Zaalima Development - Internal Project Documentation: Q4 Python Elite

To: Engineering Team Alpha

From: Lead Instructor & Senior Architect

Date: November 30, 2025

Subject: Q4 Production Grade Project Assignments - Python Elite Track-----Team Alpha, welcome to the final and most critical sprint of the development year. You have been carefully selected for the exclusive **"Python Elite" track**, signifying a move beyond foundational concepts and the self-limiting cycle of "tutorial hell." This quarter's focus is the implementation of **production-grade software**. This is not an academic exercise; the four projects detailed in this document are designed to simulate real-world demands, requiring systems that are robust enough to handle the critical "four pillars" of modern software engineering: **Concurrency, Security, Failure Recovery, and Scale.**

We are strictly adhering to a demanding **4-week delivery cycle per project**. The expectation is the delivery of industry-standard, clean, and deployable code, which mandates rigorous adherence to best practices, including comprehensive **linting, detailed documentation (docstrings and architectural diagrams), and 100% unit test coverage** (as close to 100% as practical, but aiming for >90% coverage).-----**Contents - Index**

-----1. Tech Stack - Deep Explanation: The Python-Centric Philosophy

We will be deploying a **Python-Centric Heavy Backend** stack. The selection of these tools is based on their proven ability to handle high-performance, high-traffic production environments, offering a blend of rapid development speed and raw execution performance.

Component Category	Primary Tool/Framework	Justification and Use Case
Core API/Microservices	**FastAPI**	Selected specifically for high-concurrency, asynchronous (ASGI) APIs where latency is the primary concern, such as in the time-critical Fraud Detection service.
Core Framework (Monolith)	**Django**	Utilized for its "batteries-included" capabilities, robust built-in features (like the Admin panel), and mature ORM, making it ideal for rapid, secure development of complex, data-heavy applications like the Healthcare SaaS.
Lightweight Utility	**Flask**	Reserved for highly specialized, lightweight microservices or back-end utility services where maximum flexibility and minimal overhead are necessary.
Relational Data Layer	**PostgreSQL**	The industry standard for transactional integrity, ensuring full ACID compliance for critical data like financial ledgers (Project 1) and patient records (Project 2).
NoSQL Data Layer	**MongoDB**	Employed for flexible, unstructured data needs, including large-scale log aggregation, product catalogs with varying attributes, or metadata storage.

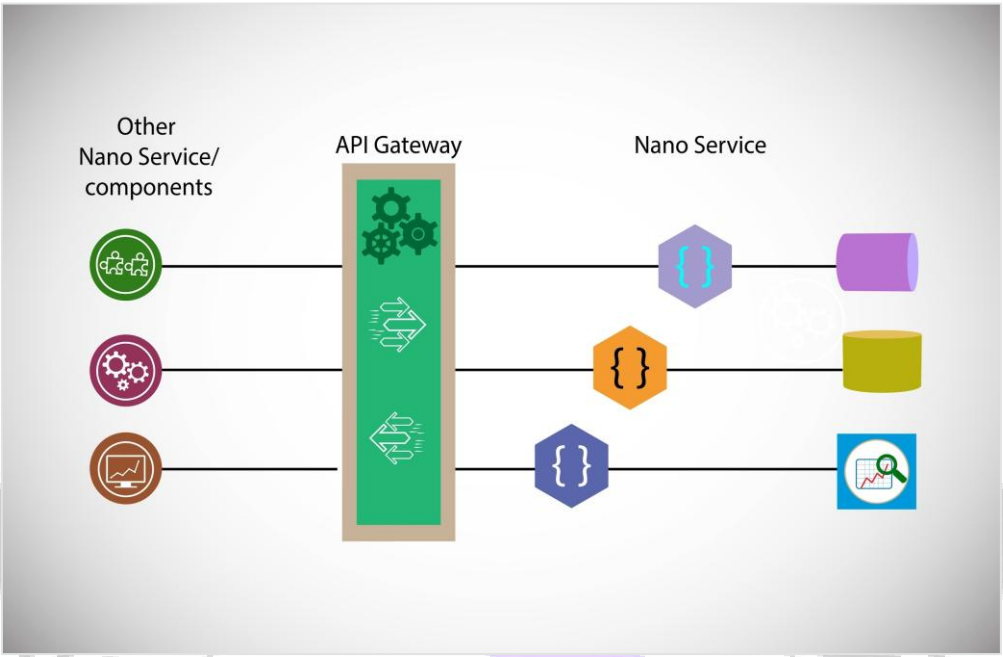
In-Memory Caching/Broker	**Redis**	Essential for caching "hot data," managing session states, and implementing fast features like rate-limiting, significantly reducing the load on the primary databases.
Asynchronous Task Queues	**Celery + RabbitMQ/Redis**	Critical infrastructure for offloading heavy, non-blocking tasks—such as image processing, machine learning model training, bulk email sending, or generating reports—thereby ensuring the core API remains responsive.
Containerization	**Docker**	Ensures environment consistency across development, testing, and production, eliminating the classic "it works on my machine" failure point.
ASGI/WSGI Server	**Gunicorn/Uvicorn**	Production-grade servers required to reliably serve the Python applications, with Uvicorn specifically for asynchronous FastAPI applications.
Edge/Proxy	**Nginx**	Acts as a reverse proxy, handling tasks like SSL termination, static file serving, compression, and sophisticated load balancing.

