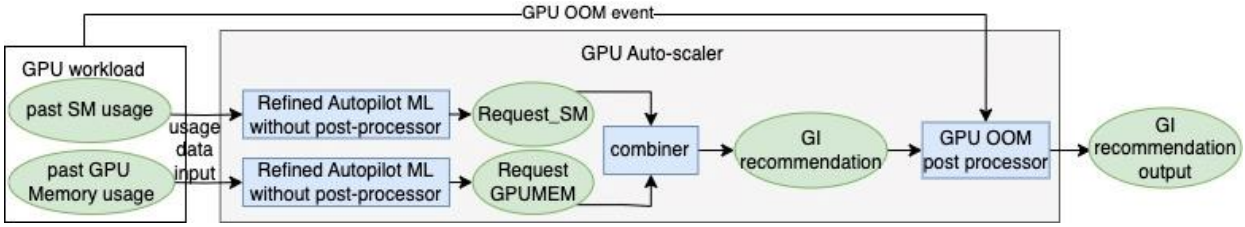

Figure 14: Illustration of the GPU auto-scaler components

On the usage data input side, at every second τ we collect SM usage data $u_i^{GPUSM}[\tau]$ and GPU Memory data usage data $u_i^{GPUMEM}[\tau]$ for each container *i*. Then, at every 5-minute window t, $u_i^{GPUSM}[\tau]$ are aggregated into histogram $s_i^{GPUSM}[t]$ with the same method as CPU in Equation 3.1, $u_i^{GPUMEM}[\tau]$ are aggregated into histogram $s_i^{GPUMEM}[t]$ as RAM in Equation 3.2. The workload level histograms $s^{GPUSM}[t][k]$, $s^{GPUMEM}[t][k]$ are calculated the same way as in Equation 3.3.

As for the bucket size *b*[1] and bucket number *K* in the histogram (remind: the bucket bounds are also the set of possible model recommendations), we first need to ensure a linear histogram. Then since our auto-scaling units are the GIs, making the recommendation granularity smaller than the GI sizes is meaningless. Therefore, for SM or GPU Memory resources, we set its histogram bucket size equal to the *greatest common divisor (gcd)* of the corresponding resource size values of all GIs. Then, the number of buckets should ensure the histogram covers the resource size of the maximum GI.

The Equation 3.21 below formalizes these bucket settings.

$$b^{GPUSM}[1] = \gcd\{A_p\}, p \in \{1 \dots P\},\ K^{GPUSM} = \frac{\max\{A_p\}}{b^{GPUSM}[1]}$$

$$b^{GPUMEM}[1] = \gcd\{B_p\}, p \in \{1 \dots P\},\ K^{GPUMEM} = \frac{\max\{B_p\}}{b^{GPUMEM}[1]} \quad (3.21)$$

Then, two "VPA-pilot without post-processor" models take histograms $s^{GPUSM}[t][k]$, $s^{GPUMEM}[t][k]$ respectively, and output raw resource re quest recommendations $R^{GPUSM}[t]$ and $R^{GPUMEM}[t]$ at every aggregation window *t*. The model remains unchanged except for fine-tuning the hyper-parameters.

Next, a combiner takes the $R^{GPUSM}[t]$ and $R^{GPUMEM}[t]$ as input. The combiner scans all types of GIs in the Profiles set in Equation 3.19 in increasing order, and chooses the smallest GI that can accommodate both $R^{GPUSM}[t]$ and $R^{GPUMEM}[t]$ as its output $GI[t]$ at the window *t*:

$$r^{GI}[t] = min_{p\in\{1\dots P\}}\{p \mid A_p \geq R^{GPUSM}[t] \wedge B_p \geq R^{GPUMEM}[t]\}$$

$$R^{GI}[t] = GI_{r^{GI}[t]} \quad (3.22)$$

Similar to RAM, when a GPU Memory allocation attempt exceeds the allocatable GPU Memory size, the GPU will raise an event similar to the RAM OOM Kill (not the same thing because RAM OOM Kill is raised by the Linux Kernel) through the NVIDIA GPU Operator, we name it *GPU OOM*. At the same time, the corresponding container will be evicted.

Thus, a post-processor is attached as the GPU auto-scaler's last component, for handling the GPU OOM events. Similar to the RAM one, this post-processor maintains a state value $pr^{GI}[t]_i$ represents the ID of post-processed GI request after considering the *i*th GPU OOM Kill event during the current 5-minute aggregation window *t*. When a GPU OOM Kill event happens, this means the GPU Memory size of the current GI is not enough, so we directly use the larger GI($pr^{GI}[t]_{i-1}+1$) to continue the workload. At the beginning of each aggregation window *t*, this value is (re-)initialized as the ID of current GI recommendation $r^{GI}[t]$. Finally, the GI recommendation is calculated from the ID of the post-processed GI request.

$$pr^{GI}[t]_0 = r^{GI}[t]$$
$$pr^{GI}[t]_i = min(pr^{GI}[t]_{i-1} + 1, P)$$
$$PR^{GI}[t]_i = GI_{pr^{GI}[t]_i} \quad (3.23)$$

If there are already *i* GPU OOM Kills in the current aggregation window *t*, then the container's recommended GI is $PR^{GI}[t]$.

