DHT Solution
迪 衡 科 技

RAPTOR AI Framework

Aigle Release 0.1

Technical System Information

Project Manager: Titan Hon
Document Author: Titan Hon
Document Contributors: Cing, Fung, George, Mimi, Robert, Nelson
Date: October 28, 2025
Version: 1.2

Version History of this document

Change No.	Date	Version	Name	Description
1	October 16, 2025	0.1	Titan	First cut for review
2	October 20, 2025	0.2	Cing	Added Cache System Design
3	October 20, 2025	0.3	George	Added Audio Processing System Design
4	October 20, 2025	0.4	Fung	Added Video Processing System Design
5	October 20, 2025	0.5	Mimi	Added Document Processing System Design
6	October 20, 2025	0.6	Robert	Added Vector Search Engine Design
7	October 20, 2025	0.7	Cing	Added Asset Management Service Design
8	October 20, 2025	0.8	George	Added Model Lifecycle Management Design
9	October 20, 2025	0.9	Mimi	Added Kafka Service Design
10	October 23, 2025	1.0	Titan	Aigle 0.1 Release System Overview
11	October 27, 2025	1.1	Titan	Update document as per peer review
11	October 28, 2025	1.2	Nelson	Update Restful API and Component Summary

DHT Solution
迪 衡 科 技

Table of Contents

1. System Overview

1.1 Executive Summary

The Contextualized Intelligence Engine (CIE) is an AI-native content intelligence platform built under the RAPTOR AI Framework. Code name **Aigle (v0.1)**, this Community Edition represents the first public release of the system.

CIE converts raw, unstructured media — documents, spreadsheets, presentations, images, audio, and video — into an indexed, queryable, semantically enriched knowledge layer. Instead of treating files as static storage objects, CIE analyzes, understands, tags, summarizes, and exposes them through intelligent APIs. The result: organizations can perform deep semantic search and automated insight extraction across multimodal assets.

Technically, CIE is implemented as a cloud-native, event-driven microservices platform. It integrates model-driven AI processing (LLM, ASR, VLM, OCR, summarization, embeddings), Kafka-based orchestration, GPU-accelerated inference, lifecycle-managed object storage, semantic vector search, and intelligent caching. The platform delivers:

Multimodal AI understanding – Documents: structure-aware parsing, hierarchical summarization – Audio: transcription, speaker diarization, acoustic classification – Video: scene analysis, OCR, vision-language description, timeline-aligned summary – Images: visual interpretation and description All outputs become searchable knowledge.

Semantic retrieval at scale All processed content is embedded (BAAI/bge-m3, 1024-dim) and indexed in Qdrant for cosine-similarity search. Results are enriched with metadata and filtered for access control.

Lifecycle governance and security Assets are versioned (lakeFS), stored in S3-compatible object storage (SeaweedFS), managed via RBAC, and exposed only via short-lived, pre-signed URLs. TTL-driven archival and deletion are built in.

Operational intelligence The system ships with observability: Prometheus, Grafana, Alertmanager, structured logs, and state tracking for every pipeline through Redis and Kafka.

The CIE Community Edition (Aigle 0.1) focuses on openness, transparency, reproducibility, and extensibility. It is designed to be deployed locally or in private infrastructure using Docker Compose, with a planned path to Kubernetes and enterprise hardening.

This document describes Aigle 0.1 from a systems, architectural, and component standpoint, and outlines the forward roadmap for both Community and Enterprise editions.

1.2 Development Roadmap

CIE is released in two product lines:

- **Community Version (Open Source)**: Aigle → Aquila → Astur → Milvus → Eryr → Adler
- **Enterprise Version (Commercial)**: Harpria → Eagle → Hawk → Kite → Harrier → Falcon

1.2.1 Community Version (Open Source Track)

- **Oct 2025 – v0.1, 0.2, 0.3 · Code Name: Aigle**

Scope: Initial public framework release under RAPTOR AI Framework

- Asset ingestion APIs
- Multimodal processing services (document/audio/video/image)
- Vector search engine backed by Qdrant
- Redis-based semantic caching
- Kafka orchestration of ~21 microservices
- Unified inference API for LLM / VLM / ASR
- Docker Compose reference deployment
- Basic MCP integration (LLM-native tool interface, ~30%)

- **Mar 2026 – v1.x · Code Name: Aquila**

Focus: Developer usability and extensibility

- Stable plugin API for third-party processors and new models
- Helm charts and Kubernetes deployment patterns
- Distributed inference routing (e.g. vLLM / TensorRT backends)
- Fine-tunable summarization templates per content type
- Expanded MCP coverage (semantic search, video/audio tooling)
- Multi-node Qdrant and Redis Cluster defaults

- **Mar 2027 – v2.x · Code Name: Astur**

Focus: Knowledge reasoning

- GPU-aware autoscaling for workers
- Policy-driven data retention and compliance workflows
- Query-time personalization / access filtering
- Cross-asset relationship graphing and entity linking
- Temporal/event-aware media reasoning
- Context stitching for long-horizon investigations

1.2.2 Enterprise Version (Commercial Track)

- **Jan 2026 – 0.x · Harpria** Compliance, RBAC, SLA packaging, SSO, audit logging
- **Jun 2026 – 1.x · Eagle** Enterprise IAM, Advanced governance, SOC2/ISO27001 alignment
- **Jun 2027 – 2.x · Hawk** Policy-driven retention, High-availability HA/DR rollout patterns

Community and Enterprise share the same architectural core. Enterprise layers on compliance, governance, tenancy, and support SLAs.

2. Technical Design: RAPTOR release Aigle 0.1

A. Overview

Agile release 0.1 is the implementation of **RAPTOR** framework of **Contextualized Intelligence Engine (CIE)** is an enterprise-grade, multimodal AI platform designed to extract, process, and enable semantic search across documents, audio, video, and images. Built on a cloud-native microservices architecture, CIE provides end-to-end asset lifecycle management, intelligent caching, and AI-driven content analysis through a unified API surface.

Architecture Philosophy

CIE implements a **layered, event-driven architecture** that separates concerns across data persistence, orchestration, AI processing, and application services:

Data Layer

Provides persistent storage (MySQL, SeaweedFS, lakeFS), vector search (Qdrant), and high-performance caching (Redis Cluster with semantic caching).

AI Engine Layer

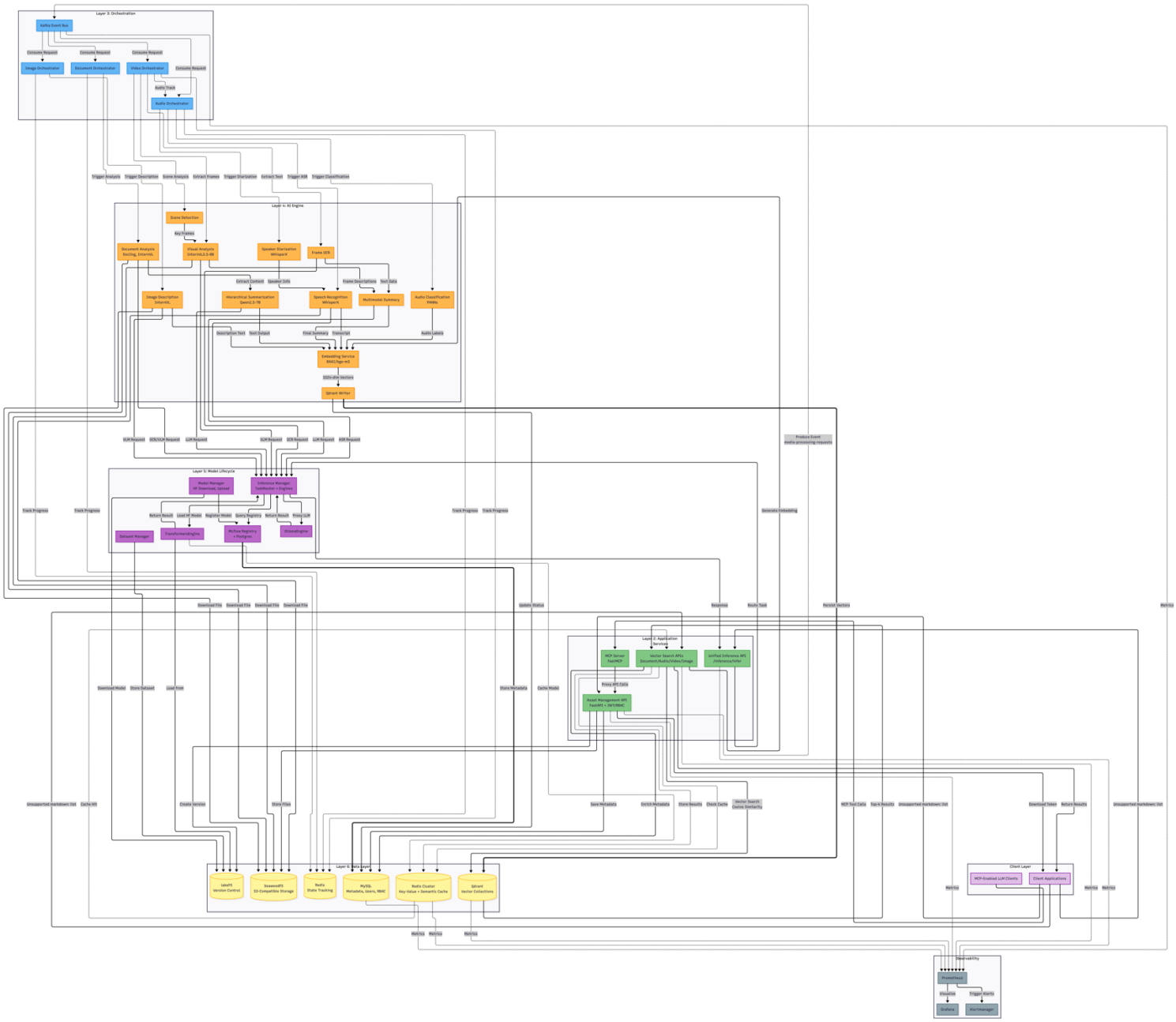
Houses specialized AI processing services for each media type, leveraging state-of-the-art models for transcription, classification, visual understanding, and summarization.

Orchestration Layer

Kafka-based event-driven workflow coordination across 21 microservices, ensuring decoupled, scalable, and fault-tolerant processing pipelines.

Application Services Layer

Exposes RESTful APIs for asset management, user authentication (JWT + RBAC), and lifecycle automation (TTL-based archival/deletion).



Solid arrows (→): Synchronous API calls/Direct data transfer | **Dashed arrows (-.->):** Asynchronous events/Cache operations | **Thick arrows (==>):** Data persistence / Storage operations

Figure 1. System Flow Diagram – RAPTOR Framework Aigle Release 0.1

(Detail Diagram https://github.com/DHT-AI-Studio/RAPTOR/blob/main/Aigle/0.1/doc/Aigle_0.1_system.svg)

