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

Grant Agreement no. 101136024

Deliverable D4.1
Low-code Application Description, Seamless Deployment, and Analytics-friendly Distributed Storage

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:	WP4
Responsible Editor:	RYAX
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
31.03.25	RYAX	Draft	0.1	Initial ToC
02.04.25	RYAX, ZSCALE	Draft	0.2	Integrate initial contributions in Sections 3, 4
08.04.25	NUBIS	Draft	0.3	Integrate updated partners contributions in Sections 5
14.04.25	ICCS, CC	Draft	0.4	Integrate final contributions in Sections 3, 4, 5
20.04.25	RYAX, CC, NUBIS	Draft	0.9	Complete version for internal review
30.04.25	RYAX	Final	1.0	Final version after internal review

Author List

Organization	Author
RYAX	Michael Mercier, Pedro Velho, Yiannis Georgiou
ZSCALE	Ivan Paez
NUBIS	Anastassios Nanos
CC	Marton Sipos
ICCS	Aristotelis Kretsis

Internal Reviewers

Marton Sipos (CC)

Jaime Fúster (NEC)

Abstract: This Deliverable presents the outcomes of Tasks 4.2 and 4.3 of the EMPYREAN project, covering the period from M4 to M15. It begins with an overview of the EMPYREAN architecture, emphasizing the integration and roles of components developed within these tasks. This deliverable provides in-depth technical documentation of several key components, including the workflow manager, dataflow programming environment, analytics-friendly distributed storage, action packaging system, and unikernel builder. Each component is described with respect to its internal architecture, contributions beyond the state-of-the-art, configuration and installation procedures, operational workflows, and exposed public APIs. Ongoing development efforts and planned integrations are discussed, and the report concludes with a roadmap outlining the next steps for Tasks 4.1 and 4.2, including reporting in future deliverables.

Keywords: EMPYREAN Architecture, Workflow Manager, Distributed Storage, Multi-clustering, Hybrid Edge-Cloud Continuum, Dataflow Programming, Unikernels, Environment Packaging

Table of Contents

List of Figures

List of Tables

Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
CI/CD	Continuous Integration / Continuous Delivery
CLI	Command Line Interface
CNCF	Cloud Native Computing Foundation
CTI	Cyber Threat Intelligence
D	Deliverable
DAG	Direct Acyclic Graph
DDL	Data Definition Language
DDS	Data Distribution Service
DFP	Data Flow Programming
ESG	Edge Storage Gateway
FaaS	Function as a Service
GDD	Generalized Data Deduplication
GPU	Graphics Processing Unit
HPC	High Performance Computing
I/O	Input / Output
IIoT	Industrial Internet of Things
IoT	Internet of Things
KPI	Key Performance Indicator
M	Month
MIG	Multiple-Instance GPU
ML	Machine Learning
MQTT	Message Queuing Telemetry Transport
OCI	Open Container Initiative
ORM	Object Relational Mapper
OS	Operating System
POSIX	Portable Operating System Interface
PV	Persistent Volume
PVC	Persistent Volume Claim
REST	Representational State Transfer
RLNC	Random Linear Network Coding
RPC	Remote Procedure Call
SDK	Service Development Kit
SotA	State-of-the-Art
SSH	Secure Shell Protocol
T	Task
TLS	Transport Layer Security
UC	Use Case
UI	User Interface
VM	Virtual Machine
VPN	Virtual Private Network
WP	Work Package

1 Executive Summary

This deliverable presents the technical outcomes of Task 4.2: *"Workflow-based Design and Low-code Application Description"* and Task 4.3: *"Seamless Deployment and Analytic-Friendly Distributed Storage"*, between M4 and M15.

First, the general EMPYREAN architecture is described, highlighting the role of the various components developed in the two tasks. The goal is to understand how each component is positioned in the EMPYREAN platform's architecture.

This overview is followed by a detailed technical description of each component, including a description of contributions and beyond state-of-the-art developments, internal architecture diagrams, installation and configuration details, key operation flows, and public APIs. Different components such as: the Workflow Manager, the Dataflow Programming, the Analytics-Friendly Distributed Storage, the Action Packaging and the Unikernels Builder are described in this deliverable. Beyond the technical descriptions, aspects related to the project's management are discussed, continuing the work reported in D2.3 (M12). The relationship of each component with project objectives is also discussed. Furthermore, a brief description of ongoing developments and planned integrations with other platform components is provided.

Finally, future steps are highlighted as part of the conclusion. This includes a roadmap for the following phase of T4.1 and T4.2, including how these contributions will be reported.

2 Introduction

EMPYREAN envisions a collaborative environment where application owners work together within Associations, sharing resources and data seamlessly across the edge-cloud continuum. Realizing this vision requires enabling the design and deployment of applications across distributed edge-cloud infrastructures, along with support for application packaging and decentralized storage solutions.

This deliverable focuses on components from Work Package 4: *"Decentralized Intelligence and Application Development and Deployment".* It provides a concise overview of its contents, outlining the purpose, structure, and intended audience of the document.

2.1 Purpose of this document

This deliverable presents the outcomes of Task 4.2: *"Workflow-based Design and Low-code Application Description"* and Task 4.3: *"Seamless Deployment and Analytic-Friendly Distributed Storage"*, covering their first 12 (M4 to M15). Four partners - RYAX, NUBIS, CC, and ZSCALE - were directly involved in the implementation of these tasks. The work builds upon the foundation laid in Work Package 2, which defined the project's requirements and established the overall EMPYREAN architecture.

As one of the project's initial technical deliverables, provides a detailed description of the components developed in Tasks 4.2 and 4.3 and how they integrate into the broader EMPYREAN architecture. It provides a detailed technical description of these components, and where applicable, typical workflows are provided to illustrate their dynamic behavior alongside the platform's static architecture.

The deliverable further outlines the initial developments, components' internals and API definitions, as well as the planned integration steps between the various components. These are essential milestones toward the integration activities that will be carried out in Work Package 5. Therefore, this document serves as a key technical reference for both platform service developers and use case providers.

A final report detailing the results of Tasks 4.2 and 4.3 during their second phase will be presented in Deliverable 4.3, scheduled for M26. While the current deliverable focuses on the individual technical specifications of each platform component, D4.3 will place greater emphasis on integration aspects, end-to-end operation flows and API details.

2.2 Document Structure

The present deliverable is split into five major sections, centred around the components developed in Task 4.2 and Task 4.3:

- EMPYREAN Architecture Mapping
- Application Design with Low-Code and Workflow-Based Abstractions
- Analytics-Friendly Distributed Storage
- Action Packaging
- Unikernels Builder

2.3 Audience

This document is publicly available and intended for anyone interested in the EMPYREAN project's work on multi-clustering, workflow management, distributed storage, environment packaging, and unikernel building. It provides an initial overview of these components along with development details and component internals including their architecture, early-stage contributions, and preliminary interfaces. Additionally, it serves as a useful resource for the general public to gain a clearer understanding of the EMPYREAN framework and the overall scope of the project.

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 (i) AI-enhanced tools for the workflow-based design and low-code description of hyper-distributed applications, (ii) novel application packaging and software delivery framework, (iii) end-to-end systems software stack for application deployment based on unikernels, (iv) secure and interoperable container runtime, and (v) analytics-friendly distributed storage solution for IoT data.

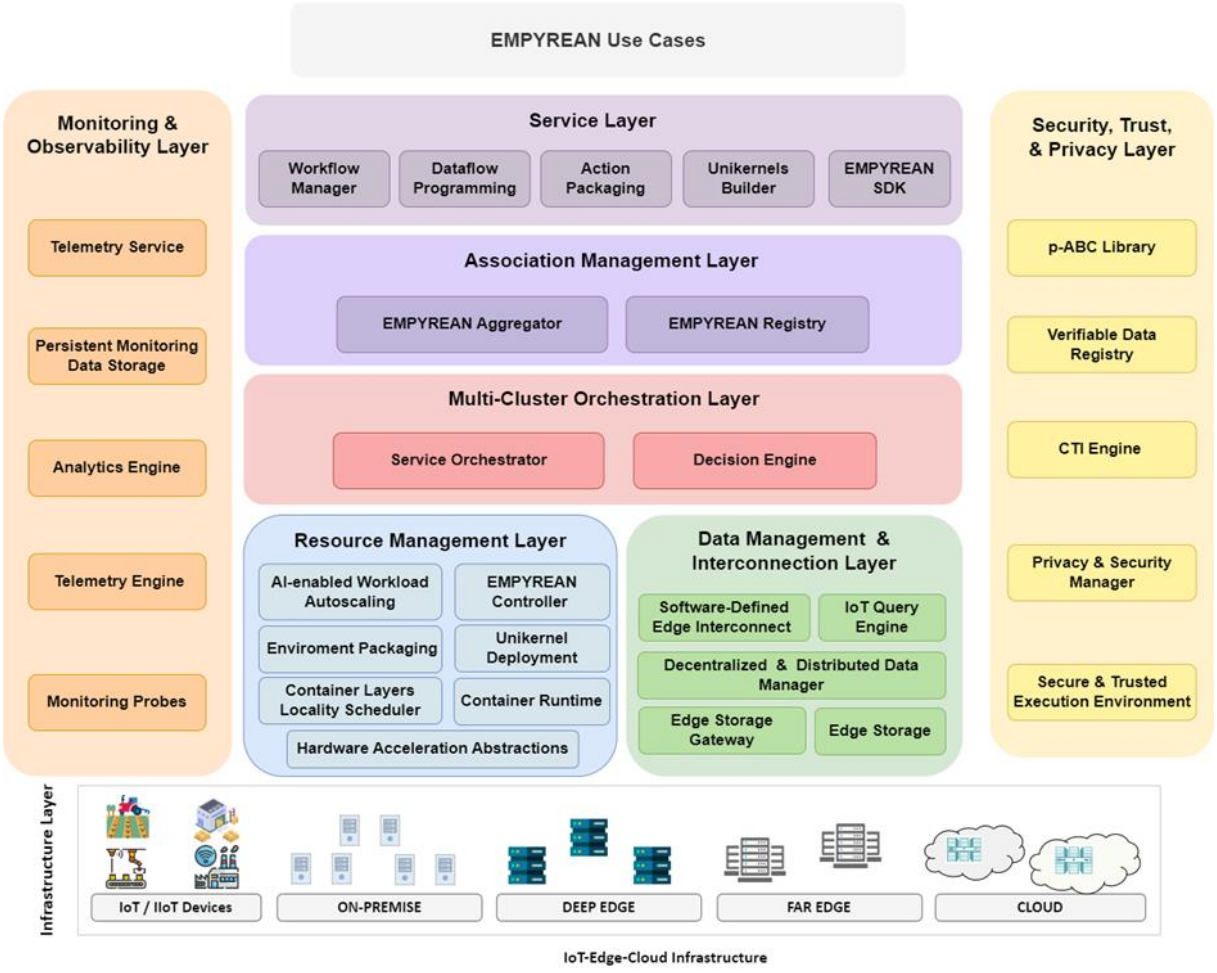

Figure 1: EMPYREAN high-level architecture

The ***Service Layer*** supports the development of Association-native applications by offering mechanisms for application-level adaptability, interoperability, elasticity, and scalability across the IoT-edge-cloud continuum. It addresses several key aspects, including: (a) the design and management of workflows for hyper-distributed applications, (b) cloud-native unikernel application development, and (c) data-flow description. This deliverable provides the detailed design and initial implementation description of this layer's components, while the EMPYREAN SDK will be introduced in deliverable D5.2 (M18).

The *Workflow Manager* (Section 4) offers tools for high-level design, development, and remote debugging of cloud-native applications, enabling seamless deployment across the Association-based continuum. The *Dataflow Programming* (Section 4) complements workflow-based application management by enabling data-centric, decentralized, and dynamic data interconnections. It allows declarative specification of data flow requirements using unified abstractions and location-transparent definitions—facilitating responsive, adaptive data management in highly heterogeneous and distributed environments. The *Unikernels Builder* (Section 5) streamlines the development and deployment of applications as unikernels in cloud-native environments. It reduces engineering complexity and enables the generation of highly efficient, secure, and lightweight applications—ideal for both edge and cloud infrastructures where performance and isolation are essential.

The ***Association Management Layer*** dynamically manages Associations across the IoT-edge-cloud continuum. By enabling the formation of resource federations, it supports seamless collaboration, resource sharing, and data distribution across diverse segments of the infrastructure. In conjunction with the Multi-Cluster Orchestration Layer, it plays a central role in EMPYREAN's distributed and autonomous management framework, establishing a resilient and adaptive Association-based continuum.

The ***Multi-Cluster Orchestration Layer*** enables efficient orchestration of services and dynamic resource management across EMPYREAN's disaggregated infrastructure. Leveraging distributed, autonomous, decision-making mechanisms, it manages the lifecycle of hyper-distributed applications and enables self-driven adaptations. Multiple instances of this layer's components provide decentralized operations, optimized resource utilization, and scalability, while also supporting energy efficiency and fault tolerance. The design and development details on this layer and the Association Management are described in D4.2 (M15).

The ***Resource Management Layer*** unifies and coordinates resource management across IoT, edge, and cloud platforms within the EMPYREAN architecture. It integrates both platform-level scheduling mechanisms (e.g., AI-enabled Workload Autoscaling) and low-level runtime mechanisms (e.g., Unikernel Deployment). Operating within Kubernetes or K3s clusters, this layer is designed to be highly modular, simplifying the integration of new hardware and software components. This deliverable focuses on the initial developments of Environment Packaging, Unikernel Deployment, and Container Runtime components. Additional related developments are discussed in deliverables D3.2 (M15) and D4.2 (M15).

The *Action Packaging* component (Section 5) supports multi-environment and multi-architecture packaging for cloud-native applications, improving the interoperability and adaptability of workloads within the EMPYREAN platform. It streamlines the creation of OCI-compatible container images that support diverse architectures and programming languages, ensuring flexibility in deployment across heterogeneous execution environments. The *Unikernel Deployment and Container Runtime* components (Section 5) enable the efficient deployment of unikernel-based applications by integrating them with standard container runtimes compatible with Kubernetes and serverless platforms. This allows seamless orchestration of lightweight and secure applications across heterogeneous infrastructures.

