Distributed Audio Processing Pipeline with Observability

Final Report for the Distributed Systems Course AY 2025/2026

Matteo Cardellini
matteo.cardellini@studio.unibo.it

May 11, 2026

This work presents a distributed, event-driven pipeline for asynchronous audio processing, designed as a practical case study for distributed systems principles and operational observability.

Users submit audio files through a FastAPI service. The API stores binary objects in MinIO, persists job metadata in PostgreSQL, and publishes processing requests to RabbitMQ. A decoupled worker consumes queued jobs, extracts core audio metadata (duration, sample rate, channels), and writes results back to the database. The same API exposes upload, job tracking, and result retrieval endpoints, enabling a complete end-to-end workflow.

The architecture separates ingestion and processing through asynchronous messaging, so API and worker instances can scale independently. Reliability is modeled through durable queue-based communication and explicit job-state transitions (PENDING, PROCESSING, DONE, FAILED), while observability is supported via Prometheus-compatible metrics for throughput, latency, queue backlog, and worker activity.

The stack is containerized with Docker and structured for Kubernetes-based deployment and scaling experiments. The project shows how message-driven coordination, persistent state management, and monitoring can be combined into a reproducible distributed audio-processing system.

Disclaimer (if needed)

During the preparation of this work, the author used GitHub Copilot for language refinement and drafting support. All technical choices, validation activities, and final content were reviewed and approved by the author, who takes full responsibility for the report.

1 Concept

Type of Product

The developed product is a **distributed, service-oriented application** for asynchronous audio processing. From an architectural perspective, it is composed of **three cooperating software components**:

- a **web API** (FastAPI) that exposes upload, job-status, and result-retrieval endpoints;
- a **background worker service** that consumes queued jobs and executes audio processing tasks;
- a **lightweight browser-based frontend** used as an interactive client for manual usage and demo scenarios.

Therefore, the project is primarily a distributed web-service system, complemented by a simple web application for user interaction, rather than a standalone library or CLI-only tool.

Use Case Description

The system supports **asynchronous audio-processing requests** submitted by users through an HTTP API. In a typical scenario, a user uploads an audio file, receives a **job identifier**, and later queries the job state and extracted results. This interaction model avoids long synchronous waits and is suitable for workloads where processing time is variable.

Users access the platform remotely through a web browser or an API client, typically from personal computers and, when needed, from mobile devices with HTTP-capable tools. Interaction frequency is naturally bursty: short upload sessions can generate multiple jobs in sequence, followed by periodic status checks until processing completes.

The platform stores both **binary** and **structured** data. Audio files are persisted in object storage (MinIO), while job metadata and processing outcomes are persisted in PostgreSQL. Stored information includes job identifiers, file references, processing states, timestamps, and extracted audio metadata such as duration, sample rate, and number of channels.

At the architectural level, the system involves distinct roles: client users submitting requests, the API service coordinating ingestion and orchestration, and worker services executing background processing. This role separation is a core part of the distributed use case and enables clearer responsibility boundaries across components.

Why Distribution Is Needed

Distribution is a **structural requirement** of this project, not only an implementation preference. Audio processing tasks can take variable and non-trivial execution time,

so coupling upload handling and processing in a single synchronous component would reduce responsiveness and limit throughput. By separating ingestion (API) from execution (workers) through a message broker, the system can accept requests quickly and process them asynchronously.

This design enables **independent horizontal scaling** of components: API instances can scale to absorb request bursts, while worker replicas can scale according to queue backlog and processing demand. It also promotes resource sharing, since multiple workers consume from the same queue and operate on a common persistent state stored in PostgreSQL and MinIO.

Distribution also improves **fault isolation** and operational resilience. If a worker instance fails during processing, new requests can still be accepted by the API, and queued jobs remain available for other workers. Combined with explicit job-state transitions and persistent storage, this model provides a robust foundation for reliability, observability, and incremental system evolution.

2 Requirements Elicitation and Analysis

This section defines **what** the system must provide from a functional and quality perspective. Requirements are identified with unique IDs and paired with explicit acceptance criteria used later in validation.

Glossary

- **Job**: one asynchronous processing request created after an audio upload.
- **Job state**: lifecycle value associated with a job (PENDING, PROCESSING, DONE, FAILED).
- **Queue backlog**: number of jobs waiting to be processed by workers.
- **Processing result**: extracted audio metadata returned by the worker and persisted by the platform.

Functional Requirements

- **FR1: Audio upload**. The system shall allow a user to submit an audio file and create a processing job. **Acceptance criterion:** after a valid upload request, the API returns success with a new job identifier and an initial job state.
- **FR2: Asynchronous execution**. The system shall execute audio processing asynchronously with respect to request submission. **Acceptance criterion:** upload requests complete without waiting for final processing, while the corresponding job continues in background.

- **FR3: Job status retrieval**. The system shall expose an endpoint to retrieve the current state of a job by identifier. **Acceptance criterion:** querying an existing job returns one valid state among PENDING, PROCESSING, DONE, FAILED.
- **FR4: Result retrieval**. The system shall expose an endpoint to retrieve processing results when available. **Acceptance criterion:** for jobs in DONE state, the API returns persisted extracted metadata; for unfinished jobs, it returns a consistent status response.
- **FR5: Persistent storage**. The system shall persist uploaded audio objects and job metadata. **Acceptance criterion:** after restart of application containers, previously created jobs and stored files remain available.
- **FR6: Failure visibility**. The system shall make processing failures observable through job state and error information. **Acceptance criterion:** if processing fails, the job is marked FAILED and failure details are retrievable through API/database inspection.

Non-Functional Requirements

- **NFR1: Scalability**. The system should support horizontal scaling of API and worker components. **Acceptance criterion:** increasing worker replicas increases processing throughput under comparable load.
- **NFR2: Responsiveness**. Upload operations should remain fast even when processing is slow. **Acceptance criterion:** upload endpoint remains responsive because processing is decoupled and queued.
- **NFR3: Reliability**. Jobs should not be lost during normal component restarts or transient worker issues. **Acceptance criterion:** queued or persisted jobs are recoverable and continue through lifecycle after restart.
- **NFR4: Observability**. The platform should expose measurable operational signals. **Acceptance criterion:** Prometheus-compatible metrics for requests, processing activity, and failures are exposed by services.
- **NFR5: Maintainability**. The architecture should keep service responsibilities clearly separated. **Acceptance criterion:** API and worker can be developed, deployed, and scaled independently without changing external contract.

Implementation Constraints

- **IR1: Service implementation language**. Core services shall be implemented in Python. **Acceptance criterion:** API and worker codebases are Python projects with reproducible dependency management.

- **IR2: Messaging middleware**. Asynchronous coordination shall use RabbitMQ. **Acceptance criterion:** job dispatch and consumption happen through RabbitMQ queues/exchanges.
- **IR3: Data technologies**. Job metadata shall be persisted in PostgreSQL and audio objects in MinIO. **Acceptance criterion:** no alternative primary store is required for the main pipeline flow.
- **IR4: Containerized deployment**. Services shall be runnable via container orchestration for reproducibility. **Acceptance criterion:** the full stack can be started with Docker Compose and deployed to Kubernetes manifests/charts provided in the repository.

2.1 Relevant Distributed System Features

For this project, distributed-system features have different priorities.

- **Scalability (high relevance).** The pipeline must absorb burst uploads and variable processing demand. Independent scaling of API and workers is therefore essential.
- **Fault tolerance and dependability (high relevance).** Worker or network issues must not cause immediate job loss. Durable messaging, persistent state, and explicit job transitions are central to reliability.
- **Resource sharing (high relevance).** Multiple workers share queue workload and common storage backends. Coordination around shared queue and shared state is intrinsic to the architecture.
- **Performance and concurrency (high relevance).** Many jobs may execute concurrently, and throughput under load is a key quality objective.
- **Evolvability and maintainability (high relevance).** Service separation enables incremental changes (API, worker, observability) with limited cross-impact.
- **Security and trust (medium relevance).** The system includes authentication and controlled API access, but it is not designed for highly sensitive regulated workloads.
- **Transparency (medium relevance).** Users are abstracted from internal distribution details, but operational distribution remains explicit for developers and operators.
- **Openness and interoperability (medium relevance).** The system relies on widely adopted protocols and components (HTTP, AMQP, SQL), which simplifies integration and portability.

