TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

Interactive User Environment Application on Kubernetes Platform

Author:
Kyriakos CHALVATZIS

Thesis Committee:
Prof.Vasilis SAMOLADAS
Prof. Nikolaos GIATRAKOS
Prof. Euripides PETRAKIS

A thesis submitted in fulfillment of the requirements for the diploma of Electrical and Computer Engineer

in the

School of Electrical and Computer Engineering

July 10, 2025

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

Interactive User Environment Application on Kubernetes Platform

by Kyriakos CHALVATZIS

The following thesis presents the design and implementation of a user-oriented analytical computing system deployed on the Kubernetes platform. The system draws inspiration from core Unix design principles, particularly in file and directory permissions, user hierarchy and authorization management.

The central ambition is to ease the operational quality of life for data analysts working with sizable or not datasets, while maintaining strict access control and support for concurrent multi-user interactions. Users can upload and share datasets, submit batch job into execution using pre-integrated applications, such as DuckDB, Bash, Octave, Pandas and others. Designed with modularity and extensibility in mind, the platform aims to abstract the complexity of orchestration, achieves abstraction of resource allocation and job scheduling, providing users with a seamless interface for analytical workflows. Each job lives and dies in a sandboxed containerized environment, ensuring reproducibility and isolation.

The solution demonstrates the feasibility of delivering a lightweight, flexible platform that leverages Kubernetes' native scalability to manage user resources, data, and jobs efficiently. It offers a pathway for democratizing access to powerful analytics tooling, especially in research and educational contexts where ease of deployment and extensibility are vital.

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

Interactive User Environment Application on Kubernetes Platform

by Kyriakos CHALVATZIS

Η παρούσα διπλωματική εργασία παρουσιάζει τον σχεδιασμό και την υλοποίηση ενός υπολογιστικού συστήματος ανάλυσης δεδομένων προσανατολισμένου στον χρήστη, το οποίο αναπτύσσεται στην πλατφόρμα Kubernetes. Το σύστημα αντλέι έμπνευση από τις θεμελιώδεις αρχές σχεδιασμού του Unix, ιδιαίτερα ως προς τα δικαιώματα αρχείων, την ιεραρχία χρηστών και τη διαχείριση εξουσιοδοτήσεων. Κεντρική φιλοδοξία αποτελεί η βελτίωση της καθημερινής εμπειρίας των αναλυτών δεδομένων που εργάζονται με μεγάλους και όχι όγκους δεδομένων, διατηρώντας παράλληλα αυστηρό έλεγχο πρόσβασης και υποστήριξη για ταυτόχρονη αλληλεπίδραση πολλών χρηστών. Οι χρήστες μπορούν να ανεβάζουν και να διαμοιράζονται σύνολα δεδομένων, καθώς και να υποβάλλουν παρτίδες εργασιών προς εκτέλεση χρησιμοποιώντας προενσωματωμένες εφαρμογές, όπως DuckDB, Bash, Octave, Pandas και άλλες. Με σχεδιασμό που δίνει έμφαση στην μοντελοποίηση και την επεκτασιμότητα, η πλατφόρμα στοχεύει στην απόκρυψη της πολυπλοκότητας της ενορχήστρωσης, της διαχείρισης πόρων και του χρονοπρογραμματισμού εργασιών, προσφέροντας στους χρήστες ένα ενιαίο και εύχρηστο περιβάλλον για ροές εργασίας ανάλυσης δεδομένων. Κάθε εργασία εκτελείται και τερματίζεται μέσα σε απομονωμένο περιβάλλον, διασφαλίζοντας της αναπαραγωγιμότητα και την απομόνωση. Η λύση καταδεικνύει τη δυνατότητα δημιουργίας μιας ελαφριάς και ευέλικτης πλατφόρμας που αξιοποιεί την εγγενή επεκτασιμότητα του Kubernetes για την αποδοτική διαχείριση των χρηστών, των δεδομένων και των εργασιών. Αποτελεί μια προσέγγιση που διευρύνει την πρόσβαση σε ισχυρά εργαλεία ανάλυσης, ιδιαίτερα σε ερευνητικά και εκπαιδευτικά περιβάλλοντα όπου η ευκολία ανάπτυξης και η δυνατότητα επέκτασης είναι καθοριστικής σημασίας.

Acknowledgements

First and foremost, I would like to express my deep respect and appreciation towards my advisor, Professor Vasilis Samoladas. His precise guidance served as a beacon of clarity and stability throughout the often chaotic process of designing and developing an entire system independently.

I am also deeply grateful for the unwavering support and patience shown by my close friends and family, who stood by me selflessly and with enduring encouragement during this demanding journey.

Last but not least, I would like to extend my heartfelt thanks to the esteemed members of the Thesis Committee, Professor Euripides. Petrakis and Professor Nikos. Giatrakos, as well as to my fellow colleagues and students on campus who shared this experience.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

API	**A**pplication **P**rogramming **I**nterface
CNS	**C**loud **N**ative **S**torage
CS	**C**omputer **S**cience
CRUD	**C**reate **R**ead **U**pdate **D**elete
CSS	**C**ascade **S**tyle **S**heets
CSV	**C**omma **S**eperated **V**alues
DAG	**D**irected **A**cyclic **G**raph
DSL	**D**omain **S**pecific **L**anguage
ETL	**E**xtract **T**ransform **L**oad
HTTP	**H**yper**T**ext **T**ransfer **P**rotocol
HTML	**H**yper**T**ext **M**arkup **L**anguange
I/O	**I**nput / **O**utput
JWKS	**J**SON **W**eb **K**ey **S**et
JWT	**J**son **W**eb **T**oken
JS	**J**ava**S**cript
JSON	**J**ava**S**cript **O**bject **N**otation
K8S	**K**uberneteS
PaaS	**P**latform **a**s **a** **S**ervice
PV	Kubernetes **P**ersistent **V**olume
PVC	Kubernetes **P**ersistent **V**olume **C**laim
RBAC	**R**ole **B**ased **A**ccess **C**ontrol
S3	Amazon **S**imple **S**torage **S**ervice
SQL	**S**tructured **Q**uery **L**anguage
UID	**U**ser **ID**entification
GID	**G**roup **ID**entification
VID	**V**olume **ID**entification
RID	**R**esource **ID**entification
JID	**J**ob **ID**entification

Dedicated to my family and friends…

1 Introduction

1.1 Purpose and Motivation

This project began as an effort to integrate a user-friendly 'environment' layer into Kubernetes—an infrastructure known for its power and scalability, yet lacking direct support for user-centric entities. The goal was to make an environment where individual users can perform analytical computing in their own 'space', especially enabling data analysts to work with large-scale datasets, by abstracting the complexity of orchestration and container management. As the system evolved, the focus deviated slightly towards a fully modular full stack platform that supports multiple users, secures access with custom authentication, and enables storage provisioning, sharing, and execution of batch jobs using familiar tools such as DuckDB, Bash, Octave, and Pandas [1]. The motivation became to deliver a system that combines the robustness of cloud-native technologies with the ease of use of desktop analytical environments—bridging the gap between DevOps infrastructure and end-user data workflows.

Ultimately, amongst some services, the bar was set to implement already existing enterprise level technology stacks to more basic and developer friendly as in ease of access and deployment versions. It had been a personal aspiration and challenge to create some utilities that are scoped for development with a laconic tone while maintaining inspiration by others.

1.2 Problem Statement

While modern data platforms such as Apache Airflow, Apache Hadoop [2], and Apache Spark address large-scale, enterprise-level orchestration needs, they often come with significant complexity and overhead, making them less accessible for individual users or small teams. These platforms are designed to handle distributed computing at scale, providing extensive features for job scheduling, resource management, and data processing. However, they require substantial infrastructure setup, configuration, and maintenance, which can be daunting for users who simply want to run analytical jobs on their data without delving into the intricacies of distributed systems.

This thesis does not seek to compete with such platforms, nor to introduce fundamentally new orchestration paradigms. Instead, it focuses on designing and implementing a lightweight, extensible system that simplifies the execution of batch data analysis jobs in a multi-user environment.

The core problem addressed is the absence of a simple, user-accessible platform where authenticated users can:

- Upload datasets to a common, structured storage layer,
- Submit analytical jobs using containerized tools such as DuckDB, Pandas, or Octave,
- View job outputs and logs through a unified interface,
- And share or manage their data and jobs in a collaborative way.

This project presents a custom application built on top of Kubernetes [3] that abstracts away infrastructure complexity, offering a modular foundation upon which analytical tools can be plugged and executed in a reproducible, isolated manner. The overarching goal is to improve the user experience in running batch data workflows—not by reimagining distributed computation, but by making its power more accessible and flexible for real-world analysis tasks.

1.3 Scope of the Project

The scope of this thesis is to design and implement a modular, microservice-based platform on Kubernetes that enables multi-user analytical computing. The project covers the following aspects:

- Development of core backend services for authentication, orchestration, and virtual filesystem abstraction.
- Integration with object storage (MinIO) for user data management.
- Provision of a frontend for real-time job monitoring and user interaction.
- Support for batch job execution using containerized analytical tools.

The project intentionally excludes advanced scheduling algorithms, dynamic autoscaling, and third-party authentication (e.g., OAuth [4]). The implementation is primarily in Go [5] using the Gin web framework [6].

1.4 Thesis Contributions

The main contributions of this thesis are:

- A novel, user-centric platform architecture that abstracts Kubernetes orchestration for end users.
- Implementation of a custom authentication service (`Minioth`) with secure, user/group-based access control.
- The design of the `Uspace` service for managing isolated user environments and secure job submission.
- Introduction of the `Fslite` virtual filesystem abstraction for hierarchical user data management.
- A real-time, WebSocket-enabled frontend for interactive job monitoring and workspace access.
- Demonstration of how cloud-native technologies can be adapted for accessible, multi-user analytical workflows.

1.5 Thesis Outline

- **Chapter 2 – Background and Theoretical Foundations:** An overview of the core technologies and concepts underpinning the system, including Kubernetes, microservices, authentication models, containerized workloads, and object storage.
- **Chapter 3 – Related Work:** A review of existing platforms and tools related to data processing, user environment provisioning, and cloud-native architectures. It highlights their limitations and positions this work within the broader landscape.
- **Chapter 4 – System Design and Architecture:** The Architectural blueprint of the system, detailing the interactions between core services such as Minioth, Uspace, Fslite, and the Frontend. The design goals of modularity, scalability, and security are emphasized.
- **Chapter 5 – System Usage & Execution Flow:** Describes how the system operates from both user and administrative perspectives. It details the deployment setup, system access points, request-response structure, and error feedback mechanisms. End-to-end workflows are illustrated through UI examples, including the job submission lifecycle.
- **Chapter 6 – Evaluation and Robustness:** An evaluation of the system's behavior under various conditions, including performance, scalability, fault tolerance, and security. Observations and informal benchmarks are included where applicable..
- **Chapter 7 – Conclusions and Future Work:** Summarizes the work, reflects on its outcomes and limitations, and discusses potential directions for further development and enhancement of the system.

2 Background and Theoretical Foundations

2.1 Kubernetes

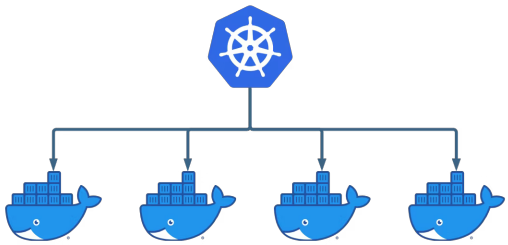


FIGURE 2.1: Kubernetes

Kubernetes is an open-source container orchestration platform originally developed by Google [7] and now maintained by the Cloud Native Computing Foundation (CNCF) [8]. It automates the deployment, scaling, and management of containerized applications across a cluster of machines. Kubernetes abstracts the infrastructure and provides primitives such as Pods, Deployments, Services, and Jobs, which allow developers to define complex systems declaratively.

What Kubernetes provides:

- **Service discovery and load balancing**
- **Storage orchestration**
- **Automated rollouts and rollbacks**
- **Automatic bin packing**
- **Self-healing**
- **Secret and configuration management**
- **Batch execution**

- **Horizontal scaling**

Kubernetes will come handy in this project to deploy and manage the microservices that make up the platform. But also provide the means to run batch jobs in a scalable and isolated manner.

2.1.1 Kubernetes API

The Kubernetes API is the primary interface for interacting with a Kubernetes cluster [9]. It exposes a RESTful interface that allows users to create, read, update, and delete resources such as Pods, Services, and Deployments. The API is designed to be extensible, allowing developers to create custom resources and controllers to manage

2.1.2 Kubernetes Jobs

Kubernetes Jobs are a resource type that allows users to run batch processing tasks [10]. Jobs ensure that a specified number of Pods successfully terminate, making them suitable for one-off tasks or batch processing workloads. Jobs can be configured to run to completion, retry on failure, and manage parallel execution and also have resource quotas and limits to control resource usage. In this system, Jobs are used to execute user-submitted analytical tasks in a controlled and isolated environment.

Batch Job execution is a key feature of Kubernetes, allowing users to run tasks that do not require continuous interaction or long-running processes. This is particularly useful for data analysis, where users may need to process large datasets or run complex computations without needing to maintain a persistent connection.

2.1.3 Kubernetes Persistent Volumes

Kubernetes Persistent Volumes (PVs) and Persistent Volume Claims (PVCs) provide a way to manage storage resources in a Kubernetes cluster [11]. PVs are cluster-wide storage resources that can be dynamically provisioned or statically defined, while PVCs are user requests for storage that can be bound to PVs. This abstraction allows users to request storage without needing to know the underlying details of how it is provisioned or managed. In this system, PVs and PVCs are used to provide persistent storage for user data and job outputs, ensuring that data is retained across job executions and Pod

restarts. Persistent storage is essential for analytical workloads, as it allows users to store and retrieve large datasets without losing data between job executions.

2.2 Containers

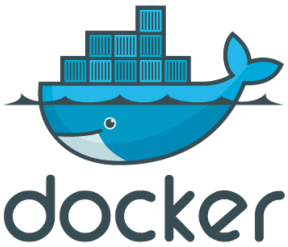


FIGURE 2.2: Docker Containers

Containers are lightweight, portable units that package software with its dependencies and run isolated from the host system [12]. Technologies like Docker and container runtimes such as containerd have made containers a standard deployment model in modern systems.

2.2.1 Containerization

Containerization is the process of packaging an application and its dependencies into a single, portable unit called a container [13]. Containers encapsulate everything needed to run an application, including the code, runtime, libraries, and system tools, ensuring that it runs consistently across different environments. This approach eliminates the "it works on my machine" problem, as containers provide a consistent runtime environment regardless of where they are deployed. In this system, containerization is used to encapsulate analytical tools and user jobs, allowing them to run in isolated environments without conflicts.

It holds also a crucial aspect of the Microservices architecture, as each service can be packaged and deployed independently, allowing for greater flexibility and scalability.

2.2.2 Docker

Docker is a popular platform for building, shipping, and running containers [12]. It provides a set of tools and APIs for creating container images,

managing container lifecycles, and orchestrating multi-container applications. Docker simplifies the process of containerization by providing a user-friendly interface and a rich ecosystem of pre-built images. In this system, Docker is used to create container images for analytical tools and user jobs, enabling easy deployment and execution in a Kubernetes cluster.

2.2.3 Container Runtimes

Container runtimes are software components responsible for running containers on a host system [14]. They provide the necessary APIs and functionality to manage container lifecycles, including starting, stopping, and monitoring containers. Popular container runtimes include Docker, containerd, and CRI-O. Kubernetes supports multiple container runtimes through the Container Runtime Interface (CRI), allowing users to choose the runtime that best fits their needs. In this system, the container runtime is used to execute user jobs and analytical tools in isolated environments, ensuring that each job runs with its own dependencies and configurations without interfering with other jobs or the host system.

2.3 Microservices Architecture

The microservices architectural style structures an application as a collection of loosely coupled services, each responsible for a specific domain or capability [15]. Microservices communicate primarily through lightweight mechanisms such as HTTP or messaging queues, and they are independently deployable.