The ***Data Management and Interconnection Layer*** ensures secure, scalable, and dynamic data communication and storage between IoT devices and computing resources. Operating at both cluster and Association levels, it integrates distributed data management mechanisms that support seamless data interaction between IoT, edge, and cloud environments. This deliverable presents the Analytics-friendly Distributed Storage, while the broader data management mechanisms are covered in deliverable D3.1 (M15), and the software-defined interconnection framework is detailed in D3.2 (M15). The *Analytics-friendly Distributed Storage* (Section 5) provides a distributed storage solution tailored for time series IoT data. It enhances the storage and retrieval of IoT time series data, employing erasure coding techniques to ensure secure, reliable, and efficient data management of large volumes of time series data.

The **Infrastructure Layer** comprises heterogeneous resources distributed across multiple administrative and technological domains including, (i) IoT/IIoT devices, robots, and on-premise edge resources where data is generated and service requests initiated, (ii) deep and far-edges, close and further from the end users/devices, for real-time processing and aggregation, and (iii) federated multi-cloud environments for enhanced robustness, cost-efficiency, and vendor independence in data storage and replication.

The **Security, Trust, and Privacy Layer** integrates distributed components to ensure secure access, privacy-preserving operations, and trusted execution across the platform. Operating at both cluster and Association levels, it establishes secure execution environments where trust relationships between data-generating and data-processing entities are continuously verified through distributed trust mechanisms. In parallel, identity and data access management components ensure controlled access and data confidentiality among different entities. The core components of this layer are presented in deliverable D3.1 (M15), while the Cyber Threat Intelligence (CTI) Engine is detailed D4.2 (M15).

The ***Monitoring and Observability Layer*** provides real-time monitoring, observability, and service assurance through distributed telemetry mechanisms and data-driven analytics. It dynamically collects and analyzes a wide range of metrics across heterogeneous infrastructures and deployed applications, ensuring system health, performance, and availability. These insights support automated control and optimization. These components are detailed in D4.2 (M15), with the service assurance mechanisms covered in D3.2 (M15).

4 Application Design with Low-Code and Workflow-based Abstractions

4.1 Introduction

In EMPYREAN, the *Workflow Manager* component is provided by the open-source RYAX[1] workflow engine. This component enables users to build, deploy and monitor their data analytics and AI applications through a low-code, workflow-based approach. This section presents an overview of Ryax's core features and architecture, and brings forward the specific enhancements developed in the context of EMPYREAN to provide fine-grained support across the edge-cloud continuum. Ryax expresses applications as one or more workflows using YAML-based abstractions, facilitating the development of distributed data analytics applications. These workflows are then deployed across the underlying hybrid computing infrastructures.

Several enhancements have been studied and are under active development. The first major enhancement introduces multi-site workflow support. It involves enabling workflows to execute across multiple clusters at both the edge and cloud. This capability requires particular networking and storage configurations to ensure seamless coordination and execution of workflows across geographically distributed environments. Furthermore, Ryax platform is being extended with the right mechanisms and abstractions to support user-defined constraints and objectives that guide optimal workload placement. While initial scheduling uses Ryax's build-in first-fit algorithms, more advanced strategies will be enabled through upcoming integrations with the Decision Engine and Service Orchestrator components of EMPYREAN.

Another important enhancement is enabling the Ryax platform to efficiently support both long-running microservices and short-duration serverless functions for data analytics and AI applications in the edge-cloud continuum.

Additionally, Ryax is being adapted to integrate with EMPYREAN Associations by interacting with the Aggregator and incorporating Zenoh[2] at the communication layer as an alternative for the message passing needed in Ryax. This integration will provide fine-grained, real-time data communication capabilities, which is essential for IoT-based EMPYREAN use cases.

This deliverable primarily focuses on the multi-site orchestration enhancements, which are essential for seamless workflow execution across the edge-cloud continuum. Additionally, it provides a status update on the ongoing work related to execution model support and integration with EMPYREAN's data communication stack.

[1] https://github.com/RyaxTech/ryax-engine
[2] https://zenoh.io

4.2 Background - The Ryax Workflow Manager

The RYAX platform developed by Ryax Technologies is an open-source low-code, API-first workflow management system for data analytics. It provides the means to create, deploy, update, execute, and monitor the execution of data processing workflows on hybrid cloud, edge, on-premises, and HPC computing infrastructures. It enables users to create their data automations and expose them with APIs through fully customizable workflows using a low-code UI. It uses a powerful, hybrid serverless/microserverices-based runtime, abstracting completely the complexity of building and deploying containers with their dependencies upon Cloud infrastructures. The software platform has been designed with a focus on data analytics. It offers a variety of built-in features and a repository of various integrations to facilitate users in industrializing cloud backend applications integrating complex data automations.

The Workflow Management system automates the orchestration and execution of task collections upon computational resources. A common pattern in scientific and cloud computing involves executing different computational and data manipulation tasks, which are usually coupled, i.e., the output of one task is used as input on another. Hence, coordination is required to satisfy data dependencies. The system handles the task execution and can be distributed among the underlying available computational resources. Consequently, this introduces further complexity on the system side related to processes such as load balancing, data storage, data transfer, task monitoring, and fault tolerance. Furthermore, on the application side, workflows provide an end-to-end view of the processing rather than focusing on a specific part of the computation, which allows users to control the whole process by abstracting the complexity of how each task is executed. Automating the aforementioned aspects of the orchestration process along with the complexity abstraction has led to the creation of workflow management systems.

The RYAX platform abstracts the complexity of (i) developing data analytics pipelines by using workflows requiring developers to describe the structure of what will be deployed and how it will be connected through a simple, intuitive web interface and visual programming tools connected to a YAML- based declarative design, (ii) building the environments to be deployed with the necessary dependencies through built-in internal mechanisms based on Nix functional package manager, (iii) deploying and monitoring containers through a fine-integration with Kubernetes, and (iv) providing efficient autoscaling capabilities based on built-in resource management techniques considering the request demands and combining both Horizontal Pod Autoscaling (Kubernetes) and Cluster Node Autoscaling (Cloud Infrastructure provider).

The initial internal architecture of Ryax is depicted in Figure 2. As illustrated, the system is composed of multiple microservices, each managed by the Kubernetes orchestrator. This design ensures a high degree of interoperability, flexibility, fault tolerance, and resilience, enabling features such as auto-healing, and easy upgrades or downgrades of individual components.

In the context of EMPYREAN, this architecture has been further adapted to address the specific challenges of the edge-cloud continuum. Enhancements have been introduced to better support heterogeneous and distributed environments, as well as to provide tighter integration with EMPYREAN Associations. These modifications are detailed in the following section.

In more detail, the principal concepts of RYAX are:

- **An action** is an independent task representing a separate building block of a broader data processing application. It may have inputs, outputs, and a specific code that manipulates the inputs with some logic to produce the outputs. There are the following types of actions:
    - **A trigger**, which allows data ingestion from the outside world. They are long-running processes (micro-services) that can trigger new workflow executions.
    - **A processor**, which is a stateless process (FaaS) that basically can perform processing. They use their inputs to ingest data from upstream produce outputs added to the downstream.
    - **A publisher**, which is a stateless process that pushes data to external services, like a database or an online service for example.
- T**he workflow,** which is a complete data processing application composed of actions linked together in the form of a DAG. The intermediate links are data streams. Each action uses some data from the input stream and adds its output data to the stream. This way, the data that an action outputs is accessible to every downstream action. In other words, any action has access to the data of upstream actions.

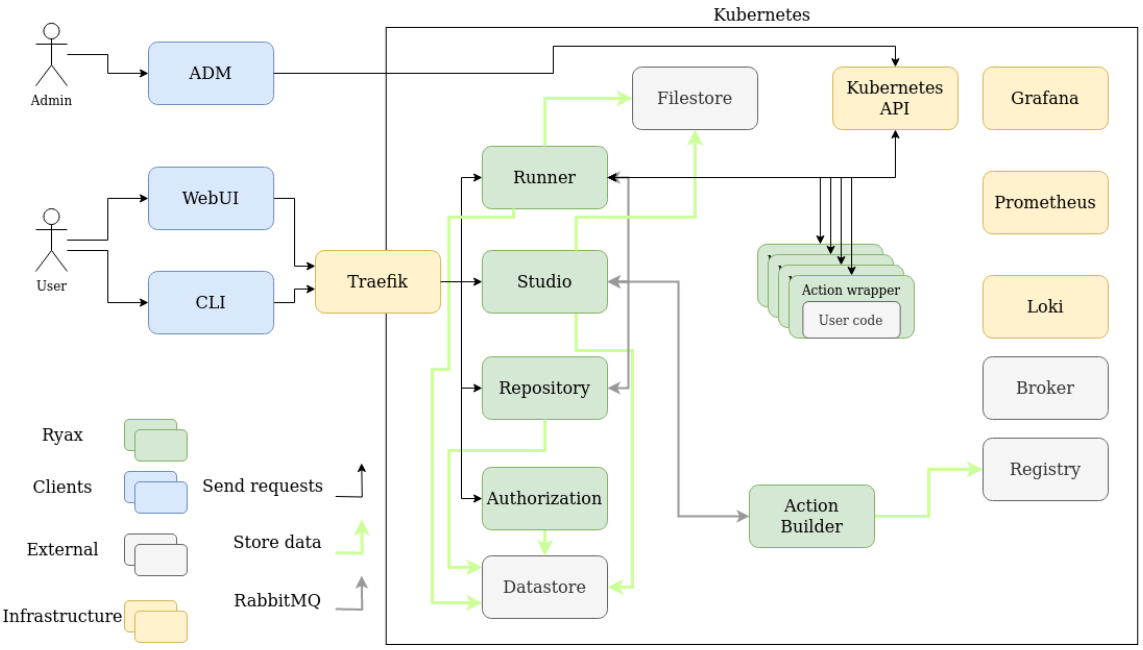

Figure 2: Ryax Architecture Overview

In particular, we have:

- The Ryax client services:
  - **WebUI,** which is a NGINX[3] server that serves our frontend written with Angular[4].
  - **ADM,** which is the administrator tool for Ryax used to deploy, update and backup/restore a Ryax instance upon a certain infrastructure.
  - **CLI,** which is an open-source Python based command line interface that enables users to build actions and create, submit and monitor workflows.
- The Ryax internal services:
  - **Authorization,** which manages user and project authorization along with roles control within Ryax.
  - **Repository,** allows the Ryax users to scan Git repositories and import Ryax actions. It also enables the triggering of the actions built through the action Builder. Once the build is finished, the actions are sent to the Studio to be placed in the action Store. It exposes an HTTP Rest API that is used by the WebUI and the CLI. It uses the Postgres datastore to persist its state using ORM.
  - **Studio,** enables Ryax users to create, edit, and deploy workflows. It exposes an HTTP Rest API that is used by the WebUI and the CLI. It uses the Postgres datastore to persist its state using ORM.
  - **Action builder,** which receives action build orders from the Repository service, does the build in a sequential and synchronous way (one at a time) making use of the Nix functional package management software for dependencies control.
  - **Runner,** which plays the role of interface between Ryax and the computing resources to launch and execute actions & run workflows. Thus, it enables the deployment, run and schedule of each action of the workflow. It also manages the execution metadata and data. This micro-service communicates with the actions through gRPC, and expose workflow runs state and results through an HTTP Rest API. http and the broker (RabbitMQ[5]).
  - **Action wrapper,** which is used between our system and the user code to be able to run it. This is not an internal Ryax service, but a wrapper that is put around user code in order to communicate with Ryax. It creates a gRPC server with a simple interface that initiates the action and then run executions. This wrapper works for both processors and trigger source actions with the same protocol: the trigger source actions are streaming execution response, while the processors are only sending one response and close the connection.

[3] https://nginx.org
[4] https://angular.dev
[5] https://www.rabbitmq.com

- The Ryax external services:
  - **Datastore,** which enables the state storage of all stateful services and a PostgreSQL[6] database server. Each service has a different access credential and a separate database.
  - **Filestore,** which is used to store execution I/O files and directories is a Minio[7] file storage service which exposes an S3 compatible API.
  - **Broker,** which enables internal communication between all services using messages serialized in Protobuf and is a RabbitMQ message broker.
  - **Registry,** which stores the users' actions created by the action builder in a Docker registry.

By default, the Ryax architecture enables the execution of actions through the Runner component on the underlying Kubernetes cluster where Ryax is deployed. For monitoring and observability, Ryax integrates a robust monitoring stack comprising Prometheus[8] for metrics collection, Loki[9] for log aggregation, and Grafana[10] for querying and visualizing operational data. Network ingress is managed by Traefik, which dynamically routes incoming HTTP requests based on path prefixes—for example, requests with the ***/runner*** prefix are directed to the Runner service.

4.3 Analysis for Ryax Multi-Site Support

This section evaluates three architectural patterns for enabling Ryax to operate seamlessly across multiple sites and cloud environments, based on our hands-on experimentation and industry best practices.

We begin with a single-cluster, multi-cloud nodes approach—demonstrated using Scaleway Kosmos[11], and comparable to solutions such as AWS EKS Anywhere[12] and Google Anthos[13]. This is followed by an exploration of true multi-cluster federation approaches, utilizing platforms like Karmada[14] and KubeSphere[15] to achieve centralized management across distributed Kubernetes clusters. Finally, we present a lightweight multi-cluster model based on persistent “worker” agents, which register with a central Ryax instance.

For each pattern, we detail setup complexity, fault-tolerance, autoscaling, storage integration, and operational overhead. The analysis culminates in identifying the multi-cluster with worker agents model as the most balanced and extensible solution to support EMPYREAN’s multi-site deployment and orchestration goals.

[6] https://www.postgresql.org
[7] https://min.io
[8] https://prometheus.io
[9] https://github.com/grafana/loki
[10] https://grafana.com

4.3.1 Single-Cluster, Multi-Cloud Nodes

Scaleway's Kosmos[11] cluster enables attaching external instances from any provider into a single managed Kubernetes control plane, unifying node pools across clouds and on-premises environments. Similar functionality can be achieved with AWS EKS Anywhere[12], which leverages the open-source EKS Distro to run Kubernetes on on-premises and edge infrastructures with automated lifecycle management, or Google Anthos[13], which provides a consistent control plane for GKE clusters on-cloud, on-premises, and across AWS/Azure via its multi-cloud APIs. In our prototype, we provisioned the Kosmos control plane via Terraform, manually added an Azure VM as an external node by registering it with Scaleway's ***node-agent_linux_amd64***, and opened WireGuard (UDP 51820), Kubernetes API (TCP 10250), and health-check ports (TCP 8134, 9100) on the VM.