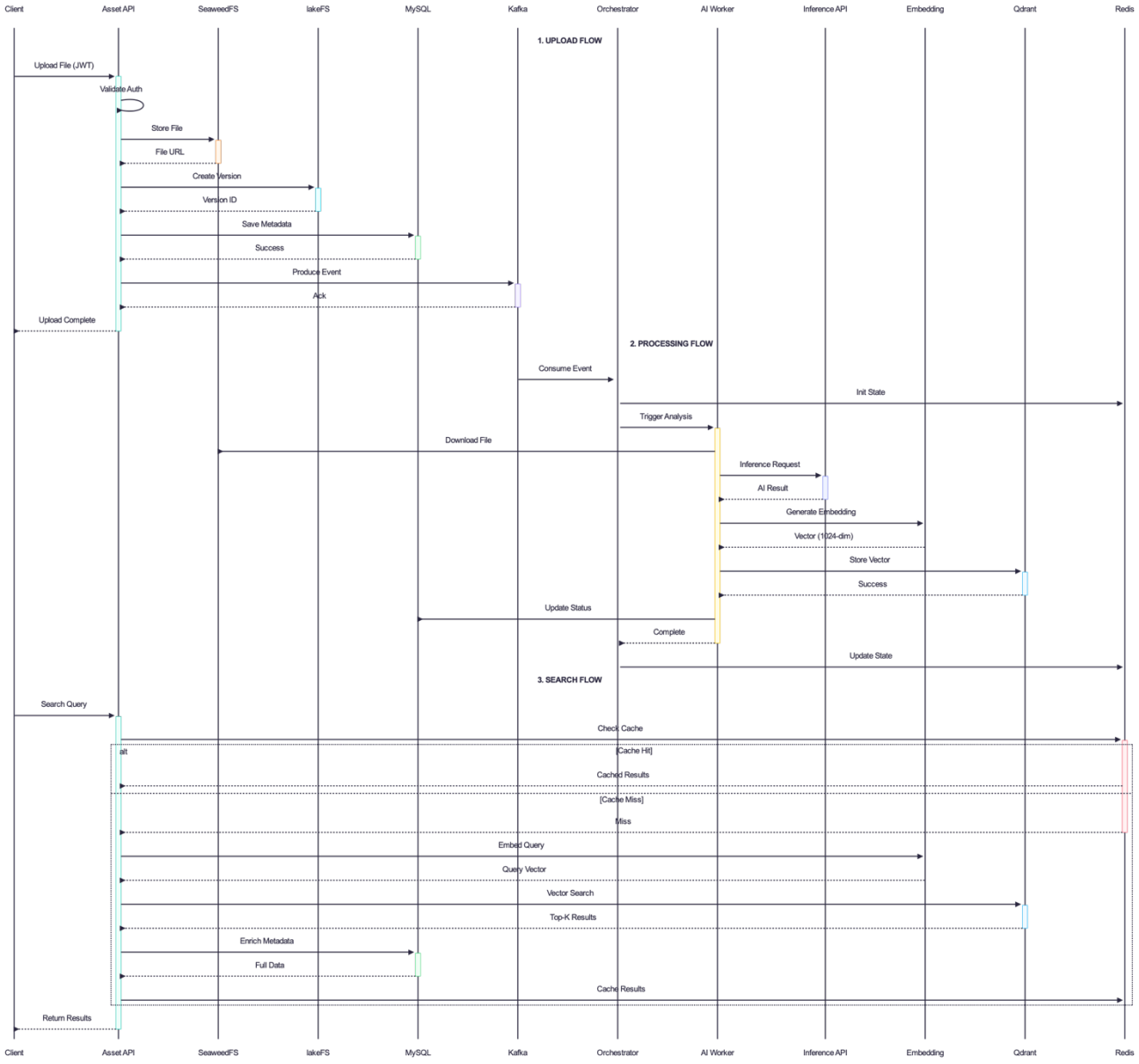


Figure 2. Data Flow Diagram – RAPTOR Framework Aigle Release 0.1

(Detail Diagram https://github.com/DHT-AI-Studio/RAPTOR/blob/main/Aigle/0.1/doc/Aigle_0.1_data.svg)

B. Key Points and Core Workflows

Key Points

1. Multi-Layer Architecture

- **Layer 2:** Application Services - Asset Management, User Management

- **Layer 3:** Orchestration Layer - Kafka-based event-driven microservices

- **Layer 4:** AI Engine Layer - Document/Audio/Video Processing, Vector Search

- **Layer 6:** Data Layer - Redis Cache, MySQL, Qdrant, SeaweedFS, lakeFS

2. Core Capabilities

- **Multimodal AI Processing:** Documents (PDF/PPT/DOC/TXT/CSV/XLSX), Audio (speech recognition, diarization, classification), Video (VLM analysis, OCR, scene detection), Images

- **Semantic Search:** Vector-based search across all media types using BAAI/bge-m3 embeddings (1024-dim) with Qdrant

- **Intelligent Caching:** Hybrid key-value + semantic caching with Redis Cluster/RediSearch

- **Asset Lifecycle Management:** Versioned storage (lakeFS), object storage (SeaweedFS), metadata (MySQL), automated archival/deletion

3. Technology Stack

- **AI/ML:** WhisperX (ASR), PANNs (audio classification), InternVL3.5-8B (VLM), Qwen2.5-7B (summarization), BAAI/bge-m3 (embeddings)

- **Orchestration:** Kafka (21 microservices), FastAPI, Docker Compose

- **Storage:** MySQL, SeaweedFS (S3-compatible), lakeFS (versioning), Qdrant (vector DB)

- **Cache & State:** Redis Cluster, RediSearch

- **ML Ops:** MLflow + Postgres, DeepSpeed (distributed training)

4. Design Principles

- **Modularity:** Microservices architecture with clear separation of concerns

- **Scalability:** Redis Cluster, Kafka workers, GPU-accelerated processing

- **Reliability:** State tracking (Redis), versioning (lakeFS), fault-tolerant pipelines

- **Security:** JWT authentication, RBAC, short-lived access tokens

- **Observability:** Prometheus, Grafana, structured logging

5. Integration Points

- **MCP (Model Context Protocol):** LLM-native interface for asset operations (30% implemented)

- **Unified Inference API:** Single endpoint for text-generation, VLM, ASR, OCR, audio/video/document analysis

- **RESTful APIs:** Asset management, vector search, model lifecycle operations

Core Workflows

1. Asset Upload & Processing

- Client uploads media via Asset Management API (JWT authenticated)

- Asset stored in SeaweedFS/lakeFS with version tracking

- Metadata persisted in MySQL

- Kafka message triggers media-specific orchestrator

- Orchestrator coordinates AI processing workers (analysis → summarization → vectorization)

- Processed results stored in Qdrant for semantic search

2. Semantic Search

- Client query → Vector Search API (cached in Redis)

- Query embedded using BAAI/bge-m3 (1024-dim)

- Qdrant performs cosine similarity search across media collections

- Results filtered by status (active only) and returned with metadata

3. Model Lifecycle

- Models downloaded from HuggingFace → committed to lakeFS

- Registered in MLflow with resource estimates (VRAM, latency)

- Unified inference endpoint routes tasks to appropriate engine (Transformers/Ollama)

- LRU model cache with GPU-aware eviction

4. Intelligent Caching

- Standard caching: SHA256-hashed function calls (key-value)

- Semantic caching: BAAI/bge-m3 embeddings + RediSearch vector similarity (threshold: 0.85)

- Dynamic TTL extension based on hit frequency

- Automatic cleanup of expired locks/counters

Integration and Extensibility

Integration Capabilities

MCP Interface

Provides LLM-native tools for asset operations:

- **upload_file:** Upload primary and associated files with TTL policies
- **add_associated_files:** Append additional files to existing assets
- **file_download:** Retrieve assets with pre-signed URLs
- **list_file_versions:** Query version history
- **file_archive:** Move assets to archived state
- **file_delete:** Soft-delete assets

Status: 30% implemented (Asset Management only)

Missing: Semantic search, video/audio/document analysis MCP tools

Unified Inference API

Single endpoint for all AI tasks:

POST /inference/infer

Supported Tasks:

- Text generation (Ollama/Transformers)
- Vision-Language Models (InternVL3.5-8B)
- Automatic Speech Recognition (WhisperX)
- Optical Character Recognition (InternVL)
- Audio classification (PANNs)
- Video/Document analysis

RESTful APIs

- **Authentication:**
  - POST /register - Register a new gateway user
  - POST /login - Obtain a gateway JWT (OAuth2 password flow)
- **Asset Management:**
  - POST /fileupload - Upload single asset to storage
  - POST /fileupload_batch - Batch upload multiple assets with concurrency control
  - GET /filedownload/{asset_path}/{version_id} - Download asset by version
  - POST /filearchive/{asset_path}/{version_id} - Archive asset
  - POST /delfile/{asset_path}/{version_id} - Destroy (delete) archived asset
  - GET /fileversions/{asset_path}/{filename} - List all versions of an asset
- **File Upload with Analysis:**
  - POST /fileupload_analysis - Upload single file and trigger Kafka processing
  - POST /fileupload_analysis_batch - Batch upload files and auto-trigger analysis via Kafka
- **Vector Search:**
  - POST /video_search - Semantic search for videos (text/summary embeddings)
  - POST /audio_search - Semantic search for audio (text/summary embeddings)

- POST /document_search - Semantic search for documents (supports CSV, PDF, DOCX)
- POST /image_search - Semantic search for images (text/summary embeddings)
- POST /unified_search – Global semantic search

- **Processing:**
  - POST /process-file - Send file processing request to Kafka topic
  - GET /processing/cache/{m_type}/{key} - Retrieve cached processing results from Redis
  - Supported m_type: document, video, image, audio
- **Health & Monitoring:**
  - GET / - Root health check (liveness probe)
  - GET /health - Health check endpoint with status details

Extensibility Features

Observability Stack

- **Prometheus:** Metrics collection from all services
- **Grafana:** Real-time dashboards and visualization
- **Alertmanager:** Automated alerting for system health
- **Structured Logging:** JSON logs for centralized analysis

Deployment Flexibility

- **Docker Compose:** Development environment (48 services)
- **Kubernetes-Ready:** Production deployment with Helm charts
- **API-First Design:** OpenAPI/Swagger documentation for all endpoints
- **Language Agnostic:** HTTP/JSON interfaces for any client

Model Extensibility

- **HuggingFace Integration:** Download and register any HF model
- **Custom Model Support:** Upload proprietary models to lakeFS
- **Multi-Engine Architecture:** Add new inference engines (e.g., vLLM, TensorRT)
- **Resource Estimation:** Automatic VRAM/latency profiling

Storage Extensibility

- **S3 Compatibility:** SeaweedFS can be replaced with AWS S3, MinIO
- **Version Control:** lakeFS provides Git-like branching for data
- **Database Options:** MySQL can be replaced with PostgreSQL
- **Cache Backends:** Redis Cluster supports sharding and replication

C. Core Components

Component Summary Table

Component	Technology	Port(s)	Purpose
Asset Management API	FastAPI + MySQL + lakeFS + SeaweedFS	host port : container port 8086 : 8000	Asset CRUD, versioning, lifecycle management
Vector Search API	FastAPI + Qdrant + Redis	host port : container port 8821-8824 : 8811-8814	Semantic search across media types
MCP Server	FastMCP	Additional configuration	LLM-native asset operations
Inference API	FastAPI + Transformers/Ollama	host port : container port 8010 : 8010	Unified ML inference endpoint
Kafka Brokers	Apache Kafka	host port : container port 19002-19004 : 19092-19094	Event-driven orchestration
Redis Cluster	Redis 7.x + RediSearch	host port : container port 7000-7005 17000-17005 : 7000-7005 17000-17005 6391:6379	Hybrid key-value + semantic cache
MySQL	MySQL 8.x	host port : container port 3307 : 3306	Metadata, users, RBAC
Qdrant	Qdrant	host port : container port 6334 : 6333	Vector storage & search
SeaweedFS	SeaweedFS	host port : container port master 9343-9345 : 9333-9335 volume 8091-8094 : 8081-8084 filer 8898 : 8888 s3 8343 : 8333	S3-compatible object storage
lakeFS	lakeFS	host port : container port 8011 : 8000	Git-like versioning for data
MLflow	MLflow + Postgres	host port : container port 5000 : 5000	Model registry & tracking
Prometheus	Prometheus	host port : container port 9091 : 9090	Metrics collection
Grafana	Grafana	host port : container port 3031 : 3000	Metrics visualization

Security & Access Control

Authentication Flow

1. **User Registration:** POST /users → MySQL (hashed password)
2. **Token Acquisition:** POST /token → JWT token (exp: configurable)
3. **API Access:** All endpoints (except /users, /token) require Authorization: Bearer header
4. **RBAC Enforcement:** MySQL tables (users, commit_history) control permissions per asset/operation

Asset Access Security

- **Short-lived tokens:** lakeFS/SeaweedFS presigned URLs configurable in .env, default 20 min
- **Version control:** Immutable commits in lakeFS prevent unauthorized modifications
- **Status filtering:** Only status=active assets returned by default (archived/deleted hidden)

🚀 Deployment Architecture

Docker Compose (Development)

Services: 48 containers

- Redis Cluster (6 nodes + cluster creator)

- Kafka + Zookeeper

- 27 Kafka microservices / 10 containers (orchestrators + workers)

- MySQL, Qdrant, SeaweedFS, lakeFS, MLflow

- Prometheus, Grafana, Alertmanager

Kubernetes (Production - Planned)

- Helm charts for service deployment

- Horizontal Pod Autoscaling for Kafka workers

- StatefulSets for Redis Cluster, Kafka, databases

- GPU node pools for AI processing services

- Istio service mesh for mTLS + traffic management

📈 Performance Characteristics

Caching Performance

- **Standard Cache Hit Rate:** ~85% (typical workloads)

- **Semantic Cache Similarity Threshold:** 0.85 (configurable)

- **TTL Management:** Dynamic extension based on hit frequency

- **Cleanup Cycle:** Hourly automated cleanup of expired locks/counters

Vector Search Performance

- **Embedding Dimension:** 1024 (BAAI/bge-m3)
- **Similarity Metric:** Cosine similarity
- **Index Type:** Qdrant HNSW (Hierarchical Navigable Small World)
- **Query Latency:** <100ms (p95) for 1M vectors with Redis cache