This approach was adopted in the system to improve modularity and separation of concerns. For instance, authentication, job orchestration, and storage abstraction are implemented as separate services, allowing them to evolve independently and scale according to their individual loads.

Microservices also enable the use of different technologies and programming languages for each service, allowing teams to choose the best tools for each specific task. In this system, the microservices are implemented in Go using the Gin web framework [6], which provides a lightweight and efficient foundation for building RESTful APIs.

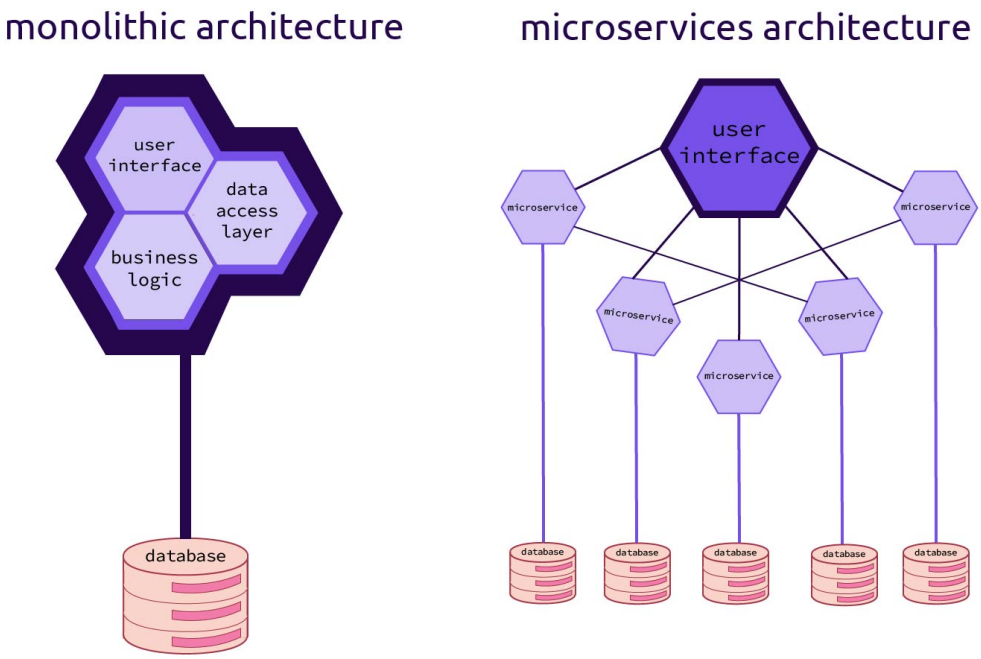


FIGURE 2.3: MicroServices vs Monolithic

2.3.1 Microservices Communication Patterns

Microservices rely on well-defined communication mechanisms to enable interaction between services and between clients and services. These mechanisms can broadly be classified into *request-response* patterns, such as REST over HTTP, and *persistent bidirectional* patterns, such as WebSockets.

REST and JSON

One of the most prevalent communication paradigms in microservice architectures is **REST** (Representational State Transfer). REST is a stateless, resource-oriented communication pattern that leverages standard HTTP methods — such as `GET`, `POST`, `PUT`, and `DELETE` — to operate on resources identified by URLs.

RESTful APIs facilitate loose coupling between services by exposing clear, consistent interfaces. These APIs are widely supported, easily testable, and inherently scalable due to the stateless nature of HTTP.

Data exchange in RESTful services is commonly performed using **JSON** (JavaScript Object Notation) — a lightweight, human-readable data interchange format. JSON provides a flexible structure for representing hierarchical data, making

it suitable for encoding complex objects and payloads in a microservices environment. Its widespread adoption across languages and platforms further reinforces its suitability for distributed systems.

While REST and JSON are effective for a broad range of use cases — particularly CRUD (Create, Read, Update, Delete) operations — they are inherently *request-response oriented*, meaning the client must initiate every interaction. This pattern works well for transactional operations but is less efficient for real-time updates or event-driven communication.

WebSockets

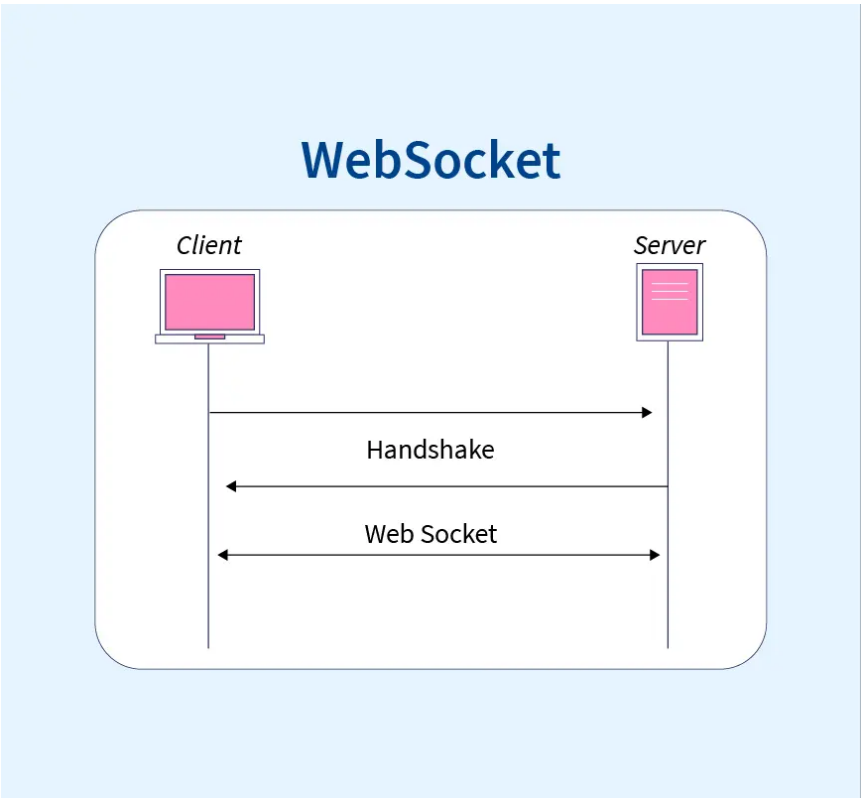


FIGURE 2.4: WebSockets communication model

WebSockets complement REST by offering a *persistent, full-duplex communication channel* over a single TCP connection. Once established, a WebSocket connection allows both the client and server to send messages independently of one another, enabling real-time communication with minimal latency [16].

This system utilizes WebSockets for features requiring real-time interaction, such as:

- Job monitoring and progress updates.

- Streaming logs or computation outputs.
- Interactive shell sessions or command execution interfaces.

WebSockets reduce the need for constant polling or repeated HTTP requests, enhancing the responsiveness of the system and improving the user experience in scenarios where timely feedback is essential.

2.3.2 Microservices Storage Patterns

Embedded Databases in Microservices

A notable approach within this paradigm is the use of **embedded databases**, where the database engine is packaged directly within the microservice. This contrasts with traditional client-server databases, which operate as standalone services accessed over the network.

Embedded databases are particularly well-suited to microservices that:

- Require local, service-scoped data.
- Avoid the complexity of managing external database clusters.
- Prioritize simplicity, portability, and ease of deployment.

SQLite as a Storage Solution

FIGURE 2.5: SQLite database engine

SQLite is a lightweight, serverless SQL database engine that operates directly on a single file without requiring a separate database server [17]. Its simplicity and zero-configuration nature make it a practical choice for embedded use within microservices.

In this system, SQLite serves as the primary datastore for services handling:

- Authentication and user management.
- Metadata tracking for resources.
- Job submission and execution logs.

By embedding SQLite within each microservice, the architecture achieves:

- **Decoupled data ownership**: Each service manages its own schema and data lifecycle.
- **Improved portability**: Services are self-contained, simplifying deployment and replication.

- **Reduced operational overhead**: No need to manage external relational database clusters for most services.

SQLite in Kubernetes Deployments

When deployed in a container orchestration environment like **Kubernetes**, SQLite databases are backed by **Persistent Volumes (PVs)**. This ensures that the underlying database files persist even if the container is rescheduled or restarted.

The advantages of this design include:

- **Data durability**: PVs ensure the state is preserved across pod lifecycles.
- **Service isolation**: Each microservice maintains its own isolated state without the risks associated with shared database servers.
- **Simplicity and scalability**: Microservices can be deployed, scaled, or updated without dependency on a centralized data layer.

While SQLite is not suited for highly concurrent, write-intensive workloads or distributed SQL scenarios, it is highly effective for microservices with:

- Localized data needs.
- Moderate concurrency.
- A preference for simplicity over complex clustering.

This storage pattern aligns with the broader microservices philosophy of decentralized ownership and independently deployable components, while Kubernetes provides the durability and resilience traditionally associated with larger database systems.

2.4 Cloud-Native Storage

FIGURE 2.6: CNS

Cloud-native storage refers to storage systems designed to operate within cloud environments, typically containerized and orchestrated via platforms like Kubernetes [18]. Unlike traditional block or file-based storage, cloud-native storage solutions are optimized for dynamic workloads, scalability, and high availability.

In this system, MinIO—a cloud-native object storage service compatible with Amazon S3—is used to store user data and job outputs. Object storage offers flexibility in managing large, unstructured datasets, and integrates seamlessly with Kubernetes via Persistent Volume Claims (PVCs). Cloud-native storage allows the system to dynamically provision isolated storage volumes for each user or job, supporting scalable and fault-tolerant operations.

2.4.1 Object Storage

Object storage is a data storage architecture that manages data as objects, rather than files or blocks. Each object contains the data itself, metadata, and a unique identifier. This model is particularly well-suited for unstructured data such as images, videos, and large datasets, as it allows for efficient storage and retrieval without the constraints of traditional file systems. Object storage systems are designed to scale horizontally, providing high availability and durability by distributing data across multiple nodes. In this system,

MinIO is used as the object storage backend, providing a scalable and flexible solution for storing user-uploaded files and job outputs. Its compatibility with the S3 API allows for easy integration with various analytics tools and frameworks, enabling users to perform data analysis directly on their datasets without needing to manage a traditional POSIX filesystem.

2.4.2 MinIO

FIGURE 2.7: MinIO

MinIO is a high-performance, Kubernetes-native object storage system compatible with the Amazon S3 API [19]. It is often used in cloud-native applications to store unstructured data such as logs, images, or datasets.

In the system, MinIO acts as the backend storage for user-uploaded files and job outputs. Its compatibility with S3 allows for flexible integration with analytics tools and easy management of large datasets without a traditional POSIX filesystem.

Additionally, the imaged application tools used in the system are set with I/O on MinIO.

2.5 Analytical Databases and Batch Processing

Analytical databases are designed to efficiently process large volumes of data for analytical queries, often involving aggregations, filtering, and complex transformations [20]. Unlike transactional databases, which are optimized for frequent, small updates, analytical databases leverage columnar storage and vectorized execution to accelerate read-heavy workloads typical in data analysis.

Batch processing refers to the execution of jobs that process data in large chunks, typically without user interaction, and is well-suited for analytical

workloads. In the context of this system, batch jobs are executed as Kubernetes Jobs, leveraging containerized analytical tools to process user data in an isolated and scalable manner.

DuckDB fits into this architecture as the primary analytical database engine for batch processing tasks. Its embeddable nature and support for direct file access make it ideal for running SQL-based analytics on user-uploaded datasets within short-lived Kubernetes Jobs. By integrating DuckDB as a batch analytical tool, the system enables users to perform complex data transformations and analyses efficiently, without the need for a persistent database server.

This approach aligns with the overall design of the platform, which emphasizes modularity, scalability, and ease of use by combining cloud-native orchestration (Kubernetes), containerization, and modern analytical engines like DuckDB.

2.5.1 DuckDB

DuckDB is an in-process SQL OLAP (Online Analytical Processing) database management system optimized for analytical queries on columnar data [21]. Unlike traditional database servers, DuckDB runs directly within the host process and is designed for interactive analytics on local files such as CSV and Parquet. Its columnar storage model and vectorized query execution engine make it particularly suitable for analytical workloads involving large datasets.

DuckDB draws conceptual inspiration from systems like PostgreSQL and MonetDB, but emphasizes lightweight deployment and embeddability. It supports complex SQL queries, window functions, joins, and aggregations with high performance, while avoiding the operational complexity of client-server architectures.

Why DuckDB?

DuckDB was chosen for this system due to several compelling advantages:

- **Embeddability:** DuckDB can be run inside a container with no setup or external dependencies, making it ideal for sandboxed job execution in Kubernetes.

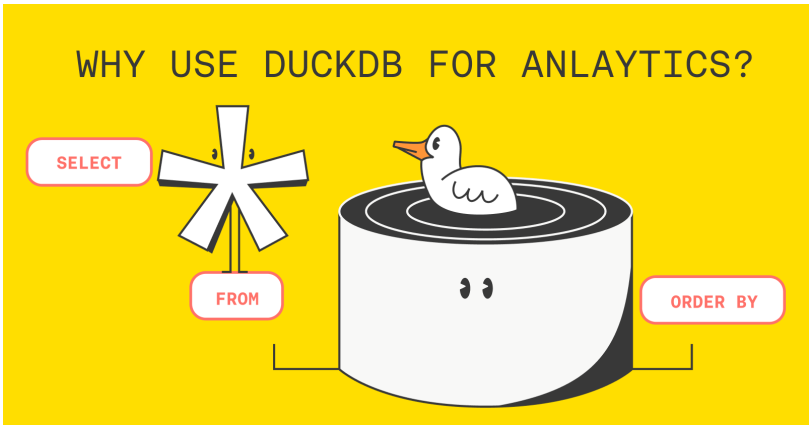


FIGURE 2.8: Why DuckDB

- **Zero Configuration:** Users can query structured files (e.g., CSV, Parquet) without needing to first load data into database tables, simplifying the user workflow.
- **Columnar Execution:** Optimized for analytical queries, DuckDB's columnar execution model enables efficient processing of large datasets, particularly in filtering and aggregation operations.
- **File-Native Access:** Direct support for querying local data files stored on persistent volumes or object storage simplifies integration with the storage layer (MinIO).
- **Lightweight and Fast:** Compared to heavier analytical engines, DuckDB has a small footprint and fast startup time, which makes it suitable for short-lived Kubernetes jobs.

In this system, DuckDB is one of the containerized tools available for users to execute SQL-based data analysis jobs on uploaded datasets. Its integration enables users to perform complex transformations and analytics directly on their data without managing database infrastructure.

2.6 Multi-User System and Authentication Models

In addition to the architectural and infrastructural foundations discussed in previous sections, this platform is fundamentally designed as a multi-user system. Supporting concurrent access by multiple independent users requires careful consideration of identity management, authentication mechanisms, authorization, and secure resource isolation.

These concepts are not specific to microservice or cloud-native architecture alone but draw inspiration from traditional operating system designs, particularly UNIX-like user and group models. This section introduces the core principles of multi-user system design and authentication models that inform the platform's architecture.

2.6.1 Multi-User System Design

A multi-user system supports concurrent access by multiple users, each with distinct identities, permissions, and levels of isolation. Designing such systems involves addressing several key concerns, including [22]:

- **Authentication**: Verifying user identities.
- **Authorization**: Controlling access to resources based on roles, groups, or permissions.
- **Resource Isolation**: Preventing interference or conflicts between users.
- **Secure Sharing**: Allowing controlled data sharing where appropriate.

The platform developed in this thesis adopts a multi-user architecture in which each user is assigned an isolated and secure environment. Users are able to:

- Upload datasets and manage resources.
- Submit and execute analytical jobs.
- Share data selectively, based on group memberships and permissions.

This design is influenced by the UNIX model of user and group-based access control, where permissions are managed according to ownership (user), group association, and global access rights. The use of these classical access models provides a clear, interpretable foundation for resource protection and operational boundaries within the shared infrastructure.

2.6.2 Authentication and Authorization Model

Authentication is the process of verifying the identity of a user before granting access to system resources [23]. In multi-user systems, robust and reliable authentication forms the foundation for all other security mechanisms.

The platform implements a custom authentication service named `Minioth`. This service adopts a modern, stateless authentication approach using **JSON**

Web Tokens (JWT). Upon successful login, a user is issued a digitally signed token that encodes their identity and associated claims, such as group memberships and roles.