Storage & Scheduling Considerations

Out-of-the-box Scaleway CSI is only installed on managed Scaleway Instances; external nodes require deploying third-party CSI drivers or a Local Path Provisioner to support PVCs. To mitigate cross-cloud latency—critical for database queries and low-latency RPCs—we pinned all Ryax services (datastore, file store, RabbitMQ, etc.) to the local node pool via ***nodeSelector***, draining external nodes during installation and uncordoning them afterward to enforce locality.

Pros & Cons

- **Pros:**
  - **No Ryax code changes** required, enabling rapid proof-of-concept deployment.
  - **Unified control plane** simplifies cluster administration and monitoring.
  - **Fast setup**, achievable within ~2 days using Terraform and minimal manual steps.
- **Cons:**
  - **No autoscaling/auto-healing** on external node pools, necessitating manual node management.
  - **Storage integration gaps**, with CSI support inconsistency across providers.
  - **Cross-cloud latency**, forcing service colocation strategies.
  - **Vendor lock-in**, as each provider uses proprietary multi-cloud tooling.

[11] https://www.scaleway.com/en/kubernetes-kosmos/

[12] https://aws.amazon.com/eks/eks-anywhere/

[13] https://cloud.google.com/anthos

4.3.2 Multi-Cluster Federation

Karmada[14] (Kubernetes Armada) is a CNCF incubating project offering a federated control plane to automate deployment, scaling, and failover across multiple Kubernetes clusters and clouds without modifying applications. It speaks native Kubernetes APIs and provides advanced scheduling, cross-cluster rolling upgrades, and policy-driven placement.

KubeSphere[15] integrates KubeFed to present a central control plane ("host cluster") managing multiple member clusters, with unified monitoring, logging, and an App Store for cross-cluster application deployment. It supports both direct connections and agent-based membership for network-isolated environments.

Pros & Cons

- **Pros:**
  - **True cluster isolation**, each site retains its own API server and resource quotas.
  - **Rich federation features**, including traffic scheduling, policy enforcement, and disaster recovery.
- **Cons:**
  - **High control-plane complexity**, requiring installation and maintenance of federation controllers.
  - **Network dependencies** between clusters, often requiring VPNs or overlay networks.
  - **Steep learning curve** for federation APIs and operational procedures.

4.3.3 Multi-Cluster with Worker Agents

In this pattern, Ryax runs a single master control plane exposed publicly, while each site deploys a lightweight worker agent that:

1. **Registers** to the master using a bootstrap token.
2. **Maintains** a persistent, encrypted overlay using **Skupper**[16] to route all control-plane and data-plane traffic through secure, multi-site tunnels without exposing local clusters directly.
3. **Exposes** site capacity and executes Ryax Actions via a minimal RPC protocol carried over the Skupper mesh.

[14] https://github.com/karmada-io/karmada

[15] https://kubesphere.io/docs/v3.4/multicluster-management/introduction/kubefed-in-kubesphere/?utm_source=chatgpt.com

[16] https://github.com/skupperproject/skupper

By leveraging Skupper for inter-site networking, all RPC calls, logs, metrics, and file transfers traverse a single outbound connection per site, simplifying firewall configuration and avoiding full VPN meshes.

Pros & Cons

- **Pros:**
  - **Simplified networking**: Skupper handles service-to-service connectivity and secure routing over existing cloud or on-premise networks.
  - **Low operational overhead:** Only one control plane to maintain, and no federation controllers.
  - **Site autonomy:** Local policies, quotas, and resource isolation remain fully under each site's control.
  - **Scalability:** New sites join simply by deploying the Worker and wiring it into the Skupper network.
- **Cons:**
  - **Implementation effort:** Requires building the Worker registration logic and RPC protocol on top of the Skupper fabric.
  - **Skupper dependency:** Relies on maintaining the Skupper overlay—though this also centralizes connectivity and reduces tunnel sprawl.
  - **Master availability:** The public control plane must be highly available, as all worker agents depend on it for registration and task dispatch.

4.3.4 Conclusions and Recommendation

We recommend adopting a **multi-cluster worker-agent** model with all inter-site traffic tunneled over a **Skupper** overlay network. In this setup, each remote site runs a lightweight Ryax Worker that:

1. **Securely bootstraps** to the central Ryax master using a token-based authentication mechanism.
2. **Joins the Skupper service mesh**, enabling all control-plane and data-plane RPCs to traverse over mutual-TLS–protected, Layer-7 tunnels, eliminating the need for VPNs or public service exposure.
3. **Reports local capacity and executes Actions** through a minimal RPC protocol.

Skupper's dynamic, cost- and locality-aware routing provides high availability and performance across hybrid clouds and on-premise sites, while also simplifying connectivity management.

This pattern strikes the ideal balance between the simplicity of single-cluster, multi-cloud approaches (e.g., AWS EKS Anywhere's customer-managed on-prem/cloud clusters) and the heavy control-plane complexity of full federation (e.g., Karmada). The design mirrors proven CI/CD architectures—such as GitHub Actions' self-hosted runners—where distributed agents connect outbound to a central service, maintain autonomy over local resources, and scale independently without exposing local APIs.

By combining a single Ryax master with per-site Workers interconnected via Skupper, EMPYREAN establishes a secure, provider-agnostic, and extensible multi-site fabric with minimal operational burden.

4.4 Ryax Extensions for Multi-Site Support

The initial version of Ryax supported workflow execution within one site either on-premise or in the cloud. However, in the context of EMPYREAN, Ryax has been significantly extended to support the deployment of workflows across multiple sites. Based on these extensions Ryax will have the ability to deploy parts of workflows on different geographically distributed sites, enabling a truly distributed orchestration model. The only requirement for a participating site is the presence of Kubernetes as the orchestrator, which allows seamless integration into the Ryax-managed continuum.

The updated architecture is illustrated in Figure 3, where the new remote sites are connected to the central Ryax master site through Skupper, shown in the top-right of the diagram. Additionally, as part of the effort to enable Ryax to operate across the edge-cloud-HPC continuum, preliminary design changes have been introduced to support HPC site integration via SSH-based connectivity (shown at the bottom-right of the figure). However, HPC integration lies outside the scope of this particular deliverable.

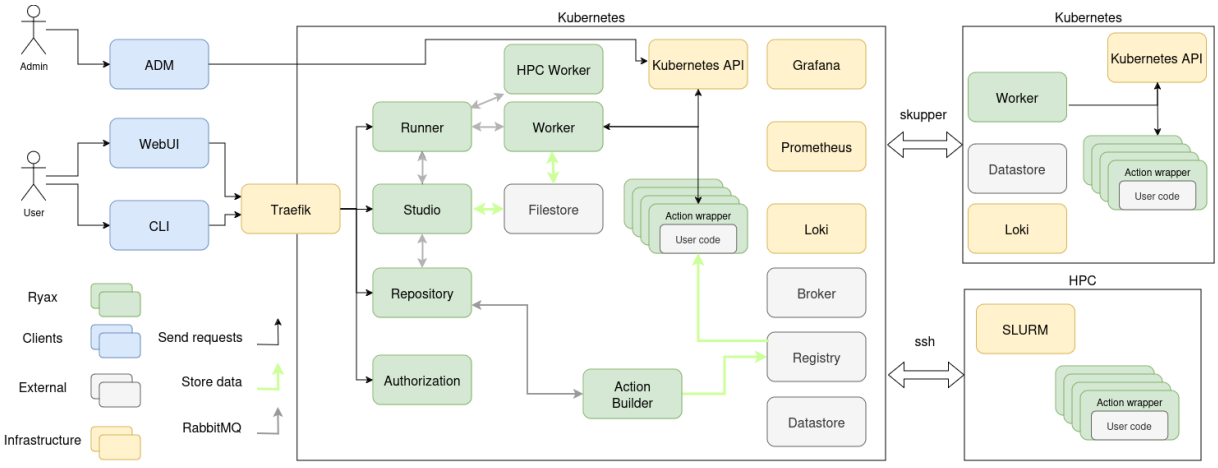

Figure 3: Ryax New Multi-Site Architecture Overview

4.4.1 Runner

The Runner service is the core of the Ryax tool, serving as the execution engine responsible for managing and orchestrating user workflows. It receives workflow deployment and un-deployment orders from the Studio component and provides execution feedback and history on the workflow runs through an HTTP API. Internally, the Runner uses the PostgreSQL datastore to persist its state, accessed through an Object-Relational Mapping (ORM). As the key interface between Ryax and the underlying computing resources, the Runner handles the launching of actions and execution of workflows. Thus, it does the deploying, running, and scheduling of workflows. It also manages the execution metadata and data flow.

In terms of communication, this microservice uses gRPC for interacting with actions, RabbitMQ for asynchronous messaging with the Studio, and HTTP to support direct, standalone access. This multi-protocol design ensures efficient and flexible interactions with other Ryax components and deployed workflows, forming the backbone of Ryax's low-code, distributed orchestration capabilities.

In the new architecture, its responsibilities have been adapted to become the following:

- Deployment
  - Communicate with the worker to get computing resources
  - Deploy/undeploy actions
- Manage the workflow execution:
  - Get new execution triggers
  - Fetch/push execution data from/to the filestore (Minio)
  - Push data to actions so that they create executions
- Scheduling
  - Communicate with the worker to scale the infrastructure
  - Scale actions
  - Trigger executions (if necessary)
  - Store waiting executions, and has an algorithm to decide which execution to start first
- Archiving
  - Holds and manages execution metadata
  - Keep track of all executions and expose an API for querying execution metadata
  - Keep track of the workflow deployment states

4.4.2 Worker

The worker service makes the link between Ryax and the local sites. The worker registers itself in Ryax using a gRPC interface. Its responsibilities are the following:

- Get access to computing resources
- Receive execution trigger
- Deploy the action if not already done (might be done beforehand depending on the deployment policy)
- Fetch execution data from remote filestore
- Activate the action with its inputs and grab the outputs
- Push execution data to local/remote storage
- Send results and their metadata to Ryax
- Undeploy the action (might be done beforehand depending on the deployment policy)
- Trigger the next execution (go back to step 2)
- Deployment scaling, depending on the underlying infra it can have different policies. For example, it can only allocate resource when a new execution is coming or keep all actions always deploy for performance reasons.

Figure 4 provides the architecture of the Ryax worker showing how the internals of this new microservice take place.

Regarding executions, the Worker gets I/O from global when an action is triggered from an external action (action from another site) and pushes I/O to local storage. The Worker pushes the data to the global storage if an execution is not on its site. Since the I/O files are distributed over multiple sites, we need a way to share data between sites. To do so, we introduce a new public-facing storage.

In our case, we use our fileStore (Minio), which we expose publicly, but it could also be any cloud storage. Based on this, the data can be shared between sites using this public storage with the following policy:

Outputs:

- always push I/O files in local storage.
- if one of the next executions is in another site also push data to the global storage. This is done by the Worker.

Inputs:

- if an I/O file comes from an execution done on another site pull I/O files from global storage
- otherwise pull I/O files from the local storage

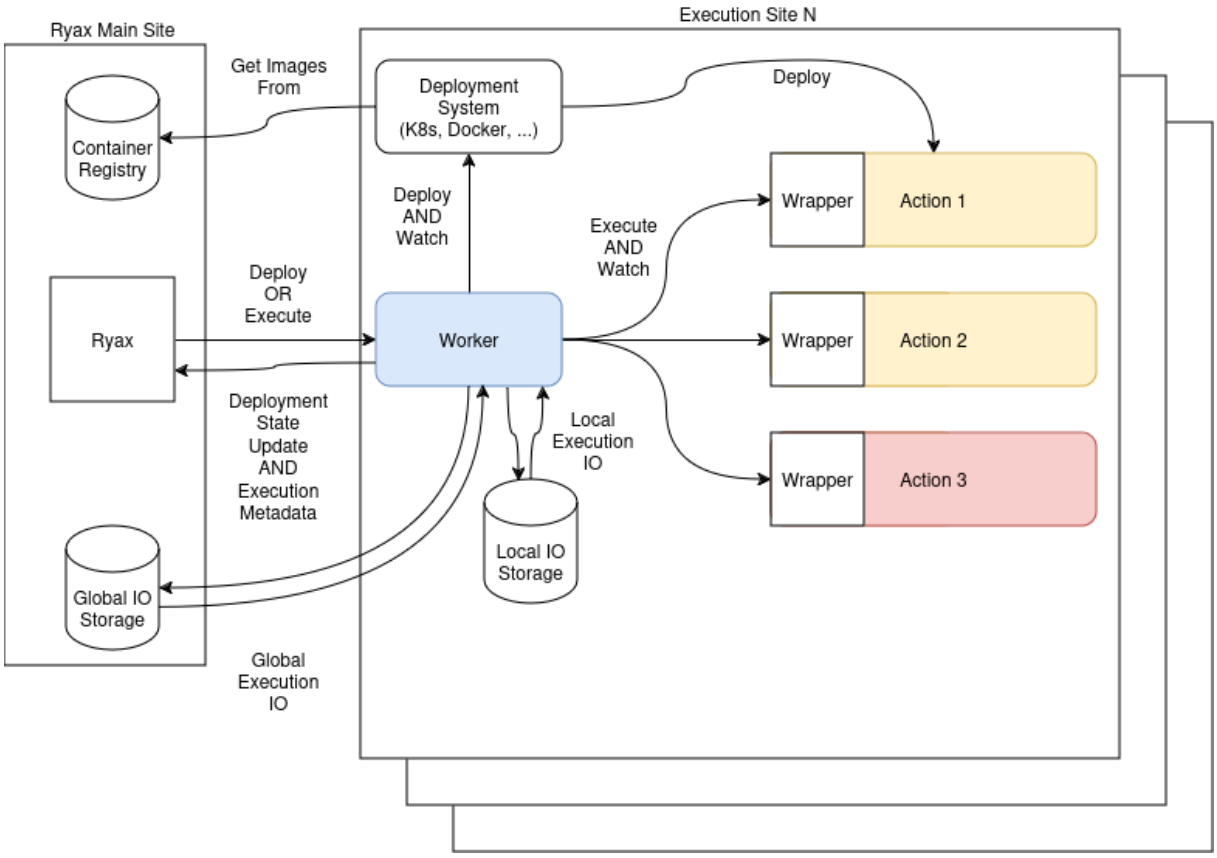

Figure 4: Ryax Worker Architecture

4.4.3 Installation and configuration for Ryax multi-site

This section provides details about the installation and configuration of Ryax multi-site version.