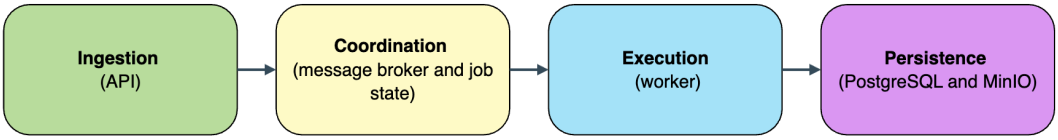

Figure 1: Four-block pipeline: ingestion, coordination, execution, and persistence.

- **Economy and costs (medium relevance).** Containerization and local orchestration reduce setup and experimentation costs while preserving realistic deployment patterns.

3 Design

This section describes how the system design addresses the requirements in section 2, with a focus on scalability, fault isolation, resource sharing, and observability. The architecture is message-driven: ingestion, background processing, and persistence are separated into dedicated components so each part can evolve and scale independently. This separation keeps API requests responsive under load, while worker capacity can increase as queue backlog grows. Components coordinate asynchronously through explicit job lifecycle states persisted in shared storage.

The next subsections present the design in order: architectural style, infrastructure, modelling, interaction, behaviour, data consistency, fault tolerance, availability, and security.

3.1 Architecture

The system follows an **event-driven, service-oriented architecture**. It combines two interaction styles:

- **synchronous request-response** for user-facing API operations (upload, status, result retrieval);
- **asynchronous message-driven processing** for background job execution through RabbitMQ.

This architectural style was chosen to satisfy the most important requirements. First, it keeps the API responsive because request handling is separated from long-running processing tasks. Second, it enables independent scaling of API and worker components, which directly supports throughput growth under variable load. Third, it improves fault isolation: temporary worker issues do not block request intake, and queued jobs remain processable by other workers.

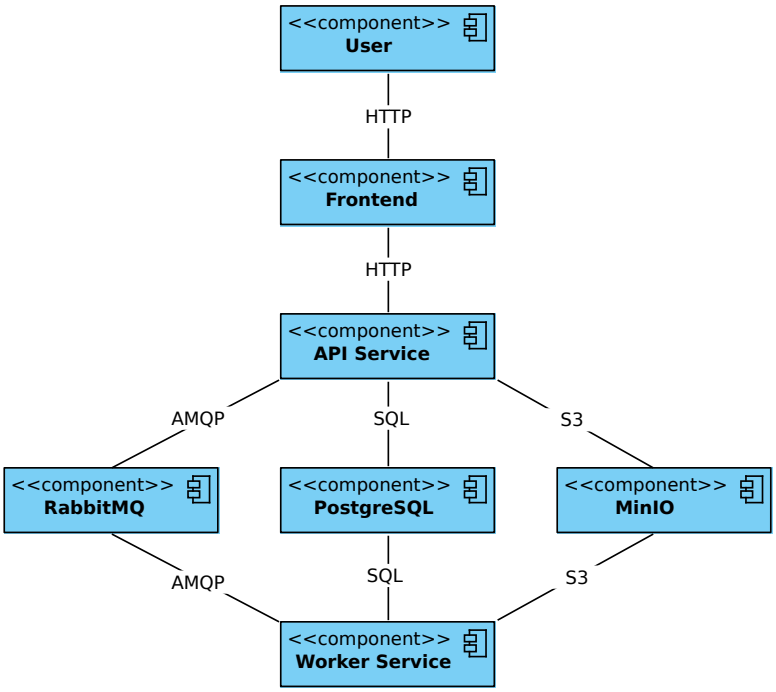

Figure 2: Runtime components and their communication protocols.

In practical terms, the architecture can be seen as a small pipeline with clear stages: **ingestion** (API), **coordination** (message broker and job state), **execution** (worker), and **persistence** (PostgreSQL and MinIO). This separation also improves maintainability, since each stage can evolve with limited impact on the others. A visual summary of this **four-block pipeline** is shown in fig. 1.

3.2 Infrastructure

The infrastructure is built around six runtime components. Figure 2 provides a simplified overview of their roles and primary communication protocols (HTTP, AMQP, SQL, S3).

- **Frontend client**: a lightweight web interface for manual interaction;
- **API service**: handles HTTP requests, upload orchestration, and status/result queries;
- **Worker service**: consumes queued jobs and executes audio processing;
- **RabbitMQ**: message broker used for asynchronous coordination between API and workers;
- **PostgreSQL**: persistent relational store for job metadata and processing outcomes;

- **MinIO**: object storage for uploaded audio files.

Kubernetes is the default deployment model: each component runs as one or more pods, while Services provide stable endpoints for internal and external traffic [1]. This setup enables horizontal scaling of stateless parts, especially workers, without changing application logic. Service discovery is DNS-based through Kubernetes Service names, which removes hard-coded addresses and improves portability. Traffic distribution is handled by platform primitives: API HTTP requests are balanced at Service level, and processing load is distributed as workers compete for RabbitMQ messages. An external load balancer remains optional, depending on the target cluster environment.

3.3 Modelling

Figure 3 captures the core domain model of the asynchronous pipeline. The primary **domain entities** are `User`, `AudioJob`, and `ProcessedAudio`, with `AudioFeatures` as a value object associated with processing results. In this model, each uploaded file creates exactly one `AudioJob`, identified by a unique ID, owned by a user, and tracked through the lifecycle states `PENDING`, `PROCESSING`, `DONE`, and `FAILED`. When processing completes successfully, `ProcessedAudio` stores extracted metadata such as duration, sample rate, and channel count.

Entity-to-infrastructure mapping follows component responsibilities. Identity and job metadata are persisted in PostgreSQL, binary audio content is persisted in MinIO, and execution coordination is delegated to RabbitMQ. At runtime, API-side services create jobs, validate access, and expose query operations, while worker-side services consume queued jobs, execute processing logic, and apply lifecycle transitions to persistent state.

The **domain events** considered by the model are: *job created*, *job queued*, *processing started*, *processing completed*, and *processing failed*. These are reflected in three **message categories** exchanged among components: *commands* (e.g., enqueue processing), *events* (state transitions), and *queries* (status and result retrieval).

The **overall system state** includes user identity and role data, job ownership and lifecycle metadata, object-storage references, extracted audio features, timestamps, and error diagnostics. This state is intentionally externalized in durable stores so that execution remains recoverable after restarts and observable across API and worker components.

3.4 Interaction

Component interaction is structured around four recurrent runtime flows that cover the complete user-facing lifecycle: authentication, upload and job creation, background processing, and status/result retrieval. Within these flows, synchronous HTTP is used for client-facing request-response operations, whereas asynchronous AMQP messaging decouples ingestion from execution between API and workers [2]. This hybrid interaction model provides immediate feedback to clients while allowing processing to complete in the background.

