Distributed Audio Processing Pipeline with Observability

Presented by: Matteo Cardellini

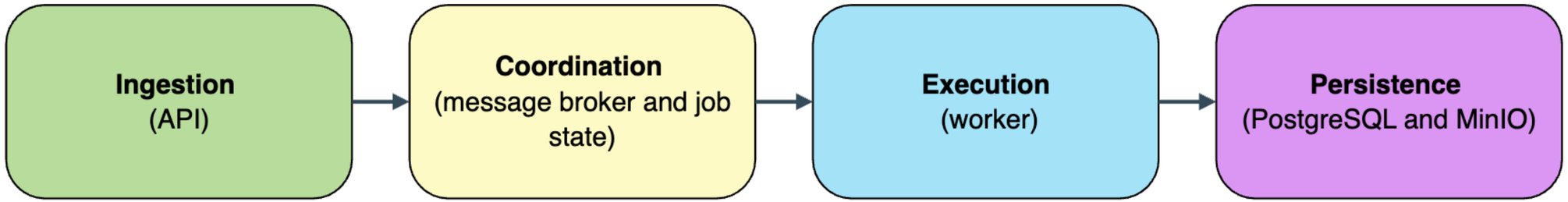
Student ID: 0001186864

Concept

- **Type of product:** Distributed, service-oriented application
- **Why distribution:**
  Decouple ingestion from processing → independent scaling, fault isolation, responsiveness
- **Three core components:**
  - Web API (FastAPI),
  - Background Worker,
  - Lightweight Frontend
- **Use case:** Asynchronous audio file processing and metadata extraction

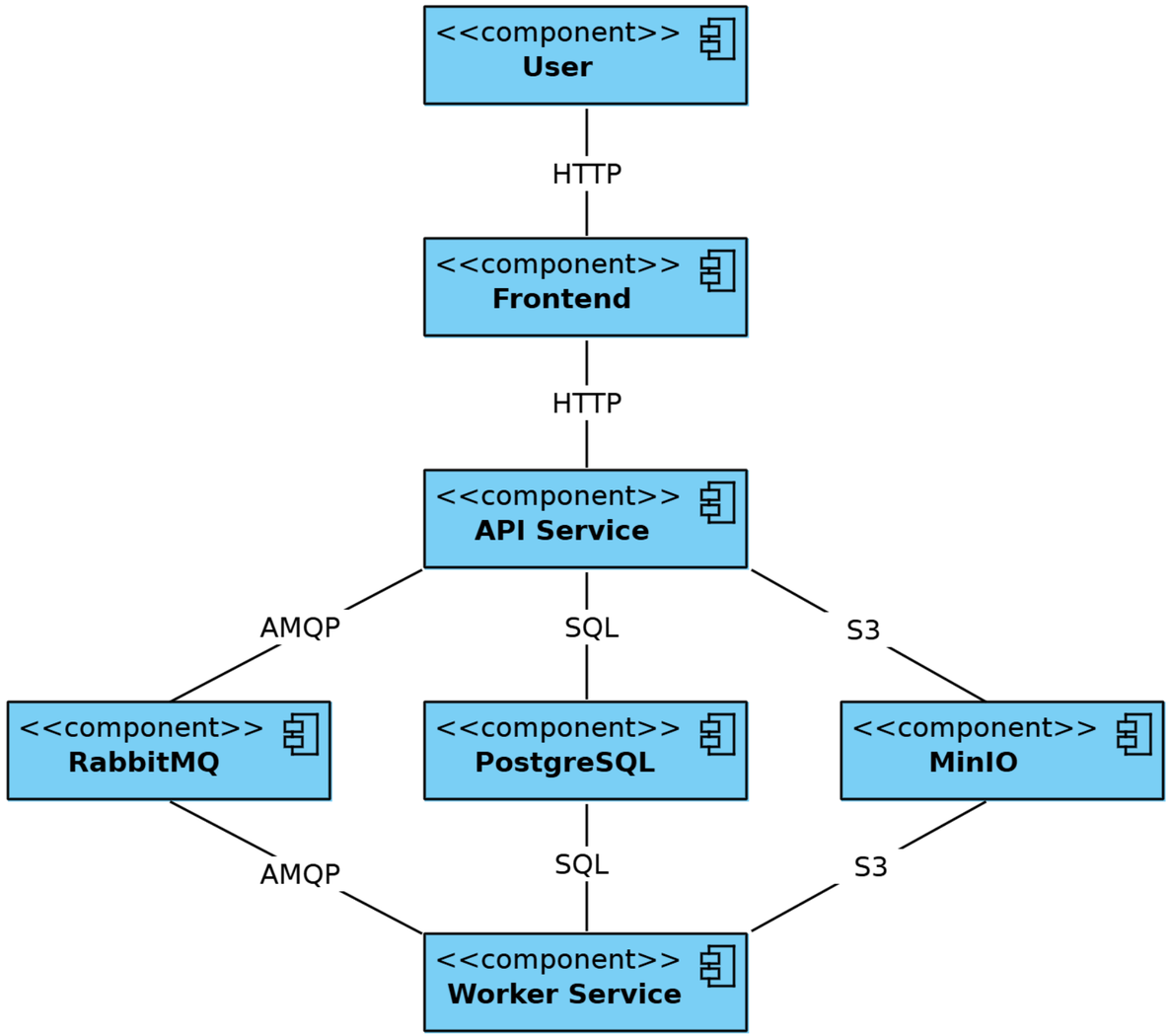
Architecture

- **Architectural style:** Event-driven, service-oriented
- **Hybrid interaction:** Synchronous HTTP (client-facing) + Asynchronous AMQP (internal coordination)
- Separation enables independent scaling, fault isolation and maintainability