Prerequisites

A Kubernetes cluster (v1.19 +) with a working default StorageClass for dynamic PersistentVolume (PV) provisioning must be available for each site. To verify or set a default, the administrator can run:

```
kubectl get storageclass

kubectl patch storageclass <YourStorageClassName> \ -p '{"metadata":{"annotations":{"storageclass.kubernetes.io/is-default-class":"true"}}}'
```

This leverages the built-in Kubernetes annotation mechanism to mark the desired class as the default for PVCs. Helm 3.x must be installed on the operator's workstation (or CI host) to manage chart-based deployments. Each Worker node should have at least 2 vCPUs, 2 GiB memory, and 1 GiB disk. The actual resource needs will scale with the execution of the tasks.

Configuration

In order to configure the Worker, the administrator will need to select one or more node pools (set of homogeneous nodes) and give the Worker some information about the nodes. We use node pools on Kubernetes because they allow Ryax to leverage the Kubernetes node autoscaling. This approach allows the executions to scale up - if the workload needs more resources - or down - if the workload does not need any more resources. It can even scale down to zero when there are no workload requests. This brings forward the real benefits of serverless, which is the core of the Ryax runtime.

The Worker's ***worker.yaml*** must define the site identity and node pools. For AWS clusters, an example snippet is:

```
config:
  site:
    name: aws-k8s-cluster-1
    spec:
      nodePools:
      - name: small
        cpu: 2
        memory: 4G
        selector:
          eks.amazonaws.com/nodegroup: default
```

For Azure AKS-managed pools, the selector will differ:

```
config:
  site:
    name: azure-k8s-cluster-2
    spec:
      nodePools:
      - name: small
        cpu: 2
        memory: 4G
        selector:
          kubernetes.azure.com/agentpool: default
```

Values for CPU and memory must match each node's ***Allocatable*** fields; selectors must uniquely identify the pool by its provider label.

Skupper Networking Setup

To establish a secure, application-level mesh between the central Ryax control plane and site Workers, Skupper is deployed in the ***ryaxns*** namespace on both clusters. After creating the namespaces:

```
kubectl create namespace ryaxns
kubectl create namespace ryaxns-execs
```

the master site initializes Skupper with:

```
skupper init -n ryaxns --site-name main
skupper token create ~/main.token -n ryaxns
```

and each remote site links using:

```
skupper init -n ryaxns --ingress none --site-name site-1
skupper link create ~/main.token -n ryaxns
```

This installs the Skupper router and configures an encrypted, mutual-TLS overlay for all subsequent service traffic.

Exposing Ryax Services

The Worker requires access to the image registry and messaging broker. On the master site, these are made available over Skupper by exposing the in-cluster services:

```
skupper -n ryaxns expose service minio --address minio-ext
skupper -n ryaxns expose service ryax-broker --address ryax-broker-ext
```

Operator secrets (e.g., Docker credentials, broker tokens) are then extracted from the master namespace and applied to the remote cluster's ***ryaxns*** namespace.

Helm Deployment of the Worker

With networking and secrets in place, the Ryax Worker chart can then be installed via:

```
helm upgrade --install ryax-worker oci://registry.ryax.org/release-
charts/worker \
  --values worker.yaml -n ryaxns
```

Using ***helm upgrade --install*** ensures idempotent deployment: first install if absent, or an upgrade if already present. Once the Worker pods are Running, the site will automatically register with the central Ryax UI and appear as a target for Kubernetes-based Action execution.

4.4.4 Multi-constraint and multi-objective scheduling

RYAX scheduler employs a multi-stage, multi-criteria optimization framework that evaluates available node pools across cloud providers and on-premise sites. Each node pool is scored from 0 to 100 on three orthogonal axes—performance, energy efficiency, and cost—and these scores drive both the initial filtering of candidates and the final placement decision via a weighted objective function. Ryax implements this through a sophisticated multi-stage scheduling system that combines constraint satisfaction with objective-based optimization that balances user priorities and infrastructure characteristics. The initial scheduling is based on simple first-fit algorithms but more sophisticated approaches will be brought through the integration with EMPYREAN's Service Orchestration and Decision Engine. Furthermore, integration with real-time telemetry may be performed to allow the refinement of all scores over time, enabling adaptive, data-driven workload distribution.

Scoring System

The **Performance Score** reflects raw computational capability and real-world throughput. Initially seeded from hardware specifications (e.g., GPU model, network bandwidth), it is continuously calibrated against actual task execution times and throughput metrics to capture application-specific behavior.

The **Energy Score** represents datacenter power efficiency. It incorporates published PUE (Power Usage Effectiveness) values, cooling effectiveness, and observed resource utilization. Node pools in regions with advanced cooling or high renewable energy penetration accordingly receive higher scores, incentivizing greener scheduling choices.

The **Cost Score** aggregates financial factors such as on-demand pricing, sustained-use discounts, and resource granularity (e.g., support for fine-grained GPU partitioning via NVIDIA MIG). Pools offering finer allocation units and volume discounts score higher, ensuring that workloads are matched to the most cost-effective infrastructure.

The implementation has been performed in a modular way allowing scores to be adapted or other scores to be added (e.g. network congestion or latency scores).

Multi-Criteria Optimization Process

The initial filtering is done using the following:

1. **Architecture Compatibility**: Discards node pools that do not match required CPU architectures (x86_64, arm64, etc.).
2. **Resource Availability**: Verifies that each candidate pool can satisfy requested CPU, memory, GPU, and storage quotas.
3. **Constraint Enforcement**: Applies any user-specified placement constraints (e.g., geographic location, compliance zones).

Then the Objective Function Computation is done through the following path. For each remaining node pool, the scheduler computes a composite score using the formula:

```
total_score = (perf_weight  × perf_score) +
              (energy_weight × energy_score) +
              (cost_weight   × cost_score)
```

User-defined weights allow dynamic prioritization—favoring performance, cost savings, or energy efficiency as needed. The final placement routine integrates temporal and spatial analyses:

- Builds a **temporal map** by correlating average function durations with container startup times to anticipate peak resource demands.
- Weighs **dynamic energy patterns**, combining static power draw with measured utilization spikes.
- Executes a first-fit search across candidate pools based on the computed ***total_score***, ensuring that each workload is assigned to the node pool that maximizes the weighted objective while honoring capacity and policy constraints.

This initial version of RYAX scheduling introduces a 2-level scheduling schema: one to select the site to run each action and one within the site itself performed by the underlying Kubernetes cluster (or SLURM in the case of the HPC site). It brings an initial first-fit algorithm and the whole framework is now setup to open the way for more sophisticated algorithms through the integration with EMPYREAN's Decision Engine and Service Orchestration.

4.5 Decentralised and Distributed Data Management Protocol

The steady increase in network-connected devices in the last couple of years has brought a new level of heterogeneity concerning computing, storage, and communication capabilities and new challenges concerning the scale at which data is produced and needs to be consumed.

The goal is to interconnect a system that spans from the data center down to the microcontroller and needs to smoothly operate across the continuum with increased performance, efficiency, improved privacy, and security, thus data should be processed as close as possible to the source, while at the same time not hindering access to geographically remote applications. In other terms, we are experiencing an architectural switch from cloud-centric paradigms in which data is stored, processed, and retrieved from the cloud to an edge-

centric paradigm where data is stored and processed where it makes the most sense for performance, energy efficiency, and security matters.

EMPYREAN leverages the Eclipse Zenoh[17] open-source project. Zenoh has been designed to address the requirements of those applications that need to deal with data in movement, data at rest, and computation in a scalable, efficient, and location-transparent data manner. Eclipse Zenoh unifies data in motion, data in use, data at rest, and computations. It carefully blends traditional pub/sub with geo-distributed storages, queries, and computations, while retaining a level of time and space efficiency higher than the mainstream stacks.

Key features include:

- Zenoh provides a small set of primitives to deal with data in motion, data at rest, and computations. i.e. pub, sub, queriable, query.
- Give total control of storage location and back-end technology integration.
- Minimize network overhead – the minimal wire overhead of a data message a data message is 5 bytes.
- Support extremely constrained devices – its footprint on Arduino Uno is 300 bytes.
- Supports devices with low duty-cycle by allowing the negotiation of data exchange modes and schedules.
- Provides a rich set of abstractions for distributing, querying, and storing data along the entire system.
- Provide extremely low latency and high throughput. We also provide analytical and empirical comparison of Zenoh ’s efficiency against mainstream protocols such as DDS and MQTT.
- Make available a lower-level API that gives full control on Zenoh primitives to the developers.

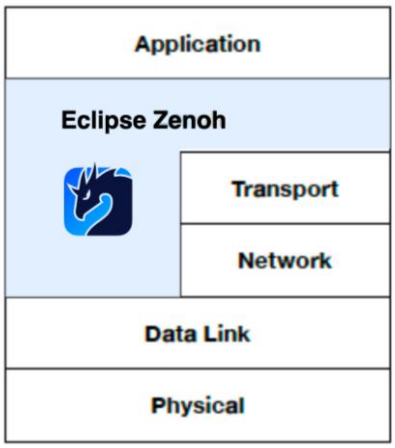

Figure 5: Eclipse Zenoh protocol stack positioning.

[17] https://github.com/eclipse-zenoh/zenoh

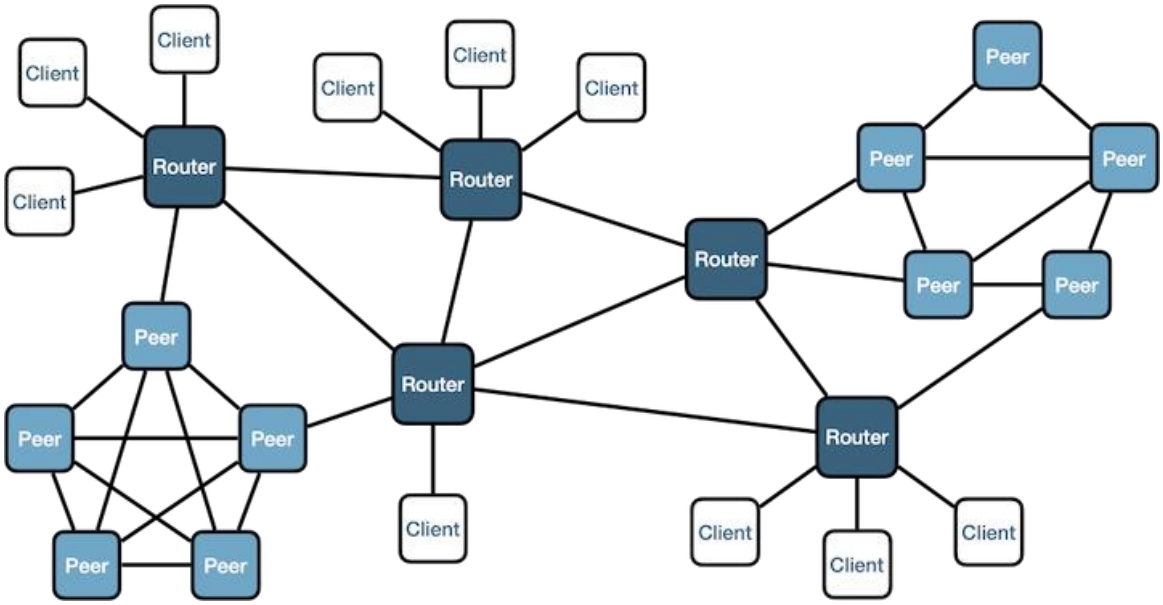

Figure 6: Eclipse Zenoh supported topologies.

Protocol abstractions

- **Resources, Key Expression, and Selectors:** Zenoh operates over resources. A resource is a (key, value) tuple, where the key is an array of arrays of characters. When representing keys we usually use the "/" as a separator. Thus, *home/kitchen/sensor/temp* is a resource's key. A set of keys can be expressed using a key selector, which may include $*$ or $**$ represent a wildcard.
- **Publisher, Subscriber, and Queryable:** The Zenoh protocol defines three different kinds of network entities, publisher, subscribers, and queryables. A publisher should be thought of as the source for resources matching key expressions. As an example, a publisher could be defined for a key, such as *home/kitchen/sensor/temp*, or for a set of keys, such as *home/kitchen/sensor/** or *home/kitchen/*** . Symmetrically, a subscriber should be thought, of as a sink for resources matching key expressions. As an example, a subscriber could be defined for a key, such

  as *home/kitchen/sensor/temp*, or for a set of keys, such as *home/**/sensor/** . A **queryable** should be thought of as a well for resources whose key matches a key expression. As such a queryable for *home/kitchen/*** essentially promises that if queried for keys that match this key expression it will have something to say.
- **Primitives:** Zenoh has a very constrained number of orthogonal primitives, these are:

  Declarations: these primitives, namely, declare_resource, declare_publisher, declare_subscriber, and declare_queryable allow declaring a resource, a publisher a subscriber, and a queryable respectively.

  Producing Data: the put operation is used to produce a (key, value). This operation provides options that allow to specify the congestion control applied to it, the associated priority, and a few other non-functional properties.

Deleting Data: zenoh provides a delete operation that makes it possible to indicate the desire for a resource shall be deleted.

Query. Zenoh provides a get operation that allows to issuing of a query. This query will be served by a set of queryable that cover, in a set-theoretical sense, the key expression portion of the query.

The Eclipse Zenoh protocol does not impose any topological constraints on how an application may communicate. As shown in Figure 6, Zenoh supports peer-to-peer over complete connectivity graphs as well as over arbitrary mesh. It supports routed communication and both routers as well as peers, can broker communication for clients. This generality allows the developer to support a multitude of use cases and to scale the protocol at the Internet scale. Finally, it is worth mentioning that Zenoh's routers are software-based and can run very efficiently on a Raspberry Pi.