7.4 VPA-Pilot Implementation

7.4.1 Auto-scaler implementation in Kubernetes cluster

Now we discuss the efficient and accurate implementation of the VPA-pilot model in the Kubernetes cluster. Since this work focuses more on the algorithm part, we choose not to develop an auto-scaler from scratch. Instead, we implement VPA-pilot for RAM and CPU based on an efficient open-source vertical auto-scaler framework called Kubernetes VPA (Vertical Pod Autoscaler), in Golang. Due to time constraints, this work also did not implement the GPU auto-scaling in the Kubernetes cluster. The evaluation of GPU auto-scaling will only be performed on the simulator in the next section. In future, the whole VPA-pilot auto-scaler for CPU, RAM and GPU will be implemented as part of the optimizations in the Ryax open-source platform.

This section first presents the Kubernetes VPA Framework, and how the VPA-pilot will be implemented based on this framework. Next, we present the detailed implementation of the formalized model, focusing on optimizing the resource consumption to adapt to the edge-cloud environment.

7.4.2 Implementation with Kubernetes VPA framework

Kubernetes VPA is an open-source implementation of a vertical auto-scaler for the Kubernetes cluster. It contains a built-in, simple threshold-based auto-scaling algorithm, which can provide rough container size recommendations. This algorithmic component has a well-defined interaction API with the system, making it ideal for adapting and implementing custom auto-scaling algorithms.

Kubernetes VPA consists of 3 main components, as shown in Figure 15:

- ***Recommender*** is the component where the core algorithm resides. It receives real-time usage information for all workloads through the Kubernetes Metrics Server API and extracts the workload usage specific to the containers (pods) of the deployment currently being auto-scaled. This usage data is input into the core algorithm implemented by the developer. Finally, the algorithm's recommendation outputs are stored in a Kubernetes Custom Resource Definition (CRD) object.
- ***Updater*** is the component that compares the latest container size recommendation in the CRD with the current actual container size. If it determines that the current container size needs to be updated, it gracefully evicts the pods hosting the containers that require resizing according to specific rules. This allows the pods to be restarted with the updated container size.
- ***Admission controller*** is a common component that intercepts requests to the Kubernetes API server, to validate or modify requests to create, delete, and modify objects. In this VPA framework, the admission controller modifies the pod creating requests to create the new pod with an updated container size.

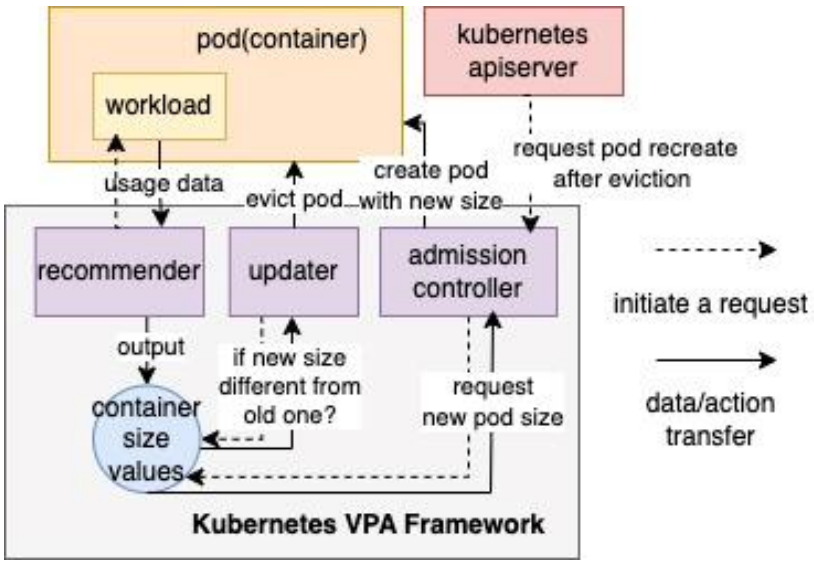

Figure 15: Architecture of Kubernetes VPA Framework

We implement VPA-pilot in the recommender component, by *replacing its built-in threshold-based algorithm*. For each resource type in a deployment being auto-scaled, we maintain a VPA-pilot instance. The instance starts running as an endless loop since the recommender is initiated. Every second, this instance takes the usage data of each related container as the input signal $u_i[\tau]$. Every 5 minutes, the instance aggregates the input signals of all containers in a deployment, runs a round of VPA-pilot, and outputs the final request and limit recommendation to the CRD.

7.4.3 Implementation complexity of VPA-pilot

Next, we describe the specific implementation of the VPA-pilot algorithm, focusing on minimizing time and space complexity. This ensures low resource consumption of the auto scaler with a large number of sub-models.

We start by analysing the equations presented before. We notice that:

1) The aggregation signal $s[k]$ is only used for calculating the number of samples above/under different bucket bounds. Therefore, instead of repeating this calculation in each sub-model, we pre-calculate them using dynamic programming right after we get the $s[k]$ signal.
2) Among the $N_{dm} \cdot N_{mm}$ sub-models, the ones with the same d_m values share the same $o(r)[t]$ and $u(r)[t]$ values in Equation 3.12, and the same $r'_m[t]$ values in Equation 3.13. Thus, we can share these values among the sub-models with the same d_m, thus saving N_{mm} times of resource consumption.
3) All time series (variables with $[t]$) are iterated only based on the latest value, so we don't need to store the entire time series. Instead, we just keep updating on a single variable for each time series.