AI Processing Throughput

- **Document:** ~2-5 pages/sec (Docling extraction)
- **Audio:** Real-time ASR (WhisperX with GPU)
- **Video:** ~1-2 FPS for VLM analysis (InternVL3.5-8B)
- **Batch Processing:** Kafka workers scale horizontally

D. System Flow and Data Flow

🔄 Flow Sequences

1. Upload & Processing Flow

Client → Asset API → SeaweedFS (store) + lakeFS (version) + MySQL (metadata)
↓
Kafka Event → Orchestrator → AI Workers → Inference API
↓
Embedding Service → Qdrant Writer → Qdrant (persist)

2. Search Flow

Client → Search API → Redis Cache (check)
↓ (if miss)
Embedding Service → Qdrant (vector search)
↓
Redis Cache (store) → Client (return results)

3. Inference Flow

AI Worker → Inference API → Inference Manager
↓
Query MLflow → Load Model from lakeFS → Execute (Transformers/Ollama)

4. State Tracking Flow

Orchestrator → Redis State (track progress)
↑
Orchestrator checks state before each step

5. Observability Flow

All Services → Prometheus (metrics collection)
↓
Grafana (visualization) + Alertmanager (alerts)

📤 Upload & Processing Flow (Detailed)

1. CLIENT UPLOAD

└─→ POST /fileupload (JWT token + files)

2. ASSET API

├─→ Validate JWT & RBAC
├─→ Store file in SeaweedFS
├─→ Create version in lakeFS
├─→ Save metadata to MySQL
└─→ Produce Kafka message

3. KAFKA ORCHESTRATION

└─→ Route to media-specific orchestrator
  ├─→ document-processing-requests
  ├─→ audio-processing-requests
  ├─→ video-processing-requests
  └─→ image-processing-requests

4. AI PROCESSING WORKERS

├─→ Analysis Worker
│ ├─→ Document: Docling + InternVL (OCR)
│ ├─→ Audio: WhisperX (ASR + Diarization)
│ ├─→ Video: InternVL (VLM) + Scene Detection
│ └─→ Image: InternVL (Description)
│
├─→ Summarization Worker
│ └─→ Qwen2.5-7B (Hierarchical summaries)
│
└─→ Vectorization Worker
  ├─→ BAAI/bge-m3 (1024-dim embeddings)
  └─→ Store in Qdrant

5. STORAGE

└─→ Qdrant vector database (searchable)

🔍 Search Flow (Detailed)

1. CLIENT QUERY

└─→ POST /search (query: "find videos about...")

2. VECTOR SEARCH API

├─→ Check Redis cache (semantic)
│ └─→ HIT: Return cached results ✓
│ └─→ MISS: Continue ↓
│
├─→ Generate query embedding (BAAI/bge-m3)
│
├─→ Qdrant vector search
│ ├─→ Cosine similarity
│ ├─→ Filter: status=active
│ └─→ Top-k results |
├─→ Cache results in Redis
│
└─→ Return to client

3. CLIENT RESPONSE

└─→ Ranked results with metadata

E. Flow Types and Data Workload

Data Flow Overview

The CIE system implements six primary data flow patterns:

1. **How data enters the system**

   Client → APIs (Asset Management, Vector Search, Inference)

2. **Where data is stored**

   SeaweedFS (files), MySQL (metadata), Qdrant (vectors), Redis (cache)

3. **How data flows through processing**

   Kafka → Orchestrators → Workers

4. **How data is transformed**

   AI Workers → Inference → Embeddings

5. **How data is retrieved**

   Search API → Cache/Qdrant → Client

6. **How system is monitored**

   Metrics → Prometheus → Grafana

📊 Data Flow Summary Table

Flow Type	Source	Destination	Data Format	Protocol
File Upload	Client	Asset API	Multipart/Form	HTTP POST
File Storage	Asset API	SeaweedFS	Binary	S3 API
Versioning	Asset API	lakeFS	Metadata	REST API
Metadata	Asset API	MySQL	JSON	SQL
Events	Asset API	Kafka	JSON	Kafka Protocol
Orchestration	Kafka	Orchestrators	JSON	Kafka Consumer
State Tracking	Orchestrators	Redis	Key-Value	Redis Protocol
AI Processing	Workers	Inference API	JSON	HTTP POST
Embeddings	Embedding Service	Qdrant	1024-dim Float32	gRPC
Vector Search	Search API	Qdrant	Query Vector	gRPC
Caching	Any Service	Redis	Serialized Data	Redis Protocol
Metrics	All Services	Prometheus	Time-Series	HTTP Scrape

✅ Data Workload Characteristics

Upload Workload

- **Entry Point:** Client → Asset Management API

- **Authentication:** JWT token validation

- **Processing:**

    - File: Multipart/form-data → SeaweedFS (binary)

    - Version: Metadata → lakeFS (commit)

    - Metadata: JSON → MySQL (relational)

    - Event: JSON → Kafka (message queue)

- **Typical Size:** 1MB - 1GB per asset

- **Throughput:** Limited by network bandwidth and SeaweedFS write performance

Processing Workload

- **Trigger:** Kafka event consumption
- **Orchestration:**
  - Document: 4 microservices coordinate processing
  - Audio: 7 specialized workers (ASR, diarization, classification, etc.)
  - Video: 7 workers (VLM, scene detection, OCR, audio extraction, etc.)
  - Image: 3 workers (description, OCR, vectorization)
- **State Management:** Redis tracks progress per request_id
- **AI Inference:** GPU-accelerated models (InternVL, WhisperX, Qwen2.5)
- **Output:** JSON structured data + 1024-dim vectors

Search Workload

- **Entry Point:** Client → Vector Search API
- **Cache Check:** Redis semantic cache (80-85% hit rate)
- **Embedding Generation:** BAAI/bge-m3 (1024-dim, ~50ms)
- **Vector Search:** Qdrant HNSW index (cosine similarity, <100ms p95)
- **Metadata Enrichment:** MySQL join operations
- **Cache Storage:** Redis with TTL (3600s default)
- **Typical Size:** 1KB query → 10-100KB results

Inference Workload

- **Entry Point:** AI Workers → Unified Inference API
- **Routing:** InferenceManager → TaskRouter → Engine selection
- **Model Loading:**
  - Cache check (LRU)
  - Load from lakeFS if miss
  - GPU VRAM estimation
- **Execution:**
  - Transformers: HuggingFace pipelines
  - Ollama: External service proxy
- **Response:** JSON with inference results + metrics

Monitoring Workload

- **Metric Collection:** All services → Prometheus (scrape every 15s)
- **Storage:** Time-series database (retention: 15 days default)
- **Visualization:** Grafana queries Prometheus
- **Alerting:** Alertmanager evaluates rules every 1 minute
- **Typical Metrics:**
  - Request rate (req/s)
  - Latency (p50, p95, p99)
  - Error rate (%)
  - Cache hit rate (%)
  - GPU utilization (%)
  - Queue depth (Kafka lag)

🔄 Data Lifecycle Summary

```
UPLOAD → STORE → PROCESS → VECTORIZE → INDEX → SEARCH → CACHE → RETRIEVE
   ↓       ↓        ↓          ↓          ↓       ↓        ↓        ↓
Client  SeaweedFS  AI      Embedding   Qdrant  Redis   Client  Client
         lakeFS   Workers   Service             Cache
         MySQL    Kafka
```

3. Redis Cache System

The Redis Cache System is a high-performance caching layer within CIE’ s Layer 6: Data Layer, designed to optimize data retrieval and reduce latency for AI-driven workloads. It supports both key-value and semantic caching, leveraging Redis Cluster or Master-Slave configurations to ensure scalability and resilience. The system mitigates cache breakdown scenarios, such as thundering herds, through intelligent locking and dynamic TTL management, enabling efficient reuse of embeddings and insights across multimodal pipelines.

3.1 System Design

The Redis Cache System is architected for modularity, resilience, and scalability, integrating seamlessly with CIE’ s semantic intelligence framework. It operates via the CacheManager, which abstracts Redis interactions and generates cache keys using SHA256 hashing of function metadata and arguments. For semantic caching, it employs the BAAI/bge-m3 model to generate embeddings stored in RediSearch indexes under the sem_cache: namespace. This hybrid approach—combining deterministic key-based caching with vector-based similarity search—enables context-aware cache hits for semantically equivalent queries, with a configurable similarity threshold (default: 0.8) to balance precision and recall.

Key Design Principles:

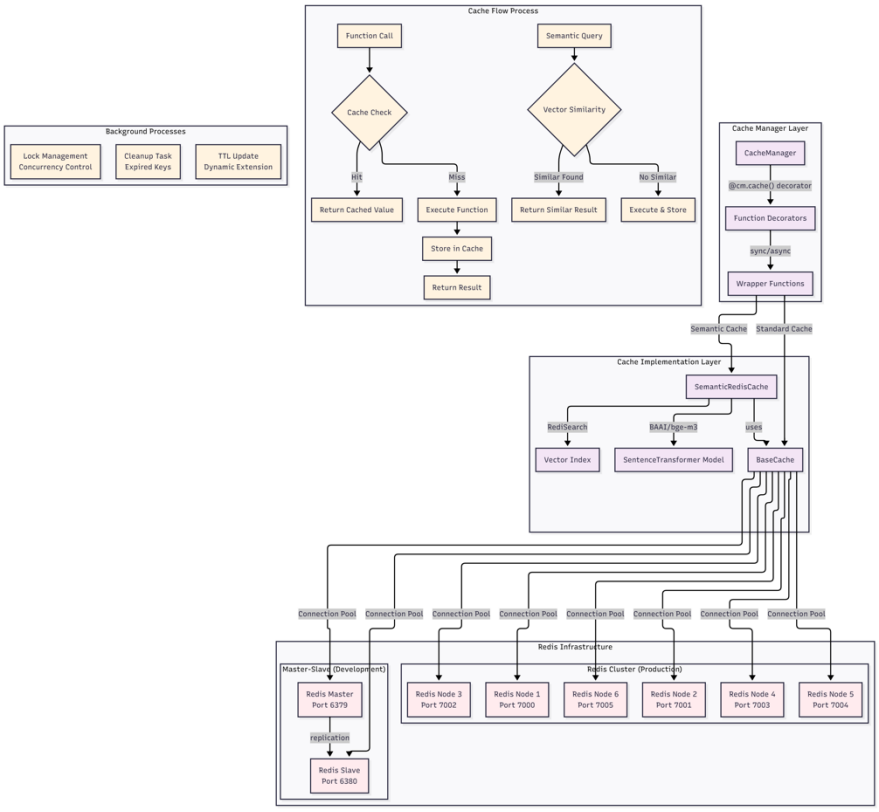
- **Modularity** – Decorator-driven caching (e.g., @cm.cache()) integrates seamlessly with sync and async functions.
- **Resilience** – In-progress task tracking and automated cleanup of expired locks and counters (e.g., hourly) maintain cache hygiene without manual intervention.
- **Scalability** – Supports Redis Cluster for production-scale deployments and Master-Slave setups for development or testing environments.
- **Semantic Intelligence** – Embedding-based caching with RediSearch vector indexes enables context-aware reuse of semantically similar inputs.

Components:

- **CacheManager:** Orchestrates caching operations, including key generation, result storage, and locking.

- **BaseCache**: Manages Redis operations for key-value storage and retrieval.
- **SemanticRedisCache**: Extends caching with vector-based similarity search using RediSearch.
- **Redis Infrastructure**: Supports scalable deployments with Redis Cluster or Master-Slave setups.