These tokens are used for authenticating subsequent requests across the distributed microservices architecture. The benefits of this stateless, token-based model include:

- **Scalability**: No need for server-side session storage; tokens are self-contained.
- **Portability**: Tokens can be verified by any microservice without centralized coordination.
- **Security**: Tokens are signed and optionally encrypted to prevent tampering.
- **Fine-grained Access Control**: Group memberships and user roles are encoded in token claims, enabling services to enforce authorization policies efficiently.

This design aligns with modern identity patterns common in cloud-native systems while maintaining consistency with the UNIX-inspired group-based access control model implemented in the platform. Together, the authentication mechanism and multi-user design provide a secure, isolated, and user-centric experience within the distributed system.

3 Related Work

3.1 Batch-Processing Platforms

Batch-processing platforms are fundamental to modern data-intensive systems, providing scalable infrastructure for executing large-scale data transformation, analytics, and machine learning workflows. These platforms typically operate by orchestrating jobs over clusters of machines, handling aspects such as task distribution, fault tolerance, and data locality.

Prominent examples include **Apache Airflow** [24], **Apache Spark** [25], and the foundational framework **Apache Hadoop** [2]. While these systems are highly effective for managing production-grade pipelines, they are often complex, requiring significant infrastructure setup and expertise.

FIGURE 3.1: Apache Hadoop and Apache Spark

3.1.1 Apache Hadoop

Apache Hadoop is one of the earliest and most influential frameworks for distributed batch processing of large-scale datasets. Its architecture is based on the **MapReduce** programming model, introduced by Google [26], which enables the decomposition of tasks into two key stages:

- **Map:** Processes input data and produces intermediate key-value pairs.
- **Reduce:** Aggregates and processes intermediate results to produce final output.

Hadoop is built around two core components:

- **HDFS (Hadoop Distributed File System)**: A fault-tolerant, scalable storage system that splits large files into blocks distributed across the cluster.
- **YARN (Yet Another Resource Negotiator)**: A cluster resource manager that schedules and orchestrates tasks.

The strength of Hadoop lies in its ability to scale to thousands of nodes, offering data locality optimizations and fault tolerance. However, Hadoop jobs are typically defined as Java programs or high-level abstractions built on top, such as Hive or Pig. The model assumes long-running batch jobs, with significant overhead in job startup time, which is unsuitable for low-latency or interactive workloads.

Moreover, Hadoop was designed for infrastructure-level users. It lacks native mechanisms for *multi-user job isolation* beyond basic OS-level permissions and does not provide end-user self-service capabilities without substantial operational overhead.

3.1.2 Apache Spark

Apache Spark evolved from the limitations of Hadoop's MapReduce model, offering a more flexible and performant engine for distributed data processing. Spark introduces an in-memory computation model, reducing the disk I/O bottlenecks inherent in Hadoop. It supports multiple workloads beyond batch processing, including:

- SQL queries via SparkSQL.
- Streaming data via Structured Streaming.
- Machine learning via MLlib.
- Graph computation via GraphX.

Spark utilizes the concept of **Resilient Distributed Datasets (RDDs)**, which are immutable collections of objects distributed across nodes. Its Directed Acyclic Graph (DAG) execution engine optimizes task execution based on data dependencies.

While Spark significantly improves upon Hadoop's performance and flexibility, it still operates within a cluster-based, infrastructure-oriented paradigm.

Deployments typically involve YARN, Kubernetes, or Mesos, requiring considerable system administration. Like Hadoop, Spark lacks built-in multi-tenant isolation at the user level; access control and job separation are delegated to external security frameworks or infrastructure controls.

3.1.3 Apache Airflow

FIGURE 3.2: Apache Airflow

Apache Airflow is not a data processing engine but an **orchestration platform** designed to define, schedule, and monitor workflows. Airflow represents workflows as **Directed Acyclic Graphs (DAGs)**, where each node corresponds to a task. It is mentioned respectively in the context of batch processing because it is often used to orchestrate Spark or Hadoop jobs, but it does not perform data processing itself. This spirit of orchestration aspires to the system proposed in this thesis but in a more primitive manner. Due to the nature of the system, which essentially sets into motion a series of batch jobs, but in a more user-friendly way.

Airflow excels at:

- Managing task dependencies.
- Scheduling periodic workflows.
- Monitoring execution status.

Airflow supports extensibility through operators, which can trigger Spark jobs, run Python scripts, submit SQL queries, or interact with APIs. However, Airflow delegates computation to external systems; it does not execute data processing itself.

While flexible for backend orchestration, Airflow is designed for infrastructure teams rather than end-users. There is no native user-level job isolation or interactive job submission model. Users must define workflows in Python

code, with access controlled at the infrastructure level rather than at the application level.

3.1.4 Relation to This Work

The platform proposed in this thesis draws conceptual inspiration from these batch-processing frameworks in terms of distributed execution as an end goal and job orchestration. However, it fundamentally differs in its focus on:

- **User-level isolation**: Each user operates within an isolated environment with sandboxed resources.
- **Simplified job submission**: Users interact with the system through web interfaces or APIs without needing to write DAG definitions, infrastructure configurations, or cluster-level code.
- **Container-native execution**: Jobs are encapsulated in containers, leveraging Kubernetes orchestration for resource management, eliminating the need for specialized distributed data processing engines.
- **Security and multi-tenancy**: Authentication, authorization, and resource isolation are built-in features, contrasting with the infrastructure-level security assumed by Hadoop, Spark, or Airflow.

Rather than replacing general-purpose data lake or ETL pipelines, this platform addresses a different niche — enabling users to execute batch jobs and manage datasets in a secure, multi-tenant, containerized environment, without requiring them to manage or interact with complex infrastructure directly.

3.2 Data Lake Platforms

FIGURE 3.3: Data Lake Platforms

Data lake platforms such as AWS Lake Formation [27], Databricks Lakehouse [28], and Google Cloud BigLake [29] provide unified storage and compute environments for large-scale data analytics. These platforms often integrate object storage, access control, and analytics tooling into a single managed service.

While they are powerful and mature, they are typically commercial, tightly integrated into specific cloud providers, and involve complex administrative overhead. Moreover, user-level job sandboxing and container-level compute customization are limited or highly abstracted.

The system described in this thesis aims to offer a lightweight, open alternative that provides user-level compute environments, shared storage, and access-controlled workflows within a Kubernetes-native ecosystem.

3.3 Platform-as-a-Service Solutions

FIGURE 3.4: PaaS

Platform-as-a-Service (PaaS) [30] offerings such as Heroku [31], Google App Engine [32], and OpenShift [33] enable developers to deploy and scale applications without managing infrastructure. Some educational platforms like JupyterHub [34] provide multi-user notebooks for teaching and research purposes.

Most PaaS platforms abstract away Kubernetes primitives, limiting extensibility and custom control. They are typically focused on application deployment rather than providing flexible, user-controlled batch job execution or modular tool integration.

This project, by contrast, builds directly atop Kubernetes, giving fine-grained control over jobs, volumes, and user environments, and supporting arbitrary containerized tools within secure, user-defined workflows.

FIGURE 3.5: CN-Auth Services

3.4 Cloud-Native Auth Frameworks

Authentication frameworks such as Keycloak [35], Auth0 [36], and Firebase Auth [37] offer scalable, cloud-native identity management for web and API-based services. These tools often support OAuth2, OpenID Connect [38], and role-based access control (RBAC) [39], and can be integrated into Kubernetes clusters using service meshes or ingress controllers.

These systems are powerful and mature enough to handle complex authentication and authorization needs, including multi-tenancy, single sign-on, and social logins. They can safely scale to multiple replicas and provide secure user management across distributed applications.

However, these systems can be complex to integrate, especially in lightweight or standalone academic settings. Moreover, their abstraction level may not offer tight coupling with Kubernetes-native resource policies such as volumes or job ownership.

The system introduced in this thesis employs a custom authentication service (`Minioth`). It aims to express simplicity and promises a expansive foundation for future extensions, while still being lightweight enough for academic and small-scale use cases.

4 System Design and Architecture

4.1 Overview diagram

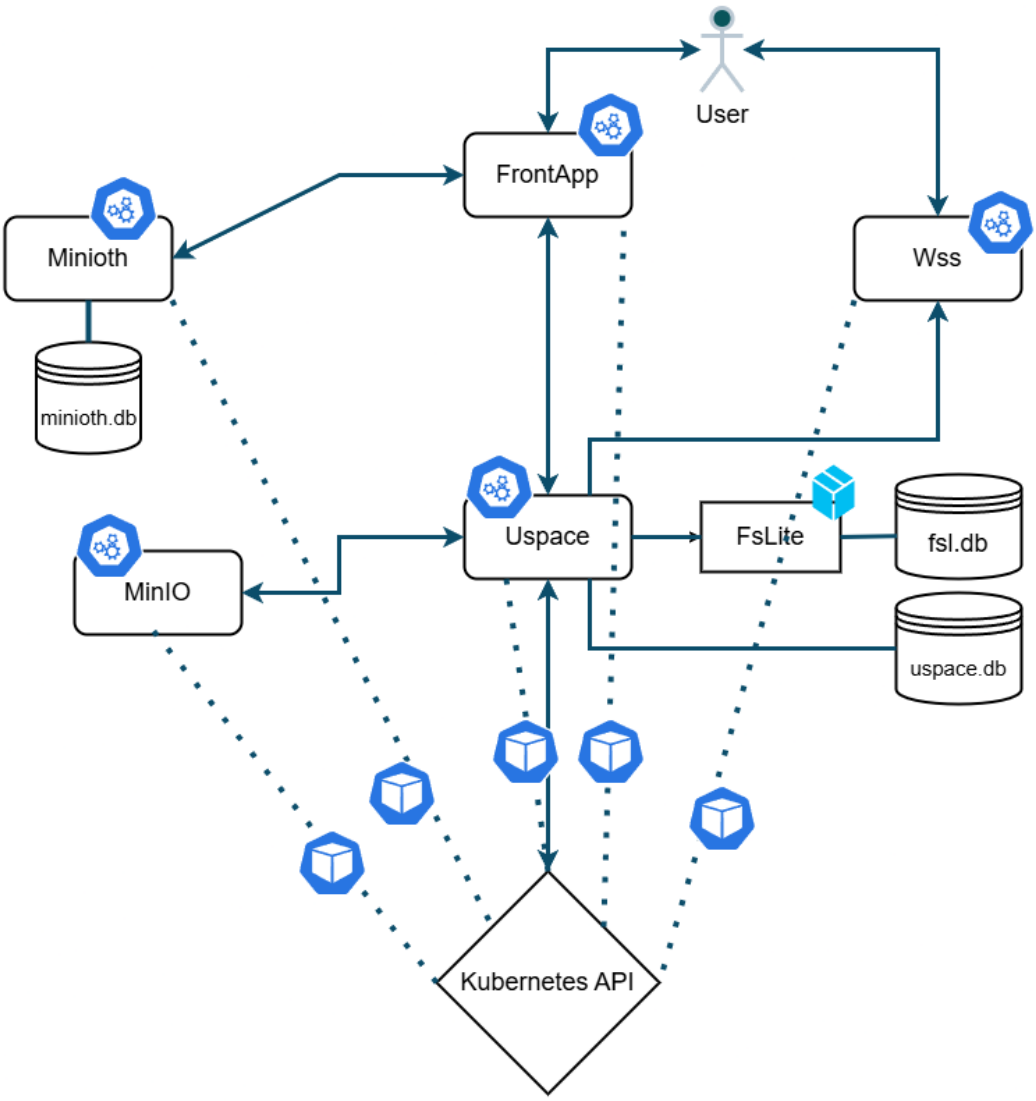


FIGURE 4.1: Kuspace System Overview

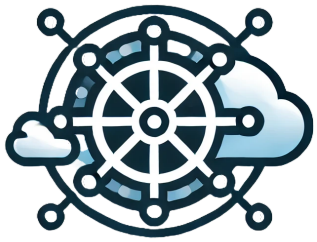


FIGURE 4.2: Kuspace logo

4.2 Uspace: Central Orchestration and Job Management Service

The **Uspace** service constitutes the core of the system's architecture, acting as the central orchestration unit that coordinates job scheduling, storage access, and user-level interaction with analytical resources. It is built with extensibility and modularity in mind, offering a consistent interface between users, applications, and the underlying infrastructure.

4.2.1 Responsibilities and Purpose

Uspace maintains the persistent state of the system through:

- A resource and volume metadata store powered by the `FsLite` module.
- A local database (SQLite or DuckDB) for tracking user-submitted jobs and available applications.
- A job scheduling and dispatch pipeline which integrates with the Kubernetes API.

Uspace exposes a RESTful API to authenticated users and administrators, supporting operations such as uploading datasets, browsing resources, and launching computational jobs.

Method	Path	Description	Requires `Access-Target`
GET	`/healthz`	Health check endpoint.	No
GET/POST	`/api/v1/job`	Submit or list jobs.	No
GET/POST	`/api/v1/app`	List available tools (applications).	No
GET	`/api/v1/resources`	List resources ("ls" equivalent).	Yes
POST	`/api/v1/resource/upload`	Upload a new resource file.	Yes
GET	`/api/v1/resource/preview`	Preview a resource.	Yes (Read)
GET	`/api/v1/resource/download`	Download a resource.	Yes (Read)
DELETE	`/api/v1/resource/rm`	Delete a resource.	Yes (Write)
POST	`/api/v1/resource/cp`	Copy a resource.	Yes (Read)
PATCH	`/api/v1/resource/mv`	Move/rename a resource.	Yes (Write)
PATCH	`/api/v1/resource/permissions`	Change resource permissions.	Yes (Owner)
PATCH	`/api/v1/resource/ownership`	Change resource ownership.	Yes (Owner)
PATCH	`/api/v1/resource/group`	Change associated group.	Yes (Owner)

TABLE 4.1: Uspace User API Endpoints

Method	Path	Description
GET/POST/PUT/DELETE/PATCH	`/api/v1/admin/volumes`	Full volume management API.
DELETE/PUT	`/api/v1/admin/job`	Administrative job control (e.g., delete jobs).
GET/POST/PATCH/DELETE	`/api/v1/admin/user/volume`	Manage user-volume mappings.
GET/POST/PUT/DELETE	`/api/v1/admin/app`	Manage available applications/tools.
GET	`/api/v1/admin/system-conf`	View current system configuration.
GET	`/api/v1/admin/system-metrics`	System metrics via Kubernetes API.

TABLE 4.2: Uspace Admin API Endpoints

4.2.2 Middleware and Access Control

The Uspace API applies layered middleware to enforce authentication, authorization, and request context binding.

In particular, API endpoints require a custom HTTP header named `Access-Target`, which encodes the target resource and the user identity. The header follows this format:

```
Access-Target: <volume_id>:<volume_name>:<resource_path>
<user_id>:[group_id1,group_id2,...]
```

The middleware parses this value to construct an `AccessClaim` object, which is then used to evaluate permissions through FsLite. This mechanism abstracts identity and access metadata away from individual endpoint logic and enforces consistency across resource operations.

Only authorized users (based on ownership or group permissions) can perform operations such as previewing, downloading, modifying, or deleting resources.

Furthermore, all API groups apply the `serviceAuth` middleware for validating a ServiceSecret common in the Service and verifying the caller is only a service in the system.

4.2.3 Core Structure

At the heart of the system lies the `UService` object, which encapsulates the configuration, runtime engine, and modular components responsible for storage, job handling, and resource management. Its composition is summarized as follows:

- **Configuration (`config`)**: Encapsulates all environment-based settings loaded during service initialization. This allows flexible configuration for both development and production deployments.
- **Web Engine (`Engine`)**: A Gin-based HTTP server that handles incoming API requests and routes them to appropriate handlers. It also includes middleware for authentication and authorization.
- **Storage Backend (`storage`)**: Implements the `StorageSystem` interface. In this system, it uses a MinIO-based implementation for interacting with S3-compatible object storage. The interface, however, allows for future pluggability (e.g., plain file volumes).
- **Job Database Handler (`jdbh`)**: Provides access to the local DuckDB or SQLite database used to track job submissions, statuses, and registered analytical applications.
- **Filesystem Metadata Layer (`fsl`)**: An instance of FsLite, used as an internal metadata engine for enforcing user-based access controls and tracking virtual filesystem hierarchies over the object store.
- **Job Dispatcher (`jdp`)**: Orchestrates the submission and lifecycle of analytical jobs. It abstracts the underlying execution backend (e.g., Kubernetes or Docker), enabling future extensibility via the `JobDispatcher` interface.