Taking these optimizations, the VPA-pilot is implemented as Algorithm 1 (Figure 16). This algorithm describes a single round in the infinite loop, i.e. the recommendation in an aggregation window t. Based on the analysis (3) above, we remove the $[t]$ in every time series and replace them with corresponding single variable. The values of these variables are retained and reused in the next round.

Because $K \geq N_{mm}$ (no need to set safety margin higher than the maximum available resource size), the time complexity of this implementation is $O(K \cdot N_{dm})$. Its space complexity is also $O(K \cdot N_{dm})$.

Algorithm 1 Implementation of 1 round of VPA-pilot calculation

Input: $s[k], k \in \{1 \ldots K\}$
Output: R
Init arrays: $s\_above[K] = \{0\}$, $s\_under[K] = \{0\}$
for $k = 2$ to K **do**
$s\_under[k] \leftarrow s\_under[k] + s[k-1]$
end for
for $k = K-1$ to 1 **do**
$s\_above[k] \leftarrow s\_above[k+1] + s[k+1]$
end for
Init arrays: $o[r][N_{d_m}]$, $u[r][N_{d_m}]$, $r'_m[N_{d_m}]$
for $n_{d_m} \in \{1 \ldots N_{d_m}\}$ **do**
$d_m = \frac{n_{d_m}-1}{N_{d_m}-1}$ (Uniform sampling in [0, 1])
for $r \in \{1 \ldots K\}$ **do**
$o[r][n_{d_m}] = (1-d_m) \cdot o[r][n_{d_m}] + d_m \cdot s\_above[r]$
$u[r][n_{d_m}] = (1-d_m) \cdot u[r][n_{d_m}] + d_m \cdot s\_under[r]$ (Eq. 3.12)
end for
$r'_m[n_{d_m}] = \arg\min_{r \in \{1 \ldots K\}} (w_o \cdot o[r][n_{d_m}] + w_u \cdot u[r][n_{d_m}] + w_{\Delta R} \cdot \Delta(r, r'_m[n_{d_m}]))$ (Eq. 3.13)
end for
Init 2d arrays $r_m[N_{d_m}][N_{m_m}]$, $c_m[N_{d_m}][N_{m_m}]$
for $n_{d_m} \in \{1 \ldots N_{d_m}\}$ **do**
$d_m = \frac{n_{d_m}-1}{N_{d_m}-1}$ (Uniform sampling in [0, 1])
for $m_m \in \{0 \ldots N_{m_m}-1\}$ **do**
Record: $prevr_m = r_m[n_{d_m}][m_m]$
$r_m[n_{d_m}][m_m] = \min(r'_m[n_{d_m}] + m_m, K)$ (Eq. 3.13)
$c_m[n_{d_m}][m_m] = d(w_o \cdot s\_above[r_m[n_{d_m}][m_m]] + w_u \cdot s\_under[r_m[n_{d_m}][m_m]] + w_{\Delta R} \cdot \Delta(r_m[n_{d_m}][m_m], prevr_m)) + (1-d) c_m[n_{d_m}][m_m]$ (Eq. 3.14)
end for
end for
$M = \arg\min_{m=(n_{d_m} \in \{1 \ldots N_{d_m}\}, m_m \in \{0 \ldots N_{m_m}-1\})} (c_m[n_{d_m}][m_m] + w_{\Delta m}\Delta(M, m) + w_{\Delta R}\Delta(r, r_m[n_{d_m}][m_m]))$
$r = r_m[n_{d_m}][m_m]$ s.t. $M = (n_{d_m}, m_m)$ (Eq. 3.15 and 3.16)
return $R = b[r]$

Figure 16: Single round implementation of VPA-pilot calculation

7.4.4 Hyper-parameter tuning with simulator

In the previous subsection, we mentioned 5 hyper-parameters ($d, w_o, w_u, w_{\Delta R}, w_{\Delta m}$) in our VPA-pilot algorithm. Their values need to be determined before running the algorithm and should be passed as configuration parameters to the VPA recommender. In Rzadca's Autopilot work, *they tune these hyper-parameters of Autopilot in off-line experiments during which they simulate Autopilot behaviour on a sample of saved traces taken from representative jobs.* We adopt this way and present a detailed method for tuning the hyper-parameters of VPA-pilot.

We employ a one (1) month's Google Workload Traces[26] on CPU and RAM usage as the "sample of saved traces taken from representative jobs". In this section, we first design a VPA-pilot simulator for offline experiments and evaluation on the long-term Google Workload Traces. Then we discuss our detailed method of tuning hyper-parameters based on the simulator.

[26] Clusterdata 2019 traces google/cluster-data. https://github.com/google/ cluster-data/blob/master/ClusterData2019.md. Accessed: 25/07/2024.

7.4.5 Auto-scaler simulator design

To tune the hyper-parameters on long traces, and evaluate the long-term behaviour of the auto scaler, it would be highly time-consuming and impractical to run it for a long time in a real-world environment to collect the necessary data. We need to overcome the time constraints and obtain the long-term data within a short time. To achieve this, we design the auto-scaler simulator. This subsection focuses on the auto-scaler simulator for CPU and RAM resources. The one for GPU has a very similar structure and behaviour.