3.2 System Flow Diagram

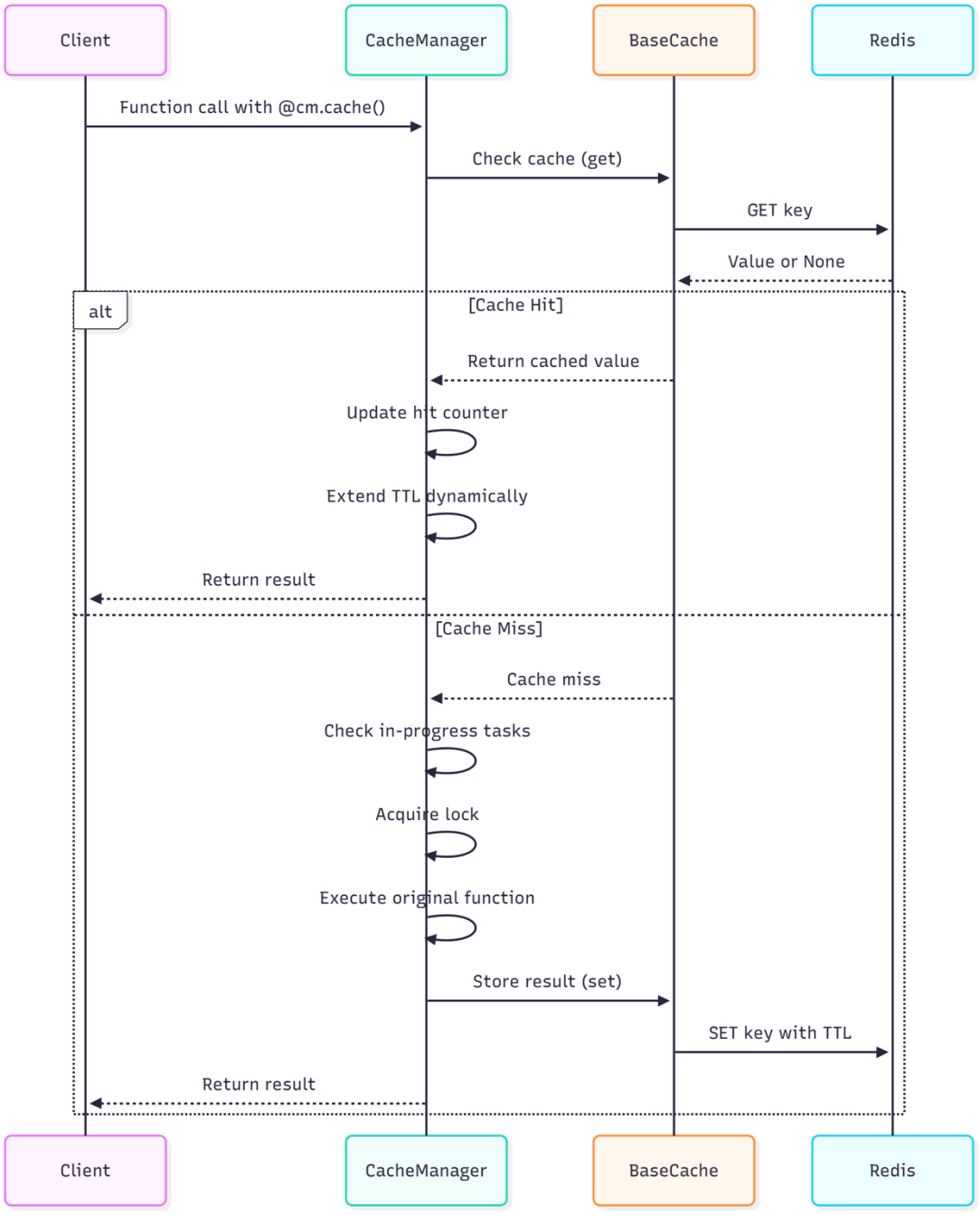


The Redis Cache System follows a layered flow to optimize caching and retrieval:

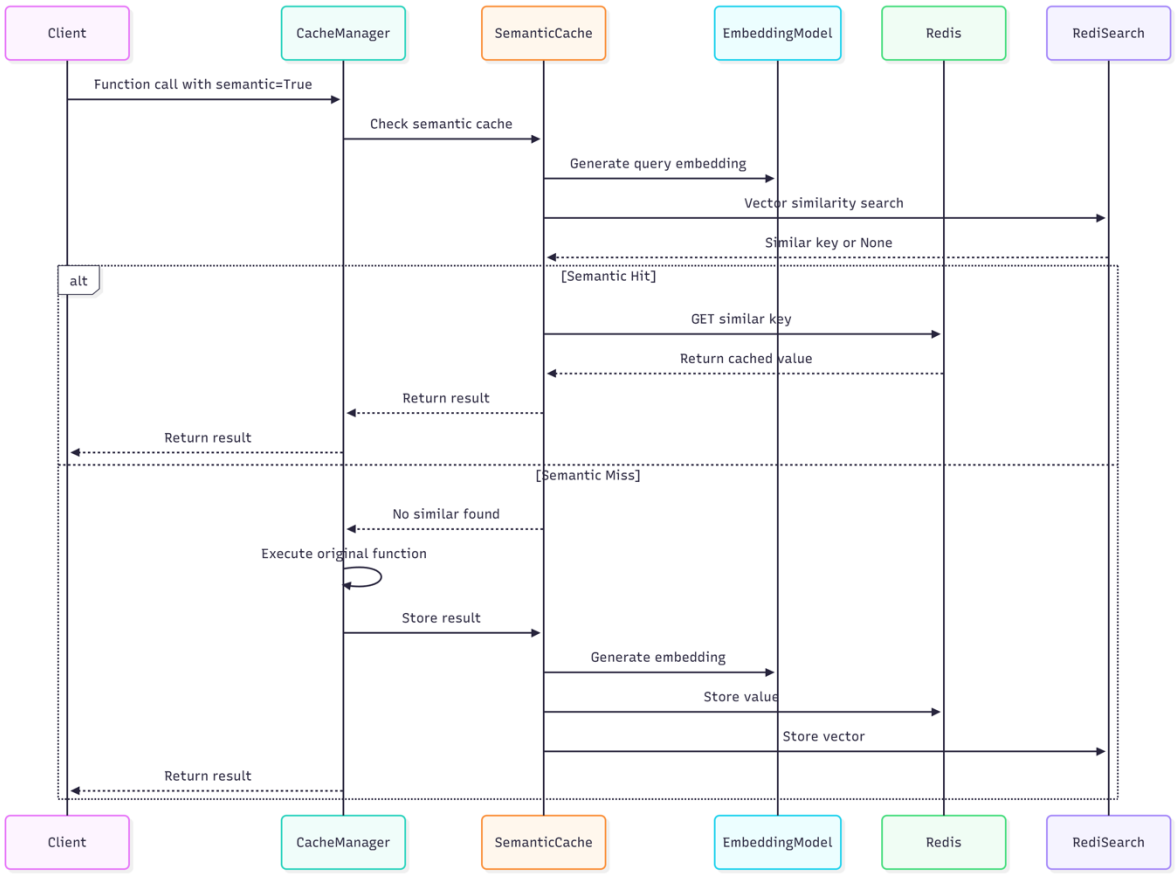
1. **Cache Flow Process**: Checks cache for existing results; returns hits instantly or computes and stores results on misses.
2. **Cache Manager Layer**: Uses decorators to handle key generation, locking, and result storage for both sync and async workflows.
3. **Cache Implementation Layer**: BaseCache manages Redis operations; SemanticRedisCache integrates BAAI/bge-m3 embeddings for semantic queries.
4. **Redis Infrastructure Layer**: Ensures consistency and scalability with Redis Cluster (production) or Master-Slave (development).
5. **Background Processes**: Asynchronous cleanup, dynamic TTL updates, and lock management maintain cache efficiency.

3.3 Data Flow Diagram

- Standard Caching Flow



- Semantic Caching Flow



The **standard caching flow** demonstrates how `@cm.cache()` handles function calls: the system first checks the cache for existing results using SHA256-generated keys, returning hits immediately. On a miss, it acquires a lock to prevent duplicate computation, executes the original function, stores the new result with a TTL, and returns it to the client.

The **semantic caching flow** extends this process with vector-based similarity search. When invoked with `@cm.cache(semantic=True)`, the CacheManager uses the BAAI/bge-m3 model to generate embeddings, performs a vector search in RediSearch, and retrieves semantically related results if available. Otherwise, it executes the function, stores both the result and its embedding, and updates the vector index for future reuse.

4. Asset Management Service

The Asset Management Service, part of CIE' s Layer 2: Application Services Layer, provides robust lifecycle management for digital assets, including versioning, access control, and automated archival/destruction. Built on FastAPI and orchestrated via Docker Compose, it integrates with MySQL for metadata, lakeFS for versioning, and SeaweedFS for object storage, ensuring scalability, reliability, and observability.

4.1 System Design

The service manages assets through RESTful APIs, enforcing JWT authentication and role-based access control (RBAC) via MySQL tables (users, commit_history). Assets are stored in SeaweedFS for high availability, with lakeFS providing immutable, branch-based versioning. Metadata, including version, asset path, MIME type, and lifecycle state, is persisted in MySQL. APScheduler automates lifecycle tasks (e.g., archival, deletion) based on TTL policies, updating Qdrant for downstream AI workflows. Observability is achieved through Prometheus, Grafana, Alertmanager, and Node Exporter, with environment-driven configuration via .env files for flexibility.

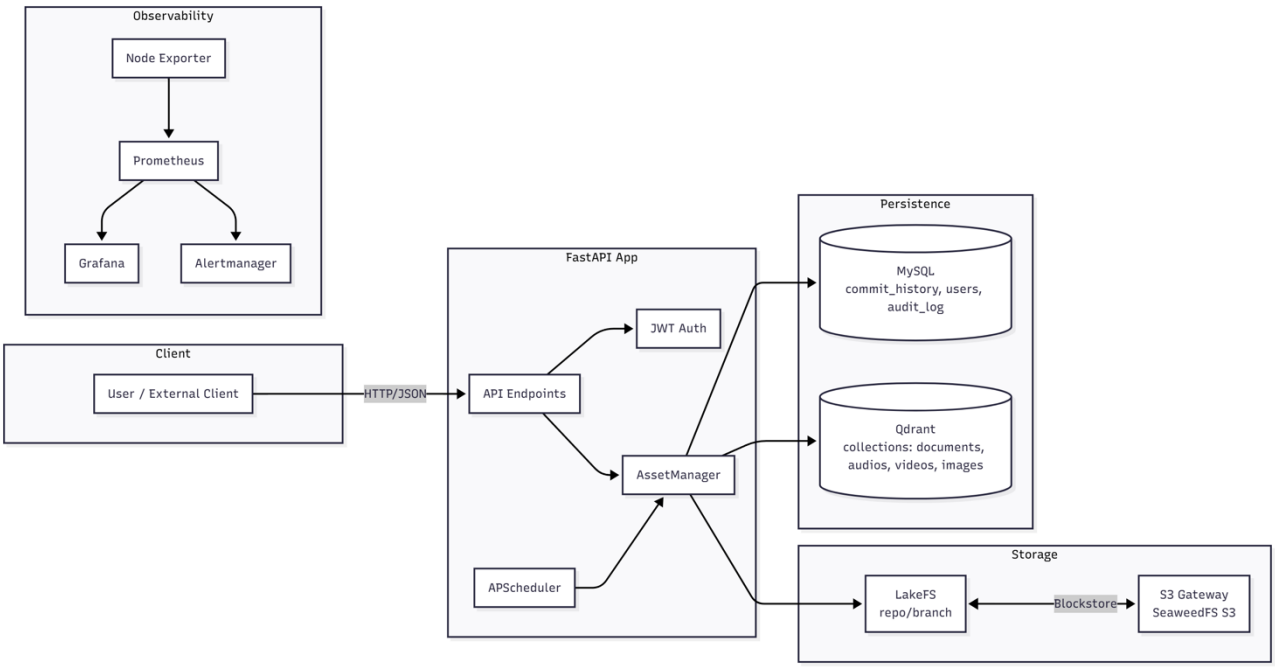
Key Design Principles:

- **Scalability**: Offers highly available, replicated object storage
- **Reliability**: Immutable versioning and consistent metadata ensure recoverability.
- **Observability**: Comprehensive metrics and monitoring provide system visibility.
- **Integration**: Seamless updates to Qdrant for AI-driven workflows.

Components:

- **FastAPI Application**: Exposes RESTful APIs for asset operations.
- **MySQL**: Stores structured metadata and enforces RBAC.
- **lakeFS**: Manages immutable versioning and metadata.
- **SeaweedFS**: Provides replicated object storage with S3 compatibility.
- **APScheduler**: Automates lifecycle tasks based on TTL policies.