4.2.4 Storage Provider Abstraction

The `StorageSystem` interface defines the contract for interacting with storage backends. Below is a description of its main methods:

Algorithm 1 Abstract `StorageSystem` Interface (pseudocode)

```
1:  function DefaultVolume(local: Bool)
2:      return default volume identifier as String
3:  function CreateVolume(volume: Any)
4:      return Error if creation fails
5:  function SelectVolumes(criteria: Map[String, Any])
6:      return matching volumes or Error
7:  function SelectObjects(criteria: Map[String, Any])
8:      return matching objects or Error
9:  function Insert(object: Any)
10:     return CancelFunc, Error
11: function Download(object: Any)
12:     return CancelFunc, Error
13: function Stat(object: Any)
14:     return metadata or Error
15: function Remove(object: Any)
16:     return Error
17: function RemoveVolume(volume: Any)
18:     return Error
19: function Update(metadata: Map[String, String])
20:     return Error
21: function Copy(source: Any, dest: Any)
22:     return Error
23: function Share(method: String, object: Any)
24:     return SharingDescriptor or Error
```

4.2.5 Storage Control

Uspace as mentioned handles storage in two layers.

- The metadata layer that includes access control is consisted of `FsLite` module which also enforces authorization on the resources.
- The `StorageSystem` for physical storage, MinIO, and its API is accessed by a custom Go MinIO client.

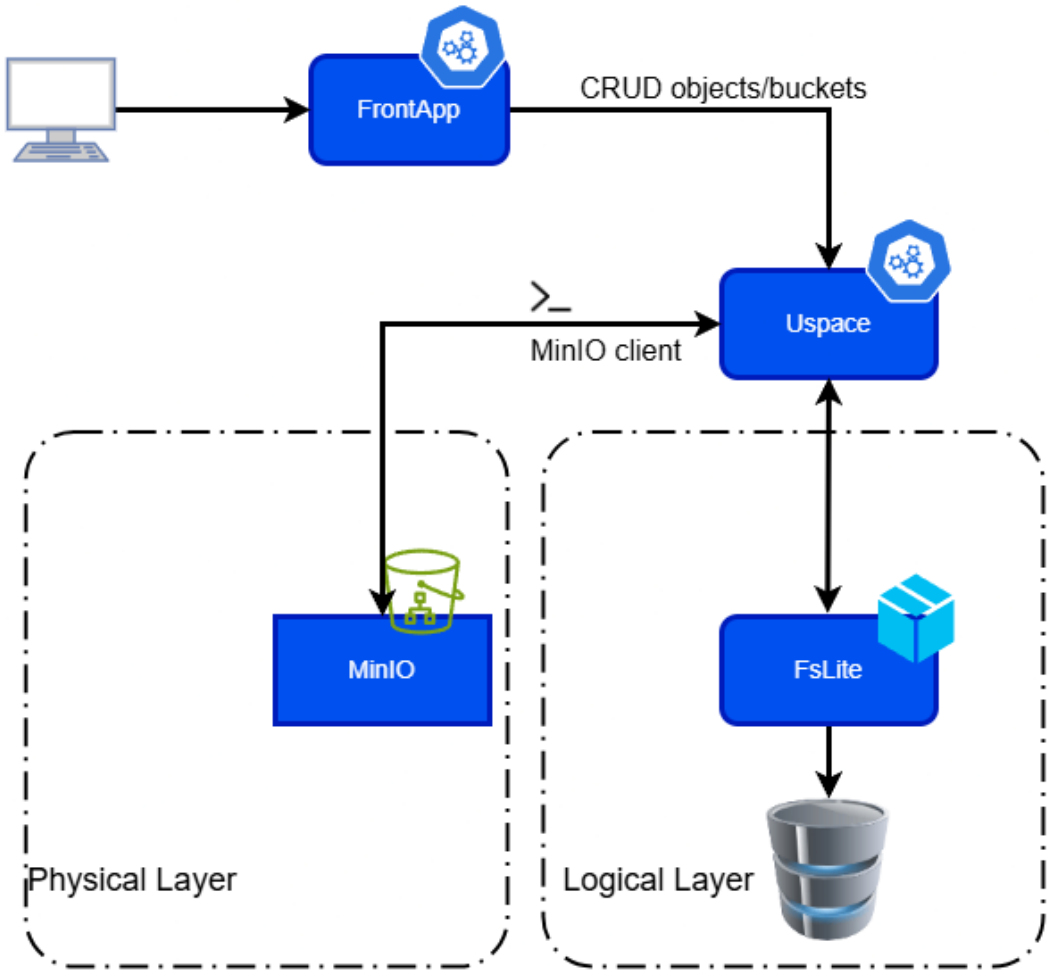


FIGURE 4.3: Uspace Storage Overview

4.2.6 Job Dispatcher and Executor

Uspace includes a pluggable job dispatching mechanism defined by the `JobDispatcher` interface, as well as a job execution mechanism defined by the `JobExecutor` interface:

Algorithm 2 Abstract `JobDispatcher` Interface (pseudocode)

```
1: function START
2:     Initialize and begin dispatching system
3: function PUBLISHJOB(job: Job)
4:     Submit a single job for dispatch
5:     return Error on failure
6: function PUBLISHJOBS(jobs: List[Job])
7:     Submit a batch of jobs
8:     return Error on failure
9: function REMOVEJOB(jobID: Integer)
10:    Cancel or remove job with given ID
11:    return Error on failure
12: function REMOVEJOBS(jobIDs: List[Integer])
13:    Remove multiple jobs
14:    return Error on failure
15: function SUBSCRIBE(job: Job)
16:    Register job for event handling or feedback
17:    return Error on failure
```

Algorithm 3 Abstract `JobExecutor` Interface (pseudocode)

```
function EXECUTEJOB(job)
    Returns: Error
function CANCELJOB(job)
    Returns: Error
```

The default implementation is a `JobManager`, which maintains queues and internal execution tracking. It uses a `JobExecutor` to actually launch jobs using one of two backends:

- **DockerExecutor**: Launches jobs in local Docker containers for testing.
- **KubernetesExecutor**: Launches Kubernetes Jobs via the API for production workloads.

This separation of concerns enables the system to evolve with future support for alternative execution engines such as serverless functions or distributed clusters.

Job Specification: Each submitted job is described by a structured payload that defines its resource requirements, execution logic, and metadata.

The job object includes:

- **Identifiers:** `JID` (Job ID), `UID` (User ID).
- **Resource Requirements:** CPU, memory, and ephemeral storage requests/limits.
- **Execution Parameters:** `Parallelism`, `Timeout`, `Priority`.
- **Paths:** `Input` and `Output` resource references.
- **Logic:** A predefined `Logic` (e.g., "duckdb") and a `LogicBody` representing the code or command to be executed.
- **Environment:** An optional key-value map of environment variables.
- **Status Tracking:** Includes current `Status`, `CreatedAt`, `CompletedAt`, and a boolean `Completed` flag.

Jobs are submitted through the Uspace API, validated, and dispatched for execution by the internal job manager. The structure supports extensibility for both containerized tools and script execution across different runtimes.

4.2.7 Job Execution Pipeline

Job Execution in Uspace follows as:

1. The user submits a job definition via `/job`.
2. The `JobDispatcher` validates and prepares the job.
3. The `JobManager` handles scheduling and preparation for Kubernetes.
4. The `JobExecutor` decides upon the existing imaged applications, and sets up the appropriate variables and connections for I/O to the Storage System
5. The `JobExecutor` then launches the containerized workload on the Engine Execution Model (e.g Kubernetes API Jobs).
6. Runtime results are streamed to WSS and the output is saved on the Storage Provider (MinIO) of the path specified.

All jobs are tracked in the local job database with associated metadata, status, timestamps, and references to input/output files.

4.2.8 Job Scheduling

The `Uspace` service includes an embedded job scheduling mechanism implemented by the `JobManager` component. It operates on a producer-consumer model, where jobs submitted by authenticated users are pushed into a bounded queue (`jobQueue`). The size of this queue is configurable via `UspaceJobQueueSize`, defaulting to 100 entries.

Job execution is handled concurrently by a worker pool, constrained by a configurable parameter (`UspaceJobMaxWorkers`). Each job is processed in a dedicated goroutine, with the pool enforced via a buffered channel (`workerPool`) to prevent system overload.

When a job is dequeued, it is dispatched to the selected `JobExecutor`—either Docker-based or Kubernetes-based—determined during service initialization. This separation allows for modular testing and cloud-native deployment.

If the job queue is full, submissions are rejected, providing backpressure to upstream services or users. This model ensures that the system maintains a predictable load and resource profile.

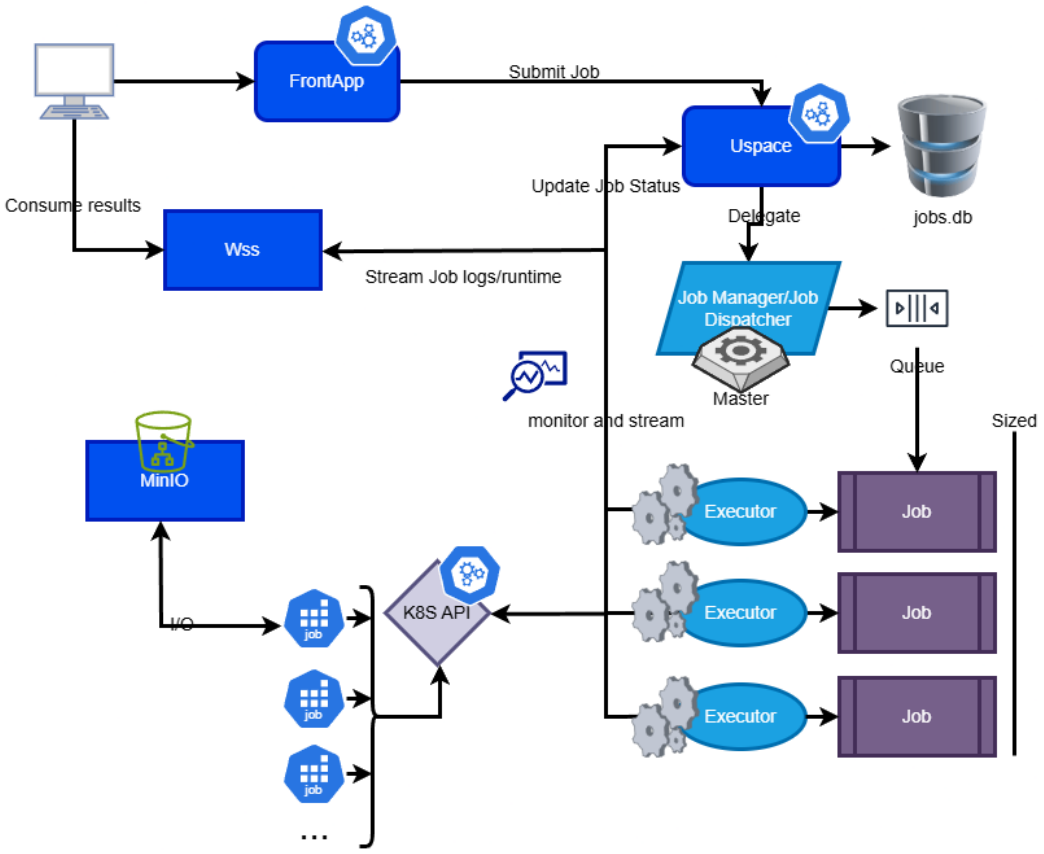


FIGURE 4.4: Job Lifecycle

4.2.9 Available Applications

The Uspace system supports a set of containerized applications that can be executed as jobs. These applications are defined as Docker images with a uniform execution contract. Each application:

- Receives the input and output paths via environment variables.
- Uses preconfigured MinIO credentials to fetch and store data.
- Executes its logic using the input file(s) and writes the results to the specified output location in MinIO.

The current applications integrated into the system are shown in Table 4.3. Each is versioned and can be expanded or modified independently.

Name	Image	Description	Version	Author	Status
duckdb	`kuspace:applications-duckdb-v1`	DuckDB SQL on MinIO object I/O	v1	k	available
pypandas	`kuspace:applications-pypandas-v1`	Python Pandas on MinIO object I/O	v1	k	available
octave	`kuspace:applications-octave-v1`	Octave code with MinIO object I/O	v1	k	available
ffmpeg [40]	`kuspace:applications-ffmpeg-v1`	FFmpeg commands on MinIO object I/O	v1	k	available
caengine [41]	`kuspace:applications-caengine-v1`	Custom engine with MinIO object I/O	v1	k	available
bash	`kuspace/applications-bash-v1`	Run Bash commands on objects	v1	k	available

TABLE 4.3: Currently supported applications in Uspace

FIGURE 4.5: Available Kuspace Applications

4.2.10 Integration and Security

Uspace authenticates user actions via JWT tokens issued by the Minioth authentication service. It verifies group memberships and resource permissions using the FsLite metadata store and enforces a Unix-style access control model.

The service is deployed as a Kubernetes `StatefulSet`, ensuring persistent identity and stable storage across restarts. Uspace stores its embedded database and FsLite metadata in a `PersistentVolumeClaim` (PVC), ensuring durability of user and job metadata.

Administrators can assign storage volumes, manage user data, inspect system metrics, and access operational logs through authenticated admin endpoints. All internal microservice communications are secured using a shared `ServiceSecret`, available at runtime to authorized components only.

4.3 Fslite: Storage Abstraction

FsLite is a modular metadata and volume management system that can operate either as an embedded library or as a standalone microservice. It is used within the Uspace module to abstract and manage virtual storage concepts, such as user volumes, datasets, and permissions, before interacting with the physical storage backend (MinIO).

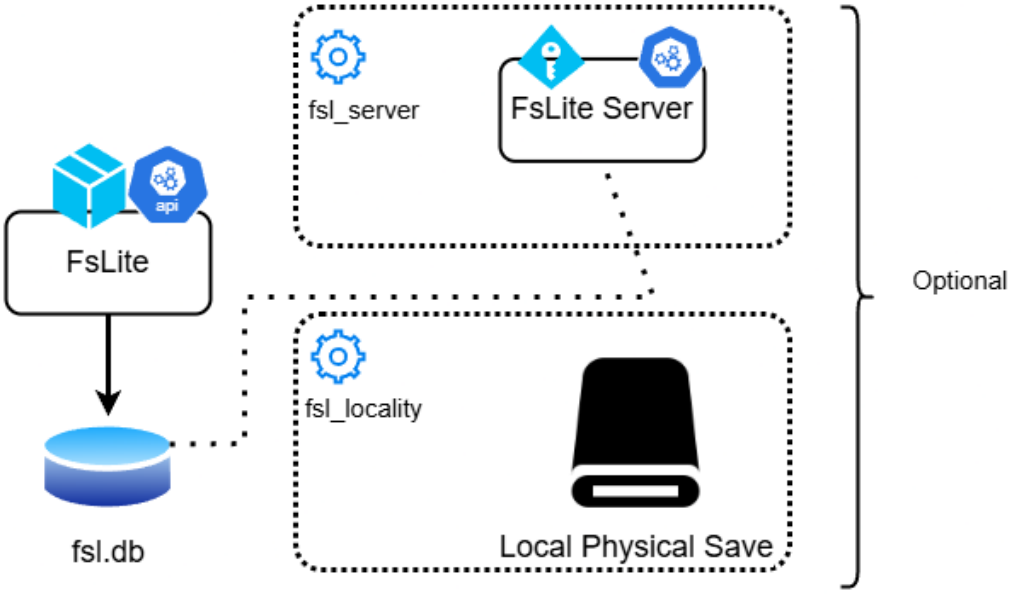


FIGURE 4.6: FsLite Block Diagram

4.3.1 Purpose and Design

The key design goal of FsLite is to provide a lightweight, configurable system to handle logical resource management, including tracking objects, managing access control, and storing volume definitions. It uses an embedded relational database (SQLite or DuckDB, depending on configuration) for metadata persistence.