The structure of the auto-scaler simulator is shown in Figure 17. We first pre-process the Google Workload Trace using Apache Spark into a trace data sequence with a frequency of 1 second, then pass the sequence into the simulator. The simulator runs in the form of iteration over the sequence. At each time step (second), a logical workload generates CPU and RAM usage according to the trace sequence value. The VPA-pilot algorithm takes the CPU and RAM usage as signals $u_i^{CPU}[\tau]$ and $u_i^{RAM}[\tau]$ per second and outputs recommended container requests and limits every 300-time steps, to simulate the 5-minutes aggregation time. A logical container is set with the recommended size, to communicate with logical workload to simulate system events like OOM Kills. Finally, on the global layer outside the per-second loop, a metrics collector gathers all required metrics to generate graphs and statistics tables after the whole execution.

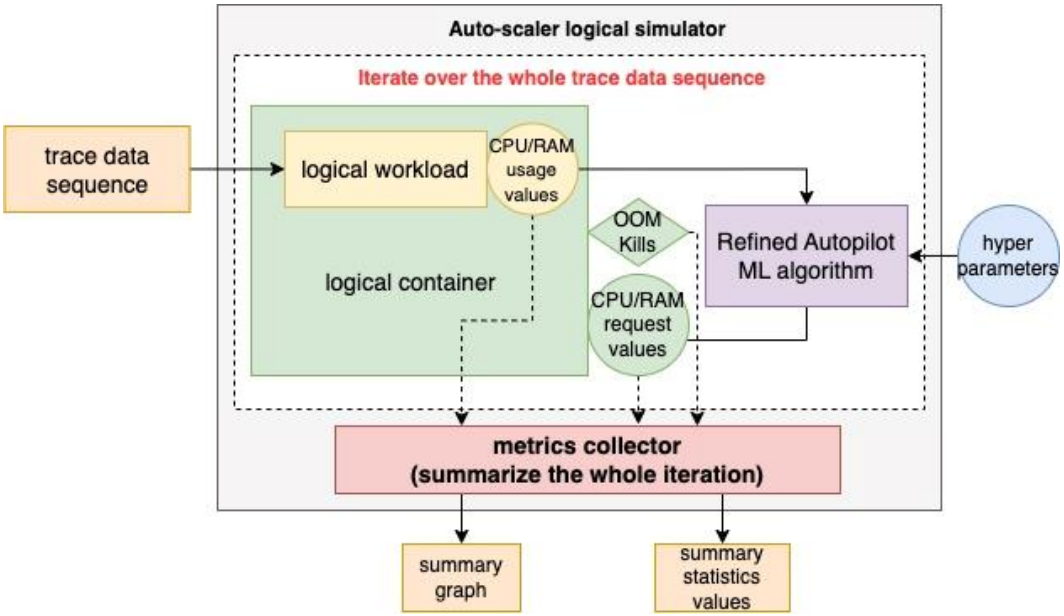

Figure 17: Structure of the auto-scaler simulator

To simplify our development, the simulator ignores the overhead of container (pod) restarts in the VPA Framework implementation. This is because we focus on the algorithm itself rather than the VPA Framework's efficiency. We can evaluate the number of restarts in the statistics and calculate their real impact. Thus, if we adjust the container size at the current time step, it will be immediately applied to the logical container in the next time step. Similarly, for OOM Kill events, the RAM post-processor can make *i* bump-ups in only *i* time steps.

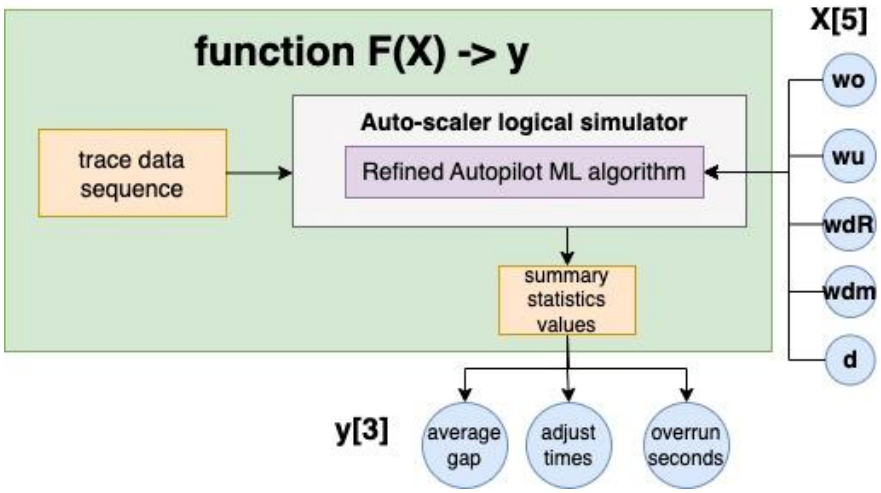

Figure 18: Black box function *F*(*X*) = *y* in hyper-parameter tuning modelling

7.4.6 Hyper-parameter tuning: Modelling and solving modelling as an operations research program

Now we present our way of tuning the 5 hyper-parameters $(d, w_o, w_u, w_{\Delta R}, w_{\Delta m})$, in VPA-pilot using the auto-scaler simulator presented in the previous subsection. This subsection also dedicates only CPU and RAM resources.