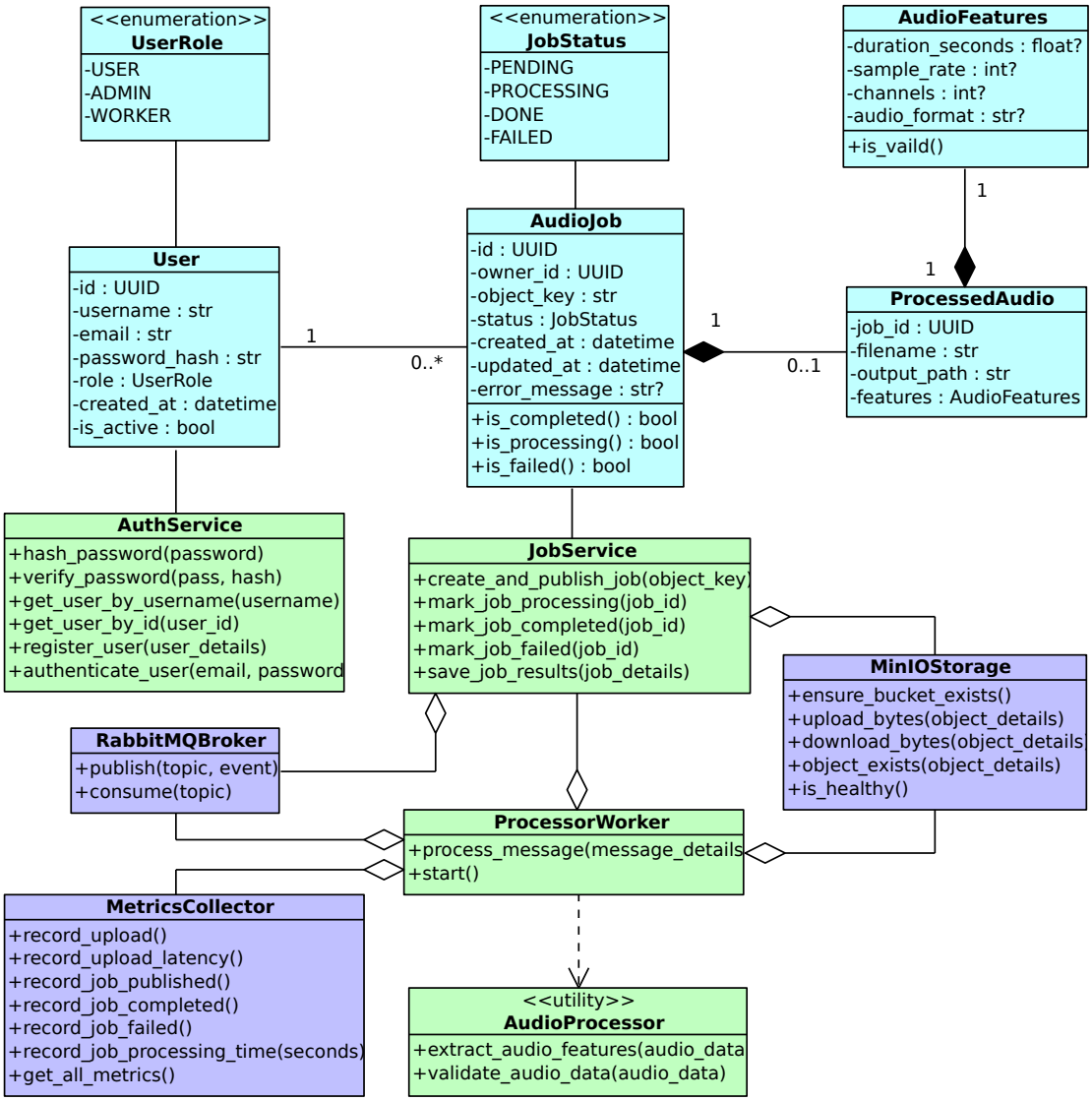

Figure 3: Class diagram showing the main domain entities, their relationships, and how they map to infrastructural components.

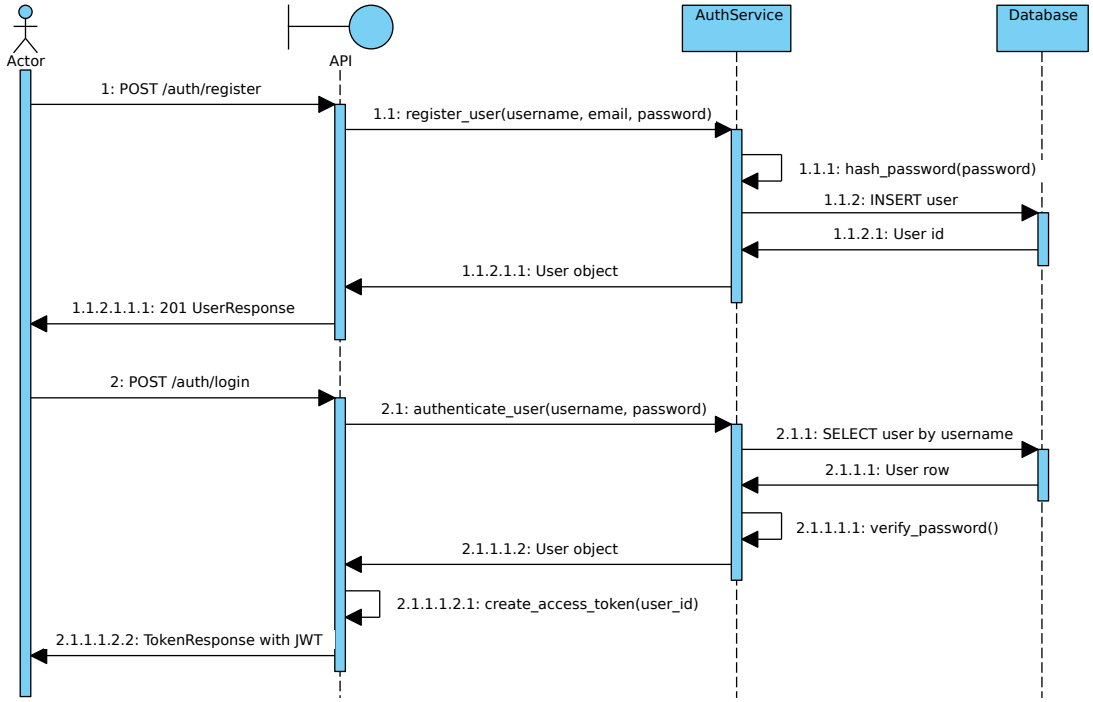

Figure 4: Sequence for user registration and login, leading to JWT issuance and authenticated API usage.

Flow 1: Authentication (register and login)

As shown in fig. 4, a new user first registers through the API. The service validates the payload, persists the user in PostgreSQL, and returns a successful registration response. For subsequent access, the login request triggers credential verification followed by JWT issuance. The client then includes the token in the `Authorization` header when invoking protected endpoints.

Flow 2: Upload and job creation

In fig. 5, an authenticated client uploads an audio file to the API. The API stores the binary object in MinIO, creates the corresponding `AudioJob` in PostgreSQL with initial state `PENDING`, and publishes a processing command to RabbitMQ. The client receives the job identifier immediately; therefore, request latency is independent of processing time. This is the primary ingestion path and the main integration point between synchronous and asynchronous communication.

Flow 3: Worker processing

Figure 6 details the execution path after a message becomes available in RabbitMQ. The worker consumes the job message, updates the job state to `PROCESSING`, fetches the

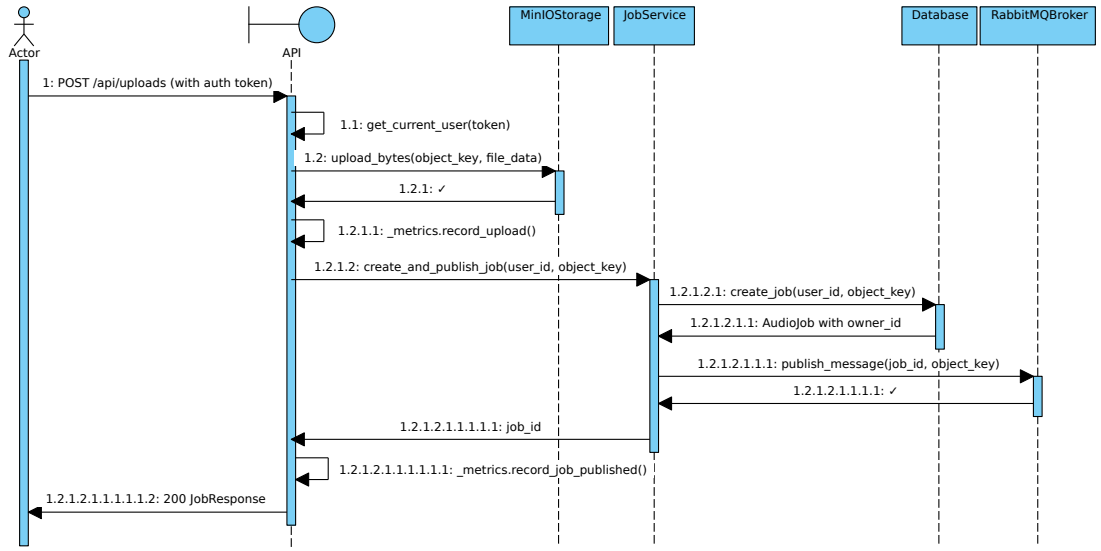

Figure 5: Sequence for file upload, persistent job creation, and asynchronous dispatch to the processing queue.