FsLite enables the system to:

- Create and track logical volumes and resource entries before physical allocation.
- Maintain metadata about files, directories, and symbolic links.
- Perform access control enforcement based on user and group ownership.
- Act as an intermediate layer before delegating large-scale I/O to MinIO.

4.3.2 Deployment and Initialization

When deployed in standalone mode, FsLite operates as a RESTful microservice using the Gin web framework. It initializes by connecting to the configured database and optionally preparing a physical volume path on the local filesystem. If enabled, it creates a default volume using the directory and capacity limits specified in the system configuration.

The admin credentials (access and secret key) are inserted during startup. Optionally, the system can bypass authentication for development or local deployments.

4.3.3 API Endpoints

FsLite exposes a structured API grouped under a versioned path. The endpoints are divided into authentication, volume management, and resource operations. Key endpoints include:

Method	Path	Description
POST	`/login`	Authenticate and receive a token.
POST	`/admin/register`	Authenticate and receive JWT token.
POST	`/admin/volume/new`	Change password for the authenticated user.
DELETE	`/admin/volume/delete`	Delete a specified volume
GET	`/admin/volume/get`	Retrieve information about a logical volume
GET	`/admin/resource/get`	Retrieve metadata of multiple resources
GET	`/admin/resource/stat`	Retrieve more detailed metadata of a specific resource.
DELETE	`/admin/resource/delete`	Delete a specified resource.
POST	`/admin/resource/upload`	Upload resource content to volume.
GET	`/admin/resource/download`	Download a resource from storage.
GET/PATCH/DELETE	`/admin/user/voumes`	CRUD user volumes.

TABLE 4.4: FsLite public API endpoints

4.3.4 Integration in the System

FsLite is integrated into the Uspace module, which delegates resource and volume tracking tasks to it. This modular separation allows Uspace to offload concerns related to volume capacity, ownership verification, and metadata persistence to FsLite.

Its database-backed design allows stateless operation from the perspective of the consuming services. It also enables simulation of filesystem-like permissions using UNIX-style ownership and mode bits.

Local vs Remote Operation

FsLite can operate in two modes:

- **Local Mode:** FsLite creates a data directory on the host machine and directly stores uploaded files, suitable for testing or single-node deployments.
- **Remote Mode:** FsLite acts purely as a metadata layer; files are stored in MinIO or other backends using volume references.

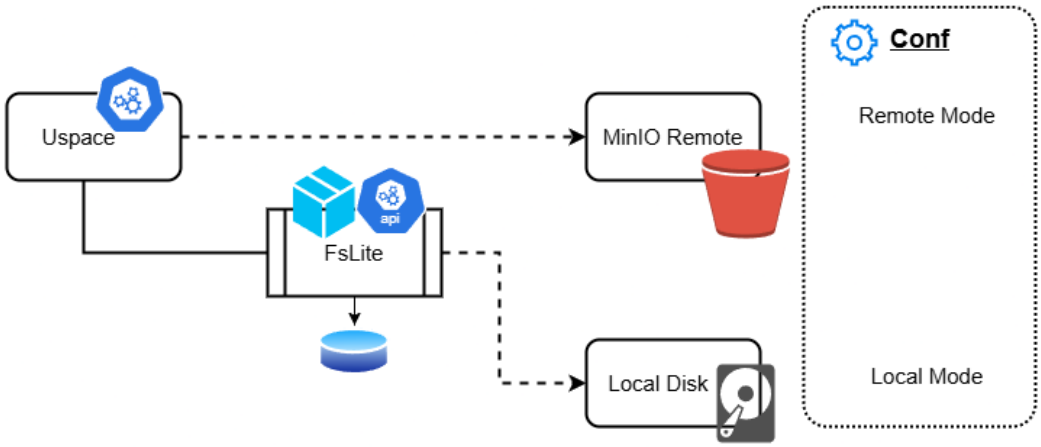


FIGURE 4.7: FsLite Integration

4.4 Storage Layer (MinIO)

MinIO serves as the system's object storage backend, offering a scalable, S3-compatible API for storing and retrieving binary data such as uploaded datasets and job output files. It is tightly integrated with Kubernetes and can be accessed from within containers via network mounts or HTTP APIs.

Each user's data is stored under isolated bucket structures or path prefixes, ensuring access control through both object path naming conventions and Fslite-based permissions. MinIO's stateless nature and support for erasure coding make it a robust choice for managing large data volumes in a distributed environment.

MinIO allows the platform to scale horizontally and reduces the need for traditional shared volumes or file servers. Its integration with the rest of the system ensures that users and jobs can read and write data in a uniform and efficient manner.

The MinIO Service is tightly connected to the Uspace Service. Uspace is authorized to perform admin operations on MinIO and also allows in its scope the Jobs spawned to fetch and load data to it.

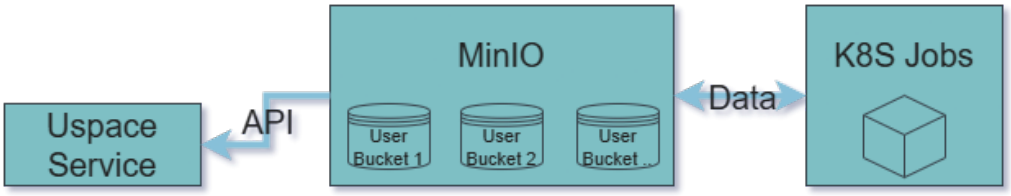


FIGURE 4.8: Minio Service Overview

MinIO is deployed as a Kubernetes StatefulSet backed by persistent volumes. This ensures durability of user data across pod restarts and facilitates scale-out configurations when needed.

Each service that interacts with MinIO is granted scoped credentials based on service identity. Access control is further enforced using path prefixes aligned with Fslite's logical resource mappings. The system may optionally use signed URLs for temporary external access to datasets.

4.5 Minioth: Authentication Service

Minioth [42] is a custom authentication and authorization microservice responsible for managing user identities and enforcing access control within the system. It implements a secure token-based authentication model using JSON Web Tokens (JWT) [43], allowing users to authenticate once and interact with other services securely.

Minioth supports essential user and group management operations such as registration, login, group assignment, and permission checking. It exposes

a RESTful API [44] that other services use to validate user tokens and retrieve identity-related metadata (e.g., UID, GID). This decouples authentication logic from the rest of the system, promoting modularity and reusability.

All user interactions begin with Minioth, and its integration is critical to enabling multi-user isolation, secure job execution, and controlled access to shared data resources.

Minioth aspires to become a fully capable identity provider!

4.5.1 Authorization Details

Minioth implements a simplified Role-Based Access Control (RBAC) model centered around user groups. Each user is assigned a unique primary group upon creation, which serves as the basis for resource ownership and sharing semantics. Group membership information is embedded in the issued JWTs, enabling downstream services to perform access checks without additional lookups.

At present, the system distinguishes between regular users and administrators. Membership in the `admin` group grants elevated privileges, including access to user and group management endpoints, system introspection, and key rotation. All other users are constrained to operations permitted within their assigned roles and groups.

4.5.2 Authentication Details

The Minioth authentication service implements secure, configurable user login and token issuance. Upon a successful login request via the `POST /login` endpoint, the system issues a JSON Web Token (JWT) containing the authenticated user's identity claims.

Token Signing Algorithm Clients can specify the desired signing algorithm by including the optional HTTP header:

`X-Auth-Signing-Alg: HS256` or `RS256` [45]

- **HS256** — HMAC using SHA-256 [46], with a system-defined shared secret.
- **RS256** — RSA signature using SHA-256, with a system-configured private/public key pair. The public key is exposed via the JWKS endpoint (`/.well-known/jwks.json`).

If the header is omitted, the system uses the default signing algorithm defined in the configuration file. This design supports interoperability with external identity providers and ensures future extensibility toward OpenID Connect.

Password Hashing User passwords are never stored in plain text. Instead, all passwords are hashed using the **bcrypt** [47] algorithm prior to storage. The hashing cost (also referred to as the computational work factor) is system-defined and configurable. It controls the computational difficulty of the hashing process, allowing the system to be tuned for performance versus security:

Algorithm 4 Password Hashing using `bcrypt`

```
1: procedure HashPassword(password, cost)
2:     salt ← GenerateSalt(cost)
3:     hash ← Bcrypt(password, salt)
4:     return hash
```

A higher cost increases the time required to compute the hash, improving resistance to brute-force attacks at the expense of login speed.

4.5.3 Pluggable Authentication Handlers

Minioth is designed with extensibility in mind, offering a flexible backend architecture based on pluggable *handlers*. A handler is a concrete implementation of a unified interface responsible for managing users, groups, and authentication state.

This architecture decouples the authentication logic from the underlying storage mechanism, enabling multiple interchangeable backends with minimal effort. All handlers implement the following interface:

Algorithm 5 Abstract Handler Interface (pseudocode)

function INIT
function USERADD(User)
  Returns: (UID, primaryGroup, Error)
function USERDEL(UID)
function USERMOD(User)
function USERPATCH(UID, Fields)
function GROUPADD(Group)
  Returns: (GID, Error)
function GROUPDEL(GID)
function GROUPMOD(Group)
function GROUPPATCH(GID, Fields)
function AUTHENTICATE(Username, Password)
  Returns: User or Error
function SELECT(ID)
  Returns: Any
function PURGE
function CLOSE

Available Handlers:

- **Database Handler:** A fully featured implementation that uses a relational database backend (such as SQLite or DuckDB). This handler persists user, group, and credential data in structured tables, enabling query flexibility, indexing, and transactional safety.
- **Plain Handler:** A minimalist implementation that stores user-related data in three flat files, following a UNIX-like structure:
  - `mpasswd` — stores basic user account data (username, UID, GID)
  - `mgroup` — stores group records (name, GID, members)
  - `mshadow` — stores password hashes

  This mode is particularly useful for lightweight deployments, testing, or portability.

Each handler supports core operations such as user/group creation, deletion, modification, and password-based authentication. By implementing the same interface, they are interchangeable at runtime, allowing the system to

be easily configured for different deployment targets or performance/security trade-offs.

New handlers suitable for other databases can be implemented.

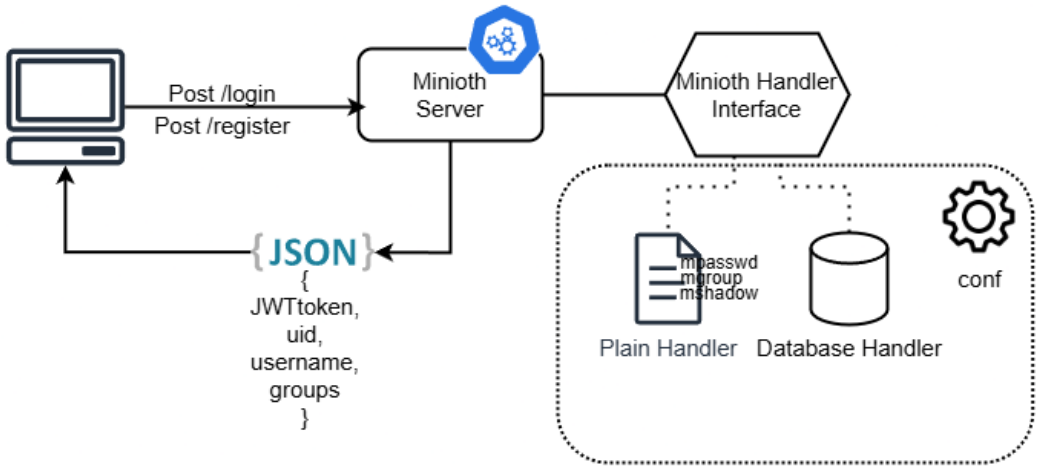


FIGURE 4.9: Minioth Block Diagram

4.5.4 Minioth Public API Endpoints

Method	Path	Description	Auth
POST	`/v1/register`	Register a new user.	No
POST	`/v1/login`	Authenticate and receive JWT token.	No
POST	`/v1/passwd`	Change password for the authenticated user.	Yes (user/admin)
GET	`/v1/user/me`	Get information about the token's user.	Token
GET	`/v1/user/token`	View current token details.	No
POST	`/v1/user/refresh-token`	Request a new access token using a refresh token.	No
GET	`/v1/swagger/*any`	Swagger UI for live API documentation.	No

TABLE 4.5: Minioth public/user API endpoints

4.5.5 Minioth Admin API Endpoints

Method	Path	Description
GET	`/v1/admin/audit/logs`	Retrieve system audit logs.
POST	`/v1/admin/hasher`	Generate bcrypt hashes from plaintext passwords.
POST	`/v1/admin/verify-password`	Compare a password against a stored hash.
GET	`/v1/admin/users`	List all registered users.
GET	`/v1/admin/groups`	List all groups and their members.
POST	`/v1/admin/useradd`	Add a new user to the system.
DELETE	`/v1/admin/userdel`	Delete a user by UID or username.
PATCH	`/v1/admin/userpatch`	Update selected fields of a user.
PUT	`/v1/admin/usermod`	Fully replace a user record.
POST	`/v1/admin/groupadd`	Create a new group.
PATCH	`/v1/admin/grouppatch`	Update specific fields of a group.
PUT	`/v1/admin/groupmod`	Fully update a group definition.
DELETE	`/v1/admin/groupdel`	Remove a group from the system.
POST	`/v1/admin/rotate`	Rotate the JWT signing key in use.
GET	`/v1/admin/system-conf`	View current server configuration.

TABLE 4.6: Minioth public/admin API endpoints

Essentially admin features include full user/group lifecycle management and audit logging, key rotation and token inspection mechanisms.

An Authorization header is required with the JWT token as a Bearer.

4.5.6 Integration

Minioth is integrated into the overall system architecture as a Kubernetes StatefulSet, with user and group data persisted on a dedicated Persistent Volume Claim (PVC). It serves as the central authentication authority by issuing JWT tokens consumed by other services, primarily the Frontend application, to authorize user actions.

To ensure secure inter-service communication, all microservices, including Minioth, share a common `ServiceSecret` key. This key enables mutual trust and token verification across services. Additionally, the JWT signing key is generated at deployment time via a configurable secret generator and securely injected into the Minioth service configuration.

4.6 Frontend and WebSocket Server

4.6.1 Overview

The Frontend service acts as the primary user interface of the system, offering a web-based environment where users can register, log in, upload datasets, browse their resources, and submit batch analytical jobs. It is implemented using standard web technologies and communicates with backend services via RESTful APIs.

Beyond serving static HTML templates and handling client-side rendering, the Frontend plays a critical role in request mediation. It intercepts client requests, performs sanitization and validation, and attaches authentication tokens or service secrets where necessary. This enables secure and orchestrated communication between the user interface and core backend services, especially the Uspace API.

4.6.2 WebSocket Server (WSS)

The WebSocket Server (WSS) is implemented as a separate microservice that provides real-time, bidirectional communication capabilities between clients and the system. Its primary purpose is to support streaming logs, job status updates, and other event-driven feedback mechanisms related to job execution.

Clients interact with the WSS by issuing HTTP upgrade requests on a dedicated registration endpoint, specifying a Job ID (JID) and their desired role: *Producer*, *Consumer*, or *JackOfAllTrades*. The server internally manages job-specific WebSocket channels. If a connection for the specified JID does not exist, it is initialized; otherwise, the client is added to the existing communication stream.

Role	Description
`Producer`	Sends messages to the WebSocket channel. Typically used by job execution pods to stream logs, status updates, or results.
`Consumer`	Subscribes to a WebSocket channel and receives messages sent by the associated producer. Ideal for real-time job monitoring.
`JackOfAllTrades`	Can both send and receive messages within the WebSocket channel. Useful for debugging or hybrid clients.

TABLE 4.7: Supported WebSocket communication roles

Additionally, an administrative endpoint allows forced disconnection of specific channels or participants, enabling control over job-specific streams.