We start with modelling the hyper-parameter tuning problem. Our goal refers to the criteria of Rzadca's Autopilot paper: *This tuning aims to produce a configuration that dominates alternative algorithms (such as the moving window recommenders) over a large portion of the sample, with a similar (or slightly lower) number of overruns and limit adjustments, and significantly higher utilization.* Based on this description, we can model the process of finding the dominating hyper-parameters on one 1-month Google Workload Trace sample as an OR (Operations Research) program including a black box function *F*(*X*) = *y*.

The function is built as in Figure 18 that treats the whole execution of the simulator as a black box with only hyper-parameters and summary statistics exposed. *X* is the input hyper-parameters set of Refined Autopilot ML. We normalize each hyper-parameter between [0,1]. This is because *d* is the decaying weight originally between [0,1]. $w_o, w_u, w_{\Delta R}, w_{\Delta m}$ are weights that are only compared with each other, so normalizing them only brings convenience without changing their meanings.

y is composed of 3 key global performances of VPA-pilot:

- average gap is the 1 month's average of (container raw request - workload usage) values at every time step. The container raw request here is the VPA-pilot's raw output without RAM post-processor (i.e. $R[t]$ in Equation 3.16). We use this $R[t]$ because the 5 hyper-parameters only affect the behaviour of the core model part, not the RAM post-processor. The lower average gap means the *higher utilization* in Rzadca's criteria, because we have the same usage in this model for one workload sample.

- adjust times is the number of occurrences where container raw requests $R[t]$ differ between adjacent time steps. Higher adjust times mean higher *limit adjustments* and more pod (containers) restarts, which we do not want.
- overrun seconds is the number of time steps in which workload usage is larger than the container's raw request $R[t]$. This value represents the *number of overruns* in Rzadca's criteria.

The OR program modelling the hyper-parameter tuning problem on a single trace sample is shown in Equation 4.1 below. The baseline adjust times and baseline overrun seconds are the corresponding metrics generated from the simulator with the Autopilot Rule-based algorithm in subsection 7.3.1.2. ε is a small relaxation on the constraints to avoid no solution. At the beginning of our solving attempt, we let $\varepsilon = 0.5$ to only limit the order of magnitude. It can be reduced in future attempts for better results.

Minimize: $y[1]$

subject to: $F(X) = y$

$$0 \leq X[i] \leq 1, i \in \{1 \dots 5\}$$

$$0 \leq y[2] \leq baseline\_adjust\_times \cdot (1 + \varepsilon)$$

$$0 \leq y[3] \leq baseline\_overrun\_seconds \cdot (1 + \varepsilon)$$

where:

$$X = [d, w_o, w_u, w_{\Delta R}, w_{\Delta m}]$$
$$y = [average\_gap, adjust\_times, overrun\_seconds] \quad (4.1)$$

We compare the optimal value $y[0]$ of this OR program with the average gap of the Autopilot Rule-based baseline. If our solution is significantly smaller, the *X* vector in the optimal solution is our best hyper-parameters on this workload sample.

7.4.7 Feasible region sampling for dominating hyper-parameters on CPU

After successfully modelling the tuning problem as the OR program in Equation 4.1, we try to solve it. This OR program is very hard to solve because the *F*(*X*) function is not composed of mathematical formulas but is based on real program execution results. This makes *F*(*X*) a complete black box from the mathematical point of view. Therefore, it is impossible to calculate the gradient or to prove the convexity of *F*(*X*).

Because of the hardness above, this work does not successfully find the OR program's optimal solution. However, we can approximate the optimal solution by sampling within the feasible region of *X*, then filtering all samples where $y[1]$ and $y[2]]$ satisfy the constraints, and sorting them in ascending order based on $y[0]$. If in a solution obtained through sampling, the value $y[1]$ (average gap) is still significantly smaller than the baseline, then the hyper-parameters set *X* in the solution also make our VPA-pilot dominate the Autopilot Rule-based baseline. We name a such sampled set of hyper-parameters *dominating hyper-parameters*.

To sample the feasible region $X, 0 \leq X[i] \leq 1, i \epsilon \{1 \dots 5\}$, we start with uniform sampling on the whole 5D feasible region. If we sample *n* points in each dimension, then the whole uniform sampling has a complexity of $O(n^5)$. Therefore, although our simulator is fast, we can only afford to sample up to 10 points for each hyper-parameter (*n* = 10 in each dimension).

On the CPU resource, we did a round of *n* = 10 uniform sampling as above. We found that the samples with the smallest $y[1]$ (average gap) values all have $X[3] = X[4] = X[5] = 0$, which means in the dominating hyper-parameters set, $w_u, w_{\Delta R}, w_{\Delta m}$ are the most likely to be 0. Although we cannot prove this, we continue with this assumption $X[3] = X[4] = X[5] = 0$. Therefore, the dimension of the sampling space is greatly reduced, allowing us to perform more detailed sampling with *n* = 300 in $X[1]$ and $X[2]$ dimensions. After this detailed sampling, we successfully find several sets of dominating hyper-parameters. Table 4 compares the sample of the most dominating hyper-parameters, with the Autopilot Rule-based baseline.

Figure 19 compares the long-term performance of VPA-pilot under the dominating hyper-parameters set, with the Autopilot Rule-based baseline algorithm. From Table 4 and Figure 19, we can validate that the dominating hyper-parameters set ($d = 0.85714, w_o = 0.14285, w_u = 0, w_{\Delta R} = 0, w_{\Delta m} = 0$) is the best for CPU resource above the given 1 month's Google Workload Trace sample.