At the moment of writing this deliverable, Eclipse Zenoh is in the release v1.3.0, and the API has been stabilized. For further information please refer to the official Eclipse Foundation Github repository[18].

4.5.1 Towards the initial prototype of Ryax/Zenoh Integration

In particular, there are several integration scenarios under evaluation between ZettaScale's Eclipse Zenoh and Ryax's intelligent workflow orchestrator, as illustrated in Figure 7.

a. RPC-based Action Interconnectivity

One potential integration approach is to leverage Eclipse Zenoh as a generic Remote Procedure Call (RPC) mechanism for interconnecting Ryax actions across distributed orchestrators. This would enable efficient and low-latency communication between workflow components, even across disparate locations, supporting more resilient and scalable workflow execution.

b. Unified Data Layer via Zenoh-backed Storage

Another potential integration point lies in Zenoh's support for cloud and object storage systems, including the Amazon S3[19] buckets or MinIO[20] object storage through the Zenoh-backend-s3[21] plugin, or the use of SQL-based data storage backends. This storage integration offers a unified and efficient data abstraction layer across the edge-cloud continuum, enabling Ryax workflows to interact seamlessly with distributed and heterogeneous data stores.

[18] https://github.com/eclipse-zenohq/zenoh

[19] https://aws.amazon.com/fr/s3/

[20] https://min.io/

[21] https://github.com/eclipse-zenoh/zenoh-backend-s3

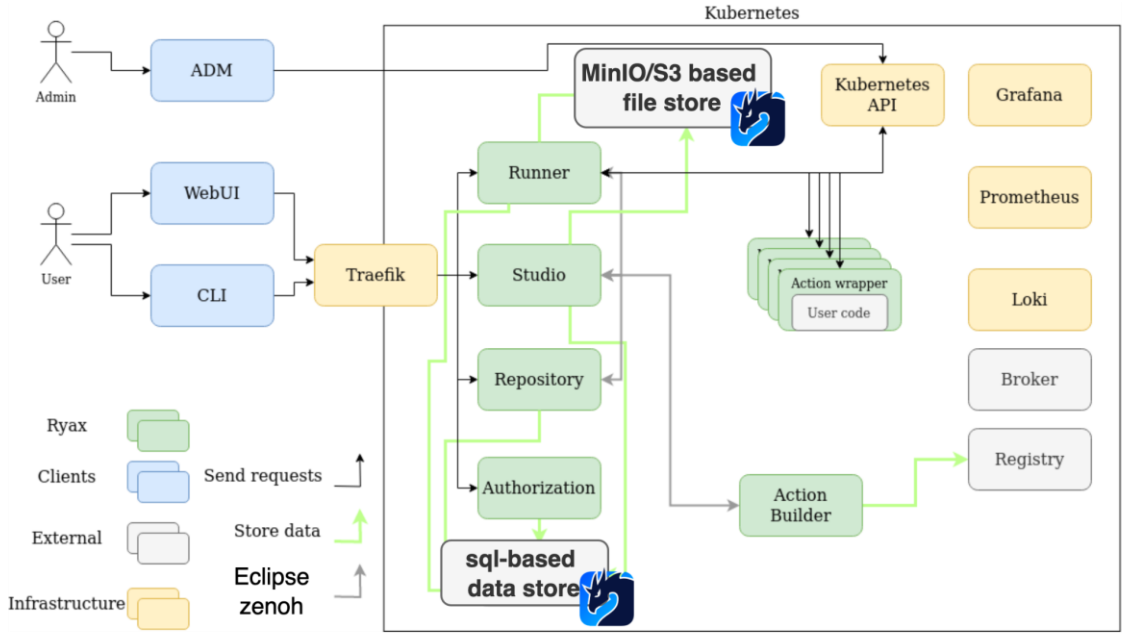

Figure 7: Initial prototype of Ryax and Zenoh integration architecture

Once the initial integration has been carried out, we plan to build an application that adopts the pipes and filters programming paradigm, which consists of splitting a complex task into a well-defined series of simple and independent processing steps called filters. While this pattern has proven to work well when the application runs on a single and isolated machine, forthcoming robotics, automotive, and edge computing scenarios are showing that (i) computations will need to span across multiple locations, from an onboard vehicle and up to the edge of the cloud, and (ii) complex and dynamic data interactions need to be taken into account, as shown in Figure 8. Data Flow Programming (DFP) appears as a potential candidate, since it generalizes pipes-and-filters by allowing applications to be represented as directed graph of components, called operators, as opposed to a linear pipeline.

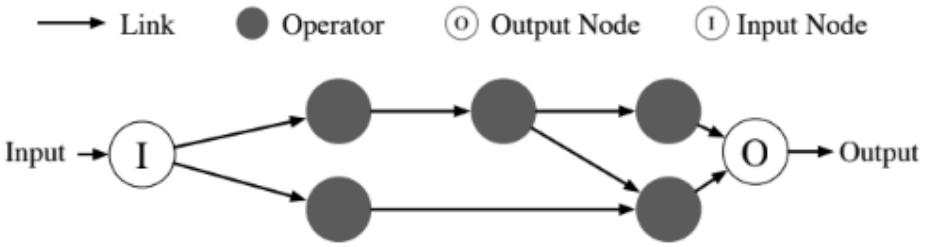

Figure 8: A generic data-flow programming in Zenoh-flow[22]

[22] Baldoni, G., Loudet, J., Guimarães, C., Nair, S. and Corsaro, A., 2023, October. A Data Flow Programming Framework for 6G-Enabled Internet of Things Applications. In 2023 IEEE 9th World Forum on Internet of Things (WF-IoT) (pp. 1-8). IEEE.

This dataflow programming paradigm could be integrated with RYAX's to operate complex event-driven streaming data flows, commonly used in Industrial IoT (IIoT) or other manufacturing 4.0 applications. Popular use cases usually involve collecting data from various machine sensors in real-time, cleaning it, filtering it, injecting it into a processing algorithm, and outputting valuable results as quickly as possible. Predictive workflows are widely used in industrial contexts to anticipate machine failures, production peaks, or plan maintenance. Ryax is well-suited for this kind of application. Collaborating efforts are ongoing to find the right collaboration mechanisms in the context of EMPYREAN and the defined UCs.

4.6 Conclusions

The enhancements provided in Ryax for the multi-site support and the framework for multi-constraint and multi-objective scheduling is now implemented and is available in the production version since the Ryax release 24.10.0[23]. The new enhancements and integrations with Decision Engine, Service Orchestrator, Zenoh and Telemetry services in the context of Empyrean are currently being developed and will appear in follow-up releases of Ryax.

[23] https://github.com/RyaxTech/ryax-engine/releases/tag/24.10.0

5 Analytics-Friendly Distributed Storage

5.1 Overview and Novel Features

Beyond the object storage solution developed in T3.2, the EMPYREAN platform introduces a second, IoT-focused storage solution: the Analytics-Friendly Distributed Storage. This system combines the cost-effectiveness and reliability of erasure-coded distributed storage with the ability to efficiently execute queries over time-series data. To achieve this capability, EMPYREAN is developing a novel data alignment and coding scheme.

Conventional state-of-the-art object storage systems support analytics-type workloads in a limited fashion. During query execution, these systems typically have to load, parse, and scan the entire objects to evaluate predicates and produce the correct results. If the objects have a predefined schema and data is replicated, byte-level access is possible, reducing the amount of data retrieved. However, many state-of-the-art storage systems rely on erasure coding instead of replication, as it provides the same level of reliability with significantly less storage overhead. To support data analytics, such storage systems have two suboptimal options: either replicate queryable files—incurring high storage costs—or reconstruct erasure-coded files on-the-fly during query evaluation—resulting in performance degradation and increased data transfer. This is because erasure-coded fragments generally need to be fully decoded even when only a small subset is required for the query.

EMPYREAN aims to advance state-of-the-art by introducing a novel storage scheme. Through a set of judiciously designed data structures and data alignment techniques, coupled with Random Linear Network Coding (RLNC), we avoid having to reconstruct complete files when evaluating queries. Critically, access to individual bytes of data can be achieved with very little to no overhead compared to replication. At the same time, we maintain the benefits of erasure coding by leveraging the Edge Storage system's core storage architecture.

A second significant contribution to the field will be explored in EMPYREAN: Can data be compressed, while maintaining some level of byte-level access? To answer this question, we have begun evaluating Generalized Data Deduplication (GDD), a special form of data deduplication that can be effective on smaller data volumes, compared to traditional deduplication techniques. Using deduplication to compress, instead of a traditional compression algorithm like DEFLATE, opens up the possibility of having a deterministic relationship between bytes in the uncompressed and compressed data. We also aim to determine whether this technique can be applied in conjunction with erasure coding to further enhance the system's cost-effectiveness.

This endeavour is a part of CC's basic research efforts. As such, it has a relatively low TRL as a goal. Instead, we aim to establish the feasibility of the aforementioned techniques and perform an initial evaluation of their effectiveness.

5.2 Relation to Project Objectives and KPIs

There are two technical KPIs - T4.4 and T4.5- that pertain to the Analytics-friendly Distributed Storage. Both require a deep scientific evaluation of the concepts involved and are driven by the very real need to limit costs and wasted time in transferring data from cloud locations. We seek formulas with proofs that characterize the overhead, along with hopefully minimal constraints on the schema or the queries.

Table 1: EMPYREAN Technical KPIs related to the Analytics-friendly Distributed Storage

ID	Indicator	Success Criteria	Objective
T4.4	Ensure that the amount of erasure coded data retrieved for a query scales linearly.	-	Obj.4
T4.5	Provide an upper limit on the overhead incurred, that is either constant or a linear function.	-	Obj.4

The component is relevant in achieving several project objectives:

- Most of the requirements described for the Edge Storage Service.
- F_DCM.2 is directly tied to this service, describing the requirements which were used during the design phase.

5.3 Implementation

Most of the Analytics-friendly Distributed Storage features are implemented as a module of the **Edge Storage Gateway.** This component provides the public APIs and performs most of data processing, including transformations, compression, and erasure coding. The gateway is also in charge of distributing the erasure coded fragments to the cloud storage locations. A detailed description of this component is available in deliverable D3.1 (M15). In order to have byte-level access to these fragments, a set of **Cloud Lambdas** offers a convenient and effective solution. The HTTP standard has a solution in the form of Range requests[24], but unfortunately cloud object storage services typically do not fully support this feature. Most only allow the client to specify a single range of bytes to retrieve. Furthermore, the extra headers and boundary bytes make HTTP Range requests very inefficient for this application.

[24] HTTP range requests: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Range_requests

5.3.1 Components shared with the Edge Storage Service

The Analytics-friendly Distributed Storage shares its core storage functionality with the Edge Storage Service. The Edge Storage Gateway (ESG) provides the user-facing APIs and performs most of data processing. The additional functionality required to serve IoT data queries is provided through a separate module. The SkyFlok.com backend handles tasks such as user management and authentication, file metadata, and storage resource management, to name a few.

For the moment, storage locations are limited to those provided by the three largest cloud providers: Amazon, Google, and Microsoft. It would also be possible to support edge locations, but there is less incentive to do so as there are no transfer costs inside associations, and connections should, generally, have high bandwidth and low latency.

5.3.1.1 Cloud Lambdas

A key difference from the Edge Storage Gateway is how data is read from the storage locations. The ESG offers object storage semantics and thus - with the exception of HTTP Range queries - always returns the entire object as part of an S3 GetObject request. Analytics-related queries, on the other hand, require byte-level access to the erasure coded fragments. To accomplish this, we introduced a set of new components on the cloud storage locations.

For each major cloud storage provider, Amazon, Google, and Microsoft, we have implemented a FaaS component, AWS Lambda, Cloud Function and Azure Function respectively. These lambdas are deployed in the same cloud region as the object storage services. When the ESG requests specific bytes from the storage, it uses a standard HTTP request with a *Range* header, as defined in RFC7233[25]. The lambda supports the full multipart syntax, e.g., "Range: bytes=0-50, 100-150", in contrast with the storage provider APIs themselves, which either have limited or no support of this feature. The lambdas parse the header, download the originally requested object from the storage service and return only the requested bytes. To further increase efficiency, the response uses a custom format, which has significantly less overhead compared to what is defined by the HTTP standard.

The data transfer between the storage service and the lambda does not leave the region and is, therefore, free. Thus, by only returning the typically small number of specified bytes, the billed data egress can be significantly reduced in exchange for a small computational overhead.

[25] https://datatracker.ietf.org/doc/html/rfc7233

5.3.1.2 Workflows

Three separate workflows can be defined based on the time series data's lifecycle.

To begin with, the application developer must provide a schema that defines the different dimensions of the data, specifying the type and number of bytes required for each field. Conceptually, this process is similar to using a Data Definition Language (DDL) in a relational database (e.g., SQL). However, in EMPYREAN, schema definition is facilitated through a simplified JSON-based interface, enabling intuitive and flexible specification of data layouts suitable for time-series workloads.

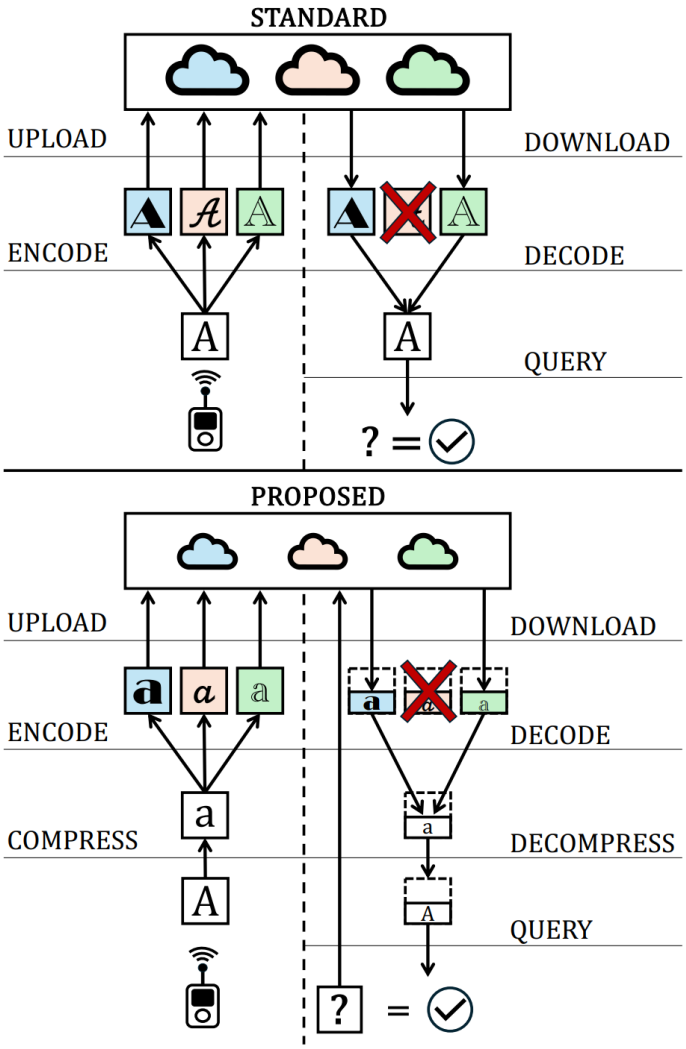

Figure 9: An illustration of the data ingest and query workflows of the Analytics-friendly Distributed Storage. Top image shows a naive approach which retrieves the entire data during queries. Bottom image shows our proposed schema, with byte-level access and compression.