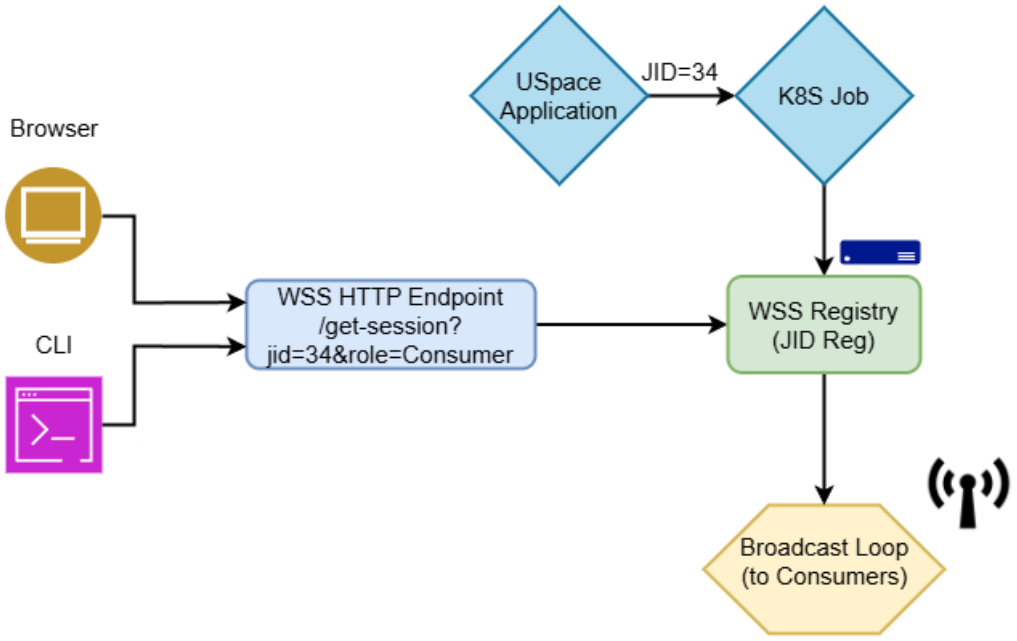


FIGURE 4.10: Wss Block Diagram

4.6.3 Frontend–WSS Separation of Concerns

While both the Frontend and WSS are part of the user-facing layer, they are deployed as independent services with different responsibilities. The Frontend focuses on rendering the user experience and forwarding REST-based job and resource operations to Uspace, while the WSS focuses on real-time feedback for asynchronous job execution.

This separation allows the system to maintain a clear modular boundary between interactive request-response behavior and event-driven streaming, simplifying system maintenance and scaling.

4.6.4 Frontend Technology Stack

The Frontend service is designed to offer a minimal yet responsive user experience, relying on standard web technologies and a server-side rendering model.

FIGURE 4.11: FrontEnd Technologies

- **HTMX** [48] is used as the main HTTP client, enabling dynamic content updates by issuing declarative AJAX requests embedded in HTML attributes. This reduces the need for complex JavaScript frameworks.
- **Go Templates** render HTML pages server-side using Go's `html/template` engine. This allows safe and dynamic construction of views tied directly to backend logic.
- **Vanilla JavaScript** is used for basic DOM manipulation, file previews, and UI behaviors. The system avoids heavy client-side frameworks to maintain performance and reduce complexity.
- **CSS** is used to apply styling and layout. The design aims for clarity and usability, with lightweight responsive behavior for handling multiple screen sizes.
- **Gin Framework** in Go powers the backend of the Frontend service, handling routing, middleware, and template rendering.

Security is enforced via JWT-based middleware and service-to-service secret headers. All critical interactions (e.g., job submissions, resource uploads) are protected with authentication and validated on the server.

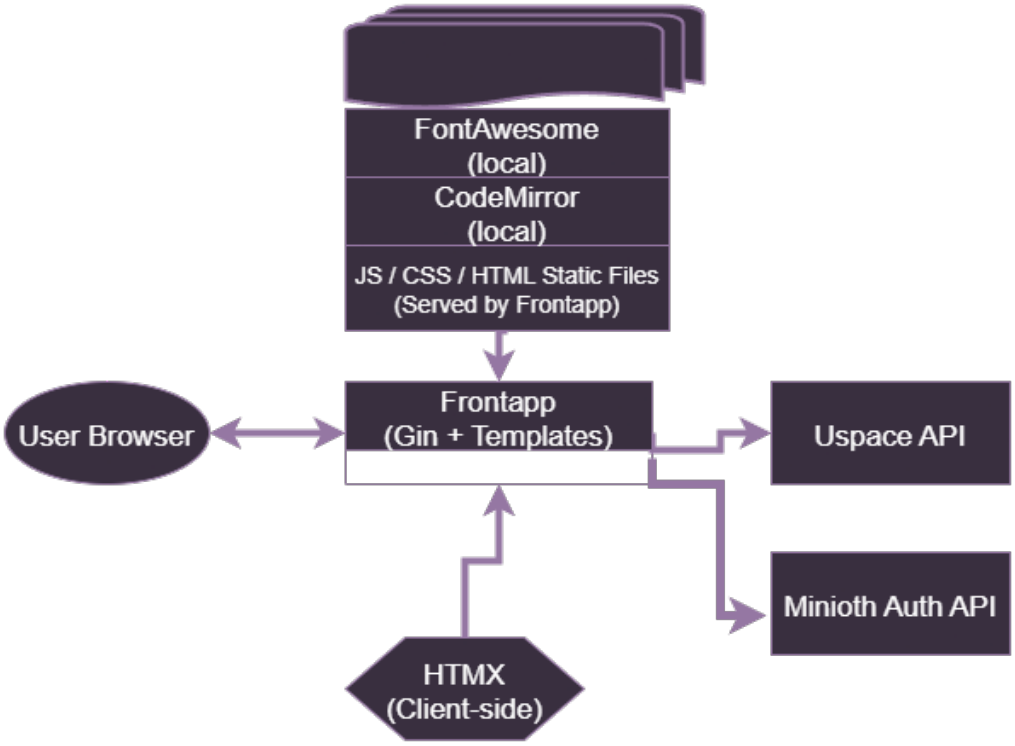


FIGURE 4.12: Frontapp Overview diagram

To support syntax-highlighted editing for user-submitted scripts (e.g., SQL, Python), the frontend integrates CodeMirror, a versatile in-browser code editor [49].

Interface icons, tool indicators, and status badges are rendered using the Font Awesome Free icon library [50], providing visual consistency and accessibility. Both technology stacks are served from Frontapp, bypassing any cdn.

4.6.5 Integration

Both the Frontapp and the Wss are integrated in the system as Kubernetes Deployments remain stateless. There is also ephemeral logging, which can be used in the future.

4.7 Kubernetes Integration

Kubernetes functions as the orchestration backbone of the system. It is responsible for deploying, scheduling, and executing containerized jobs submitted by users. Uspace leverages Kubernetes Jobs to launch sandboxed environments in which user-specified tools (e.g., DuckDB, Octave) are executed. It also deploys each Microservice itself, tightening overall security and exposes only the desired service to the end user. In this case everything should be used and accessed via Frontapp.

Additionally, Kubernetes' Persistent Volume Claims (PVCs) or object storage access is used in alignment with StatefulSets to persist data. Everything deployed on K8S is namespaced.

Each service is included with a Kubernetes Service and configured accordingly so that there is inter-service communication. There is also use for NodePort Services for the system entrypoints.

Overall, the integration with Kubernetes allows the system to inherit properties such as fault tolerance, scaling, and container lifecycle management. It enables infrastructure-level abstraction so that users do not need to interact directly with Kubernetes primitives.

5 System Usage & Execution Flow

5.1 Deployment and Tooling

This section outlines how the system is deployed, the development tools used throughout the implementation, and the underlying infrastructure model.

5.1.1 Deployment Procedure

The system is built to run on Kubernetes, with support for local development and testing via Minikube and Docker Desktop. While it has not been extensively validated on production-grade Kubernetes clusters, all features and microservices operate successfully in local Kubernetes environments.

Deployment is orchestrated using a dedicated Go-based tool named kuspacectl. This CLI tool simplifies building, deploying, and tearing down the entire stack. It handles:

- Building Docker images for each microservice.
- Creating or destroying the Kubernetes namespace `kuspace`.
- Applying all Kubernetes manifests found under the `/deployment/kubernetes` directory.
- Generating secrets and initial configurations.

In addition to kuspacectl, traditional Makefiles are available to facilitate tasks such as code building, static analysis, cleaning, documentation generation, and running unit tests.

The development process incorporates:

- **Golang** with modules for all backend microservices.
- **golangci-lint** [51] for linting and static analysis.
- **go test** for unit and integration testing.
- **Swagger** [52] documentation generation for HTTP APIs.

- **Golds** [53] for Golang documentation generation.
- **Air** [54] for live hot reload in Go apps/services. Used mostly for the Frontapp development.
- **AI Assistance Tools**: During development and refinement, AI-assisted code review and language models (e.g., ChatGPT) were used to prototype logic, validate syntax, and improve structural clarity. These tools were used judiciously, with all code and documentation critically evaluated and verified by the author.

5.1.2 Infrastructure Overview

The deployment manifests are defined in the `/deployment/kubernetes` directory. This includes the full Kubernetes configuration for each service:

- **Namespaces:** All resources are grouped under the `kuspace` namespace.
- **ConfigMaps:** Service-specific configuration values, including environment variables and runtime parameters.
- **Secrets:** Secure key material such as JWT signing secrets and service authentication keys.
- **Deployments & StatefulSets:** Stateless services (like the Frontend) use `Deployment` objects, while stateful components (like `Minioth` and `Uspace`) are defined as `StatefulSets` with persistent volume claims (PVCs).
- **Services:** ClusterIP services are exposed internally for inter-service communication, while the Frontend may optionally be exposed externally.

The system architecture encourages modularity and scalability. All microservices are independently containerized and can be redeployed or scaled individually.

5.2 System Access and Communication

5.2.1 Access Points and Interfaces

The system is primarily designed to be accessed through the web-based `Frontapp` user interface, which provides a convenient entry point for typical end-users. However, developers or system integrators can interact directly with the

backend services via command-line tools or custom HTTP clients, such as `curl`, `httpie`, or language-specific libraries.

All core services expose their functionality over RESTful APIs. These APIs are documented using Swagger (OpenAPI), with each service providing a dedicated endpoint for live exploration and testing (e.g., `/swagger/index.html`). These interfaces offer comprehensive descriptions of all available routes, expected parameters, response structures, and authorization requirements.

5.2.2 Request-Response Model

Communication with the backend services follows the standard HTTP request-response model. Beyond typical REST conventions, several HTTP headers are critical for correct operation:

- `Authorization`: Contains the Bearer token issued by the authentication service (`Minioth`). This token encodes the user ID, group membership, and role, and is required for all authenticated actions.
- `X-Service-Secret`: Used to authenticate requests between microservices or to authorize trusted developer actions. Each internal service shares a secret key that validates privileged operations.
- `Access-Target`: A custom header required by the `Uspace` service. It encodes both the target of the action (volume ID, resource path) and the identity context (user ID, group IDs) to enforce resource-level authorization. Its format follows:

  ```
  <volume_id>:<volume_name>:<resource_path> <user_id>:[group_id,group_id,...
  ```

  This design allows `Uspace` to separate identity verification (via JWT) from fine-grained access control at the resource level.

5.2.3 Error Handling and Feedback

Each microservice is responsible for validating input, authenticating requests, and returning appropriate error codes and messages using standard HTTP status codes (e.g., `401 Unauthorized`, `403 Forbidden`, `404 Not Found`, `500 Internal Server Error`).

On the frontend, the `Frontapp` uses HTMX event hooks to capture responses and dynamically update the user interface. Errors are caught and displayed

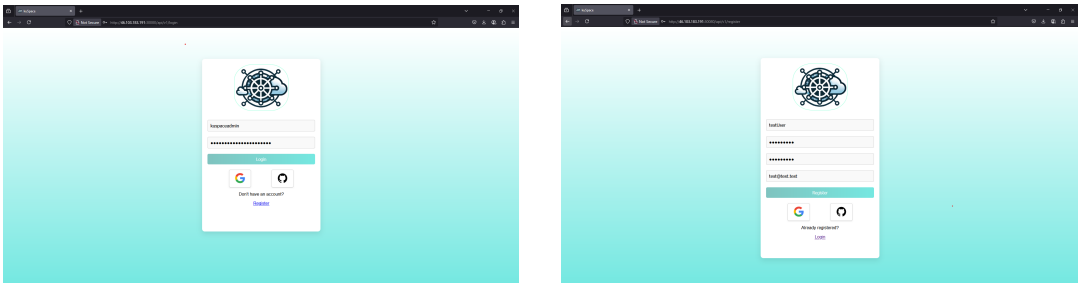
as user-friendly feedback elements, while success events trigger content refreshes, loading indicators, or UI transitions. This approach enhances responsiveness and usability without relying on full client-side frameworks.

5.3 End-to-End Workflow

5.3.1 Overview and User Roles

Currently there is only a distinction between admins and users. Admins are the users that bear the **admin** role. Every new registered user is simply a regular user. Admins can edit and promote other users to admins, by simply giving them the **admin** group.

There is a plan to incorporate the **mod** (moderator) group which will grant users more privileges than regular users but fewer than admins.



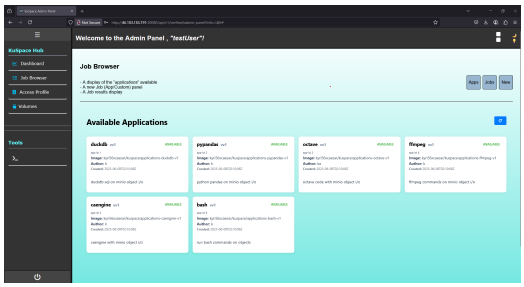
(A) Login Page (B) Register Page

FIGURE 5.1: Kuspace Login and Register Pages

The OAuth by Google and Github structure is implemented, yet not fully functional at this point.

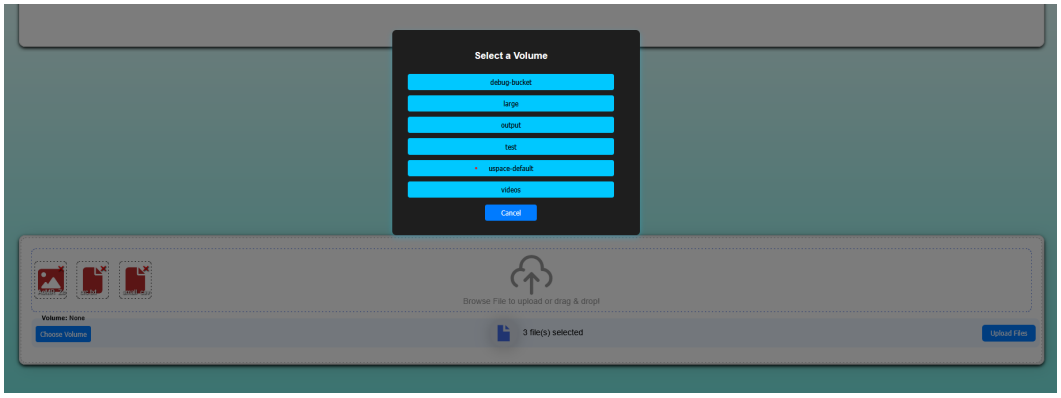
There are restrictions on user registration: passwords must match, meet a minimum length, and contain a variety of character types. Usernames are also constrained by minimum and maximum length requirements, and certain names are prohibited.