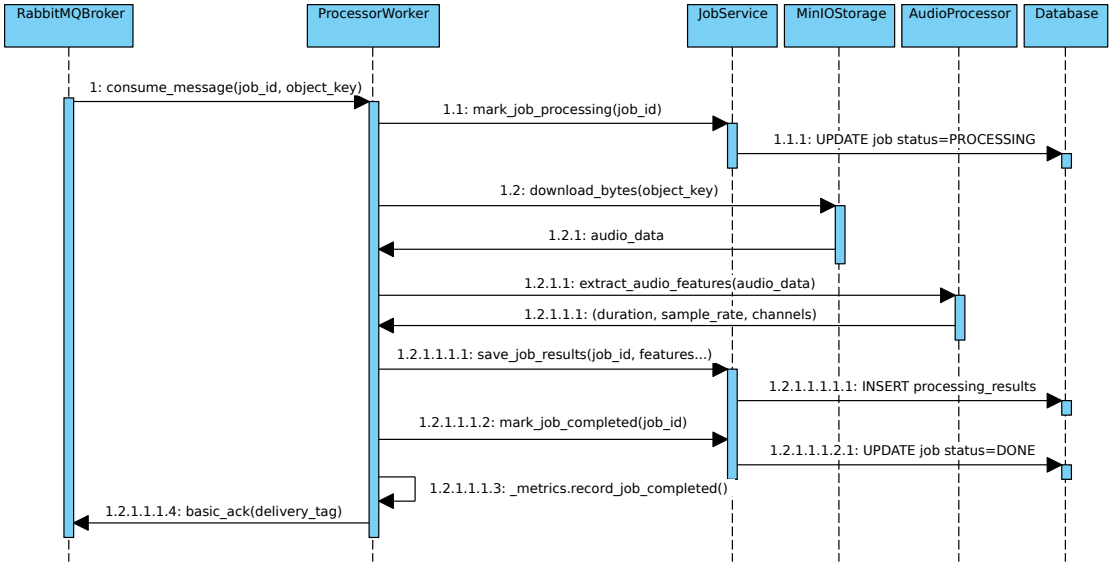

Figure 6: Sequence for queue consumption, processing execution, and final state transition update.

source object from MinIO, extracts audio metadata, and persists outputs in PostgreSQL. On success, the state transitions to `DONE`; on failure, it transitions to `FAILED` with diagnostic details. Broker acknowledgement is sent only after the processing attempt is finalized, supporting robust at-least-once consumption semantics.

Flow 4: Job status and result retrieval

Finally, fig. 7 describes the read path used after submission. The client periodically queries job status through the API, which retrieves the current lifecycle value from PostgreSQL. While processing is ongoing, the API returns consistent intermediate responses; once the state is `DONE`, the result endpoint returns persisted metadata (duration, sample rate, channels).

Taken together, these four flows implement a coherent interaction pattern: state-changing ingestion operations remain short and synchronous, compute-intensive work is decoupled and asynchronous, and completion is exposed through explicit state queries.

3.5 Behaviour

Behaviour in this system is centered on two complementary state machines. The first one describes the **business lifecycle** of an `AudioJob`, while the second one describes the **runtime behavior** of the worker while it consumes and processes queue messages. Together, they clarify which component updates state, which state is persistent, and which state is only internal to execution.

Figure 8 shows the persistent job lifecycle stored in PostgreSQL. When the API receives a valid upload, it creates the job in `PENDING` state. The worker then advances the job to `PROCESSING` when it starts consuming the corresponding message. If the processing succeeds, the job becomes `DONE` and the extracted metadata are persisted. If the processing fails, the job is either retried or, after the retry budget is exhausted, marked `FAILED`. These terminal states are terminal only for the job record; they do not stop the service itself.

Figure 9 captures the internal control flow of the worker service. The worker is intentionally long-running and therefore does not have a final lifecycle state in the usual sense: after handling one message, it returns to its idle listening state and waits for the next one. This diagram models a single message-processing cycle, starting from message reception, moving through parsing and processing, and then branching either to successful completion or to retry scheduling. On transient failures, the worker republishes the message with an incremented attempt counter and resets the job to `PENDING`; once the retry limit is reached, the worker records the final failure and stops retrying that job.

3.6 Data and Consistency Issues

The system stores both **persistent business data** and **ephemeral coordination data**. Persistent business data includes user accounts, authentication-related information, job metadata, job ownership, job state transitions, error messages, and extracted audio results. Binary audio files are not stored in the relational database: they are

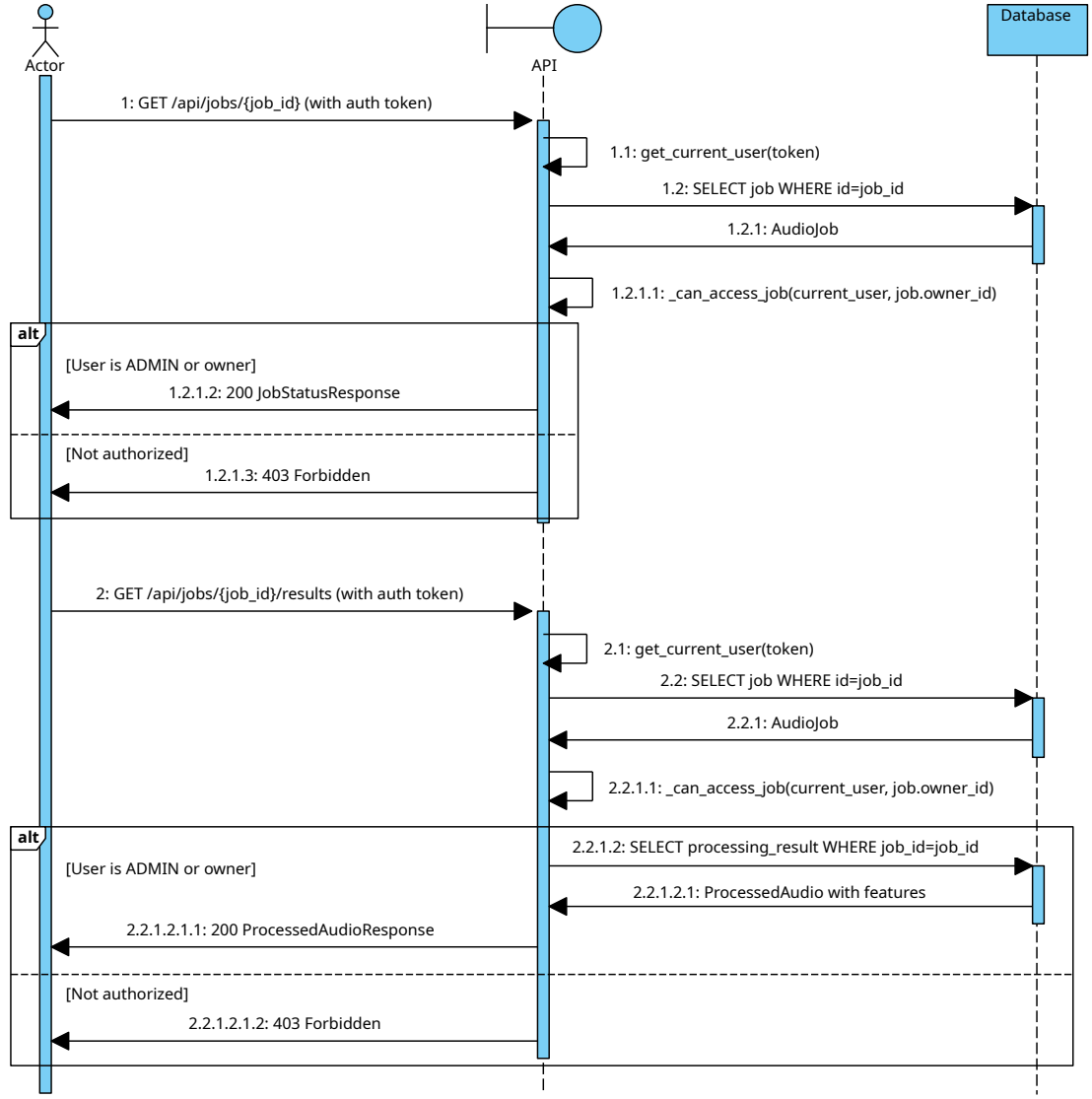

Figure 7: Sequence for polling job status and retrieving processing results once completion is reached.