Once the schema is established, data can be stored in batches. Ideally, batches should be at least on the order of megabytes. Smaller batches may lead to reduced performance and increased metadata storage. This phase is conceptually equivalent to SQL *INSERT* operations.

After data ingestion, the data becomes available for querying. Applications may provide filtering conditions along with a list of projected dimensions. This is equivalent to the basic SQL *SELECT* operation, though many features such as aggregation or cursors are not supported.

Figure 9 shows a schematic representation of the data ingestion and querying processes.

5.3.1.3 Results

We have conducted an initial evaluation of the system[26]. The paper goes into great detail explaining the different techniques we proposed and evaluated over several different datasets. Figure 10 shows results for a dataset[27] with 500k+ rows and several columns.

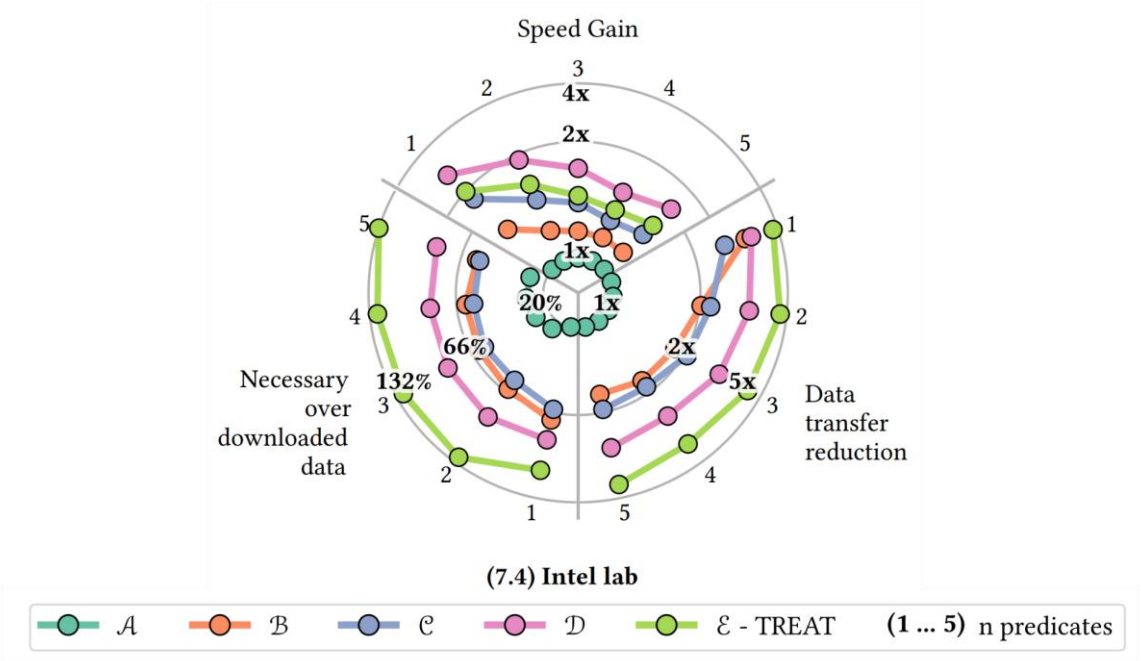

Figure 10: An initial evaluation of our proposed technique over a representative dataset. Three metrics are shown on the different parts of the circle. The different letters correspond to increasingly more complex techniques. A is the naive baseline approach of retrieving the entire erasure coded object and E - TREAT is the most complex approach which downloads only what is needed of the compressed and rearranged dataset, while filtering out unfeasible deduplicated (compressed) base ids.

[26] Francesco Taurone, Marcell Fehér, Márton Sipos, and Daniel E. Lucani. 2024. TREAT - Two wRongs makE A righT: efficient distributed storage and queries of IoT datasets with erasure coding and compression. In Proceedings of the 18th ACM International Conference on Distributed and Event-based Systems (DEBS '24). Association for Computing Machinery, New York, NY, USA, 147–158. https://doi.org/10.1145/3629104.3666039

[27] Berkeley Research Lab. 2004. Intel Berkeley Research Lab Sensor Data. https://db.csail.mit.edu/labdata/labdata.html

A is the naive baseline approach of retrieving the entire erasure coded object and E - TREAT is the most complex approach which downloads only what is needed of the compressed and rearranged dataset, while filtering out unfeasible deduplicated (compressed) base ids.

The main goal of making queries cost-efficient is best achieved with the most complex technique, both in terms of the absolute data transfer reduction and the relative query efficiency (shown as necessary over downloaded data on the figure and expressing the overhead of fetching 'unneeded' data). In terms of the speed gain that can be expected, the picture is less clear as the added complexity can lead to worse performance. The paper performs the evaluation over five different datasets and goes into details on each of the proposed techniques.

5.4 Public APIs

The Analytics-friendly Distributed Storage will provide three separate APIs, each tied to the lifecycle of the data:

Schema definition API

Each dataset that is to be stored needs to have a well-defined schema that specifies the dimensions of the data, as well as the type and size of each dimension. This will likely be provided by the user as a JSON file. We have previously implemented a more complex API to manage schemas, but for the moment at least, it is superfluous, when considering the project's objectives.

Data ingest API

Uploading data will likely be handled using a simple REST API. Applications must also specify the dataset to which they are uploading data.

Query API

Data retrieval will happen using queries, not too dissimilar to SQL SELECT statements. Applications will use a REST API, specifying: (i) the queried dataset, (ii) filtering conditions, such as the dimensions, comparison values and relationship between the conditions, and (iii) the projected dimensions that will make up the result set.

5.5 Integration with EMPYREAN Platform services

The Analytics-friendly Distributed Storage is implemented through the Edge Storage Service, described in Deliverable 3.1 (M15), submitted concurrently with the current document. The two offer substantially different services to EMPYREAN applications, reflected in their respective APIs. However, the core storage layer is shared, bringing the core benefits such as hybrid storage defined using storage policies and erasure coding to both.

5.6 Relation to use cases

The Analytics-friendly Distributed Storage provides a way for the three use cases to cost-effectively store time series data. By erasure coding and compressing data, it can be stored reliably, with high availability over a long period of time. Subsequently, when analytics workloads need to be run on the accumulated data to evaluate different queries, parts of the time series can be retrieved with relatively little overhead, without the need to decode and decompress the entire series or entire erasure coded blocks in advance. This provides a clear advantage over current systems, where application designers must choose between having the cost, availability and reliability benefits of erasure coding and query efficiency.

UC1 - Anomaly Detection in Robotic Machining Cells

The first use case stores measurements from machining equipment as time series data. Processes try to detect anomalous events based on this data in order to determine when a machine component has reached the end of its lifetime and must be replaced. The accuracy of this process is paramount, as machining objects with defective/worn out tools carries a heavy financial cost. One possible way the Analytics-friendly Distributed Storage can further enhance this process is by reliably storing the data over a long period of time. When a new type of anomaly is found, previously stored data can be queried to ascertain whether the event had happened before. Following this, measurements surrounding these past events can be retrieved efficiently.

UC2 - Proximal Sensing in Agriculture Fields

The second use case stores different types of information regarding the state of agricultural fields. Measurements like soil organic carbon levels are performed over a span of several years and thus should be stored reliably and cost-efficiently. When the need arises to present trends across seasons and different years, the data can be queried efficiently using the Analytics-friendly Distributed Storage. With a well-defined schema, only the dimensions that are relevant to the analysis need be retrieved. There is no need to know which of the dimensions will be relevant at the time of data ingest.

Another option is to train machine learning models from the measurement datasets. For example, following an initial analysis on which dimensions are most important (e.g., using Principal Component Analysis), the input to the model can be queried, projecting only the dimensions with the greatest weights. Similarly, if the need arises to train a model based only on the parts of the dataset that meet some criteria (e.g. only when temperatures are above a certain threshold), then these parts can be efficiently retrieved by setting the appropriate filter.

6 Action Packaging

6.1 Introduction

EMPYREAN leverages the Nix[28] functional package manager as a foundational component of its unified action packaging tool. This approach enables a declarative and reproducible process for building lightweight, OCI-compliant containers, ensuring consistency across diverse environments.

The EMPYREAN action packaging tool, which is part of Ryax Workflow Engine, has been significantly enhanced with various new features to support seamless packaging of diverse workloads. Key enhancements include: (i) support for multiple architectures, extending beyond x86 architectures to include arm64, thus enabling deployment across a broader range of edge and embedded platform, (ii) improved reproducibility through finer-grained control over environment dependencies, and (iii) tight integration with NUBIS' unikernel builder, enabling packaging of microservices, serverless functions, and even IoT firmware into OCI-compatible images.

These enhancements promote architecture-agnostic deployment across the edge-cloud continuum and support a modular, flexible application design in polyglot environments.

6.2 Background

The action packaging tool has been implemented as part of the Ryax Workflow Engine, supporting the packaging of actions in several languages. The initial focus was on Python3, NodeJS, and C#. Within Ryax, an action or a trigger requires at least two files: (i) the python code in a *.py* file, and (ii) the Ryax metadata file: ***ryax_metadata.yaml***.

If the action has external dependencies, such as PyPi packages or anything outside the Python standard library (e.g., pandas, TensorFlow), these must be listed in a standard *requirements.txt* file. Each dependency should appear on a separate line in plain text. To specify the Python version, the *ryax_metadata.yaml* file should include the filed ***spec.option.python.version***, following the format *major.minor* (e.g., 3.11). The action packaging tool automatically looks for a *requirements.txt* file at the root of the action directory. This file informs the tool about the external Python libraries needed to properly package and execute the user's code. An action description requires the inclusion of a *ryax_metadata.yaml* file at the root of the action directory. This file provides a high-level definition of the action, including its inputs and outputs. It follows the YAML format and must conform to the schema expected by the Ryax Workflow Engine.

[28] https://nixos.org

Below is an example of a basic *ryax_metadata.yaml* file:

```
apiVersion: "ryax.tech/v2.0"
kind: Processor
spec:
  id: tfdetection
  human_name: Tag objects
  description: "Tag detected objects on images using Tensorflow"
  type: python3
  version: "1.0"
  logo: mylogo.png
  resources:
    cpu: 2
    memory: 4G
  dependencies:
  - opencv
  inputs:
  - help: Model used to tag the images
    human_name: Model name
    name: model
    type: enum
    enum_values:
     - ssdlite_mobilenet_v2_coco_2018_05_09
     - mask_rcnn_inception_v2_coco_2018_01_28
    default_value: ssdlite_mobilenet_v2_coco_2018_05_09
  - help: An image to be tagged; in any format accepted by OpenCV
    human_name: Image
    name: image
    type: file
  outputs:
  - help: Path of the tagged image
    human_name: Tagged image
    name: tagged_image
    type: file
```

Below is the reference definition of each field in the *ryax_metadata.yaml* file:

- kind: to tell the *kind* of the action: accepted values are **Trigger, Processor, Publisher**
- id: unique name of the action (must contains only alphanumeric characters and dash)
- version: unique version of the action (must contains only alphanumeric characters and dash)
- human_name: a short name, readable by a human
- description: a description of the action
- type: the programming language. Supported values are: python3, python3-cuda, nodejs, csharp-dotnet6

- logo (optional): a relative path to a logo file
- resources: to set the amount of resources that your Action will ask by default
    - cpu (float): the number of cpu core (can be a fraction of cpu, i.e O.5)
    - memory (string | int): the amount of memory in bytes. Allowed units are (K,M,G,T)
    - time (string | int): the maximum time in seconds that should be allocated to the action execution before it is cancelled. Allowed units are (s,m,h,d)
    - gpu (int): the number of GPU
- dependencies (optional list(string)): list of packages to add to your Action's environment (See above for more details)
- categories (optional list(string)): list of labels to be added to your action. This is only used as metadata (e.g. filtering in the dashboard)
- dynamic_outputs (boolean): *optional, and only for advanced usage*. (default to false). Only triggers may have dynamic outputs. This is useful in some cases such as when using an online form which can be filled out, to trigger workflows. This feature allows for the reuse of all the code for that trigger, while also allowing users to re-define the fields on that form in different workflow
- inputs: list of inputs values injected in the action context:
    - name: the name of your variable in your code. Is must not contain spaces or special characters except for _. The input dict of you handler contains an entry with this name
    - human_name: a human readable name
    - help: describes your variable usage
    - type: the action IO types. See the following table
    - optional: whether your IO is optional or not. If true it will accept a None value
    - enum_values (Only for enum type): a list of values accepted by your enum
    - default_value (Optional and only for inputs): give a default value for this input
- outputs: list of outputs returned by the action. Uses the same format as inputs

6.3 Ryax Types

Each input and output declared in *ryax_metadata.yaml* must specify a type, chosen from the supported types that are listed in Table 2.

Table 2: Available input and output types in Ryax Actions

Type	Description
string	String of characters
longstring	Use for long text (Larger UI inputs)
password	String hidden on the UI
integer	64-bit integer
float	floating-point number
boolean	True of False
enum	Enumeration with a list of possible values
file	File (imported and exported by Ryax)
directory	Directory containing a set of files (imported and exported by Ryax)

Ones the action is built, Ryax executes it by importing a Python module and running a specific function within it.

