Multi-Tenant Async File Management & Extraction System — Technical Specification

1. Project Overview

Name: Multi-Tenant Async File Management & Extraction System

Purpose: A fully **asynchronous, multi-tenant, containerized microservices system** for file management and structured data extraction. Tenants provide their own **immutable codes**. File storage is organized per client and month, with **Redis caching** for shared operations, fully tested and production-ready.

2. Goals & Objectives

- Multi-tenant support with **immutable tenant codes**
- File operations: **upload, download, delete, list, update metadata**
- Extraction service for **structured data from unstructured files**
- **Async backend** for all operations
- **Redis caching** for repeated queries and shared data
- **Microservices-oriented architecture**, each service with its own port
- Containerized and deployable in **Kubernetes**
- Full **unit, integration, and async testing** for every component

3. Functional Requirements

3.1 Tenant Management

Feature	Description
Create Tenant	Tenant provides a **unique code** at creation (cannot be changed)
Get Tenant Info	Retrieve configuration, file stats, metadata
Delete Tenant	Deletes tenant and **all associated files** from storage + DB

3.2 File Management

Feature	Description
Upload File	Async, single file per request, validates size, extension, media type, nested zip depth
Download File	Async, retrieve by file ID
Delete File	Async, removes from storage + DB
List Files	Async, list files per tenant with metadata
Update Metadata	Async JSONB metadata updates

3.3 Extraction Service

Feature	Description
Extract Structured Data	Async parsing of files (PDF, DOCX, TXT, images)
Save Extraction Results	Store in PostgreSQL; optionally cache in Redis

Feature	Description

4. Non-Functional Requirements

- **Immutable tenant codes**
- **Storage layout:** `storage_base_path/<tenant_code>/YYYY-MM/`
- **Redis caching** for shared metadata, repeated lookups
- Fully **async operations** (FastAPI + async DB)
- **Microservices architecture:** each service runs independently with its own port
- **Containerization:** Docker + Kubernetes deployment ready
- **Testing:** unit + integration + async; deterministic, covers all edge cases

5. Technical Stack

Layer	Technology
Backend	Python 3, FastAPI (async)
ORM	SQLAlchemy (async)
Validation	Pydantic
Database	PostgreSQL (JSONB for metadata/config)
Cache	Redis
Containerization	Docker, Kubernetes
Messaging	Optional: Kafka / RabbitMQ for async events
Testing	Pytest, Async TestClient, coverage

6. Database Schema

Tenant

Column	Type	Description
_id	UUID	Auto-generated primary key
code	String	Tenant-provided, unique, immutable
configuration	JSONB	Allowed/denied extensions, max size, media types

File

Column	Type	Description
id	String	Globally unique file ID (`fs_<random>`)
tenant_id	UUID	FK to Tenant, cascade on delete
file_name	String	Original file name
file_size_bytes	Integer	Size in bytes
media_type	String	MIME type

Column	Type	Description
time_created	Timestamp w/ timezone	Creation time in UTC
tags	JSON	Optional tags
metadata	JSONB	Optional metadata
Storage Path	Path	`storage_base_path/<tenant_code>/YYYY-MM/`

---

7. Microservices Design

Service	Port	Responsibilities
File Service	8001	Upload/download/delete/list files, metadata management, storage
Extraction Service	8002	Extract structured data, save results, caching
API Gateway	8000 (optional)	Routing requests to services, rate limiting
Redis Cache	N/A	Shared caching for repeated queries, extraction results

Principles:

- Each service is **fully async**
- Proper **error handling & logging**
- Containerized individually
- Each service **runs independently**, can scale horizontally
- Optional async messaging for inter-service events (Kafka/RabbitMQ)

---

8. Storage & Directory Layout

```
storage_base_path/
├── <tenant_code>/
│   └── YYYY-MM/
│       └── fs_<random>.ext
```

- Tenant folder per month
- Deleting a tenant removes all files + folders
- Deleting a file removes storage + DB metadata

---

9. Redis Caching

- Store **shared metadata**
- Cache **extraction results**
- Implement **TTL for stale data**
- Async-safe operations

---

10. Testing Strategy

Type	Scope
Unit Tests	Models, utility functions, validation logic
Integration Tests	Services + DB + Redis
Async Tests	All endpoints called concurrently, check for race conditions

Type	Scope
Edge Cases	File size violations, extension violations, nested zip depth, tenant deletion cascade
Deterministic	Ensure reproducible results, no flakiness
Coverage	100% for critical components (tenant, file, extraction, caching)

---

11. Folder Structure

```
src/
├── file_service/
│   ├── models.py
│   ├── schemas.py
│   ├── crud.py
│   ├── routes.py
│   └── services.py
├── extraction_service/
│   ├── models.py
│   ├── schemas.py
│   ├── crud.py
│   ├── routes.py
│   └── services.py
├── shared/
│   ├── db.py
│   ├── redis_client.py
│   └── utils.py
├── tests/
│   ├── unit/
│   ├── integration/
│   └── async/
├── Dockerfile
└── k8s/
    ├── file_service.yaml
    ├── extraction_service.yaml
    └── redis.yaml
```

---

12. Deployment Guidelines

- Each service has its **own Docker container**
- Kubernetes manifests per service
- Redis deployed as a separate pod
- Use **Persistent Volume Claims (PVC)** for file storage
- Optional: use **Helm charts** for scaling and configuration
- All services must support **graceful shutdown and async cleanup**