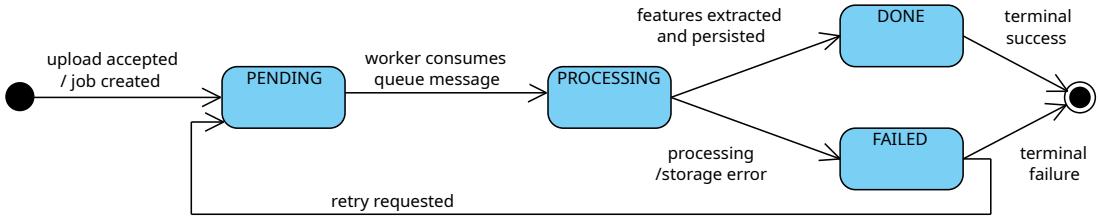

Figure 8: State diagram for the `AudioJob` lifecycle, from creation to completion or failure.

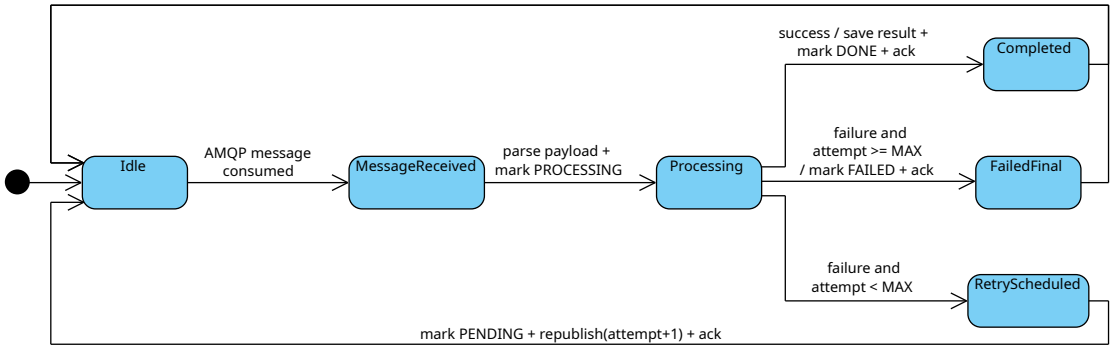

Figure 9: State diagram for the worker execution loop, including bounded retry handling.

persisted separately in MinIO as objects, while PostgreSQL keeps only the references needed to reconstruct the processing flow. This separation keeps the relational model compact and makes large file handling independent from metadata queries.

Persistent data is modeled as a relational schema because the main entities have clear identifiers, strong relationships, and structured lifecycle transitions. PostgreSQL is therefore the source of truth for users, jobs, and processing outcomes (as illustrated in Figure 10), while MinIO is the source of truth for uploaded audio content. The message broker is intentionally not used as the system of record: RabbitMQ carries work commands, but the durable state is always reconstructed from the database and object storage.

Several components perform queries on the database, but they do so with different responsibilities. The API service writes user and job data during registration, authentication, and upload handling, then reads jobs and results when serving status or result requests. The worker service updates job state during execution and writes processing results after a successful extraction attempt. Because the worker and API may access the same records concurrently, the design favors small, explicit updates rather than long transactions spanning multiple components.

Concurrent access is mostly read-heavy, with writes concentrated on job creation, status transitions, and result persistence. The main contention point is the `jobs` table, since the API initializes jobs and the worker advances their state. This is acceptable because each job is updated by a small number of transitions and the transitions are monotonic in normal execution. The `processing_results` table is updated with an upsert pattern, which keeps retries idempotent and avoids duplicate result rows when a message is reprocessed.

Some data must be shared between components, but only through durable storage and message payloads. The shared identifiers are the job ID, owner ID, storage object key, current state, timestamps, and extracted features. AMQP messages carry the job ID, object key, and retry attempt counter so the worker can continue execution without embedding business state in the queue itself.

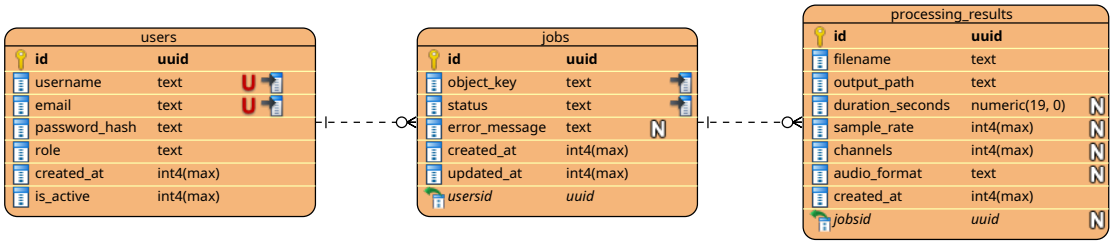

Figure 10: PostgreSQL schema entities and relationships.

3.7 Fault-Tolerance

Fault tolerance in this system is based on **durable state**, **asynchronous retries**, and **component isolation**. There is no separate replication layer for the application logic itself; instead, resilience comes from storing business data in **durable backends** and keeping the message broker as a **coordination layer** rather than as the source of truth. PostgreSQL stores job metadata, state transitions, and processing outcomes, while MinIO stores the uploaded audio objects. Because the persistent state is externalized, the API and worker services can restart independently without losing the information needed to resume processing.

Retry handling is applied to **transient failures** that can occur during processing or storage access. If a worker cannot complete a job because of a temporary issue, the message can be retried and the job remains recoverable until the retry budget is exhausted. This keeps the failure policy explicit and prevents the queue from silently discarding unfinished work.

Error handling is split across the components that can detect failures earliest. The API rejects invalid uploads and authentication failures immediately, so only valid jobs enter the system. The worker records processing failures in the job state, making them visible through the status and result endpoints and allowing operators to inspect persistent diagnostics. In this way, faults are converted into **observable job states** rather than hidden internal errors.

3.8 Availability

The system does not rely on a dedicated **caching** layer. This choice is deliberate: the most frequently accessed data are job metadata, processing state, and final results, all of which are already stored in durable backends and are small enough to retrieve directly. Avoiding cache invalidation logic keeps the system simpler and reduces the risk of serving stale job information.

Availability is instead improved through **load balancing** and **horizontal replication of stateless services**. In Kubernetes, the API can run with multiple replicas behind a Service, so incoming HTTP traffic is distributed across available instances. Worker replicas also scale horizontally and consume from the same RabbitMQ queue, which makes processing capacity more resilient to spikes in demand or temporary pod

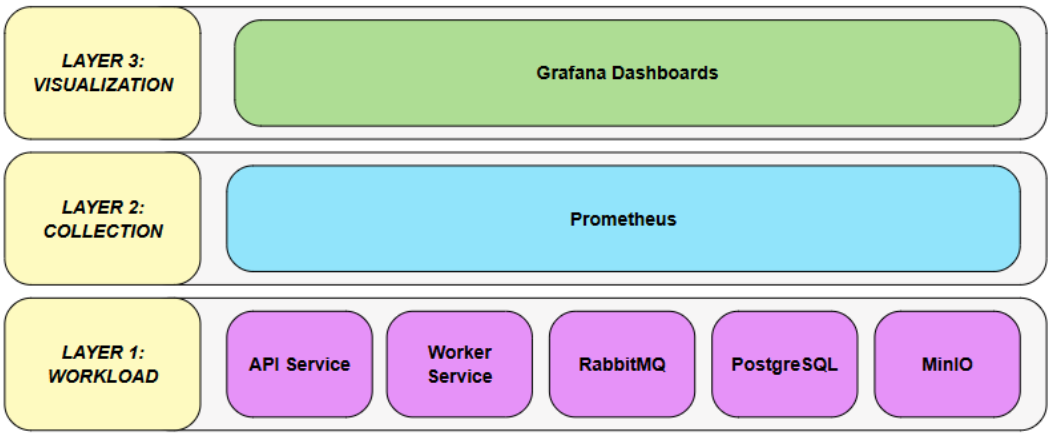

Figure 11: Simplified three-layer observability view: workload components, Prometheus collection, and Grafana visualization.

restarts.