Depending on the ***kind*** of the action, two different filenames are used:

- ***ryax_run.py*** for triggers
- ***ryax_handler.py*** for all other actions

When creating an action, Ryax copies all the files located in the action directory. This means that if the code is split across multiple Python files or includes resource files, they will be copied into the action.

If the action has non-python dependencies (called “binary dependencies”), these can be declared in the ***dependencies*** field of the ***ryax_metadata.yaml*** file. Ryax uses the Nix package manager to install these binary dependencies. Users can search for available packages and supported versions using the nixpkgs[29] search tool.

For example, to add *git* and *opencv* to an action, users may add:

```
spec:
  dependencies:
    - git
    - opencv
```

[29] https://search.nixos.org/packages

6.4 EMPYREAN Developments

6.4.1 Finer dependencies control

One of the key enhancements implemented in the context of EMPYREAN is the improved control over dependency management during the packaging process. In this context, we have enhanced the action packaging tool to enable the declaration of NIX packages and overlays to be used for the particular environment.

As part of this enhancement, users can now define the version of nixpkgs repository to be used by specifying the field ***spec.options.nixpkgs.version*** in the ***ryax_metadata.yaml*** file. The value can be either a branch name, a tag or a specific commit SHA from the nixpkgs repository. For example, to use the unstable branch of nixgpks[30], the configuration would look like:

```
spec:
  options:
    nixpkgs:
      version: nixos-unstable
```

In cases where users need to customize some packages or include new ones not available in the nixpkgs repository, Ryax supports the definition of Nix Overlays. To do so, users need to create an ***overlays.nix*** file in the action directory. This file will be automatically detected and loaded during the build process, and added to the action environment. To ensure these custom packages are available in the action environment, users have to add them in the binary dependency list of the ***ryax_metadata.yaml*** file.

For example, if an action requires ***opencv*** with ***Tesseract*** support enabled, users can define the ***overlays.nix*** file to customize the action packaging accordingly:

```
[
  (self: super: {
    opencvWithTesseract = self.opencv.override { enableTesseract = true; };
  })
]
```

And in the ***ryax_metadata.yaml***:

```
spec:
  dependencies:
    - opencvWithTesseract
```

[30] The website https://lazama.co.uk/nix-versions helps you find the nixpkgs version that contains the version of a package.

6.4.2 Reproducibility of building process

When building a Ryax action, the dependencies used are specific to the time of building. Without proper version tracking, rebuilding the same action at a later time might pull different versions of these dependencies, which can lead to breaking changes and development issues.

To solve this, Ryax automatically generates a lockfile during the build process. This lockfile captures the complete build environment, ensuring full reproducibility. It includes:

- Python version used
- Versions of all Python dependencies
- Nixpkgs version, and dependencies
- Action version

This lockfile can be committed with the action code and will be used in future builds to ensure that the exact same environment is reproduced.

The first time an action is built, it has no lockfile. To generate it, users need to initiate a build via the web application, in the *Library* section. Once the action build is complete, users can display its information on the library page (Figure 11). To view the lockfile and action details, users have to navigate to the *Library* page and click on the corresponding action image in the right-hand panel.

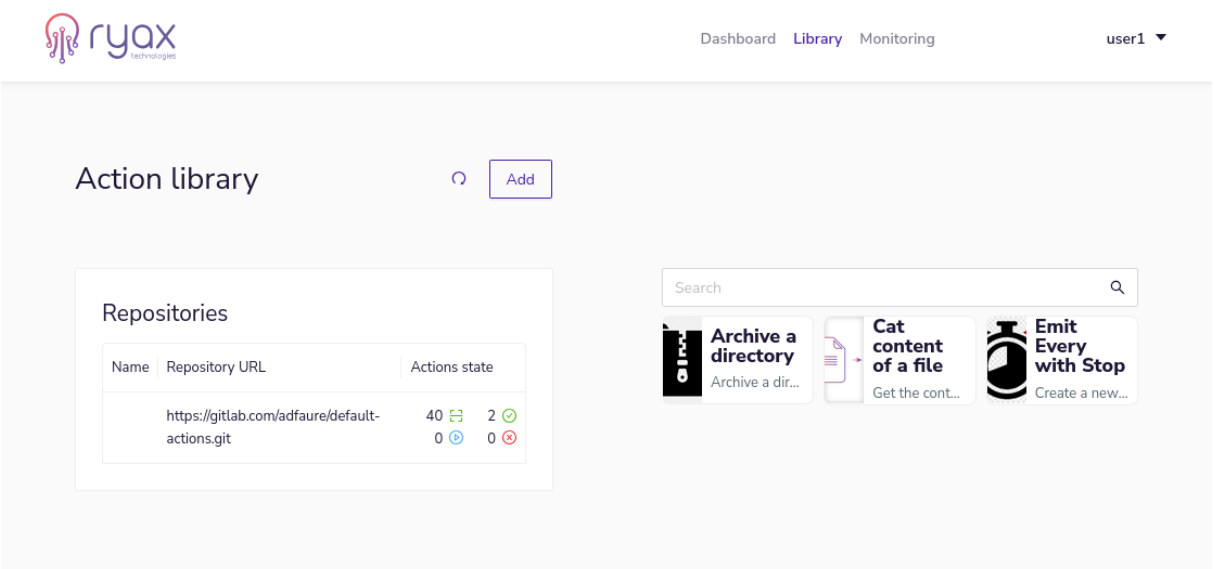

Figure 11: View of the action library featuring the available built actions

The lockfile will be named ***ryax_lock.json*** and by clicking on it we can display the lockfile content of the build (Figure 12).

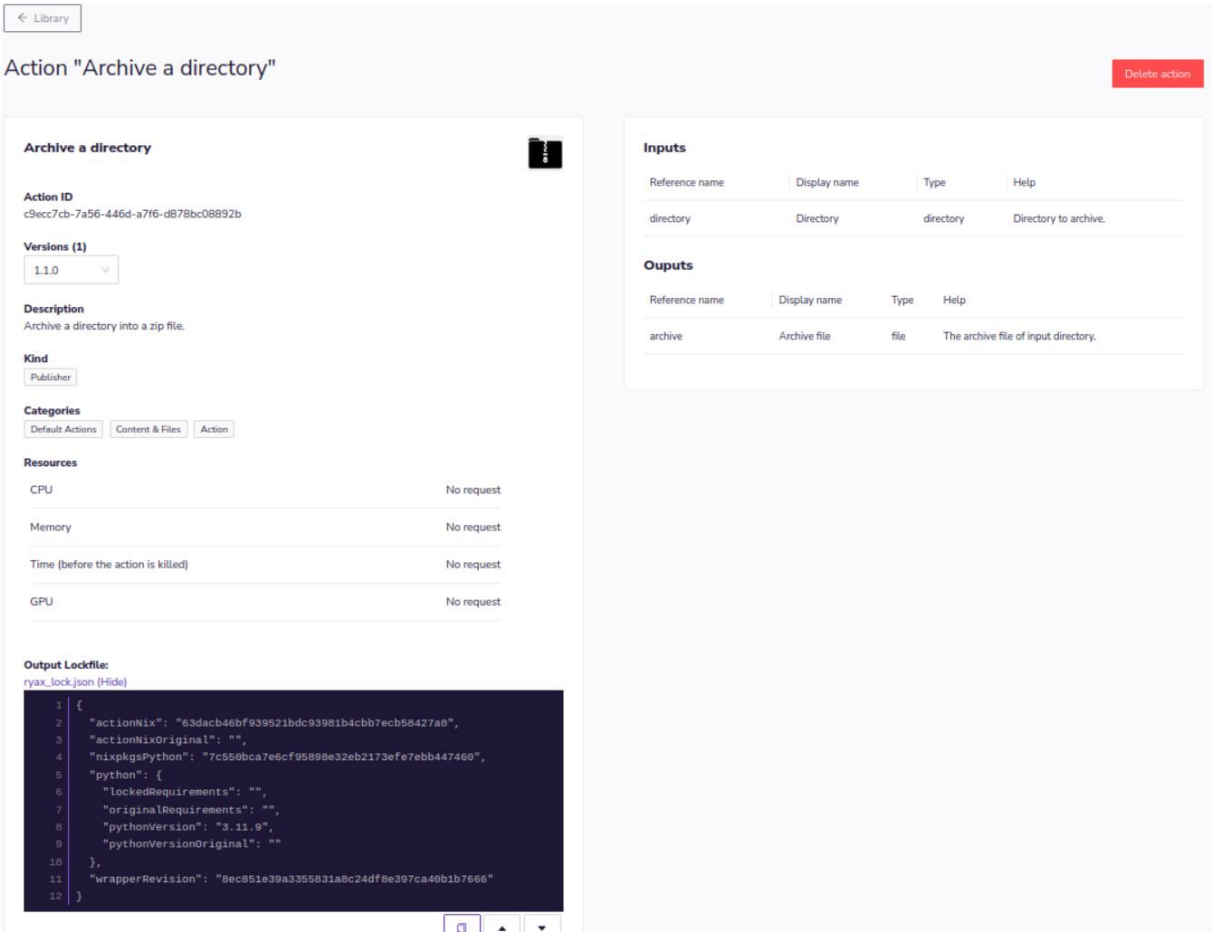

Figure 12: Example of a specific action's details mentioning info such as inputs, outputs, versions and lockfile

The lockfile looks like this:

```
{
  "actionNix": "63dacb46bf939521bdc93981b4cbb7ecb58427a0",
  "actionNixOriginal": "",
  "nixpkgsPython": "7c550bca7e6cf95898e32eb2173efe7ebb447460",
  "python": {
    "lockedRequirements": "",
    "originalRequirements": "",
    "pythonVersion": "3.11.9",
    "pythonVersionOriginal": ""
  },
  "wrapperRevision": "8ec851e39a3355831a8c24df8e397ca40b1b7666"
}
```

When the user needs to use the action for a different instance, the generated lockfile should be committed to the action's folder, under the name ***ryax_lock.json***, with the content obtained from the previous section. During repository scanning on the web interface, the lockfile will be automatically detected as part of the action. To confirm successful detection, users can navigate to the repository page of the action, where the presence of the lock file will be visible, as depicted in the image below.

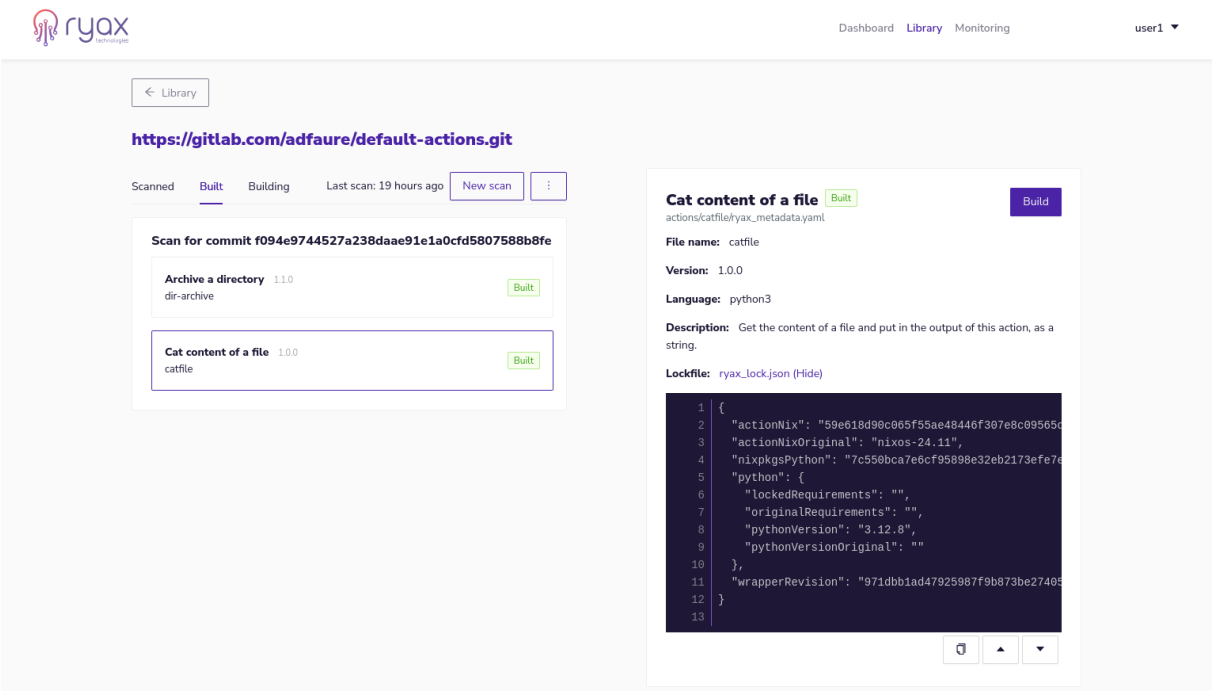

Figure 13: Example of an action with a lockfile ready to be built

While lockfiles ensure reproducible builds, they do not override the dependencies defined in the ***ryax_metadata.yaml*** (see here[31] for details). When a user rebuilds an action, Ryax follows these steps: (1) compares the current dependencies with those captured in the existing lockfile, (2) identifies any differences between the two, and (3) updates the lockfile to reflect the new specifications.

For example, if the lockfile specifies Python 3.9 but ***ryax_metadata.yaml*** specifies Python 3.12 (***spec.options.python.version***), then the build will use Python 3.12 and a new lockfile will be generated reflecting this change.

Moreover, in case one or several fields are not specified, default values will be used and will show in the resulting lockfile.

[31] https://docs.ryax.tech/reference/action_python3.html

6.4.3 Multi-architecture support

Supporting multiple CPU architectures within EMPYREAN Associations and Ryax workflows requires a strategy for building and distributing architecture-specific images. The standard approach involves creating one OCI-compatible image per architecture (e.g., x86_64, ARM64), and then pushing these images to a container registry under a common manifest. This manifest serves as a reference point, allowing the runtime environment to automatically select the appropriate image based on the node's architecture.