Table 4: Statistics comparison between VPA-pilot on CPU resource, with dominating hyper-parameters [in order ($d, w_O, w_u, w_{\Delta R}, w_{\Delta m}$)], and the Autopilot Rule-based baseline

Statistics for CPU resource	Average Gap	Adjust Times	Overrun Seconds
VPA-pilot Rule-Based	275.20660	425	12896
VPA-pilot hyper-param: (0.85714,0.14285,0,0,0)	237.71037	127	14355

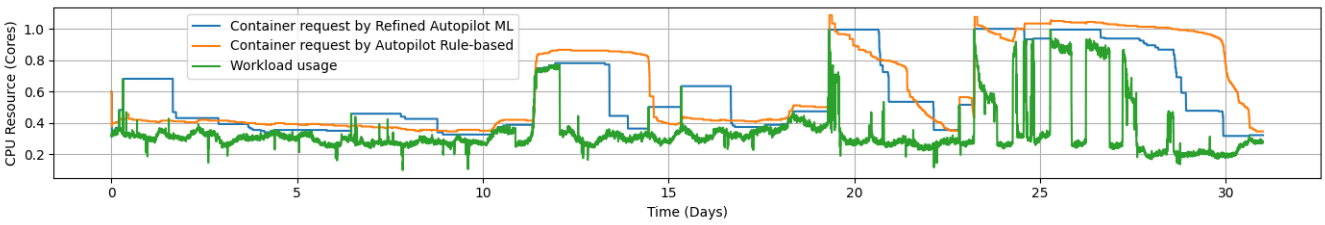

Figure 19: VPA-pilot performance on CPU resources with dominating hyper-parameters ($d = 0.85714$, $w_o = 0.14285$, $w_u = 0$, $w_{\Delta R} = 0$, $w_{\Delta m} = 0$). Compared with Autopilot Rule-bases baseline

7.4.8 Sampling and manual methods for dominating hyper-parameters on RAM

Encouraged by the success on CPU above, we start on RAM also by performing exhaustive uniform sampling of the 5D feasible region of X with n = 10 in each dimension. Unfortunately, we cannot find any patterns in the results. As a result, we are not able to reduce the dimension to perform more detailed sampling. Therefore, we propose another two sampling methods to attempt to find more accurate dominating hyper-parameters.

The first sampling method is per-dimension uniform sampling as shown in Algorithm 2 (Figure 20). This sampling starts with X_0, y_0, which are the above n = 10 exhaustive uniform sampling's solution with the minimum y[1] (average gap) value. Outputs a more fine-grained sampling solution X_{lopt}, y_{lopt}. This sampling tries to find a "local minimum" that is near to the above n = 10 exhaustive sampling's minimum solution. Although we cannot prove any relation between this local minimum and the global minimum, this local minimum should be much better than the n = 10 exhaustive sampling's solution.

The second sampling method is pure random sampling: We repeatedly randomize the 5 elements of input X as real numbers in [0,1], limit y[2], y[3] and gather the minimum y[1]. We let this random program run for 1 day on a cloud server.

Algorithm 2 Per-dimension sampling for dominating hyper-parameters on RAM

Input: X_0, y_0
Output: X_{lopt}, y_{lopt}
  $X_{lopt} \leftarrow X_0, y_{lopt} \leftarrow y_0$
  for $i \in \{1 \ldots 5\}$ **do**
    for $j \in \{0 \ldots 20000\}$ **do**
      $sample\_val_j = \frac{j}{20000}$
      Make X_{sample} by replace $X_{lopt}[i]$ by j. Make $y_{sample} \leftarrow F(X_{sample})$
      if $y_{sample}[1] < y_{lopt}[1]$, $y_{sample}[2] \leq$ baseline_adjust, $y_{sample}[3] \leq$ baseline_overrun **then**
        $X_{lopt} \leftarrow X_{sample}, y_{lopt} \leftarrow y_{sample}$
      end if
    end for
  end for
  return X_{lopt}, y_{lopt}

Figure 20: Per-dimension uniform sampling algorithm for dominating hyper-parameters on RAM

It is interesting that, although neither of the above 2 sampling methods can be proved to find solutions close to the global minimum of y[1], the y vectors (average gap, adjust times and overrun seconds) produced by the two sampling methods are relatively close, although the corresponding X (hyper-parameters) vectors are completely different. Therefore, we guess that the two methods yield nearly the optimal solution, multiple hyper-parameters sets can lead to this solution. However, we are not able to prove this.

The hyper-parameters obtained by random sampling are (d = 0.01388, w_o = 0.80714, w_u = 0.04725, $w_{\Delta R}$ = 0.25499, $w_{\Delta m}$ = 0.10142). Corresponding y values are shown in the third row of Table 5. Its corresponding VPA-pilot performance is shown in Figure 21. Unfortunately, the average gap value of VPA-pilot with randomly sampled parameters is larger than that of the Autopilot Rule-based baseline. This means for RAM resources, we cannot get dominating hyper-parameters with our sampling methods.