In case of **network partitioning**, the system degrades in a controlled way rather than hiding partial failures. If the API cannot reach the broker or the storage backends, uploads fail fast and return an error instead of accepting jobs that cannot be completed. If workers lose connectivity while processing, jobs remain persisted and can be retried once connectivity is restored. This behavior favors consistency and recoverability over masking broken communication paths, which is appropriate for an asynchronous processing pipeline.

3.9 Observability

Observability is treated as a first-class design concern because the pipeline is asynchronous and spans multiple loosely coupled components. The goal is not only to detect failures, but also to understand how requests move through the system, where work accumulates, and which component becomes a bottleneck under load. For this reason, the observability solution is built around **metric exposure**, **centralized scraping**, and **dashboard-based visualization**.

A simplified view of the monitoring architecture is shown in fig. 11. The first layer contains the main workload components of the pipeline (API, RabbitMQ, worker, PostgreSQL, and MinIO). The second layer contains Prometheus, which collects metrics from those components. The third layer contains Grafana, which visualizes the collected metrics through dashboards.

Each runtime component exposes Prometheus-compatible metrics that describe its own behavior. The API reports request volume and latency, the worker reports processing activity and failure counts, and the queue layer exposes backlog-related signals that indicate whether processing capacity is keeping up with demand. These metrics are

aligned with the most relevant quality concerns of the project: responsiveness, throughput, reliability, and scaling behavior.

Prometheus acts as the collection layer by scraping the exposed metrics at regular intervals and storing them in a time-series format. Grafana then consumes those metrics to provide a consolidated view of the system, making it easier to inspect trends over time and correlate events such as upload bursts, queue growth, and worker slowdowns. This separation between metric production and visualization keeps the application code lightweight while still giving a clear operational picture.

From a design perspective, observability also supports validation and debugging. When a job remains pending for too long or a worker begins failing repeatedly, the metrics make the problem visible before it becomes a silent functional error. In that sense, the monitoring stack complements fault tolerance: fault tolerance preserves correctness under failure, while observability makes the failure itself measurable and actionable.

3.10 Security

The system implements **stateless, token-based authentication** and **role-based access control** to protect API endpoints and ensure that users can only access their own resources.

Authentication

Authentication is realized through **JWT (JSON Web Tokens)** issued on successful login. When a user submits valid credentials (username and password), the API verifies them against the stored password hash and issues a JWT token. The token is signed using the HS256 algorithm and contains the user ID as the subject and an expiration time. The client includes the token in the `Authorization` header as a bearer token for all subsequent requests. Each protected endpoint applies a dependency that extracts and validates the token, extracting the user ID and loading the corresponding user record from the database. If the token is missing, malformed, or expired, the request is rejected with a 401 Unauthorized response. This approach keeps authentication stateless and allows API instances to verify tokens without session storage or coordination.

Authorization

The system uses **role-based access control (RBAC)** with three defined roles: `USER`, `ADMIN`, and `WORKER`. Standard users are granted the `USER` role on registration and can submit audio files, retrieve their own job status, and view their processing results. Job ownership is enforced at the application layer: queries filter results by the authenticated user's ID, so users can only see and interact with jobs they created. Administrative operations (user management, system configuration) are restricted to instances with the `ADMIN` role. The `WORKER` role is reserved for internal service-to-service communication in scenarios requiring explicit worker identification; it is not assigned to regular users. Authorization checks are implemented as FastAPI dependency injection, which transparently enforces access rules at the route level.

Cryptographic Schemas

The system employs two complementary cryptographic mechanisms:

- **Password hashing**: User passwords are hashed using bcrypt before storage in PostgreSQL. This ensures that even if the database is compromised, passwords remain protected. Verification happens through bcrypt comparison during login, not by storing plaintext or reversible encryption.
- **Token signing**: JWT tokens are signed with a shared secret key using HS256 (HMAC SHA-256). This cryptographic signature allows the API to verify that a token was genuinely issued by the system and has not been tampered with. The secret key is stored as an environment variable and is not hard-coded.

4 Implementation

The implementation mirrors the architecture: a FastAPI-based API service, a separate background worker, a lightweight browser frontend, and containerized infrastructure for local and cluster-based deployment.

The API and frontend communicate over **HTTP**, while the worker is coordinated asynchronously through **AMQP** via RabbitMQ. In-transit data is represented mostly as **JSON** for API responses and queue messages, with `multipart/form-data` for uploads and form-encoded credentials for login. PostgreSQL is queried directly with **SQL** through `psycopg`, using a relational schema for users, jobs, and processing results.

Authentication uses **JWT** access tokens with **bcrypt** password hashing, and authorization relies on **RBAC** plus job ownership checks so users only access their own data. The implementation is built with **FastAPI**, **Uvicorn**, and **Poetry**; it also uses RabbitMQ, MinIO, Prometheus-compatible metrics, and Python's standard `wave` module to extract WAV audio features.

Deployment is container-based through Docker Compose for local runs and Kubernetes/Helm assets for cluster deployment. The repository includes separate Dockerfiles for the API and worker, plus a static frontend served as a lightweight web client. The API runs as a FastAPI application served by Uvicorn, the worker runs as a separate Python process sharing the same domain modules, and observability is exposed through prometheus-client.

The source repository for the project is available at `https://github.com/matteo1610/distributed-audio-pipeline`.

5 Validation

5.1 Automatic Testing

Automatic testing is currently based on **unit and component-level tests** for the two Python services. The API test suite uses `FastAPI.TestClient` together with mocks to cover health and metrics endpoints, registration and login, upload handling, job status

and result retrieval, and the main failure paths. The worker test suite uses mocked broker, storage, database, and metrics dependencies to verify message processing, retry handling, startup wiring, and database-readiness checks.

Concretely, the main automated test files are `srcs/app/tests/test_api.py`, `srcs/app/tests/test_auth.py`, and `srcs/worker/tests/test_worker.py`.

These tests are automated with `pytest` and focus on service logic in isolation, so they validate request handling, authentication flow, message consumption, and state transitions without relying on live infrastructure. The API tests can be run from `srcs/app` with `poetry run pytest`, and the worker tests can be run from `srcs/worker` with `poetry run pytest`.

In addition, the GitHub Actions workflow `.github/workflows/python-tests.yml` executes the test suites automatically on push and pull request events when files under `srcs/app/**`, `srcs/worker/**`, or the workflow file itself are modified. A full end-to-end suite is not automated yet; complete-stack verification is still done manually during deployment and smoke testing.

5.2 Acceptance test

Manual acceptance testing was performed on the full Docker Compose stack to validate the end-to-end user workflow in a production-like local environment. The main checks were: user registration and login, authenticated upload from the frontend/API client, job status polling through lifecycle states, result retrieval after worker completion, and basic failure visibility when invalid inputs or processing errors occur.

This part is still manual because it spans multiple containers and infrastructure dependencies (API, worker, RabbitMQ, PostgreSQL, MinIO, and frontend), and the project currently prioritizes unit/component automation for service logic.

6 Deployment

Deployment is provided in two reproducible modes: Docker Compose for local execution and kind+Helm for local Kubernetes execution. The Compose path is the default for fast setup, while the Kubernetes path is used for orchestration-oriented experiments.

Required software and versions

To run the project from scratch with containers, the machine needs:

- Docker Engine and Docker Compose plugin (for local multi-container deployment);

For the Kubernetes path, the machine also needs:

- kind;
- kubectl;
- Helm 3.

Main runtime versions are pinned in images and chart values:

- PostgreSQL `16.13-bookworm`;
- RabbitMQ `4.2.5-management`;
- MinIO `RELEASE.2025-09-07T16-13-09Z-cpuv1`;
- Prometheus `v2.55.1`;
- Grafana `11.2.0`;
- Python `3.12.13` for API and worker images.