While cross-building images at the wrapper level is relatively straightforward, similar to what is done in CI pipelines, integrating this functionality directly into the ***Action Builder*** component presents more complex challenges. Cross-compilation within the Action Builder requires support for *binfmt_misc*, which allows execution of binaries compiled for different architectures. However, this capability must be enabled at the host level, and involves installing and configuring ***qemu-user-static*** and ***binfmt-support***. This setup is not typically feasible in most managed Kubernetes environments due to restrictions on modifying host configurations.

To overcome these limitations, one potential solution is to deploy a dedicated Action Builder service for each architecture present in the cluster. Each service would handle native builds for its specific architecture. While this ensures compatibility and build correctness, it introduces synchronization complexity. In particular, a coordination mechanism is required during assembling and pushing a unified multi-architecture image manifest. One of the builders must eventually aggregate the results and push the manifest, and without careful orchestration this can lead to race conditions and added complexity.

A more scalable and secure alternative leverages Nix's remote build capabilities. In this model, a single centralized Ryax Action Builder service coordinates the build processes but delegates actual compilation to remote Nix builders deployed on nodes with the appropriate hardware architectures. This setup provides several key benefits:

- ***Parallelized native builds*** across architectures, improving build efficiency and reducing total build time.
- ***Decoupling of build orchestration logic from underlying hardware***, increasing deployment flexibility and maintainability.
- ***Enhanced security and scalability***, since the main Action Builder can operate in a minimal, isolated environment while offloading potentially untrusted or resource-intensive builds to remote workers.

By leveraging these advanced capabilities of Nix, EMPYREAN establishes a robust, maintainable, and future-proof mechanism for supporting heterogeneous infrastructure across the edge-cloud continuum.

Towards the goal of supporting multi-architecture builds, development of this feature has already begun. At the time of writing this report, the feature is advanced and the action packaging tool has the ability to build successfully containers in different architectures (x86 or arm64). Tis is achieved by moving the Action Builder microservice to a node that matches the target architecture.

Currently, this feature is in experimental mode. It requires manual intervention from the system administrator of the action packaging tool to move the Action Builder microservice to the appropriate architecture. Despite this limitation, the system can already seamlessly build and produce valid architecture-specific images, on x86 or arm64, from the same code. This makes the feature usable in production environments, provided it is set by default on one architecture from the beginning. Our next objective is to eliminate the need for manual intervention and make this capability completely transparent to end-users. This will be achieved by following the architecture previously outlined, which includes support for automatic delegation to remote Nix builders based on architecture requirements.

Figure 14 shows a screenshot of an action built through the packaging tool of Ryax as part of the actions' repository, featuring in the left the availability of different versions arm64 and x86.

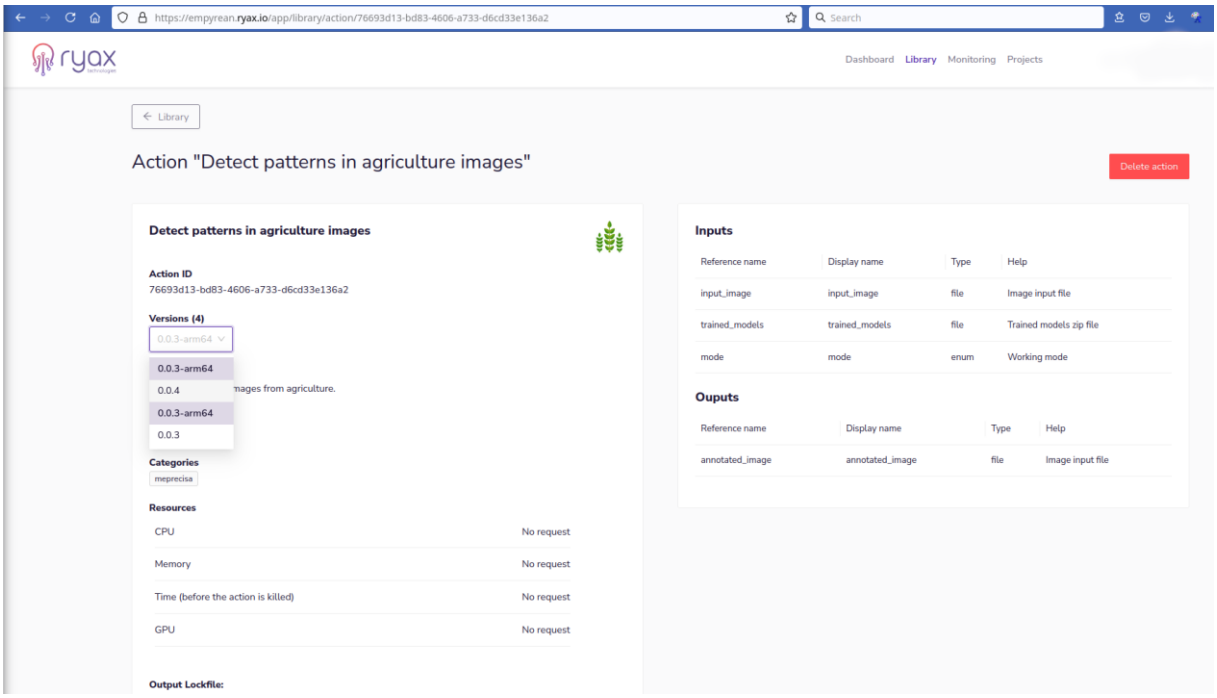

Figure 14: Action with versions of the same code in both x86 and arm64 architectures

7 Unikernels Builder

7.1 Introduction

Traditional deployment models in cloud-native environments assume POSIX-like containers and standardized runtime environments. While this suits many applications, it introduces significant overhead for lightweight, single-purpose workloads such as network services or low-level functions. The Unikernels Builder, developed under the project codename Bunny, introduces a minimalistic and high-performance framework for deploying cloud-native workloads as unikernel images. This system responds directly to EMPYREAN's objective of delivering secure, lightweight, and scalable application deployments across the IoT-edge-cloud continuum. Unlike traditional POSIX containers, which incur runtime and security overheads, unikernels are compiled into single-purpose bootable VM images with no userland, shell, or external dependencies.

Bunny facilitates this process through a fully integrated toolchain that automates the build, packaging, publishing, and deployment of unikernel applications. It adopts OCI-compatible packaging to ensure seamless compatibility with container registries while leveraging orchestration engines like Kubernetes through `urunc` (a unikernel-friendly container runtime). This enables the execution of unikernels alongside traditional workloads using common DevOps pipelines.

Through Bunny, EMPYREAN gains a flexible, composable deployment method that drastically reduces memory usage, boot time, and attack surface—critical features for edge and embedded devices operating in sensitive environments.

7.2 Lightweight Environment Packaging with Bunny

As part of the EMPYREAN platform's efforts to support novel deployment models, Bunny bridges the gap between existing container ecosystems and emerging unikernel-based runtimes. Bunny is designed to streamline the process of building, packaging, and deploying unikernels, enabling transparent integration with CI/CD pipelines and container registries. The toolchain focuses on high reproducibility, minimal developer overhead, and seamless orchestration integration (e.g., Kubernetes) while taking advantage of the efficiency and isolation provided by unikernels.

Prior tools like Unikraft[32] and MirageOS[33] have introduced unikernel build systems, but lack native integration with cloud-native infrastructure and lifecycle tooling. Systems such as OSv[34] attempted POSIX compatibility in VM-based lightweight OSes but incurred extra complexity.

[32] https://unikraft.org
[33] https://mirage.io
[34] https://osv.io

Bunny instead adopts a streamlined approach, tailored to the needs of minimalistic applications and cloud deployment constraints.

The Bunny toolchain introduces several innovations aimed at addressing the unikernel packaging gap:

- ***OCI-compatible Unikernel Packaging***: Bunny packages unikernels in OCI-compliant container images, allowing them to be pushed to standard container registries and executed using compatible runtimes (e.g., urunc).
- ***Unikernel Build Automation***: Provides a declarative, reproducible build pipeline that integrates with modern GitOps workflows and cloud-native CI/CD systems.
- ***Minimal Runtime Metadata***: Bunny strips container metadata down to essentials, replacing filesystem-based resources with static linking and memory-mapped assets. This approach drastically reduces the attack surface and boot latency.
- ***Boot as Main***: Follows the *"boot = main"* philosophy, packaging applications as bootable VMs where the unikernel is the application.
- ***Orchestration-Aware***: Compatible with orchestration layers like Knative[35], enabling on-demand execution and scale-to-zero for unikernel workloads.

7.3 Relation to Project Objectives and KPI

The Bunny toolchain directly contributes to EMPYREAN's objective of redefining workload deployment models across the IoT-Edge-Cloud continuum. The lightweight unikernel execution model is ideal for compute-constrained environments and latency-sensitive services. Bunny's contribution is most closely aligned with the KPIs in the following table.

Table 3: EMPYREAN Technical KPIs related to the Unikernels Builder

ID	Indicator	Success Criteria	Objective
T5.1	Reduce the development time of continuum-native applications	>20% decrease compared to SotA	Obj.5
T5.2	Number of supported hardware architectures for seamless deployment of an application.	>3	Obj.5
T4.3	Reduce memory and space required for deploying application in resource-constrained IoT/Edge devices.	>70% decrease of footprint	Obj.5
T5.5	Reduce memory and storage overhead for function workloads	>30% reduction over container baseline	Obj.5

[35] https://knative.dev/docs/

7.4 Implementation

7.4.1 Internal architecture

The Bunny system is composed of the following core modules, each designed to streamline the process of building, packaging, and deploying unikernel-based applications:

- ***Bunny Builder***: A modular build system comprising a set of rules and templates (currently based on Unikraft) that transforms minimal application logic into a lightweight, bootable unikernel image.
- ***Bunny Packager***: Wraps the generated unikernel binary into a minimal Open Container Initiative (OCI) image, ensuring compatibility with *urunc* and other *containerd-based* container runtimes.
- ***Bunny Publisher***: Handles image distribution by pushes the packaged unikernel image to a container registry (e.g., GitHub Container Registry), using standard tools such as docker *push* or *oras*[36].
- ***Bunny Deploy Helper***: Facilitates deployment by generating the required manifests (e.g., Knative Service YAML descriptions), which reference the packaged unikernel images and can be directly applied to Kubernetes clusters.

7.4.2 Workflows

The Bunny toolchain adheres to a structured lifecycle consisting of four key phases:

1. ***Build Phase***: The application source code is compiled into a standalone unikernel binary using Bunny's predefined build recipes.
2. ***Package Phase***: The resulting binary is encapsulated into a lightweight OCI-compliant image containing only the essential bootloader and metadata required for runtime execution.
3. ***Publish Phase***: The packaged image is pushed to a standard container registry.
4. ***Deploy Phase***: Deployment descriptors (e.g., Kubernetes Custom Resource Definitions) reference the unikernel image, enabling orchestration engines to seamlessly pull and run the unikernel on demand.

Compared to minimal Linux-based containers, Bunny unikernel packages offer significantly smaller image footprints and faster startup times, while providing enhanced isolation guarantees due to the complete absence of userland components, shell access, and traditional operating system services.

[36] https://oras.land

7.5 Public APIs and Integration

7.5.1 APIs

Bunny does not expose runtime APIs, but integrates with DevOps tooling:

- ***CLI Commands***: *bunny build, bunny package, bunny push, bunny deploy*
- ***CI/CD Integration***: GitHub Actions or GitLab CI with YAML snippets
- ***Registry Support***: Works with *docker*, *oras*, and *nerdctl-compatible* registries

7.5.2 Integration with EMPYREAN Platform Services

Bunny integrates with the EMPYREAN orchestration stack through *urunc*, a lightweight unikernel runtime designed interface with *containerd* and Kubernetes. This integration enables EMPYREAN users to develop and deploy bootable unikernel-based images using standard container workflows, treating them as regular OCI containers. As a result, it ensures smooth and consistent deployment across hybrid environments, supporting a mix of traditional containerized services and high-performance unikernel workloads with minimal adaptation.

7.6 Relation to Use Cases

The Bunny toolchain facilitates rapid deployment and high-density execution of isolated functions and microservices, fully aligned with the needs of all three EMPYREAN use cases:

- ***UC1 - Anomaly Detection in Robotic Machining Cells***: Bunny enables the deployment of trusted control logic and lightweight monitoring agents as unikernels with minimal runtime overhead, significantly reducing the system's attack surface and enhancing overall security.
- ***UC2 - Proximal Sensing in Agriculture Fields***: Through the use of minimal unikernel-based agents, Bunny allows for local processing of sensor data directly at the edge. This improves data locality, reduces communication latency, and minimizes the risk of sensitive data leakage to external entities.

8 Conclusions

The deliverable outlines how the technical outcomes of Tasks 4.2 and 4.3 play a key role in providing (i) enhanced tools for the workflow-based design and low-code description of hyper-distributed applications, (ii) novel application packaging, software delivery and deployment framework, and (iii) analytics-friendly distributed storage solution for IoT data. It provides a detailed description of each component's implementation, highlighting critical design decisions, internal architecture, and public APIs. Furthermore, it demonstrates the relevance of each component by mapping them to the project's requirements, KPIs, and use cases.

To achieve the goals of Work Package 4, and ultimately those of the broader EMPYREAN project, these components implement a wide range of features, many of which are directly exposed to EMPYREAN application developers and operators. This deliverable presents mechanisms for:

- Enabling low-code, multi-site workflow orchestration with support for user-defined constraints and hybrid edge-cloud deployments through the Ryax workflow engine.
- proposing a unified data communication layer introduced through Zenoh, supporting efficient pub/sub, distributed storage access, and computation across geo-distributed systems.
- novel coding and alignment techniques to enable efficient query execution directly on erasure-coded, time-series data without full reconstruction through the analytics-friendly distributed storage.
- contributing a reproducible and architecture-agnostic packaging system using NIX and OCI standards, brought with Ryax action packaging, supporting a broad range of workloads and plans for integration with unikernel building.
- introducing a minimal, OCI-compliant toolchain to compile and deploy secure, lightweight unikernel applications tailored for the IoT-edge-cloud continuum.

In the next phase of Tasks 4.2 and 4.3, efforts will focus on finalizing the development of each component in alignment with the initial plans, and progressing with their integration to compose a unified EMPYREAN platform. Additionally, each component will contribute to validating the achievement of the project's technical goals and the attainment of its KPI targets. Importantly, the outcomes of WP4 will remain central to the project's progression, directly supporting the activities in Work Packages 5 and 6.