5.3.2 User Perspective



(A) Job Browser and App Selection

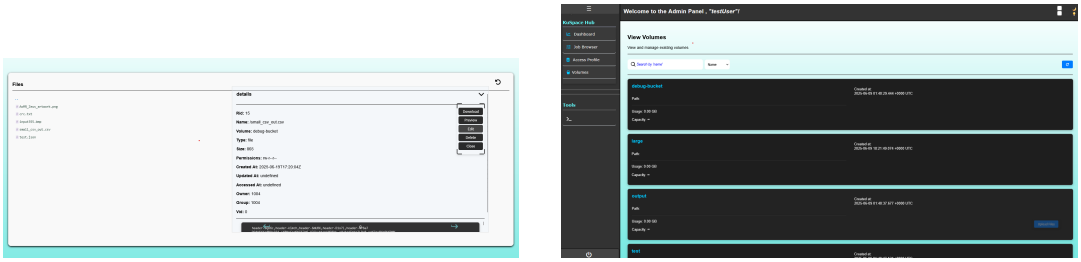
(B) Access Info and Change View



(C) File Upload with Volume Selection

FIGURE 5.2: User Interface Actions in Kuspace

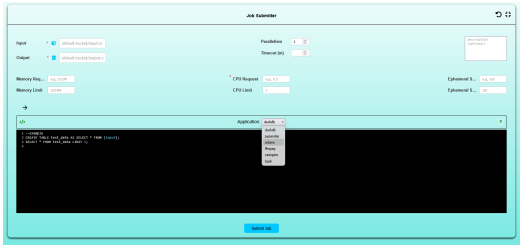
Here we can see the agency a user has to change his credentials, upload files on existing volumes and see the available applications.



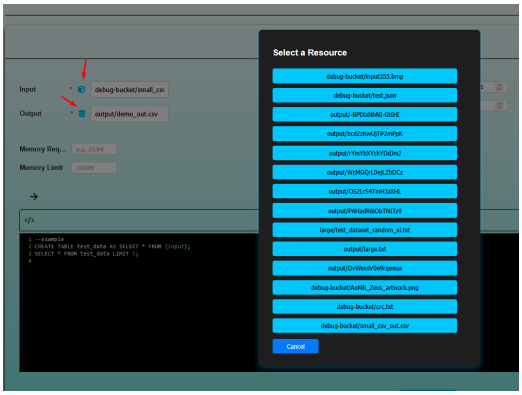
(A) File Traversal and Options (B) Volume Overview

FIGURE 5.3: Kuspace user views for navigating files and inspecting volumes.

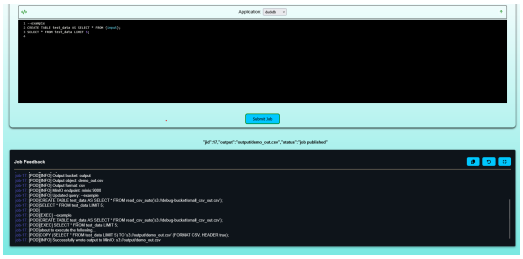
These interfaces allow users to browse directories, inspect resources, and understand volume boundaries within the system. Navigation tools and volume metadata views help contextualize jobs and uploaded files.



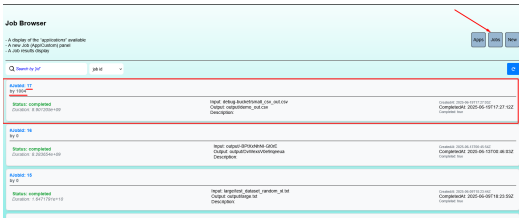
(A) Job Submitter Panel



(B) Input/Output Selection



(C) Job Published & Execution Streaming



(D) Job Execution Results

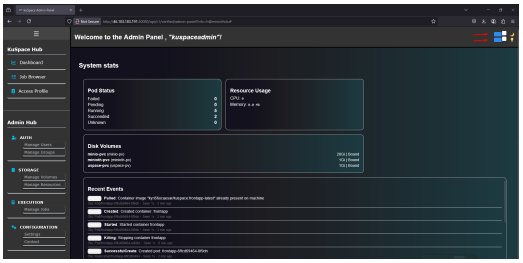
FIGURE 5.4: Kuspace user interface for submitting jobs.

The workflow includes selecting an application, specifying input/output paths, tuning job parameters, and viewing results.

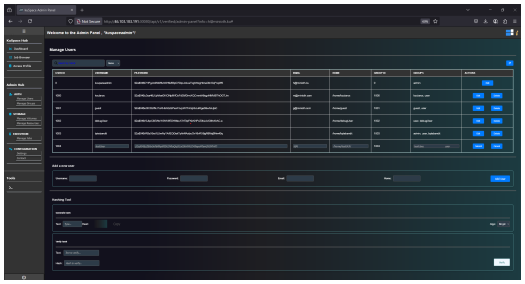
5.3.3 Admin Perspective

Kuspace advanced administrative actions, including job mutation and system configuration view. These tools help define volumes, manage jobs, and manage resource properties. These views allow administrators to control access and maintain structure.

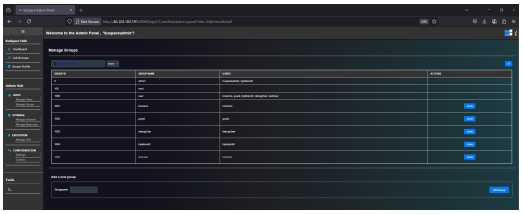
There are also stylistic options, for example dark-mode, and less verbosity in the top right corner.



(A) Admin Dashboard Overview



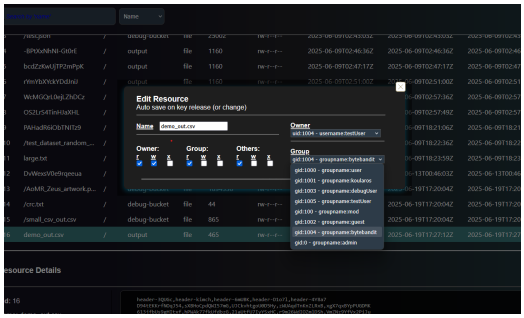
(B) User Management Panel



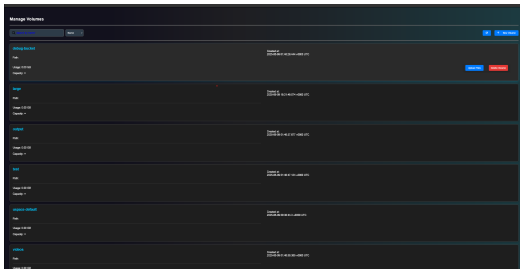
(C) Group Management

(D) Resources Management

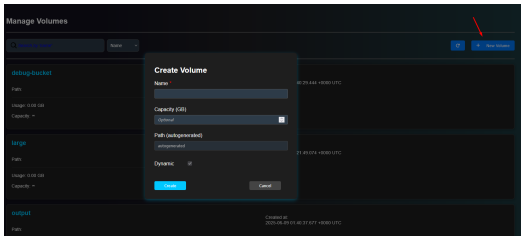
FIGURE 5.5: Kuspace admin interface for managing users, groups, and resources.



(A) Edit Resource Metadata

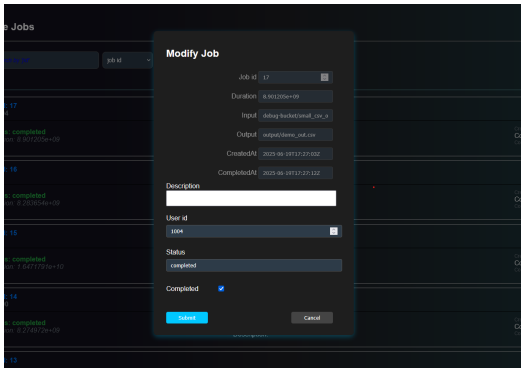


(B) Volume Overview



(C) Create New Volume

(D) Manage Jobs



(E) Modify Submitted Job

(F) Settings Panel

FIGURE 5.6: Kuspace admin views for storage and job management.

5.3.4 Job Examples

In this section, we demonstrate two distinct job examples that showcase the system's ability to execute containerized applications over uploaded data:

- A **Bash-based job** that counts the number of occurrences of the string `"bash"` within a 4GB file of random characters.
- A **DuckDB-based job** that computes a histogram of the first character of each line in a structured text file.

Example 1: Bash Job – Count Occurrences

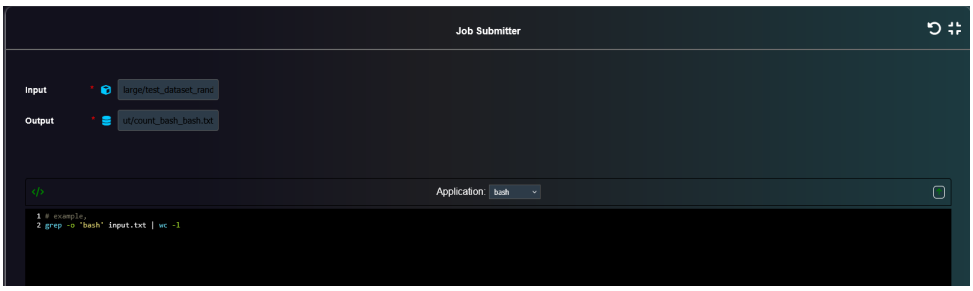
The input file consists of 4GB of random characters including the word `"bash"` scattered throughout.

Job Logic:

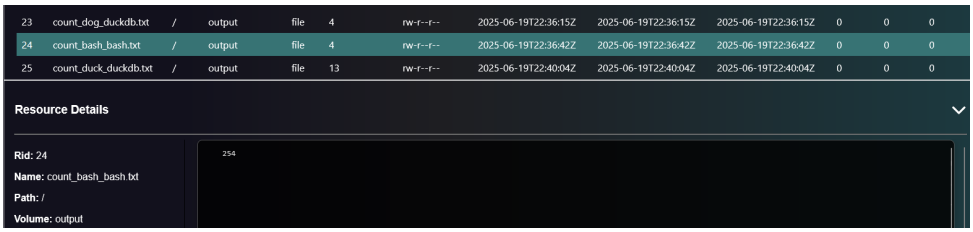
```
grep -o 'bash' input.txt | wc -l
```

Result: `254` occurrences of `"bash"` were found.

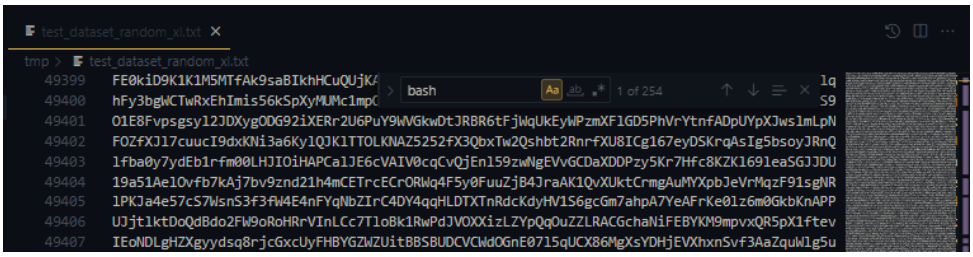
This result can be verified locally using a terminal or text editor (assuming sufficient memory).



(A) Bash App job logic submitted via UI



(B) Bash App job output



(C) Manual result verification (e.g., via editor)

FIGURE 5.7: Bash job: Counting occurrences of a word in a large file

Example 2: DuckDB Job – Line Start Histogram

This job uses a smaller text file to illustrate SQL capabilities more clearly. Each line in the file starts with a capital letter forming the line:

```
DUCKDBBASHOCTAVE
```

Job Logic:

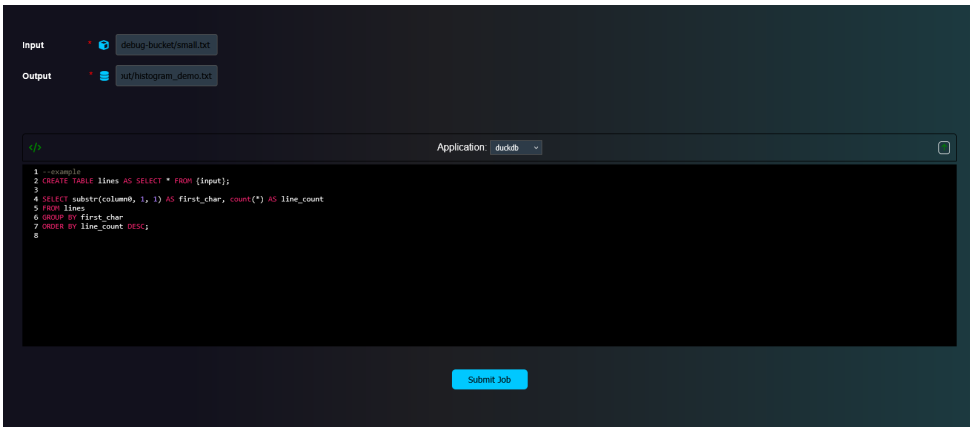
```
CREATE TABLE lines AS SELECT * FROM {input};

SELECT substr(column0, 1, 1) AS first_char, count(*) AS line_count
FROM lines
GROUP BY first_char
ORDER BY line_count DESC;
```

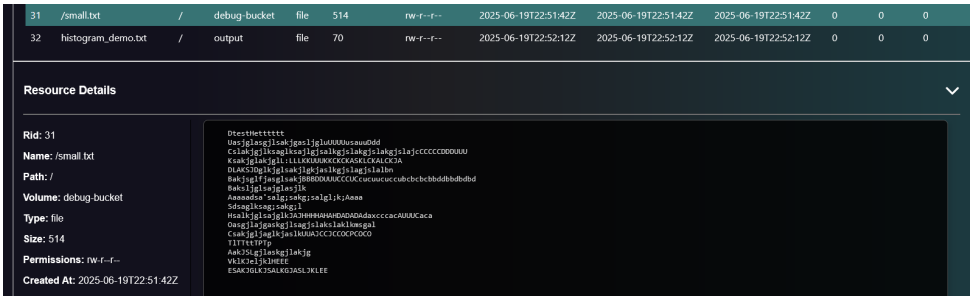
Result:

```
D,2
C,2
B,2
A,2
U,1
K,1
S,1
H,1
O,1
T,1
V,1
E,1
```

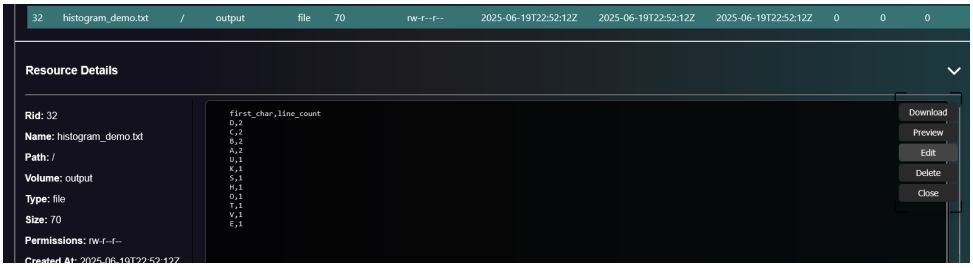
As expected, each starting letter is counted accurately and returned in descending order of frequency.



(A) DuckDB job query and configuration



(B) DuckDB job input file (preview)



(C) DuckDB job result

FIGURE 5.8: DuckDB job: Character frequency histogram from first line characters

6 Evaluation and Robustness

6.1 Scalability Testing Methodology

Scalability tests were performed using the `bombardier` load testing tool. The primary goal was to evaluate the system's ability to handle concurrent HTTP requests to key API endpoints.

Tests focused on the main user-facing endpoints, including data retrieval, job submission, and authentication. Each endpoint was tested independently under varying levels of concurrency and request volumes. Resource utilization was monitored using Kubernetes-native tools (`kubectl top`) to correlate system load with CPU and memory consumption.

Endpoint	Concurrency (C)	No. Requests Sent	Notes
`/api/v1`	100	1,000,000	(Front) Main homepage or landing data.
`/v1/login`	100	1,000,000	(Auth) Login token generation load.
`/v1/register`	100	1,000,000	(Auth) User registration endpoint.
`/api/v1/resource/upload`	50	500	(Uspace) Upload files.
`/api/v1/resources`	100	1,000,000	(Uspace) List resources.
`/api/v1/job`	50	500	(Uspace) Job submission endpoint.