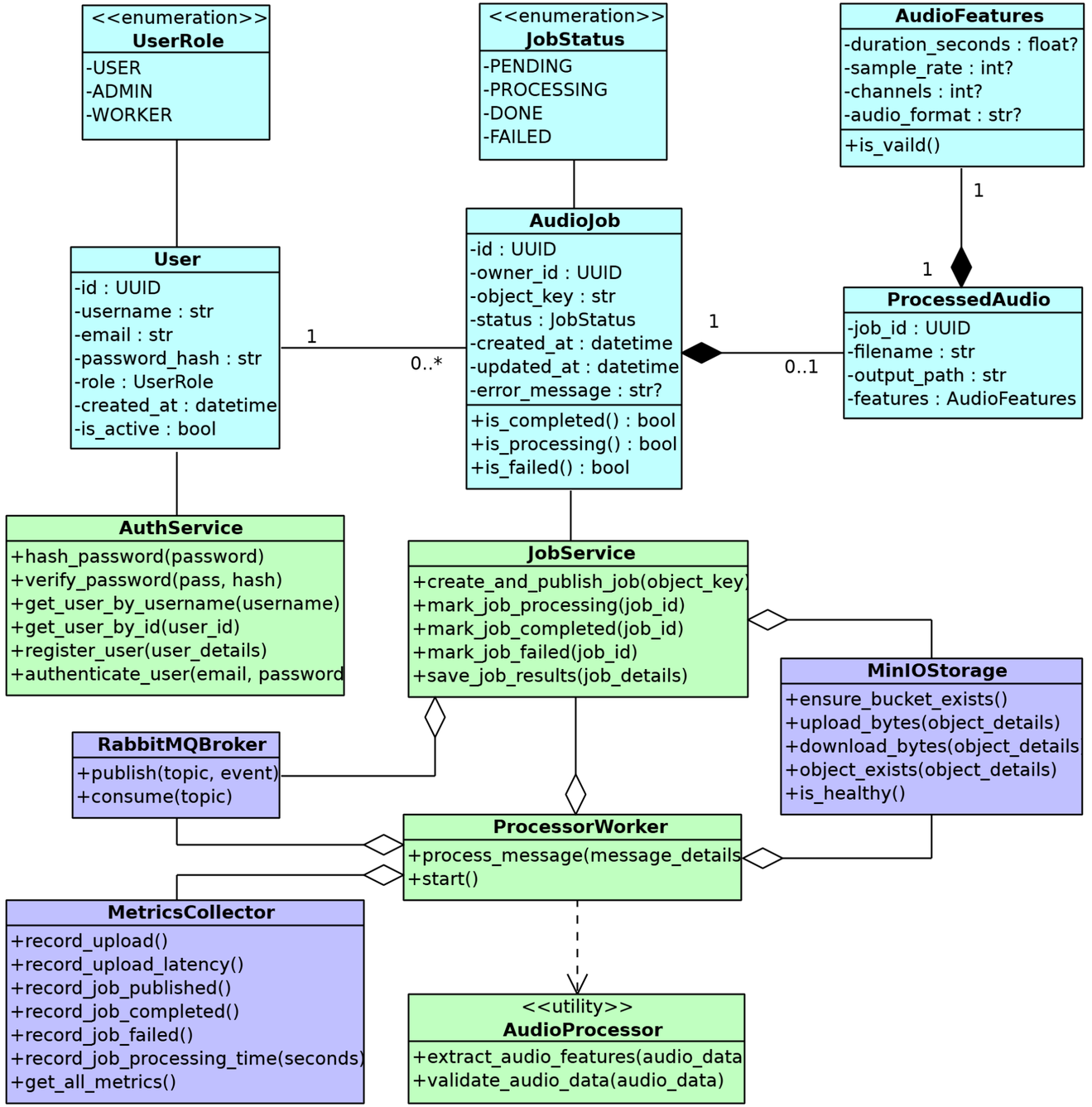
Infrastructure

- **Six runtime components:** Frontend, API, Worker, RabbitMQ, PostgreSQL, MinIO
- **Communication protocols:** HTTP, AMQP, SQL, S3
- **Kubernetes-native:** DNS-based service discovery, automatic load balancing
- Stateless services enable **horizontal scaling**