From-scratch installation with Docker Compose

From repository root:

```
cd srcs
docker compose up --build
```

Expected outcomes:

- API reachable on `http://localhost:8000`;
- Frontend reachable on `http://localhost:5173`;
- RabbitMQ management UI on `http://localhost:15672`;
- MinIO console on `http://localhost:9001`.

To include observability in the same local deployment:

```
cd srcs
docker compose -f docker-compose.yaml \
  -f observability/docker-compose.observability.yaml up -d
```

Additional expected outcomes:

- Prometheus on `http://localhost:9090`;
- Grafana on `http://localhost:3000` (default `admin/admin`).

From-scratch installation with kind and Helm

From `srcs`, create a local cluster and deploy the application chart:

```
kind create cluster --name audio-pipeline \
  --config k8s/kind/kind-config.yaml
helm install dap k8s/distributed-audio-pipeline \
  -f k8s/values-kind.yaml
```

Then optionally deploy observability:

```
helm install dap-observability k8s/observability \
  -f k8s/observability/values-kind.yaml
```

Expected outcomes through kind host port mappings:

- API on `http://localhost:8000`;
- Frontend on `http://localhost:30517`;
- Worker metrics on `http://localhost:9100/metrics`;
- RabbitMQ management on `http://localhost:15672`;
- MinIO API/console on `http://localhost:9000` and `http://localhost:9001`;
- Prometheus and Grafana on `http://localhost:9090` and `http://localhost:3000`.

Environment variables and configuration files

The Compose deployment defines runtime variables directly in `srcs/docker-compose.yaml`. For API local development, a reference template is provided in `srcs/app/.env.example`, covering database, broker, object storage, and API host/port settings.

In Kubernetes, configuration is managed through Helm values files:

- `srcs/k8s/values-kind.yaml`;
- `srcs/k8s/distributed-audio-pipeline/values.yaml`;
- `srcs/k8s/observability/values-kind.yaml`.

These files define service types, NodePorts, image tags, credentials, and storage settings.

How deployment assets are engineered

Deployment assets follow a **layered design**. For Docker Compose, the base stack is defined in `srcs/docker-compose.yaml` (API, worker, PostgreSQL, RabbitMQ, MinIO, frontend), with health checks and service dependency conditions to enforce startup ordering. Observability is added as a **compose overlay** in `srcs/observability/docker-compose.observability.yaml`, so monitoring can be enabled without modifying the base file.

For Kubernetes, the project uses **two separate Helm charts**: one for the application stack (`srcs/k8s/distributed-audio-pipeline`) and one optional chart for observability (`srcs/k8s/observability`). This separation is intentional and fundamental: the **core system and observability infrastructure are decoupled**, allowing independent deployment, scaling, and lifecycle management. The application can run without

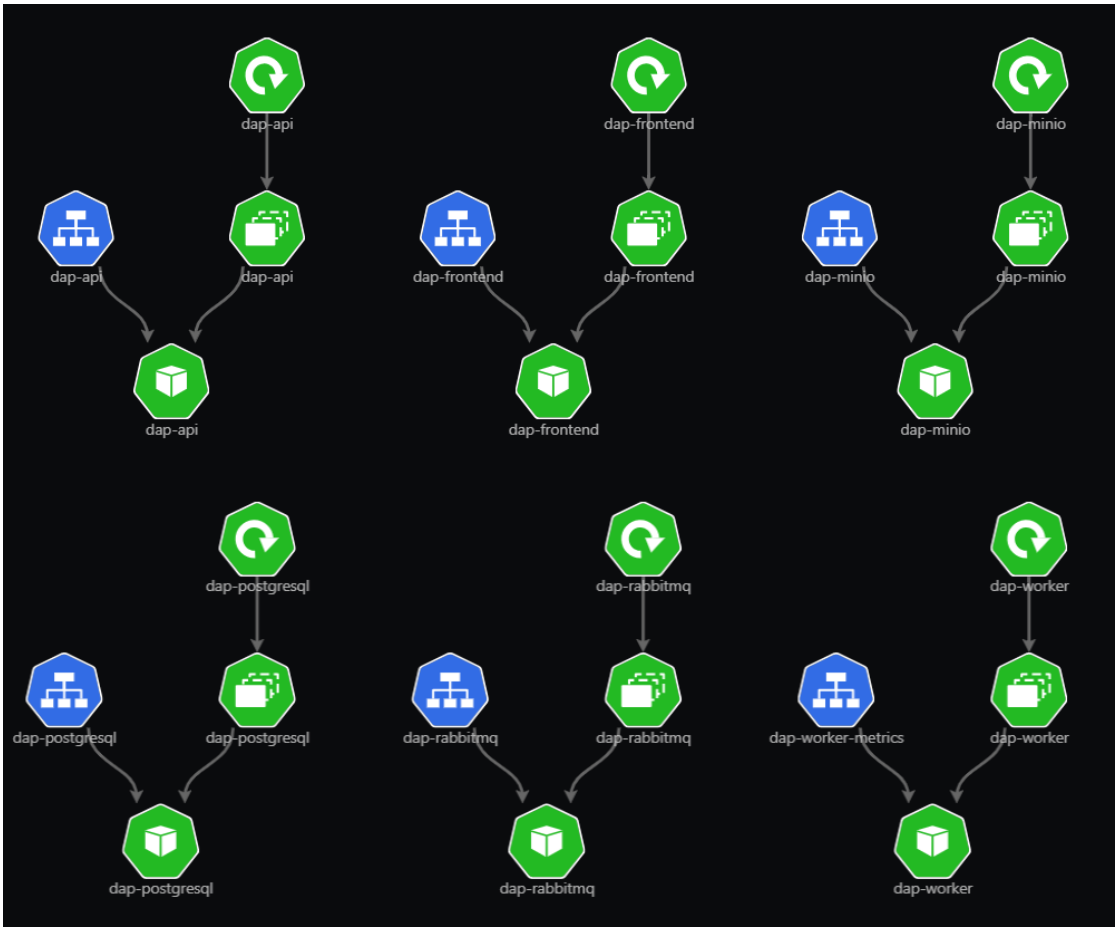

Figure 12: Application chart deployment in kind (release `dap`).

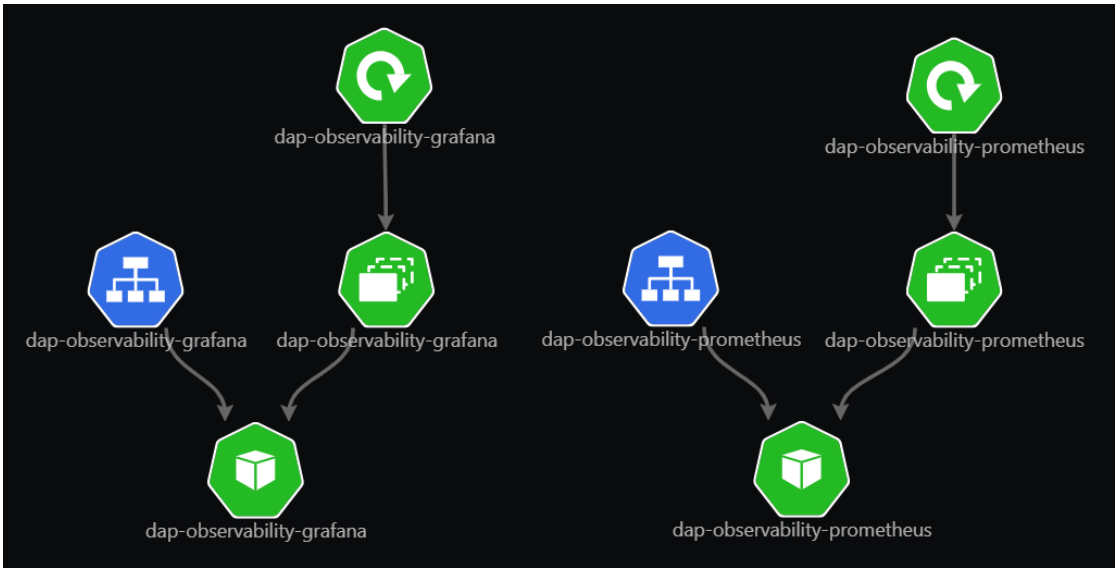

Figure 13: Observability chart deployment in kind (release `dap-observability`).

observability, and monitoring can be added or updated without touching the main workload. To keep chart-embedded assets synchronized, the script `srcs/scripts/sync_k8s_assets.sh` copies SQL and observability configuration files into chart `files/` directories before release updates.