4.2 System Flow Diagram

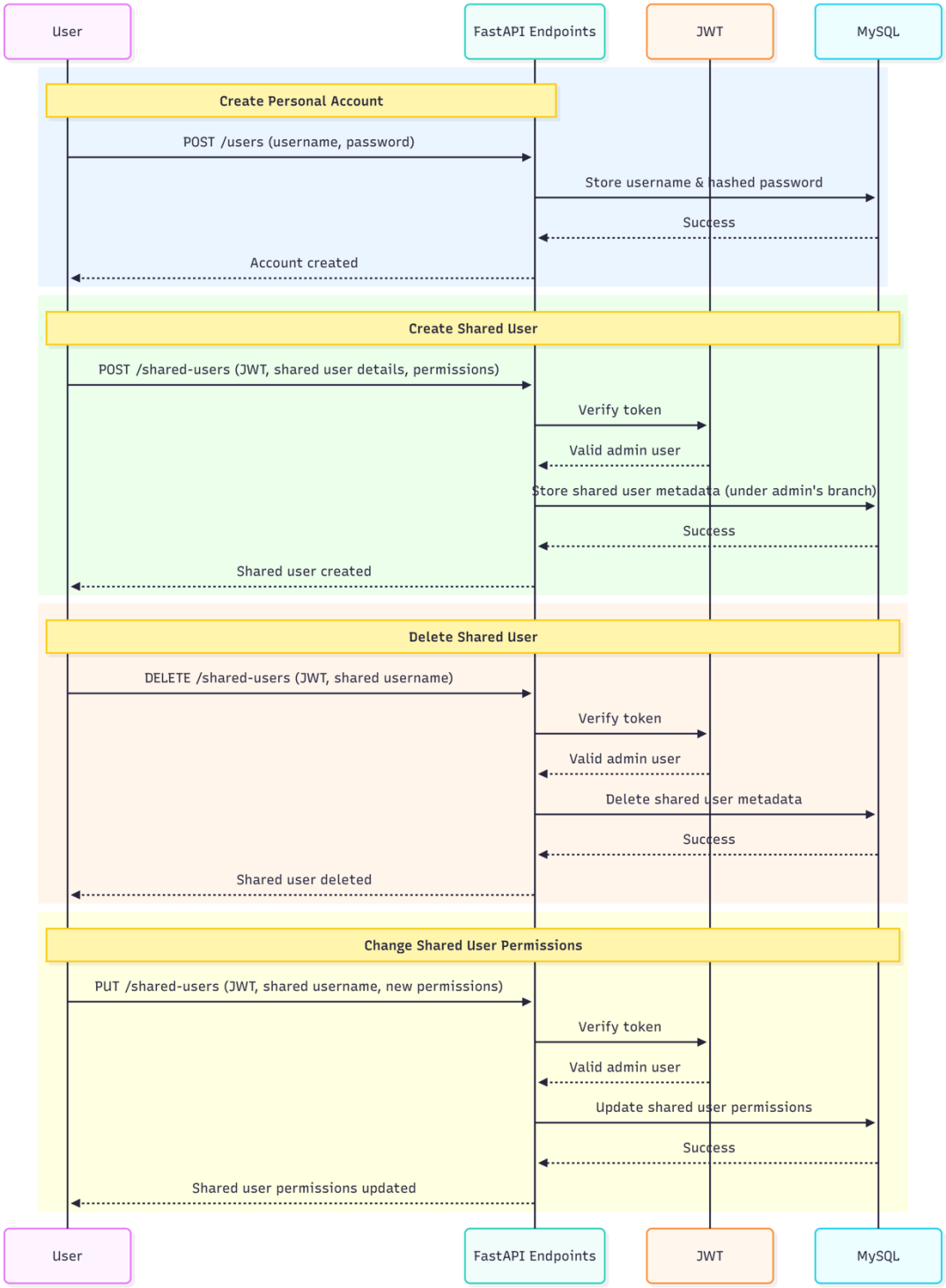


The service follows a modular flow for asset management:

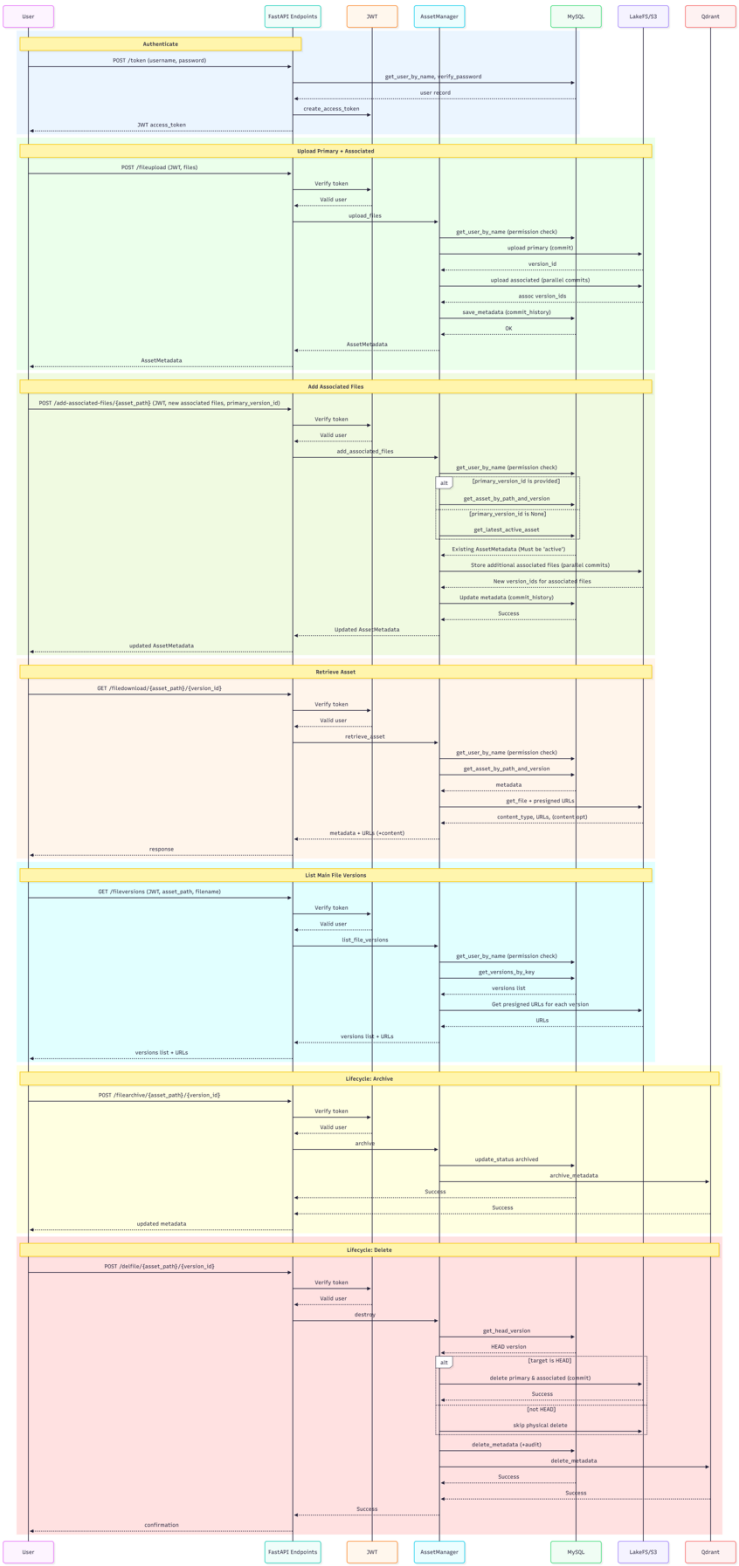
1. **Client Interaction**: Users and external systems interact via HTTP/JSON requests, authenticated with JWT and RBAC.
2. **AssetManager Core**: Handles upload, download, versioning, and lifecycle operations, ensuring metadata consistency.
3. **Persistence Layer**: MySQL stores metadata (commit_history, users, audit_log); Qdrant updates reflect lifecycle changes.
4. **Storage Layer**: SeaweedFS stores assets; lakeFS manages versions and generates presigned URLs for secure access.
5. **Observability Layer**: Prometheus and Grafana provide metrics and visualization; Alertmanager handles alerts.

4.3 Data Flow Diagram

- User Management API Data Flow



- **Asset Management API Data Flow**



- **User Management API**:
  - Support user creation (POST /users)
  - Support Shared user management (POST/DELETE/PUT /shared-users)
  - Authentication (POST /token)
- **Asset Management API**:
  - Uploading (POST /fileupload)
  - Appending additional associated files to an existing asset (POST /add-associated-files)
  - Downloading (GET /filedownload)
  - Version listing (GET /fileversions)
  - Archival (POST /filearchive)
  - Deletion (POST /delfile)

Except for the user registration (POST /users) and authentication (POST /token) endpoints, all API operations require a valid JWT token and are protected by role-based access control (RBAC). Metadata is stored in MySQL, assets in SeaweedFS/lakeFS, and Qdrant is updated for lifecycle consistency.

5. Document Processing Service

The Document Processing Service is designed to handle documents of various formats and transform them into structured, chunked representations with hierarchical summaries. Each document is processed to extract text, tables, and images, which are then tokenized, summarized at multiple levels, and output as JSON records suitable for embedding or downstream semantic search tasks.

5.1 System Design

The Document Processing Service supports multiple document formats (PDF, PPT, DOC, TXT, CSV, XLSX) and generates JSON arrays with two record types: final document summaries (`embedding_type=summary`) and per-chunk text records (`embedding_type=text`). It leverages specialized processors for each format, ensuring robust parsing, normalization, and summarization.

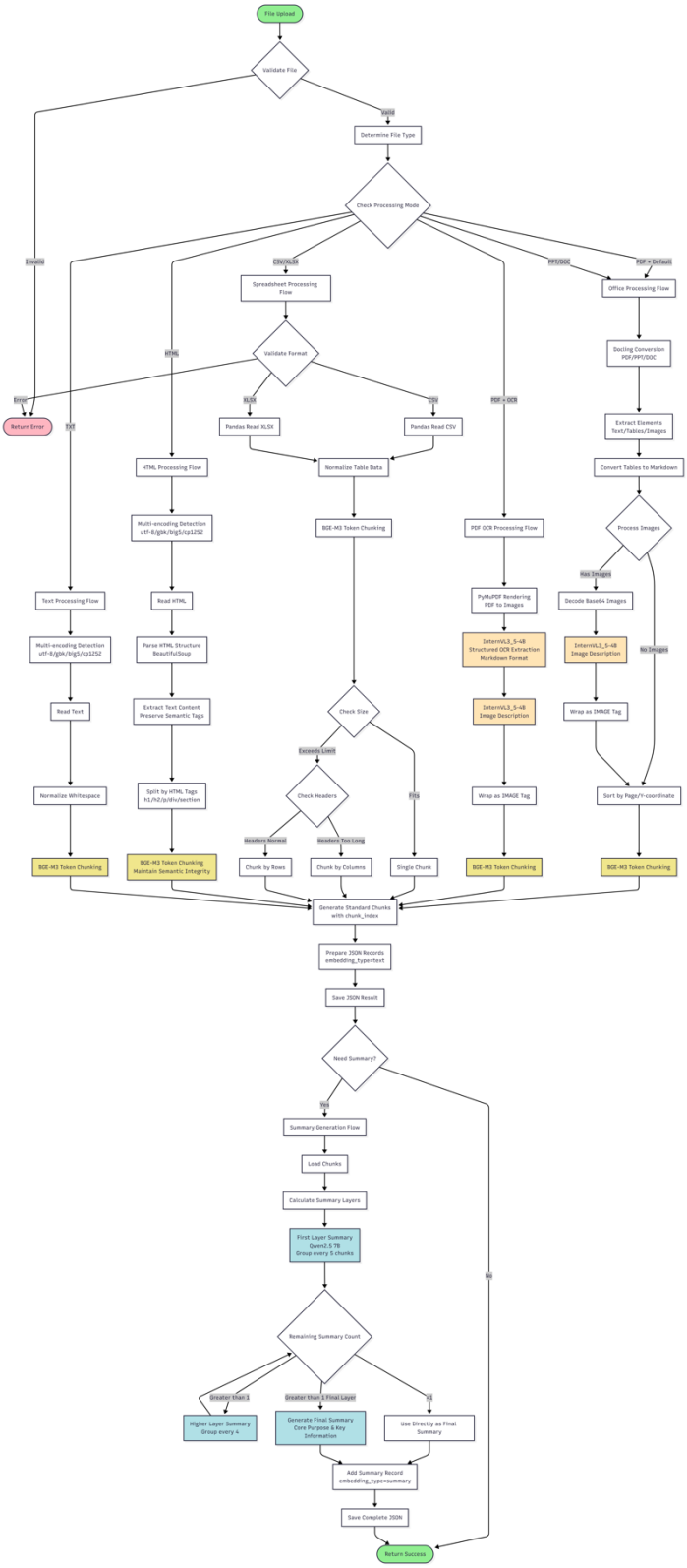
Key Design Principles:

- **Versatility**: Handles diverse input formats with consistent output.
- **Precision**: Preserves document structure and reading order.
- **Scalability**: Processes large documents efficiently with token-based chunking.
- **Integration**: Outputs JSON records optimized for Qdrant and semantic search.