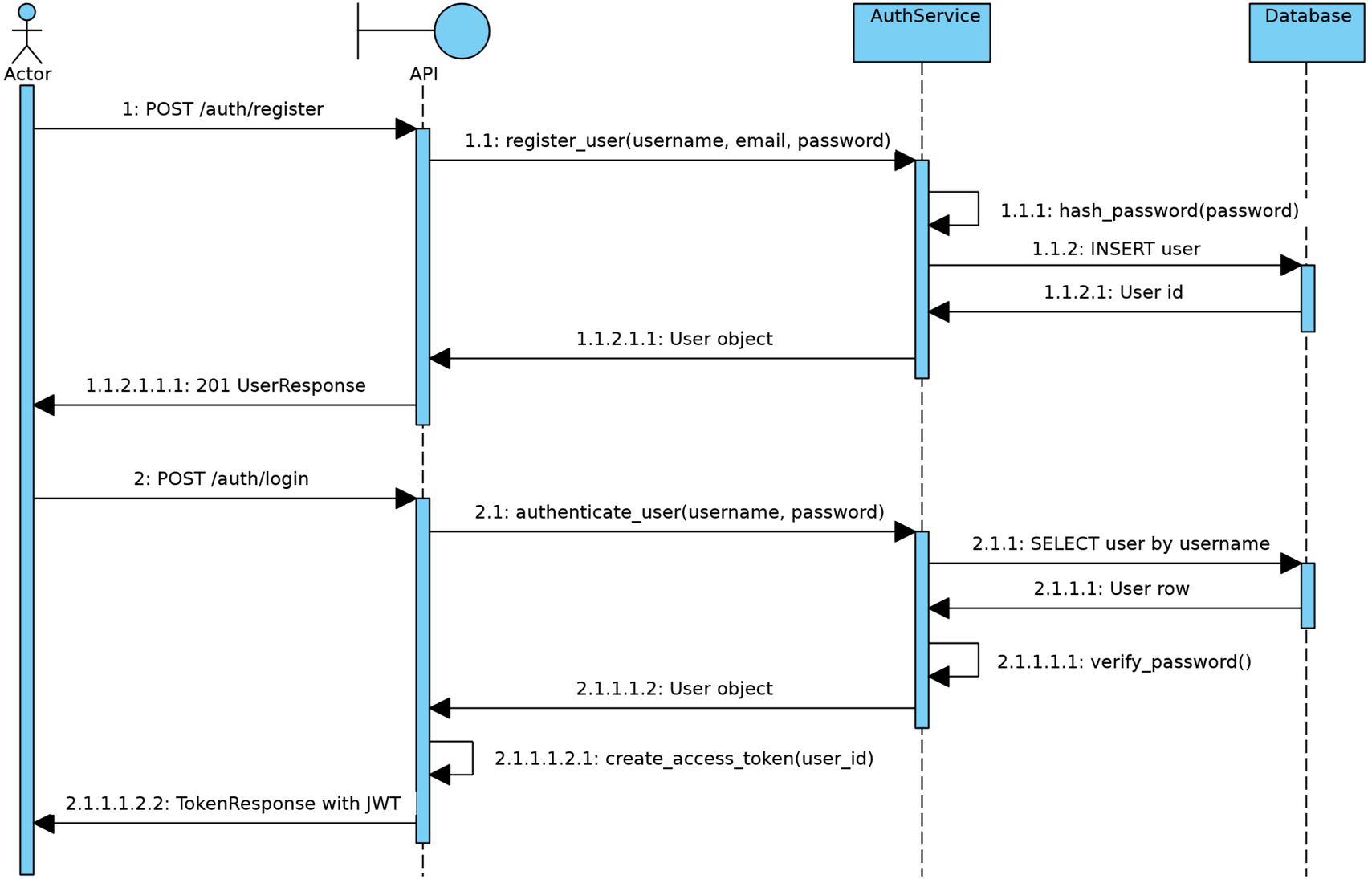


Modeling

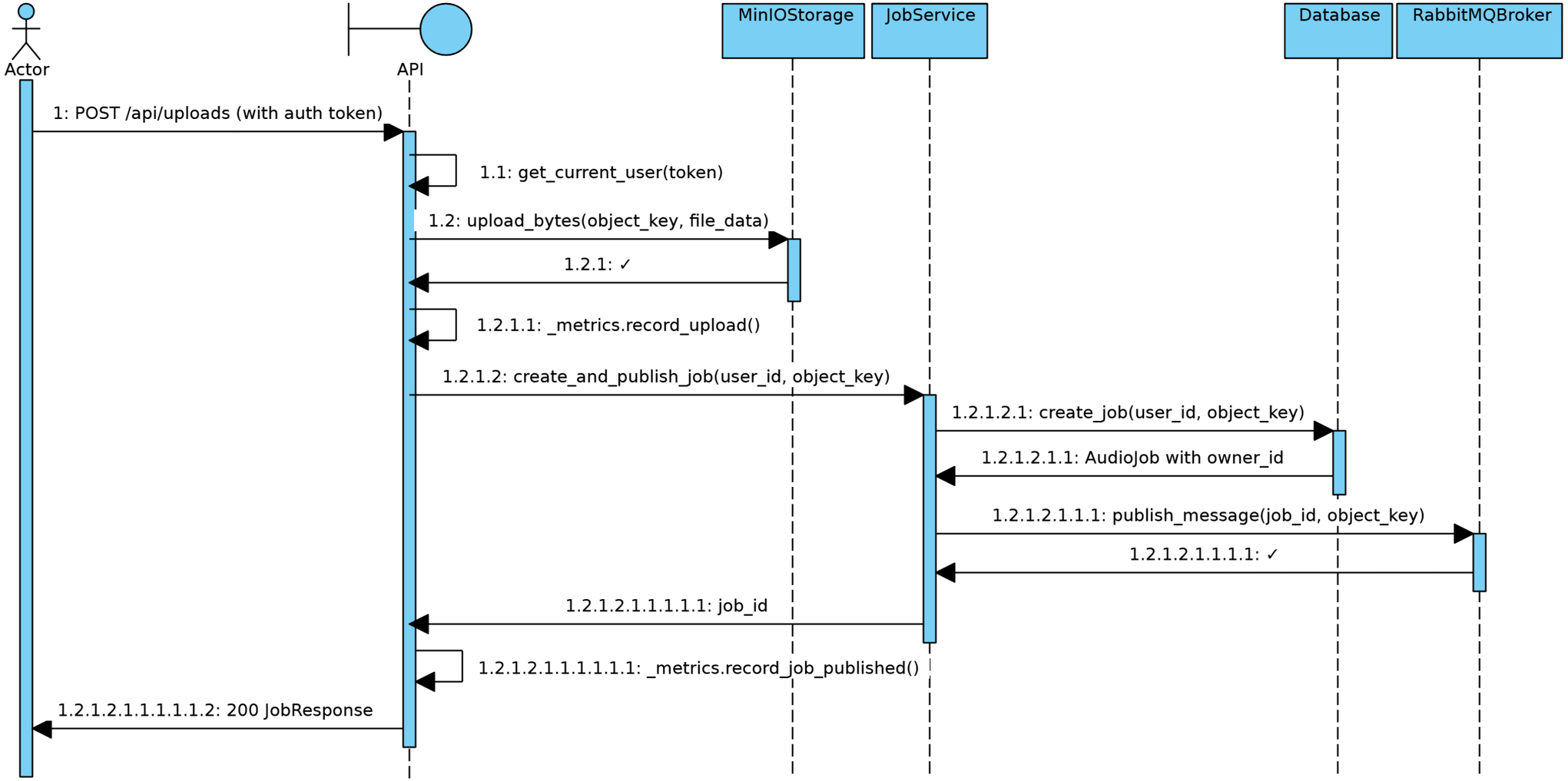
- **Domain entities:** User, AudioJob, ProcessedAudio, AudioFeatures
- **Entity-to-infrastructure mapping:** PostgreSQL (metadata), MinIO (binaries), RabbitMQ (coordination)
- **Domain events:** job created, queued, processing started, completed, failed