Table 5: Statistics comparison among VPA-pilot for RAM resources, with hyper parameters fixed by samplings (d = 0.01388,w_o = 0.80714,w_u = 0.04725,$w_{\Delta R}$ = 0.25499,$w_{\Delta m}$ = 0.10142), with hyper-parameters fixed manually (d = 0.612,w_o = 0.9,w_u = 0.01,$w_{\Delta R}$ = 0.0099,$w_{\Delta m}$ = 0.00009), and the Autopilot Rule-based baseline

Statistics for RAM resource	Average Gap	Adjust Times	Overrun Seconds	OOM Kills
Autopilot Rule-Based	261822531	63	301	3
VPA-pilot hyper-param: (0.01388, 0.80714, 0.04725, 0.25499, 0.10142)	332630700	92	385	20
VPA-pilot hyper-param: (0.612, 0.9, 0.01, 0.0099, 0.00009)	111383357	83	9835	47

Besides sampling, we also attempted to manually search for hyper-parameters and observe their performance, regardless of the strict constraints in Equation 4.1, because we roughly understand the practical meaning of each hyper-parameter thanks to the good interpretability of VPA-pilot. We manually discovered a set of hyper-parameters that are worth dis cussing: (d = 0.612,w_o = 0.9,w_u = 0.01,$w_{\Delta R}$ = 0.0099,$w_{\Delta m}$ = 0.00009), whose statistics is shown in the third row of Table 5 and whose performance is shown as the bottom graph of Figure 21.

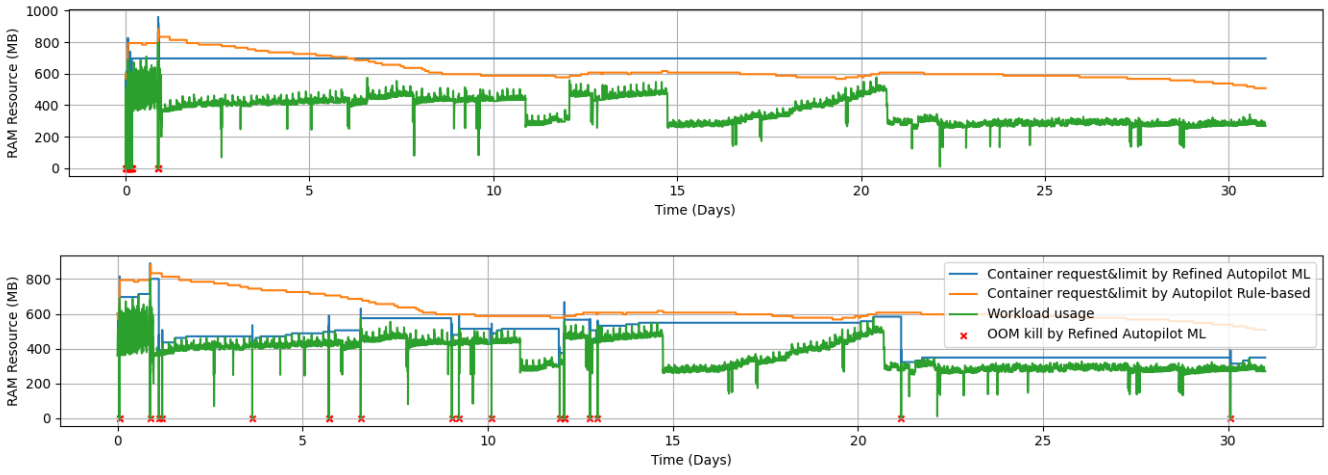

Figure 21: VPA-pilot performance on RAM resource with hyper-parameters fixed by samplings (the top graph), with hyper-parameters fixed manually (the bottom graph). Compared with Autopilot Rule-bases baseline

Now we concern the 3 key global performances (average gap, adjust times, overrun seconds) brought by our manually discovered hyper-parameters: In Table 5 compared to the Autopilot Rule-based baseline, our manually discovered hyper-parameters leads to significantly smaller average gap, slightly reduced adjust times, but a great increase in overrun seconds. These make the hyper-parameters good but not standard dominating. Although not dominating, based on the overall assessment of the results, this is the best solution we can find. In the following section, we discuss the reasons for this situation, what we can learn from the positive aspects, and how to improve the bad overrun seconds.

For a RAM auto-scaler, OOM Kills is actually a more critical performance than overrun seconds in our OR program, as it directly leads to pod eviction. Therefore, for memory resources, our simulator additionally generate OOM Kills performance for RAM:

- OOM kills (for RAM) is the number of events where the OOM Kills triggers the RAM post-processor.

These OOM Kill statistics are shown in the last column of Table 5. It shows that the Refined Autopilot ML with 2 different hyper-parameters sets both have significantly higher OOM Kills compared to the baseline, which is not good. This indicates that the lack of OOM Kill control in Rzadca's Autopilot paper criteria and our OR program is problematic. In the following "Future improvements" part, we will attempt to add OOM Kill control in our hyper-parameter tuning model.

7.4.9 Future improvements for a more balanced hyper-parameter tuning model

We argue and study whether the VPA-pilot with sampled hyper-parameters (top of Figure 21) or the one with manually found hyper-parameters (bottom of Figure 21) performs better. Based on our analysis, we tend to favour the manually found hyper-parameters, although the sampled ones are selected entirely following Rzadca's Autopilot paper criteria and the OR model. Next, we discuss this conflict in detail.