Components:

- **OfficeDocumentProcessor**: Uses Docling to process PDFs, PPTs, and DOCs, extracting text, tables, and images.
- **PDFOCRProcessor**: Employs InternVL3_5-4B for OCR and text extraction from scanned or image-heavy PDFs.
- **TxtProcessor**: Supports multiple text encodings (UTF-8, GBK, etc.).
- **CSVXLSXProcessor**: Uses Pandas for structured data extraction.
- **Summarization Engine**: Qwen2.5 7B generates hierarchical summaries.

5.2 System Flow Diagram



The Document Processing Service follows a modular pipeline with distinct stages:

1. Parsing and Extraction
    - **OfficeDocumentProcessor**: Converts PDFs, PPTs, and DOCs into structured data. Text blocks and page numbers are collected; tables are reconstructed in Markdown-like format; images are decoded and described via InternVL3_5-4B, wrapped as <IMAGE> blocks.
    - **PDFOCRProcessor**: Renders each PDF page as an image using PyMuPDF, extracts visible text with structured prompts via InternVL3_5-4B, and produces brief image descriptions appended as <IMAGE> blocks.
    - **TxtProcessor**: Reads text from multiple encodings (utf-8/utf-8-sig/gbk/big5/cp1252) and normalizes whitespace.
2. Normalization
    - Converts extracted elements into a **unified internal representation**.
    - Preserves the reading order by page number and approximate Y-position, ensuring chunks follow the natural sequence of the document.
3. Token-based Chunking (using BAAI/bge-m3)
    - Each chunk contains text, element types, page numbers (if any), and a deterministic chunk_index.
4. Hierarchical Summarization
    - **First-level summaries**: Chunks are grouped (default group size: 5) and summarized individually.
    - **Higher-level summaries**: First-level summaries are grouped by 4 for subsequent layers.
    - **Final summary**: If multiple summaries remain, the model generates a final summary highlighting the document's core purpose and key information; if only one summary remains, it is used directly.

6. Audio Processing Service

The Audio Processing Service, part of CIE’ s Layer 4: AI Engine Layer, enables robust processing of audio assets, including speech recognition, speaker diarization, and audio event classification. Integrated with the broader CIE ecosystem, it leverages FastAPI for API exposure, WhisperX for transcription and diarization, and PANNs for classification, ensuring scalable and reliable audio analysis for semantic search and downstream tasks.

6.1 System Design

The Audio Processing Service is designed for modularity, scalability, and integration with CIE’ s orchestration and storage layers. It processes audio files (and audio extracted from videos) to produce structured JSON outputs, which are stored temporarily and forwarded to Qdrant for vector search. The system operates within a containerized environment using Docker Compose, with GPU support for compute-intensive tasks. It integrates with the Kafka Service for asynchronous processing and the Redis Cache System for performance optimization.

Key Design Principles:

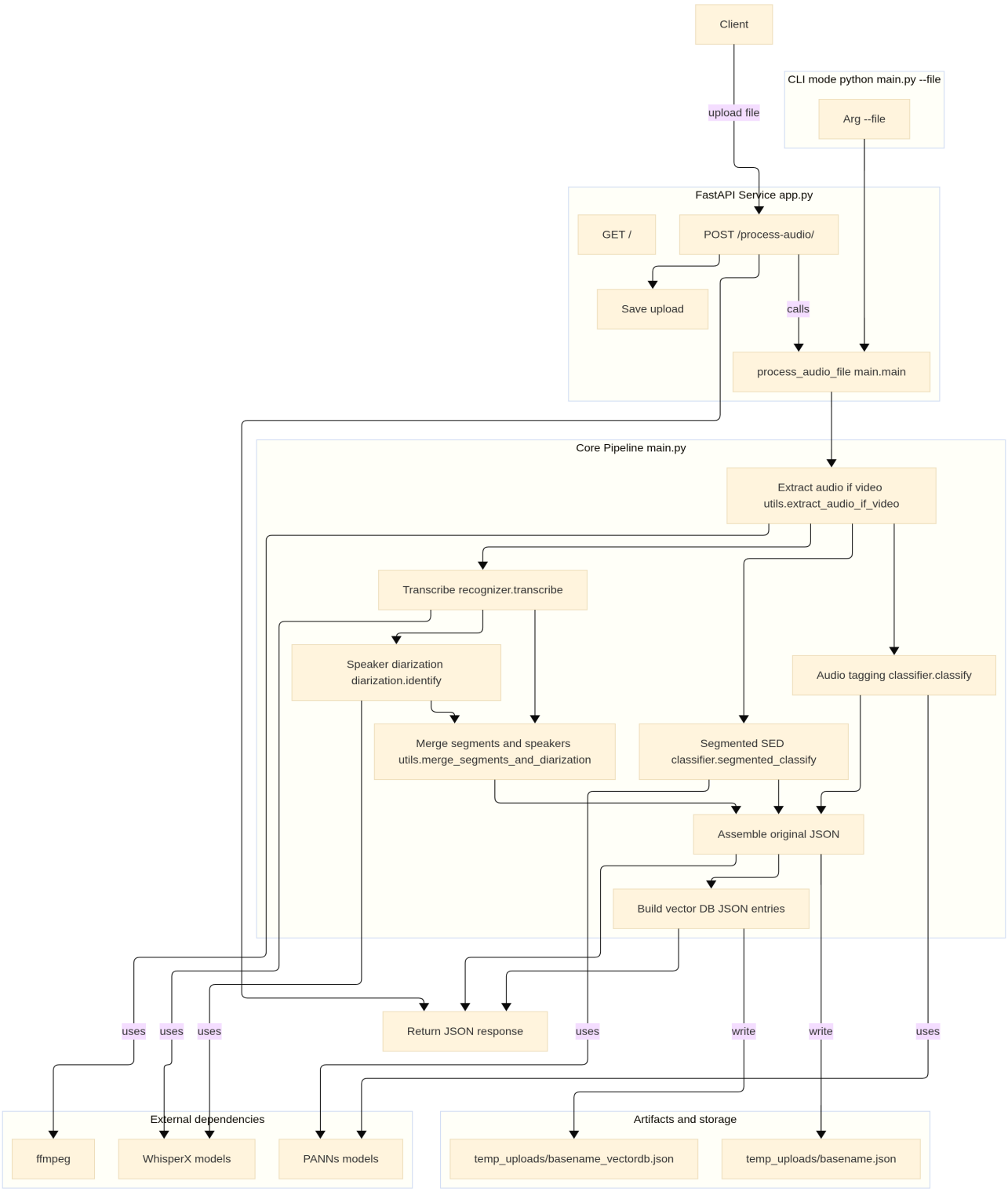
- **Modularity**: Independent modules for recognition, diarization, and classification enable flexible pipeline configuration.
- **Scalability**: Containerized deployment with GPU support ensures efficient processing of large audio datasets.
- **Reliability**: Temporary storage and Kafka integration ensure fault-tolerant processing.
- **Integration**: Structured outputs align with CIE’s unified data format for seamless Qdrant storage and semantic search.

Components:

- **API Service**: FastAPI application (AudioProcess/app.py) exposes health and processing endpoints, running under Uvicorn.
- **Pipeline Orchestrator**: AudioProcess/main.py coordinates end-to-end processing, from input handling to output generation.
- **Speech Recognition**: AudioProcess/recognizer.py uses WhisperX for transcription and language detection.
- **Speaker Diarization**: AudioProcess/diarization.py leverages WhisperX for identifying speakers and their speaking intervals.

- **Audio Classification**: AudioProcess/classifier.py employs PANNs AudioTagging and SoundEventDetection for labeling audio events (overall and segmented).
- **Utilities**: AudioProcess/utils.py handles audio extraction from videos using ffmpeg and merges transcripts with diarization data.
- **Artifacts/Storage**: JSON outputs are written to AudioProcess/temp_uploads/ for temporary storage before Qdrant persistence.
- **Deployment**: AudioProcess/docker-compose.yaml and AudioProcess/dockerfile containerize the service with GPU support and mounted volumes.
- **Examples / format**: AudioProcess/format.json shows downstream payload shapes for vector search systems.

6.2 System Flow Diagram



7. Video Processing Service

The Video Processing Service, part of CIE' s Layer 4: AI Engine Layer, transforms raw video content into actionable insights through advanced AI-driven analysis. It integrates audio extraction, visual-language model (VLM) processing, and multimodal summarization to deliver structured, timeline-aligned outputs optimized for semantic search and downstream applications. Built with a cloud-native microservices architecture, it ensures scalability, fault tolerance, and seamless integration with enterprise systems.

7.1 System Design

The Video Processing System implements a comprehensive pipeline that integrates audio extraction, Vision-Language Model (VLM) analysis, and multimodal summarization to deliver structured video insights. It leverages InternVL3.5-8B for visual understanding, FFmpeg for audio extraction, and sophisticated prompt engineering for coherent summary generation.

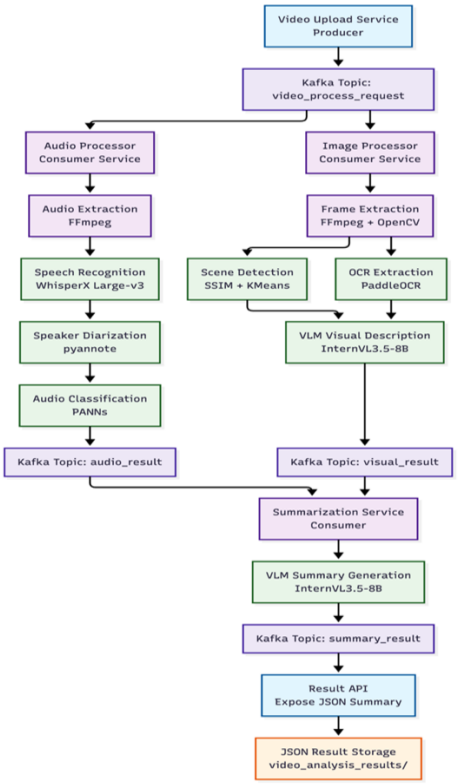
Key Design Principles:

- **Multimodal Integration**: Seamlessly combines visual, textual (OCR), and audio analysis with timeline synchronization.
- **Intelligent Sampling**: Uses scene-based and uniform frame sampling strategies for optimal analysis efficiency.
- **Fault Tolerance**: Implements fallback mechanisms to ensure pipeline continuity under failure conditions.
- **Scalability**: GPU-accelerated processing with dynamic resource allocation for high-throughput video workloads.

Components:

- **Main Processing Workflow**: Orchestrates audio extraction, VLM-based visual description, and multimodal summary generation.
- **Audio Extraction Module**: Uses FFmpeg to extract 16kHz mono WAV audio from video files.
- **Visual Analysis Engine**: InternVL3.5-8B VLM processes video frames for event detection and description.
- **Multimodal Summarizer**: Integrates VLM outputs, OCR text, and audio transcripts into cohesive summaries.

7.2 System Flow Diagram



The Video Processing System follows a robust, fault-tolerant pipeline:

1. **Audio Extraction**: FFmpeg extracts audio from video files, converting to standardized WAV format with comprehensive error handling and 5-minute timeout protection.
2. **Visual Description Generation**: VLM analyzes video frames using intelligent sampling strategies—either uniform sampling across 32 segments or targeted scene-change frame analysis (max 16 key frames).
3. **Multimodal Integration**: Combines VLM visual descriptions, scene OCR text, and audio transcripts (speaker segments with timestamps) into structured inputs.
4. **Summary Generation**: Advanced LLM processes multimodal data to produce detailed, narrative-style Chinese summaries preserving chronological relationships and key events.
5. **Output Aggregation**: Produces comprehensive JSON results including success status, visual descriptions, final summaries, and processing metadata.

Data Flow:

- **Input Processing**: Video files are received via API Gateway, validated, and routed to Kafka's video-processing-requests topic.
- **Pipeline Execution**: Orchestrator coordinates extraction, VLM analysis, and summarization, storing intermediate WAV files and scene data locally.
- **Multimodal Fusion**: Visual descriptions, OCR timestamps, and audio segments are synchronized by timeline and fed into the summarization engine.
- **Output Storage**: Structured JSON payloads containing summaries, transcripts, timestamps, and metadata, designed for seamless integration with downstream systems and semantic indexing.

8. Vector Search Engine

The Vector Search Engine, part of CIE’ s Layer 4: AI Engine Layer, enables semantic search across videos, audios, documents, and images using Qdrant (localhost:6333). It employs BAAI/bge-m3 for 1024-dimensional embeddings, with cosine similarity for matching, and integrates with the Redis Cache System for performance optimization.

8.1 System Design

The Vector Search Engine uses a unified data format with UUID-identified records, storing embeddings and metadata (e.g., text/summary, status, asset_path) in Qdrant collections (videos, audios, documents, images). FastAPI services (ports 8811 – 8814 local, 8821 – 8824 Docker) expose search and insertion APIs, with payload indexes accelerating filtered searches. Only active (non-archived) content is returned by default.