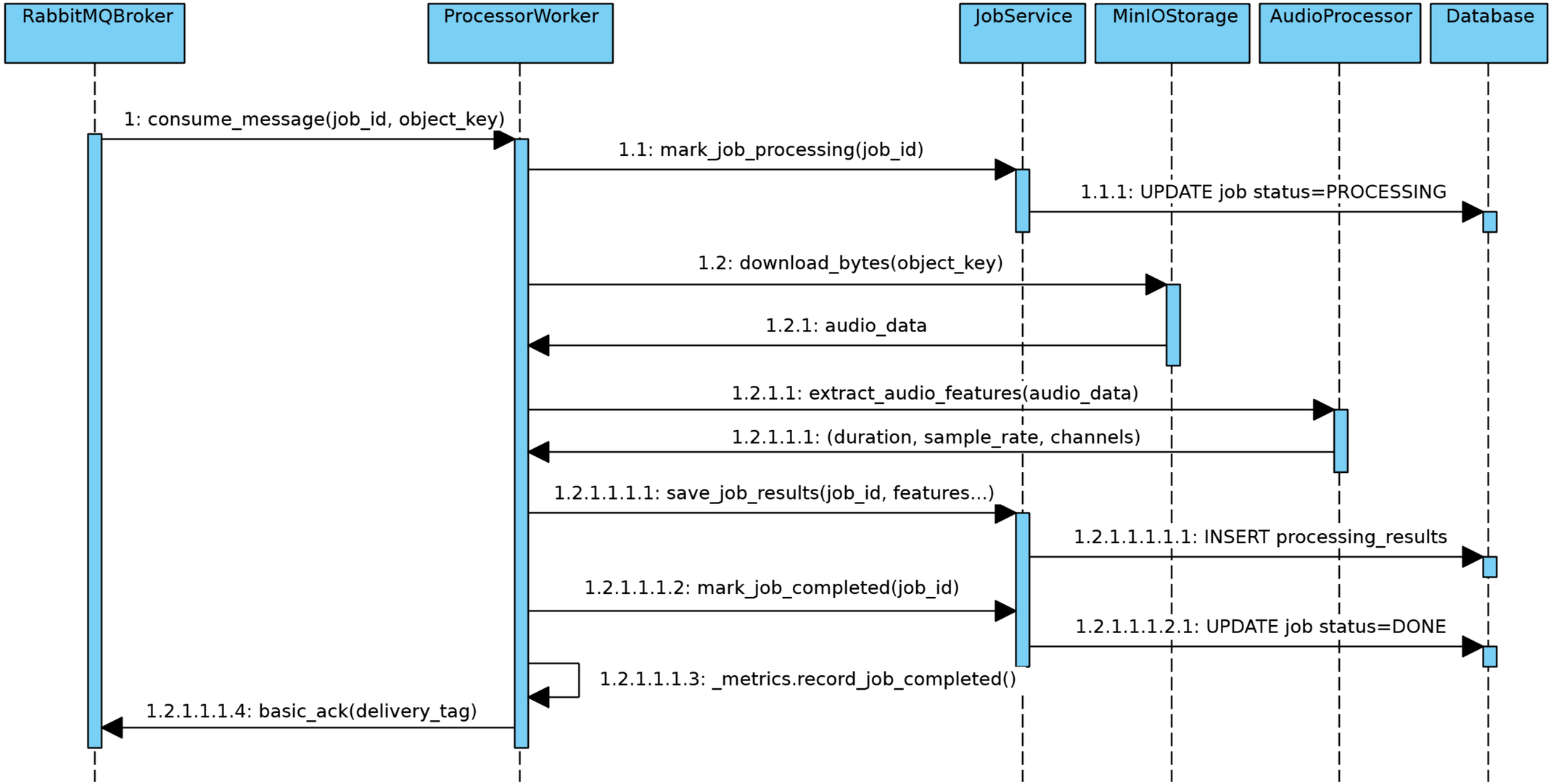
Interaction: Authentication



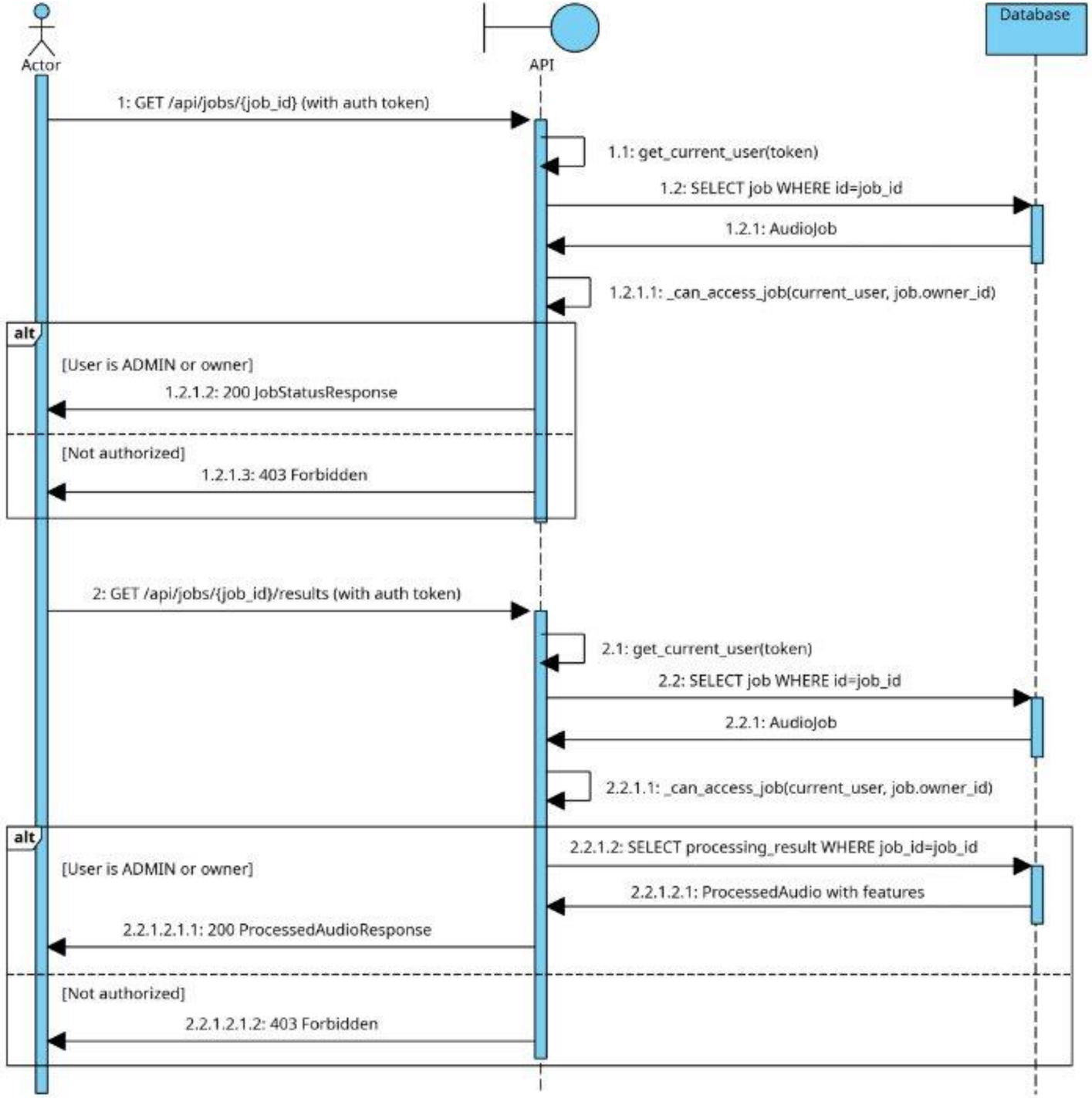
Interaction: Upload and job creation



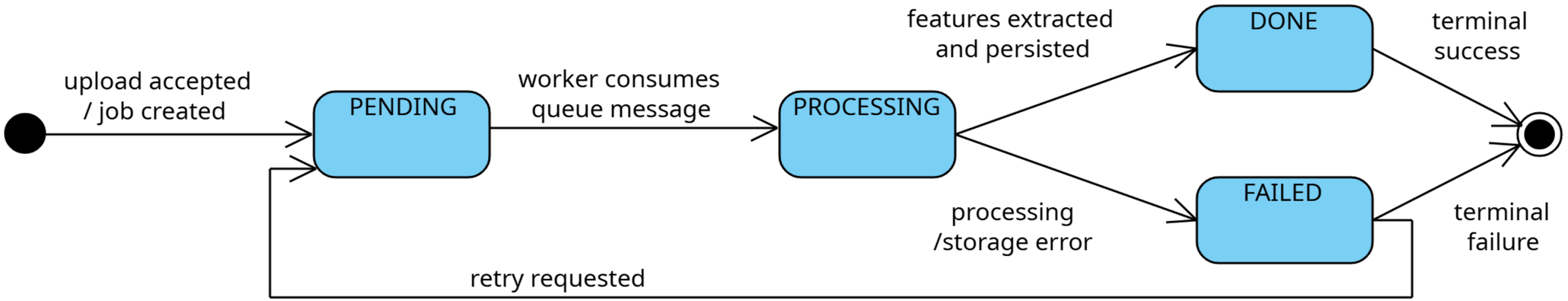
Interaction: Worker processing



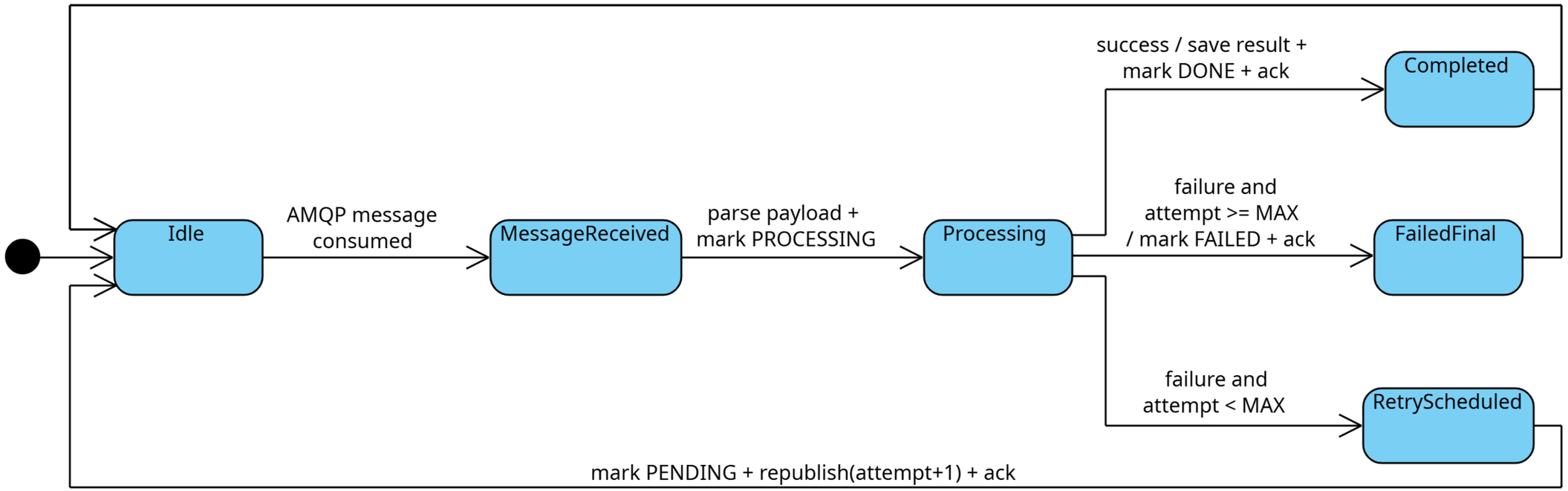