Container image delivery is **automated** through CI: a workflow builds and pushes updated API and worker images to Docker Hub whenever significant project changes are merged, and deployment configurations then **pull tagged images from Docker Hub** (instead of rebuilding on-cluster), keeping rollouts reproducible across environments and simplifying update propagation.

The Helm-based packaging also provides a **standard and repeatable deployment method across Kubernetes environments**. While kind is used in this project mainly for local experimentation convenience, the same chart structure and values-driven configuration can be applied with minimal changes to any conformant Kubernetes cluster. The default deployment profile is **intentionally basic** (single-replica settings and local-oriented resource choices) to keep execution lightweight and fast on a developer machine. Even so, the current chart and service decomposition already support straightforward evolution toward multi-replica workloads, stronger distribution across nodes, and more production-oriented scaling policies.

7 User Guide

The user-facing workflow is intentionally simple: authenticate, upload an audio file, monitor the job, and retrieve the extracted metadata once processing completes. The workflow is exercised through the lightweight frontend (see Figure 14), the preferred

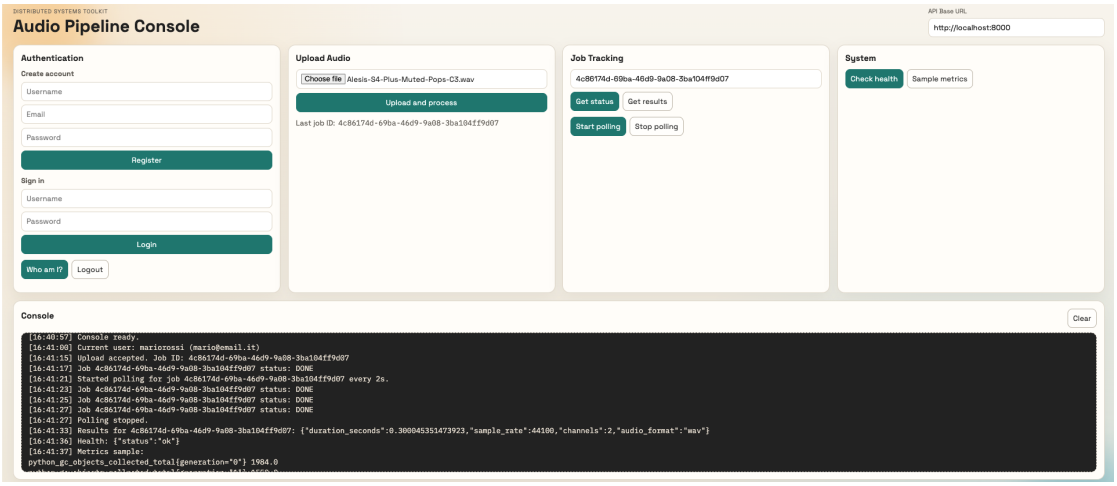

Figure 14: Lightweight frontend used to upload files and inspect job status.

interface for manual testing of backend functionality.

Main Workflows

1. Register and log in

Create an account or log in with existing credentials to obtain a JWT access token. This token is required for upload, status, and result requests.

2. Upload an audio file

Use the frontend file picker or the API interface to submit an audio file. The system stores the file in MinIO, creates a job record in PostgreSQL, and queues the processing request asynchronously.

3. Monitor job progress

After upload, the job moves through the usual lifecycle states: `PENDING`, `PROCESSING`, and `DONE` or `FAILED`. The frontend shows the current status and lets the user follow the processing progress in real time.

4. Retrieve the extracted metadata

When the job reaches `DONE`, the user can open the result view to see the extracted audio metadata, such as duration, sample rate, and number of channels.

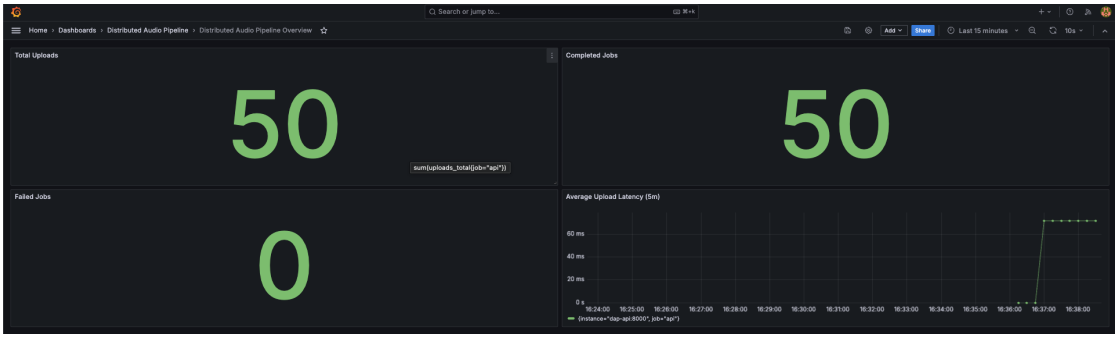

Figure 15: Example Grafana dashboard showing total uploads, completed jobs, failed jobs, and 5-minute average upload latency.

Observability

If observability is deployed, you can run the provided setup and then inspect the collected metrics in Grafana. Grafana is available at `http://localhost:3000` (default credentials: `admin/admin`).

The main dashboards summarize request rate, worker throughput, queue backlog, and the general health of the stack. An example Grafana dashboard is shown in Figure 15; it highlights total uploads, completed jobs, failed jobs, and the 5-minute average upload latency, providing a quick operational view of pipeline health. To populate these panels with sample data, the repository includes a simple stress script that uploads example files: `srcs/scripts/stress_uploads.sh`.

8 Self-evaluation

The distributed audio pipeline successfully demonstrates a complete, production-oriented architecture combining clean separation of concerns, asynchronous message-driven communication, and built-in observability.

This work spans full-stack implementation from domain modeling and FastAPI backend with JWT authentication and RBAC, through background workers, to containerized infrastructure supporting dual deployment strategies: Docker Compose for local development and Kubernetes/Helm for orchestration. The system achieves portability across environments while maintaining clear architectural boundaries between core services and optional monitoring layers.

Observability and monitoring are integrated from the start, with Prometheus metrics and Grafana dashboards providing visibility into request rate, job completion, and queue health. Automated unit and component testing with CI/CD integration through GitHub Actions ensures code quality, and comprehensive technical documentation explains architectural decisions, implementation choices, and deployment procedures.

9 Future works

This system serves as a solid foundation for a production-grade distributed audio processing pipeline. The project successfully addresses the core proposal objectives: distributed architecture, asynchronous processing, fault tolerance, and observability. However, several items proposed in the initial specification remain unimplemented or only partially realized.

Key areas for extension and enhancement include:

- **Automated integration and end-to-end testing** (proposed but partially implemented): Currently, unit and component tests are automated with pytest and GitHub Actions, but full integration and end-to-end test suites remain manual. Automated testing should be extended using container-based test harnesses (e.g., TestContainers or Docker Compose-based e2e scenarios) to validate the complete workflow across all services, catch regressions early, and improve overall reliability.
- **Frontend implementation** (partially implemented): The current implementation is a minimal browser-based client. The proposal intended a fully-featured web interface; this should be enhanced with real-time progress indicators, detailed error messages, job history, and rich state visualization. Consider progressive loading, offline support, and improved user experience.
- **Kubernetes and infrastructure enhancements**: Extend the Helm charts with scaling policies (e.g., autoscaling based on queue depth or worker CPU), resource limits, and production-oriented network policies. Document recommended configurations for multi-replica deployments and cloud-native environments.
- **Observability enhancements**: Add distributed tracing (e.g., OpenTelemetry) to track requests across service boundaries and simplify debugging across the pipeline. Implement granular logging for error cases and state transitions. Extend Grafana dashboards with more advanced metrics and alerting rules for operational visibility.

References

[1] B. Burns. *Designing Distributed Systems.* O'Reilly Media, 2018.

[2] M. Fowler and J. Lewis. Microservices. *martinfowler.com*, 2014.