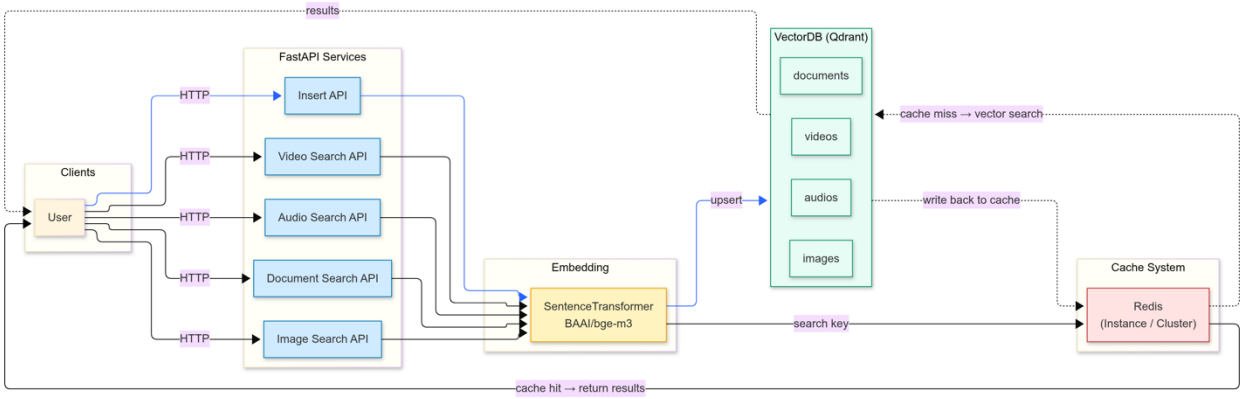
Key Design Principles:

- **Consistency**: Unified data format across media types ensures predictable search behavior.
- **Performance**: Redis caching and Qdrant indexing optimize query latency.
- **Flexibility**: Customizable payload fields accommodate media-specific metadata.

Components:

- **FastAPI Services**: Expose search and insertion APIs for each media type.
- **Qdrant**: Stores embeddings and metadata in media-specific collections.
- **BAAI/bge-m3**: Generates 1024-dimensional embeddings for semantic search.
- **Redis Cache System**: Accelerates query performance with cached results.

8.2 System Flow Diagram



The Vector Search Engine follows a streamlined flow for search and insertion:

1. **Client Interaction**: HTTP requests via FastAPI for search or data insertion.
2. **Caching Check**: Redis Cache System returns cached results for matching queries.
3. **Embedding Generation**: BAAI/bge-m3 generates embeddings for queries or new data.
4. **Search/Insertion**: Qdrant performs vector searches or stores new embeddings and metadata.
5. **Result Caching**: Search results are cached in Redis for future queries.

Data Flow:

- **Search**: Incoming queries are first embedded using the BAAI/bge-m3 model and checked against the Redis cache to reduce redundant vector computations. If a cache miss occurs, the system performs a cosine similarity search in Qdrant to retrieve the most relevant results. The resulting matches are then written back to Redis for subsequent queries, improving overall search latency and efficiency.
- **Insertion**: When new data is ingested, textual content is converted into 1024-dimensional embeddings. The embeddings, along with associated metadata and information (e.g., asset_path, version_id, and embedding_type), are stored in Qdrant. Each record is automatically indexed to enable efficient similarity-based retrieval in subsequent search operations.

9. Model Lifecycle Management

9.1 End-to-End ML Operations

CIE provides a cohesive model lifecycle built around versioned storage (LakeFS), model registry/metadata (MLflow + Postgres), and a unified inference stack:

Model repository and registry:

- Versioned artifacts via LakeFS (immutable commits for models and datasets)
- MLflow for model registration, metadata, and tags; Postgres as MLflow backend store
- Resource metadata generation (e.g., VRAM/latency estimates) via ModelResourceEstimator, stored as MLflow tags
- Clear separation of temp vs persistent storage: /tmp/models, /tmp/datasets for staging; /data/models, /data/datasets for durable use

Datasets and artifacts flow:

- **DatasetManager** and **ModelManager** handle uploads/commits to **LakeFS**
- Optional source sync from HuggingFace Hub (download → commit → stage)
- Model versions registered in **MLflow**; metadata available to upstream services

Training Pipeline:

- Standardized training scripts with best practices
- Distributed training support via DeepSpeed for large models
- Automated hyperparameter tuning and experiment tracking

Unified inference serving:

- Single FastAPI endpoint: **/inference/infer** covering tasks like text-generation, VLM, ASR, OCR, audio-classification, video-analysis, document-analysis
- **InferenceManager** orchestrates execution; **TaskRouter** selects the appropriate engine (**TransformersEngine** or **OllamaEngine**) based on task/engine inputs
- **ModelExecutor** coordinates model load, preprocess/infer/postprocess via task-specific handlers
- **ModelCache** provides LRU caching with GPU-aware cleanup; GPUManager can clear cache and report device status

- **OllamaEngine** proxies to an external **Ollama** service for LLMs; **TransformersEngine** loads from local /data/models for HF pipelines
- **ModelResourceEstimator** supports capacity planning (estimated VRAM, latency, throughput) and model selection logic

9.2 Model Monitoring and Drift Detection

The platform exposes operational metrics and hooks to integrate with your monitoring stack; drift detection is supported through extensible patterns:

Operational monitoring:

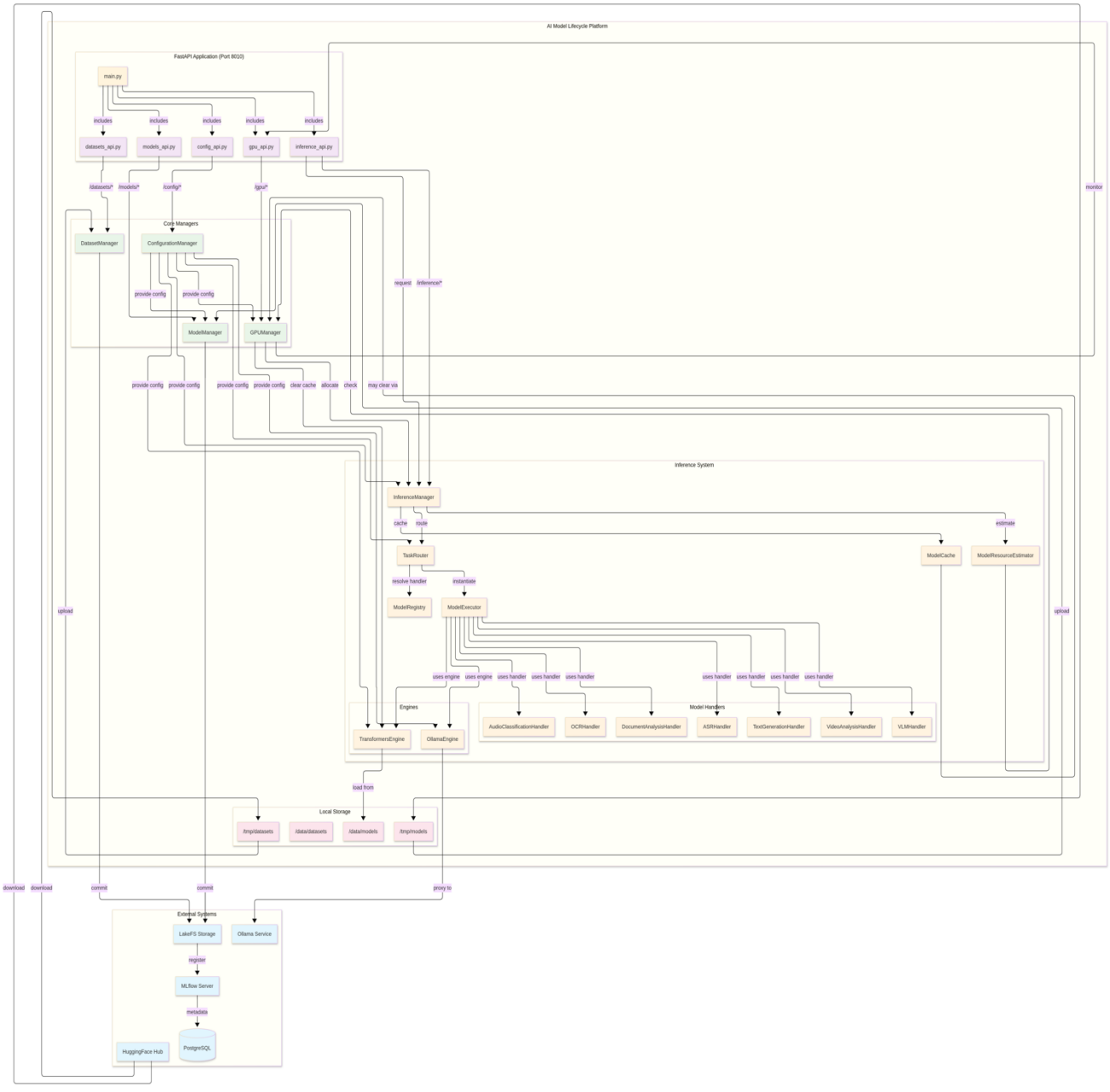
- Real-time service metrics available from inference responses and stats:
- processing_time, api_processing_time, success/failure counts
- cache hit/miss, evictions, loaded models (from ModelCache stats)
- health status for core components (router/registry/cache/model_manager/gpu_manager)
- GPUManager integration enables cache clears and device checks; suitable for GPU health dashboards
- Health and stats endpoints are scrape-friendly for Prometheus and can feed alerts

Drift and performance tracking:

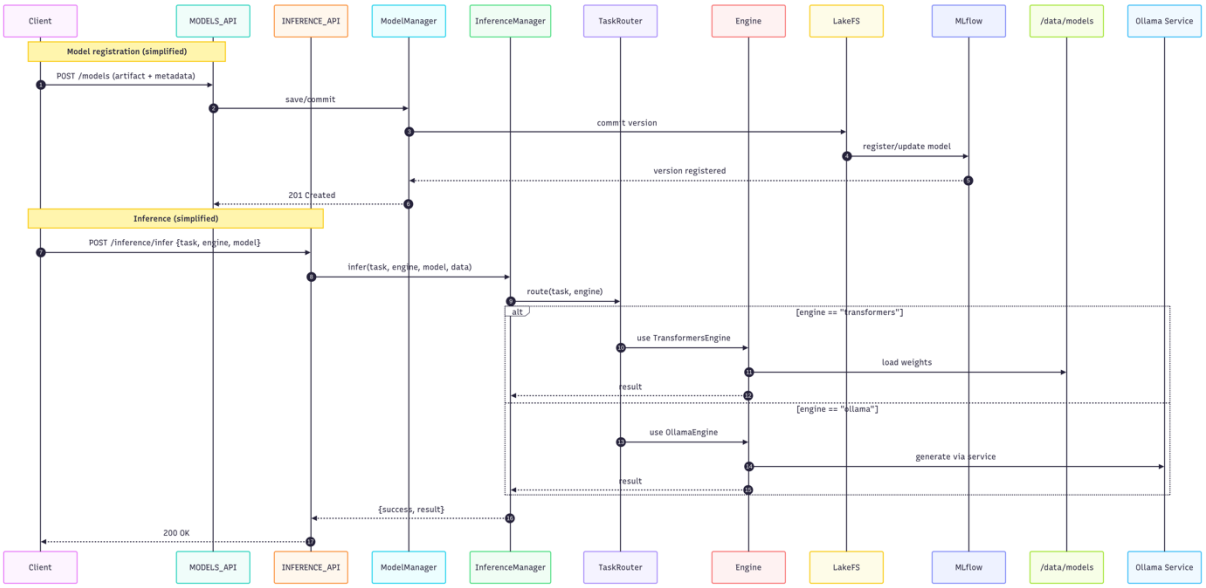
- Leverage **MLflow** to log and compare model metrics over time; store evaluation runs alongside registered versions
- Implement inference logging (inputs/features, predictions, confidence) to a data sink for offline drift checks
- Compare live distributions vs. training (or champion) baselines to detect covariate/label drift
- Close the loop by retraining on new data and registering updated versions in **LakeFS** + **MLflow**, then promote via your rollout strategy

9.3 Comprehensive System Flow & Sequence Diagram

- flowchart



- Sequence diagram



10. Kafka Service

The Kafka Service, within CIE’ s Layer 3: Orchestration Layer, provides an asynchronous, scalable framework for processing documents, images, audio, and video assets. It uses Kafka topics for decoupled microservices, with LakeFS/SeaweedFS for storage, Redis for state tracking, and Qdrant for retrieval, ensuring fault tolerance and security.