Interaction: Job status and result retrieval



Behaviour: AudioJob lifecicle



Behaviour: Worker execution



Data Consistency

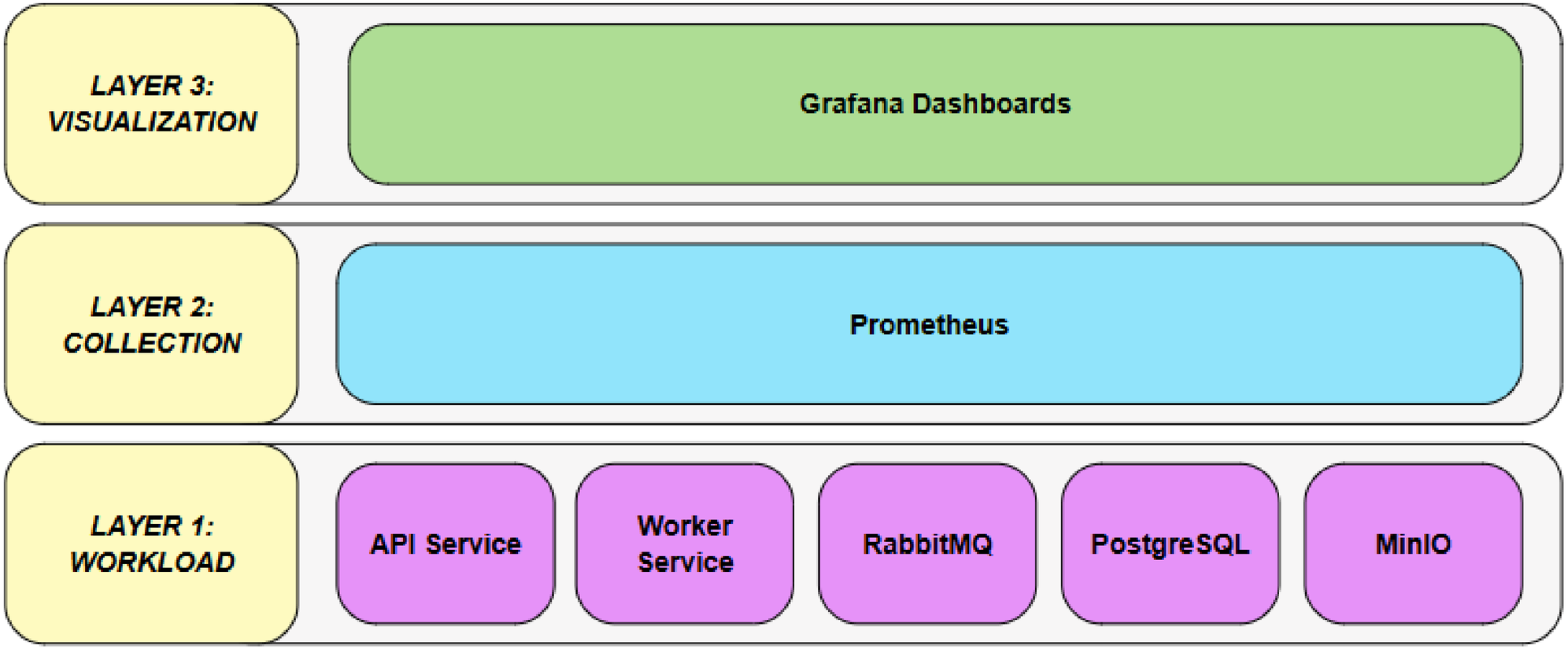
- **Persistent business data:** Users, jobs, metadata, audio references, timestamps, errors
- **Storage strategy:** PostgreSQL for structured metadata, MinIO for binary files
- Message broker **NOT** system of record (only coordination layer)
- **Concurrency approach:** Many readers, few writers; small explicit updates
- **Shared data:** Job ID, owner ID, storage key, state, timestamps maintained across components via DB and messages