First, we should admit that the VPA-pilot is not universally good for all types of workloads. Due to the lack of decaying weight in the resource amount itself like in Autopilot Rule-based, when faced with a workload usage that suddenly decays and then persists for a long time as in Figure 21, VPA-pilot often tries to fit by making abrupt changes in recommended container size, rather than gradually reducing it. This naturally makes the auto-scaler generate more overruns on this specific type of workload.

The issue with the sampling hyper-parameters method is that it relies too strictly on the 1 month's global performance metrics but rarely cares about the specific auto-scaling behaviour. Therefore, when faced with the workload in Figure 21, if we strictly adhere to the criteria and the OR model we have established to tune hyper-parameters, to force VPA-pilot generates less overruns than the Autopilot Rule-based. Then the auto-scaler will attempt to aggressively increase the container size at the beginning of the trace and then maintain a high container size to avoid overruns, as shown in the top graph of Figure 21.

This is not good auto scaler behaviour in a production environment. However, the global criteria and our OR model have no punishment for this behaviour. In contrast, the manually selected hyper-parameters do not strictly adhere to the 1 month's global performance metrics, especially on "overrun seconds". However, this actually results in better auto-scaling behaviour.

The comparison above is enough to demonstrate that the limits on adjust times and overrun seconds in our current OR model (Equation 4.1) and Rzadca's original criteria are sometimes too strict, particularly for hyper-parameters tuning on RAM resources.

However, completely relaxing these restrictions is also unacceptable. The exploding overrun seconds by manually selected hyper-parameters is the result. Therefore, even if strictly limiting them is not feasible, we still need to apply some appropriate constraints. The same issue applies to the OOM Kills caused by VPA-pilot. Thus, we also need to add OOM Kills limit to the current 3 key global performances.

Based on all the above discussions, we designed a more balanced and flexible approach to improve the OR program: to assign weights to each element in vector *y* in the OR program of Equation 4.1, based on practical requirements, thereby constructing a new single objective value *y* in the OR program.

The improved OR program is shown in Equation 4.2 below:

Minimize: y

subject to:

$$F(X) = y'$$

$$y = y' \times [weight_1, weight_2, weight_3, weight_{4\ (RAM\ only)}]^T$$

$$0 \leq X[i] \leq 1, i \in \{1 \dots 5\}$$

where:

$$X = [d, w_o, w_u, w_{\Delta R}, w_{\Delta m}]$$

$$y' = [average\_gap, adjust\_times, overrun\_seconds, OOM\ Kills(RAM\ only)] \quad (4.2)$$

To effectively limit the number of OOM kills, we add "OOM Kills" support in the simulator thus adding the corresponding element to the vector y'. Besides, to further evaluate the details of the auto-scaler actions, we can add more detailed metrics like "the balance of the gap values during all periods of the trace", etc. into y' vector in Equation 4.2, and assign corresponding weight in the calculation of *y*.

Besides, to get more robust behaviour in production, we still need to introduce more representative workload traces, and tune hyper-parameters in the same way as we discussed on these representative traces. To be specific, we planned to use Apache Spark[27] to extract multiple (at least 10+) representative traces from the Google Workload Trace[28] by filtering keywords and merging sub-tasks, then randomly shuffle fragments of these traces to create new sequences for running hyper-parameters tuning.

However, the Google Workload Trace is a massive dataset (2.4TB, compressed). Due to the lack of time and enough resources to process such a size, we only processed 1 one-month trace used above and did not go further for multiple traces.

Moving such a system to production we need to consider the workload trace (historical data) of the system we aim to optimize. To be specific, for the future hyper-parameters tuning in the edge-cloud continuum, we can broadly collect usage data from representative workloads running on different edge-cloud clusters, and use Apache Spark to filter and categorize those from similar clusters. After some human review, we can shuffle and concatenate the data fragments from similar clusters. The hyper-parameters will be tuned respectively on these concatenated data for the corresponding clusters.

The development of the complete methodology to consider when analysing historical workload traces using Apache Spark to better prepare the hyperparameters for each production case remains a future work.

[27] https://spark.apache.org
[28] Clusterdata 2019 traces google/cluster-data.
https://github.com/google/cluster-data/blob/master/ClusterData2019.md

8 Conclusions

The EMPYREAN platform pursues a very ambitious goal: to unify operations across multiple layers of the computing stack, spanning from low -level interconnects and container virtualization to high-level resource management, autoscaling, and service assurance.

The platform is designed to support, among others, the deployment of large-scale distributed and collaborative systems in both the scale-up (horizontal scaling, software-defined interconnects, and hardware acceleration abstractions for containerized workloads) and scale-out (dynamic autoscaling of containers on Kubernetes clusters) dimensions. These capabilities enable fine-grained and efficient resource sharing across heterogeneous environments within the IoT-edge-cloud continuum.

The developments presented in this report reflect the progress made during the first iteration of the implementation phase (M4-M15) under Tasks 3.3 *"Software-Defined Edge Interconnect for Distributed Computations and Hardware Acceleration"* and Task 3.4 "*Autoscaling, Service Assurance, and Computing Management*" of Work Package 3, laying the foundation of EMPYREAN's unified and collaborative platform. Moreover, the detailed mechanisms empower EMPYREAN to achieve key technical and performance objectives. Final implementations and integration outcomes will be detailed in the forthcoming Deliverable D3.3, scheduled for M26.