TABLE 6.1: Scalability Test Configuration for API Endpoints

Endpoint	Avg RPS	Max RPS	Avg Latency	Max Latency	Error Rate (%)	Throughput	Duration	Notes
`/api/v1`	3057	24599.13	33s	950ms	0	12.32MB/s	328s	Very stable
`/v1/login`	6087	11730	16 17ms	244ms	0	5.13MB/s	164s	Stable
`/v1/register`	3692	8129	27 90ms	2s	0.01	1.37MB/s	270s	98 timeouts
`/api/v1/resources`	1470.31	4296.58ms	661ms	68.02	0	151.14MB/s	680s	Stable
`/api/v1/job`	115.57	549.25	387.99ms	2s	6.4%	87.98KB/s	4s	load-sensitive

TABLE 6.2: Scalability Test Results for API Endpoints: Performance Metrics

Endpoint	Concurrency (C)	Throughput (files/sec)	Avg Latency (ms)	Max Latency (ms)
/api/v1/resource/upload	50	111.22	322.29	3058.23

TABLE 6.3: Resource Upload Endpoint Performance

6.1.1 API Endpoint Performance

The frontend endpoint (`/api/v1/`) was subjected to a load of 1,000,000 requests using 100 concurrent connections. The service sustained an average throughput of approximately 3,058 requests per second, with an average latency of 33 milliseconds. No errors were recorded during the test. his result demonstrates the scalability of stateless endpoints under high concurrency. Resource monitoring indicated that CPU utilization remained within acceptable limits, with no service crashes or degradation observed.

6.1.2 Authentication Endpoint

Login

The authentication endpoint (`/v1/login`) achieved a sustained throughput of approximately 6,089 requests per second with 100 concurrent connections. Average latency was 16.4 milliseconds, with a maximum observed latency of 245 milliseconds. The system processed one million requests with zero errors or failed connections.

These results confirm that the authentication service can handle significant load while maintaining low latency. The observed throughput reflects the stateless nature of the login process, where token issuance relies primarily on CPU and lightweight database read/write operations. Resource usage remained within acceptable thresholds during testing.

It is noted that throughput and latency may vary based on deployment configuration and backend load; however, this test demonstrates the scalability of the API authentication layer under typical demand scenarios.

Register

The user registration endpoint (`/v1/register`) was tested under a high-contention scenario where one million requests attempted to register the same username concurrently. As expected, only the first request succeeded, while all subsequent requests were rejected with HTTP 4xx errors due to violation of the username uniqueness constraint.

Despite the high failure rate being the correct functional behavior, the service sustained an average throughput of 3,692 requests per second, with an average latency of 27 milliseconds. The maximum observed latency reached 2 seconds during periods of elevated database constraint checking. A total of 98 timeouts occurred during the test, likely attributable to transient database locking under contention.

This test demonstrates that the authentication service correctly enforces username uniqueness under extreme load, while maintaining high throughput and acceptable latency for failure responses. The observed timeouts suggest a potential opportunity for optimization in handling high write contention, particularly in future deployments with distributed or more robust database backends.

Uspace Main Endpoint

The resource upload endpoint (`/api/v1/resource/upload`) was tested with 500 file uploads using 50 concurrent clients. The system achieved a throughput of approximately 111 files per second, with an average latency of 322 milliseconds per upload. The maximum observed latency peaked at 3.05 seconds, likely due to disk I/O or database locking under concurrent load. All requests succeeded with no errors.

This result demonstrates that the system's upload service handles concurrent uploads reliably. While upload operations naturally exhibit higher latency compared to lightweight API calls, the overall performance remains stable and predictable. Further scalability can be achieved with distributed storage or parallelized backends.

The resource listing endpoint (`/api/v1/resources`) was tested with one million requests at a concurrency level of 100. The system sustained an average throughput of 1,470 requests per second, with an average latency of 68 milliseconds and a maximum observed latency of 661 milliseconds.

No errors were recorded during the test, demonstrating the system's ability to handle high read loads efficiently. The high throughput of 154 MB/s suggests that the response payloads are non-trivial, likely depending on the number of resources returned per request. These results indicate that the metadata querying and database read paths are highly scalable in the current architecture.

Job Submission Performance

The job submission endpoint (`/api/v1/verified/jobs`) was evaluated with 500 concurrent requests using 50 parallel connections. The system achieved an average throughput of 115 requests per second, with an average latency of 388 milliseconds and a maximum latency of 2 seconds.

Out of 500 requests, 468 completed successfully while 32 failed due to timeouts, indicating a 6.4% error rate under load. This behavior is expected given the nature of job submission, which involves database writes, resource validation, and interaction with the job execution backend.

These results highlight that the job submission pathway is more sensitive to load compared to stateless API endpoints. Performance bottlenecks are due to the configuration of the jobs queue size on configuration file, which is set to 100. This limits the number of concurrent job submissions that can be processed, leading to increased latencies and timeouts under high load.

6.1.3 Scalability Evaluation

The current scalability evaluation is constrained to a single-node deployment due to the use of embedded databases (SQLite, DuckDB) which do not support distributed scaling. Consequently, while stateless API components can scale horizontally, the database remains a bottleneck for write-heavy workloads. Future scalability tests on a multi-node deployment with a distributed database backend are planned as future work.

6.2 Security

6.2.1 Security Model

The security model of this platform is designed around fundamental security principles, with a focus on user isolation, access control, and operational integrity within a multi-tenant environment. While the current implementation covers key aspects such as authentication, authorization, and availability, other areas like comprehensive audit trails and advanced runtime defenses are identified as areas for future development.

The main security components are summarized in Table 6.4.

TABLE 6.4: Security Aspects of the System

Aspect	Description	Implementation
Authentication (AuthN)	Verifies user identity before granting access.	JWT-based authentication using the `Minioth` service. Tokens encode user identity and group memberships.
Authorization (AuthZ)	Determines user permissions and access to resources after authentication. Controls actions like data access, and resource sharing/editing.	Unix-like permission model implemented in `FsLite`, using user/group-based access control (owner, group, other).
Availability	Ensures that services remain operational and resilient in the face of failures.	Leveraged via Kubernetes native fault tolerance: automatic pod restarts, health checks, and job rescheduling. Stateless services can be redeployed with minimal disruption.
Audit and Logging (Partial)	Provides traceability of user actions and security events. Essential for accountability, debugging, and incident response.	Currently limited to authentication events (login, token issuance). No full audit trail for resource access or job execution yet. Logs are primarily written to stdout for aggregation.

6.2.2 Security Evaluation

The system properly distinguishes between user and admin roles, enforces ownership and group permissions on data access, and guards access to privileged routes. However, a formal security audit has not been conducted. Future improvements could include:

- Apply different inter-service communication protocols (e.g., gRPC) to enhance security.
- Rate limiting and brute-force protection.
- Formal threat modeling and penetration testing.

7 Conclusions and Future Work

7.1 Conclusions

This thesis presented the design and development of a modular, microservice-oriented job execution system tailored for Kubernetes-based infrastructures. Through a combination of lightweight services, clearly defined APIs, and a user-focused frontend, the platform enables authenticated users to upload data, submit analytical jobs, and retrieve results in a secure and controlled environment.

The system was implemented entirely in Go, using technologies such as Gin for HTTP routing, MinIO for object storage, and Kubernetes for orchestrated execution. A pluggable job architecture and storage abstraction layer allow for flexibility and future adaptability. Even though a formal evaluation was not conducted, the system demonstrates real-world viability through functional integration and local deployments on Minikube and Docker Desktop.

7.2 Future Work

Despite its completeness as a prototype, several areas of the system lend themselves to meaningful expansion and improvement:

- **Caching and Optimization:** Introducing caching mechanisms (e.g., Redis) could reduce redundant computations and disk I/O, improving performance for repeated access to commonly used files or job results.
- **Job Pipelines:** Extending the job model to support pipelined execution—where the output of one tool feeds directly into another—would allow users to build more complex analytical workflows natively within the platform.
- **Distributed Job Models:** More sophisticated orchestration strategies could be explored, enabling distributed and cooperative execution of

job chains, possibly integrating with task queues or event-driven architectures like Kafka or NATS.

- **Application Generation Framework:** A meta-application that allows administrators or advanced users to define new containerized applications (with their input/output schema and logic) via a UI or DSL would significantly expand the platform's usability and extensibility.
- **CLI Client Tool:** To complement the web-based frontend, a command-line interface (CLI) tool could be developed, enabling power users and automation scripts to interact with the system more efficiently.
- **Evaluation and Benchmarking:** A structured performance and scalability analysis remains an open task. Real-cluster deployments and load testing would help validate assumptions about the system's robustness and identify further bottlenecks.

In conclusion, the system serves as a strong foundation for scalable, secure, and user-friendly batch processing on Kubernetes. The outlined future work suggests a wide potential for transforming it into a general-purpose analytical platform adaptable across domains.

Appendix: Source Code Access

The full source code of the project, including the backend services, FrontApp UI, and deployment configurations, can be found at:

https://github.com/kyri56xcaesar/kuspace

References

[1] Wes McKinney. “Data Structures for Statistical Computing in Python”. In: *Proceedings of the 9th Python in Science Conference* (2010). Ed. by Stéfan van der Walt and Jarrod Millman, pp. 51–56. DOI: 10.25080/Majora-92bf1922-00a.

[2] *Apache Hadoop*. https://hadoop.apache.org/. Accessed: 2025-06-11.

[4] D. Hardt. *The OAuth 2.0 Authorization Framework*. https://datatracker.ietf.org/doc/html/rfc6749. RFC 6749, IETF. Accessed: 2025-06-11. 2012.

[5] The Go Authors. *The Go Programming Language*. https://golang.org. Accessed: 2025-06-11. 2012.

[6] Manu Mtz-Almeida and Gin Contributors. *Gin Web Framework*. https://gin-gonic.com/. Accessed: 2025-06-17. 2024.

[7] *Kubernetes Documentation*. https://kubernetes.io/docs/. Accessed: 2025-06-11.

[9] *Kubernetes API Reference*. https://kubernetes.io/docs/reference/generated/kubernetes-api/. Accessed: 2025-06-17.

[10] *Jobs - Kubernetes Documentation*. https://kubernetes.io/docs/concepts/workloads/controllers/job/. Accessed: 2025-06-17.

[11] *Persistent Volumes - Kubernetes Documentation*. https://kubernetes.io/docs/concepts/storage/persistent-volumes/. Accessed: 2025-06-17.

[12] *Docker Documentation*. https://docs.docker.com/. Accessed: 2025-06-11.

[13] Dirk Merkel. “Docker: Lightweight Linux Containers for Consistent Development and Deployment”. In: *Linux Journal* 2014.239 (2014). Accessed: 2025-06-17, p. 2. URL: https://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment.

[14] *containerd Documentation*. https://containerd.io/docs/. Accessed: 2025-06-17.

[15] Sam Newman. *Building Microservices*. O’Reilly Media, 2015.

[16] Ian Fette and Alexey Melnikov. *The WebSocket Protocol*. RFC 6455, IETF. https://tools.ietf.org/html/rfc6455. 2011.

[17] SQLite Consortium. *SQLite Documentation*. https://www.sqlite.org/docs.html. Accessed: 2025-06-11.

[18] Cloud Native Computing Foundation. *CNCF Cloud Native Storage Whitepaper*. https://github.com/cncf/tag-storage/blob/main/whitepapers/cloud-native-storage.md. Accessed: 2025-06-11.

[19] *MinIO Documentation*. https://min.io/docs. Accessed: 2025-06-11.

[20] Andrew Pavlo et al. "A Comparison of Approaches to Large-Scale Data Analysis". In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (2009), pp. 165–178. DOI: 10.1145/1559845.1559865.

[21] Mark Raasveldt and Hannes Mühleisen. "DuckDB: an embeddable analytical database". In: *Proceedings of the 2020 ACM SIGMOD* (2020).

[22] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. 4th. Pearson, 2015.

[23] D. Richard Kuhn David Ferraiolo and Ramaswamy Chandramouli. *Role-Based Access Control*. Artech House, 2003.

[24] *Apache Airflow*. https://airflow.apache.org/. Accessed: 2025-06-11.

[25] *Apache Spark*. https://spark.apache.org/. Accessed: 2025-06-11.

[26] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2004, pp. 137–150. URL: https://research.google/pubs/pub62/.

[27] Amazon Web Services. *AWS Lake Formation*. https://aws.amazon.com/lake-formation/. Accessed: 2025-06-11. 2025.

[28] Michael Armbrust et al. "Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics". In: *CIDR* (2021).

[29] *Google BigLake Documentation*. https://cloud.google.com/biglake. Accessed: 2025-06-11.

[30] Peter Mell and Timothy Grance. *The NIST Definition of Cloud Computing*. Tech. rep. Special Publication 800-145. Defines PaaS, IaaS, SaaS models. Accessed: 2025-06-11. National Institute of Standards and Technology, 2011. URL: https://doi.org/10.6028/NIST.SP.800-145.

[31] Heroku Inc. *Heroku - Cloud Application Platform*. https://www.heroku.com. Accessed: 2025-06-11. 2025.

[32] Google Cloud. *Google App Engine Documentation*. https://cloud.google.com/appengine. Accessed: 2025-06-11. 2025.

[33] *Red Hat OpenShift*. https://www.openshift.com/. Accessed: 2025-06-11.

[34] *JupyterHub*. https://jupyter.org/hub. Accessed: 2025-06-11.

[35] *Keycloak Documentation*. https://www.keycloak.org/. Accessed: 2025-06-11.

[36] Auth0 Inc. *Auth0 Identity Platform*. https://auth0.com/. Accessed: 2025-06-11. 2025.

[37] Google Firebase Team. *Firebase Authentication*. https://firebase.google.com/products/auth. Accessed: 2025-06-11. 2025.

[38] OpenID Foundation. *OpenID Connect Core 1.0*. https://openid.net/specs/openid-connect-core-1_0.html. Accessed: 2025-06-11. 2014.

[39] Ravi S. Sandhu et al. "Role-Based Access Control Models". In: *IEEE Computer* 29.2 (1996), pp. 38–47. DOI: 10.1109/2.485845.

[40] FFmpeg Developers. *FFmpeg*. https://ffmpeg.org/. Version used: 6.x. Accessed: 2025-06-11. 2024.

[41] Aikaterini Tsimpirdoni. "Remote Execution of an FPGA-based Cellular Automata Accelerator on the Amazon F1 Cloud". Unpublished, available in print at the university library. MSc Thesis. School of Electrical and Computer Engineering, Technical University of Crete, 2024.

[43] Michael B. Jones. *RFC 7519: JSON Web Token (JWT)*. https://datatracker.ietf.org/doc/html/rfc7519. Accessed: 2025-06-11. 2015.

[44] Roy T. Fielding. "Architectural Styles and the Design of Network-based Software Architectures". PhD thesis. University of California, Irvine, 2000. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[45] Michael B. Jones. *JSON Web Algorithms (JWA)*. https://datatracker.ietf.org/doc/html/rfc7518. RFC 7518. Accessed: 2025-06-11. 2015.

[46] D. Eastlake. *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*. https://datatracker.ietf.org/doc/html/rfc6234. RFC 6234. Accessed: 2025-06-11. 2011.

[47] *Bcrypt Password Hashing*. https://en.wikipedia.org/wiki/Bcrypt. Accessed: 2025-06-11.

[48] Carson Gross. *HTMX: High power tools for HTML*. https://htmx.org. Accessed: 2025-06-11. 2020.

[49] Marijn Haverbeke and contributors. *CodeMirror*. https://codemirror.net/. Accessed: 2025-06-17. 2024.

[50] Inc. Fonticons. *Font Awesome Free*. https://fontawesome.com/. Accessed: 2025-06-17. 2024.

[51] *golangci-lint: Fast Go linters runner*. https://github.com/golangci/golangci-lint. Accessed: 2025-06-17.

[52] *Swagger: API Documentation Tools*. https://swagger.io/. Accessed: 2025-06-17.

[53] Chai2010. *golds: A Go documentation server*. https://github.com/go101/golds. Accessed: 2025-06-17.

[54] Cosmtrek. *air: Live reload for Go apps*. https://github.com/cosmtrek/air. Accessed: 2025-06-17.

External Links

[3] Google Inc. *Kubernetes: Production-Grade Container Orchestration*. https://kubernetes.io. Originally developed at Google. Accessed: 2025-06-11. 2014.

[8] Cloud Native Computing Foundation. *Kubernetes Project*. https://www.cncf.io/projects/kubernetes/. Accessed: 2025-06-11. 2024.

[42] Kyriakos Chalvatzis. *Minioth*. 2025. URL: https://github.com/kyri56xcaesar/kuspace/tree/on-k8s/pkg/minioth.