Fault-Tolerance and Availability

- **Fault Tolerance:** Durable state (PostgreSQL/MinIO) + async retries + component isolation
  - No separate replication layer; resilience via externalized persistent state
  - Worker restarts don't lose jobs (metadata in DB, queue messages preserved)
  - Retry handling for transient failures (bounded by retry budget)

- **Availability:** Load balancing + horizontal scaling (no caching layer)
  - API scales with multiple replicas behind Service
  - Workers compete for RabbitMQ messages (load distribution)
  - Network partitioning: Fail fast rather than mask partial failures (consistency over availability)

Observability



Security

- **Authentication:** JWT tokens (HS256), issued on successful login, validated on protected endpoints

- **Authorization:** RBAC with three roles (USER, ADMIN, WORKER)
  - Users can only access own resources (job ownership enforced in queries)
  - Admin operations restricted to ADMIN role

- **Cryptography:**
  - Password hashing: bcrypt (stored hashes, verified on login)
  - Token signing: HMAC SHA-256 (prevents tampering, validates issuer)

- **Stateless verification:** No session storage required, scales across API replicas

Implementation

- **Languages & Frameworks:** Python 3.12, FastAPI + Uvicorn, worker as separate Python process
- **Protocols:** HTTP (client-API), AMQP (API-worker coordination), SQL (metadata queries), S3 (file storage)
- **Data Encoding:** JSON (API responses, queue messages), multipart/form-data (uploads)
- **Messaging:** RabbitMQ AMQP, durable queue-based coordination
- **Dependency Management:** Poetry for reproducible Python environments
- **Audio processing:** Python standard `wave` module for WAV metadata extraction

Validation

- **Unit/Component Testing:** pytest for API and worker services
  - API tests: Health, auth, upload, status, results, failure paths
  - Worker tests: Message consumption, retry logic, startup, DB readiness

- **Test Automation:** GitHub Actions CI/CD on push/PR (python-tests.yml)
  - Runs: `poetry run pytest` for both services

- **Manual Testing:** Full Docker Compose stack acceptance testing
  - Workflow: Register → Login → Upload → Status polling → Result retrieval
  - Validates end-to-end lifecycle and failure visibility

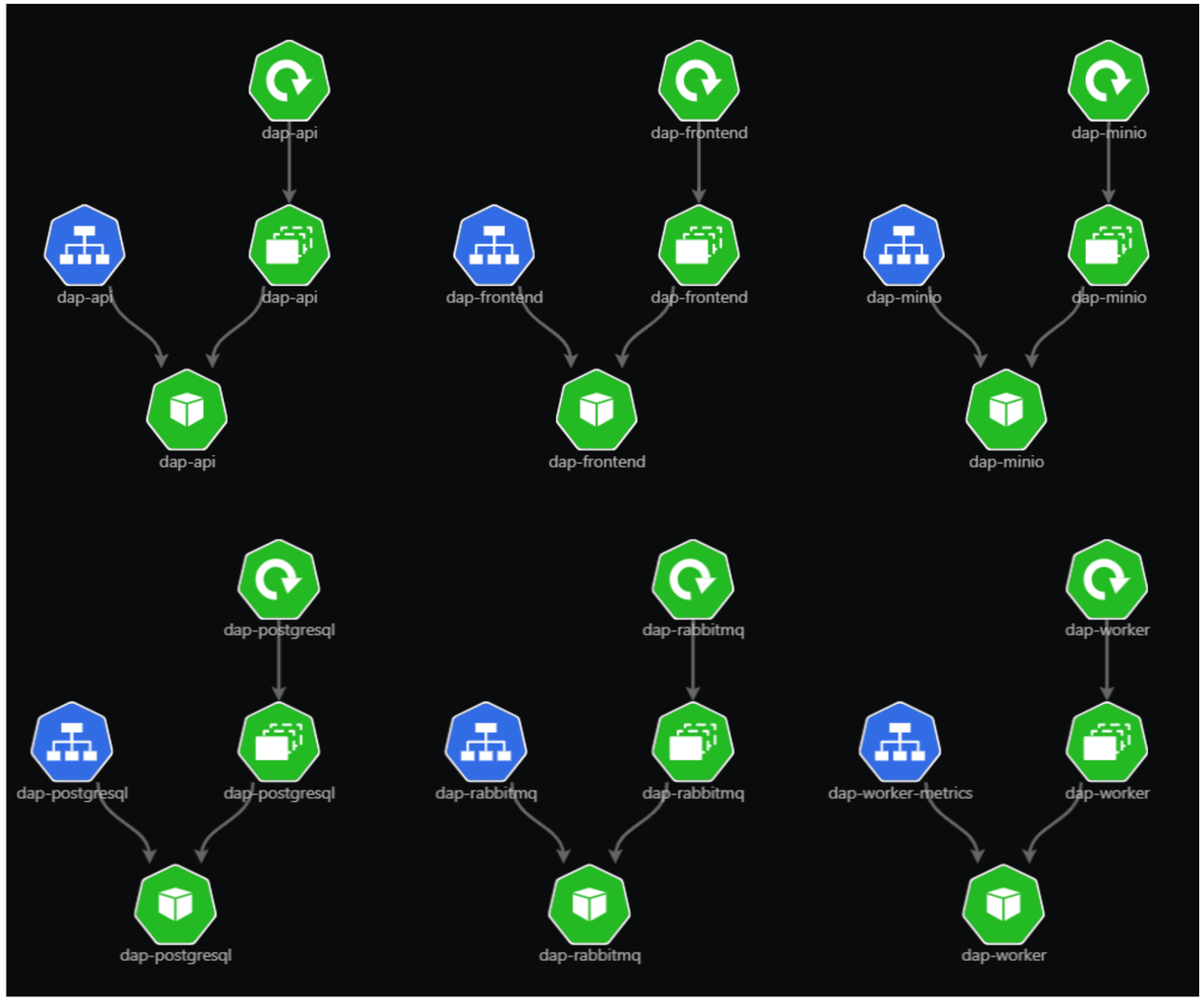
- **Coverage:** Service logic in isolation; full e2e automation pending