10.1 System Design

The service orchestrates asset processing through media-specific Kafka topics, with orchestrators and workers handling analysis, summarization, and vectorization. Access is secured via short-lived tokens, and intermediate artifacts are stored locally before final results are persisted in Qdrant.

Key Design Principles:

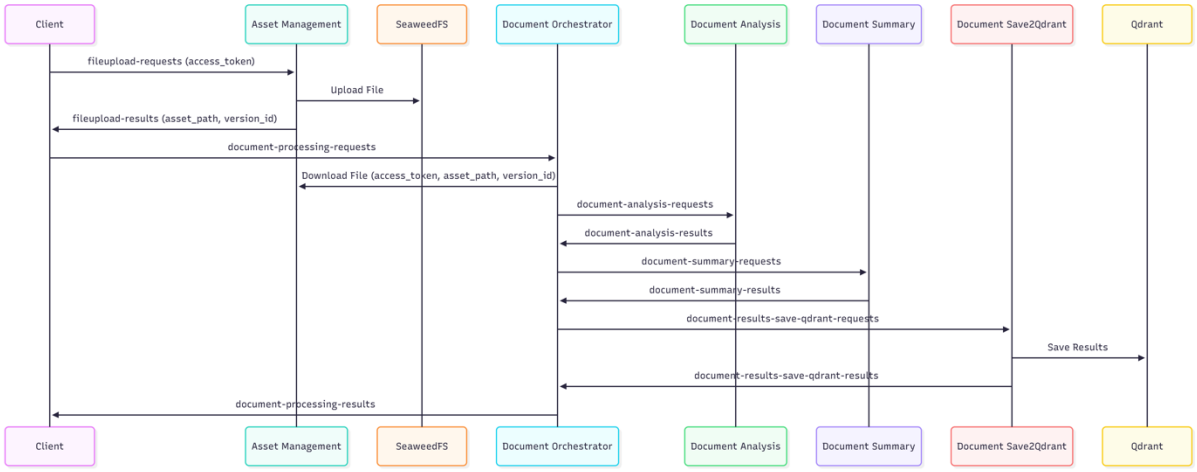
- **Decoupling**: Kafka topics enable independent microservice processing.
- **Reliability**: Redis tracks progress; request_id and step_id ensure traceability.
- **Security**: Short-lived tokens control access to storage.
- **Scalability**: Workers dynamically process tasks based on queue depth.

Components:

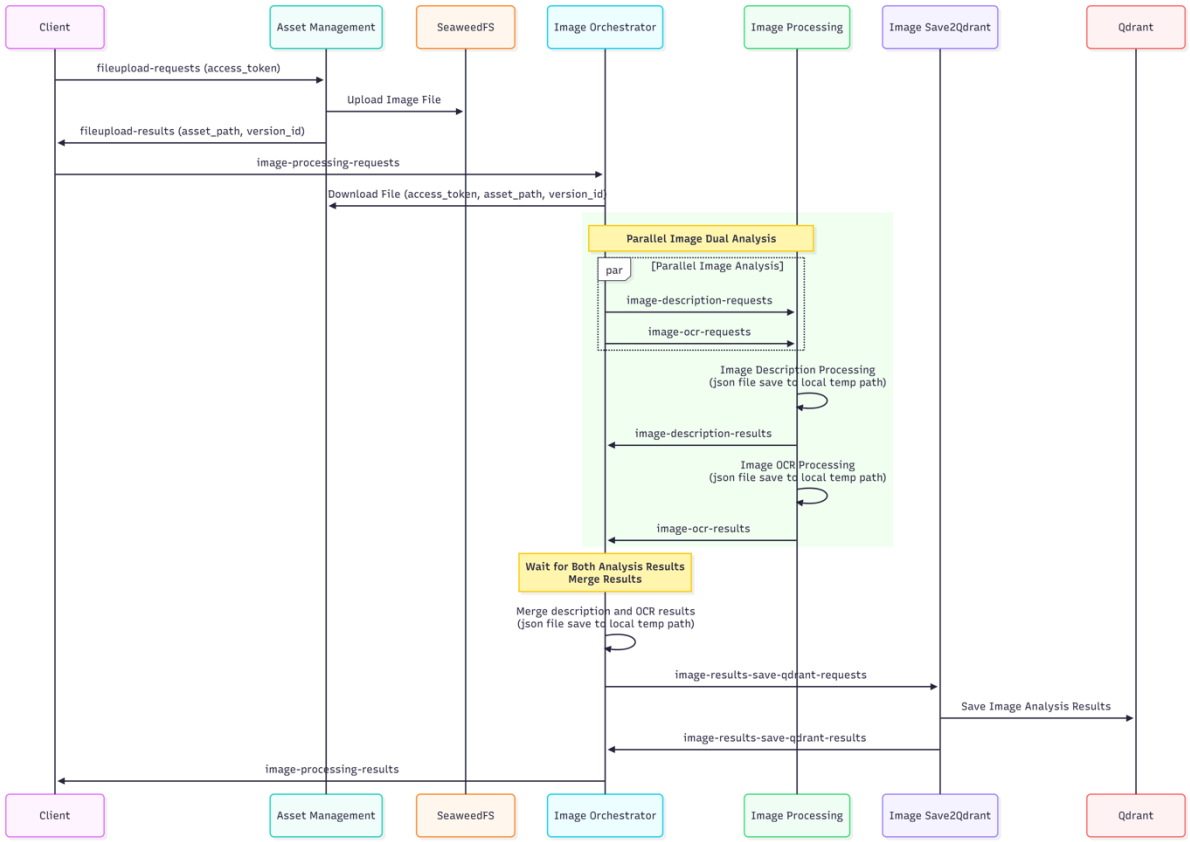
- **API Gateway/Upload Service**: Handles uploads, stores assets in LakeFS/SeaweedFS, and produces Kafka messages.
- **Orchestrators**: Coordinate media-specific pipelines (document_orchestrator_service, etc.).
- **Workers**: Perform tasks like analysis, OCR, summarization, and Qdrant storage.
- **Storage**: LakeFS/SeaweedFS for assets, local filesystem for intermediates, Qdrant for results, Redis for state.

10.2 Data Flow Diagram

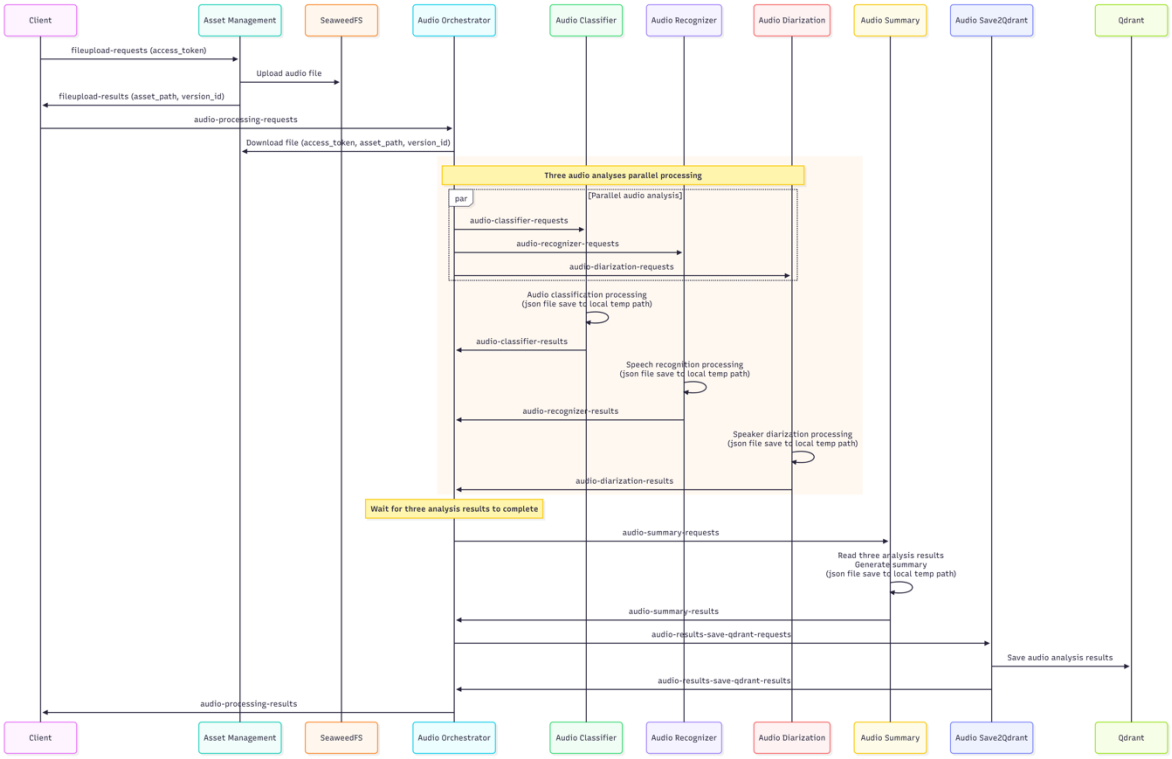
- Document Processing Pipeline



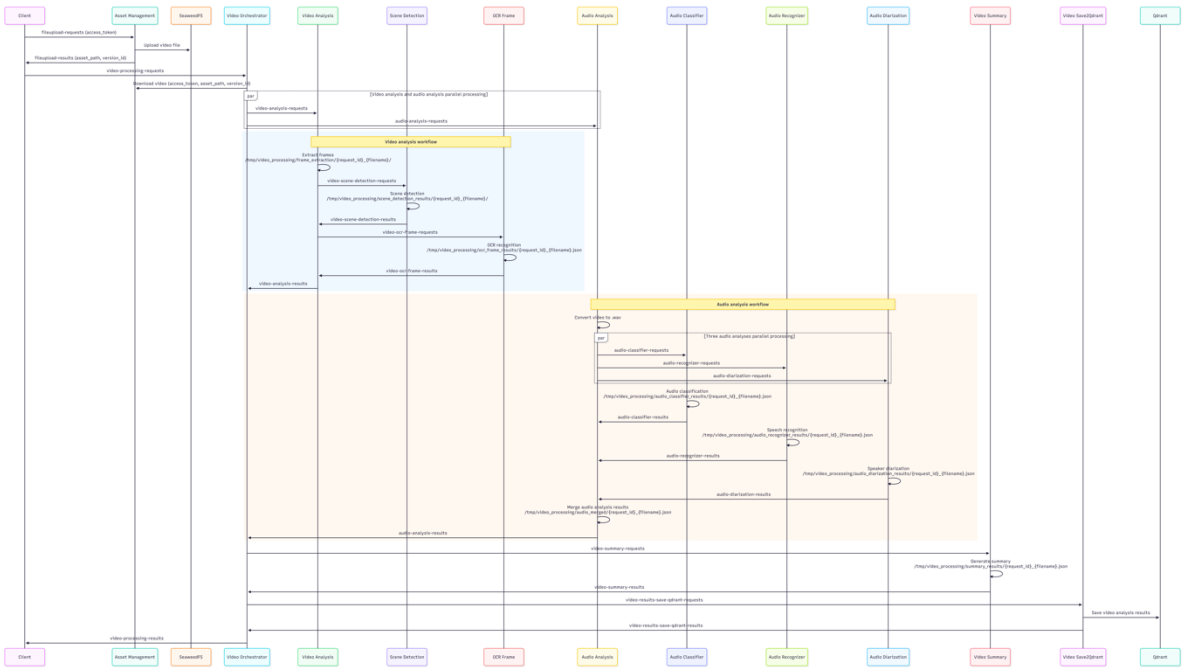
- Image Processing Pipeline



- Audio Processing Pipeline



- Video Processing Pipeline



The service follows a modular pipeline:

1. **Upload Handling**: API Gateway stores assets and sends requests to media-specific Kafka topics.sss
2. **Orchestration**: Orchestrators consume requests, coordinate workers, and track state in Redis.
3. **Worker Processing**: Workers perform media-specific tasks (e.g., scene detection, summarization), storing intermediates locally.
4. **Result Storage**: Processed data is vectorized and stored in Qdrant.

Data Flow:

- **Document Pipeline**: Uploads trigger document-processing-requests, with workers handling analysis, summarization, and Qdrant storage.
- **Image Pipeline**: Processes images via image-processing-requests, generating embeddings for Qdrant.
- **Audio Pipeline**: Handles audio tasks (recognition, diarization) via audio-processing-requests.
- **Video Pipeline**: Processes videos (frame extraction, OCR, summarization) via video-processing-requests, fanning out audio tasks to audio workers.