Deployment Strategy

- Deployment was engineered to support two goals: fast local reproduction and realistic Kubernetes orchestration.
- The stack is split into a base application layer and an optional observability layer, so monitoring can be enabled without changing the core workload.
- Configuration is externalized through Compose files, Helm values, environment variables, and CI-built container images.
- This makes the same system easy to run, inspect, and demo in different environments.

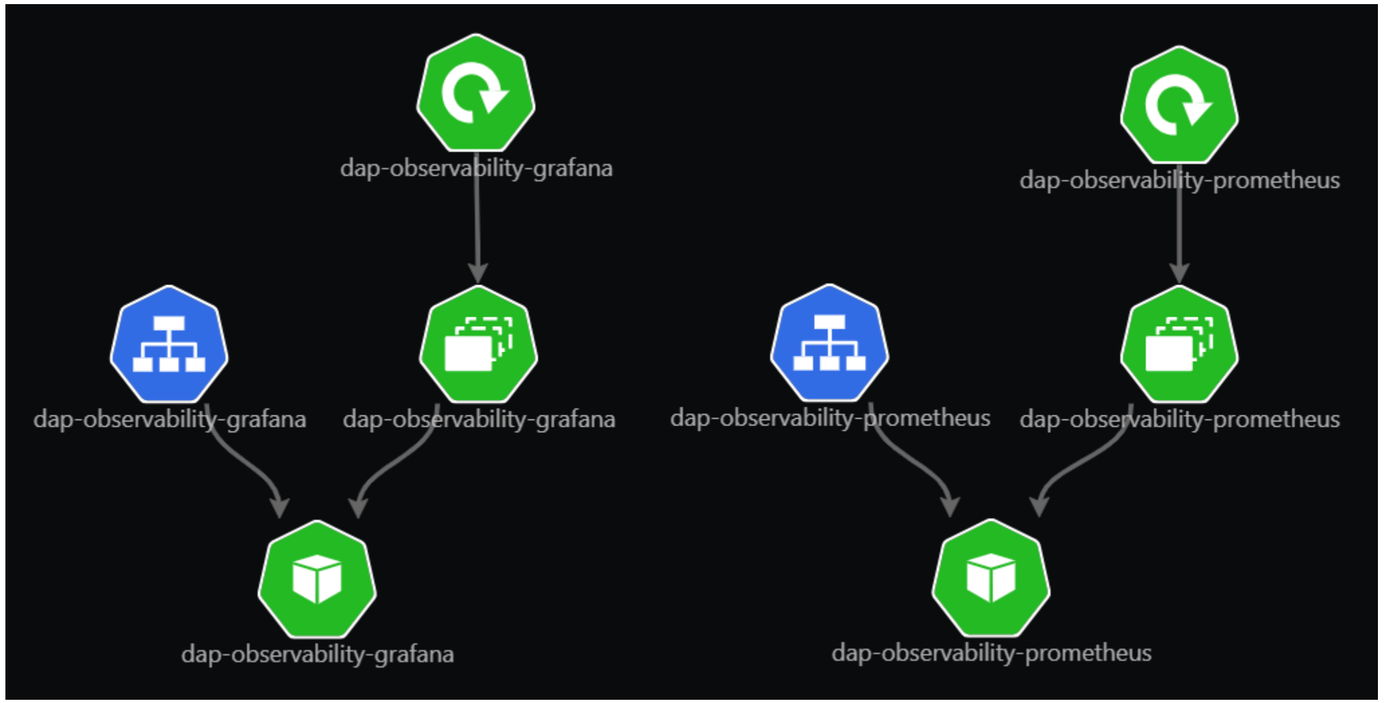
Application Deployment

- Full runtime stack in one chart.
- Kubernetes Services handle discovery and scale.
- Tuned for kind, portable elsewhere.



Observability Deployment

- Separate chart for monitoring.
- Prometheus and Grafana run independently.
- Synced and CI-friendly.



DEMO

Video (people in UNIBO organization can access)

Future works

- **Automated E2E testing:** TestContainers / Docker Compose-based e2e harnesses for full pipeline validation
- **Enhanced frontend:** Real-time progress indicators, job history, detailed error messages, rich state visualization
- **Kubernetes enhancements:** Auto-scaling policies (HPA based on queue depth), resource limits, network policies
- **Observability improvements:** Distributed tracing (OpenTelemetry), granular logging, advanced Grafana alerts

THANKS FOR YOUR ATTENTION!