-----2. Project 1: FinTech - Real-Time Fraud Detection Engine

Project Title: High-Throughput Transaction Guard
Product Brand Name: "SentinelStream"

Use Case (Production): A large-scale Neo-bank requires a sophisticated system capable of processing tens of thousands of credit card transactions per second. SentinelStream must intercede between the external payment gateway and the bank's internal ledger, analyzing and

flagging suspicious transactions in **under 200ms** using a layered approach of a static rule-based engine and a predictive, pre-trained Machine Learning model.



Feature Type	Detail	Implementation Focus
Basic Functions	User Transaction History API, Current Balance Check, Secure Transaction Logging.	Standard REST API design and secure data storage.
Deep (Production) - Concurrency	**Idempotency Keys:** Mechanism to prevent double-charging or duplicate entries if the client API request times out and is retried.	Middleware implementation to check and log unique keys.
Deep (Production) - Failure Recovery	**Asynchronous Webhooks:** Non-blocking notification system to inform the merchant or third-party service of transaction status (Approved/Declined).	Use of Celery/RabbitMQ to ensure delivery without blocking the critical transaction thread.

Deep (Production) - Rules	**Rule Engine:** A configurable service allowing non-technical staff to define dynamic fraud rules (e.g., *"If transaction amount > $5000 AND location != UserHome, flag as RISK."*).	Dedicated Python class structure and database-driven configuration.
Deep (Production) - ML	**ML Integration:** Real-time scoring of transactions (outputting a 0.0 to 1.0 risk score) by integrating a pre-trained `Scikit-Learn` pipeline for classification.	Model serialization, low-latency deployment, and fast feature engineering.

Implementation Details:

- **Architecture:** FastAPI (speed-optimized API) + PostgreSQL (The immutable ledger) + Redis (Rate Limiting and session management).
- **Diagrams Needed:** Sequence Diagram illustrating the high-speed path: `User -> Gateway -> SentinelStream (API) -> ML Model -> PostgreSQL Write -> Response`.
- **Resources:** `FastAPI-Limiter` for managing burst traffic; `Pydantic` for strict data validation on all incoming payloads.

Week	Goal & Focus	Key Tasks	Review & Deliverable
Week 1	**Planning & Architecture**	Define comprehensive SRS. Design relational Database Schema (Star Schema for efficient analytics). Setup GitHub repository with robust CI/CD pipelines (GitHub Actions). Mock-up Admin Dashboard wireframes.	Verification of Schema 3NF normalization. Confirmation of API contract stability (Swagger/OpenAPI).

Week 2	**Core Transaction Pipeline**	Develop the high-speed `POST /transaction` endpoint. Implement Redis caching logic for user profiles to minimize database hits. Connect to PostgreSQL using `SQLAlchemy AsyncIO`.	**Load Testing (Locust):** Must demonstrate the ability to sustain a minimum of **100 requests/second** on local infrastructure.
Week 3	**Intelligence Layer (ML & Rules)**	Integrate a pre-trained **Isolation Forest model** for anomaly detection. Fully implement and test the dynamic "Rule Engine" class. Create dedicated **Celery workers** for sending email alerts on high-risk flags.	**Latency Check:** The entire process must be validated to consistently stay **below 200ms**. Accuracy review of the deployed fraud engine.
Week 4	**Finalization & Deployment**	Full service containerization with **Docker**. Configure Nginx as the reverse proxy for SSL and load distribution. Implement **JWT (JSON Web Tokens)** for securing endpoints. Write comprehensive **PyTest** cases.	**Security Audit:** Mandatory checks against common vulnerabilities (e.g., SQL Injection). Code coverage must exceed **80%**. Final presentation of system performance.

-----3. Project 2: Healthcare - AI-Driven Diagnostic SaaS

Project Title: HIPAA-Compliant Telehealth & Diagnostics Platform
Product Brand Name: "CuraMind AI"

Use Case (Production): A secure, compliant platform designed to connect patients with medical specialists. The core function involves patients securely uploading high-resolution medical images (MRI, X-ray), which the system then pre-processes using Computer Vision to highlight potential anomalies *before* the radiologist conducts their formal review, accelerating the diagnostic workflow.

Feature Type	Detail	Implementation Focus
Basic Functions	Patient/Doctor Authentication, Appointment Scheduling, Secure Electronic Medical Record (EMR) Storage.	Robust Django Models and authentication backend.
Deep (Production) - Compliance	**DICOM Image Handling:** Specialized processing and viewing capabilities for the medical standard image format.	Use of the `pydicom` library for parsing and metadata extraction.
Deep (Production) - Security	**RBAC (Role-Based Access Control):** Granular access control ensuring strict data separation (e.g., Nurses cannot view Doctor-specific data; Admins cannot view private Patient medical data).	Leveraging Django's permission system and object-level permissions.
Deep (Production) - Auditability	**Audit Logging:** Every critical action (view, edit, delete, access attempt) on a medical record is logged to an immutable, time-stamped table for compliance validation.	Dedicated logging middleware/signals.

Implementation Details:

- **Architecture:** Django (Preferred for its built-in security features, mature Admin, and monolithic approach, which simplifies initial compliance) + Celery (For non-blocking Image Processing).
- **Resources:** `pydicom` for medical image handling; `Django-Guardian` for

implementing object-level permissions.

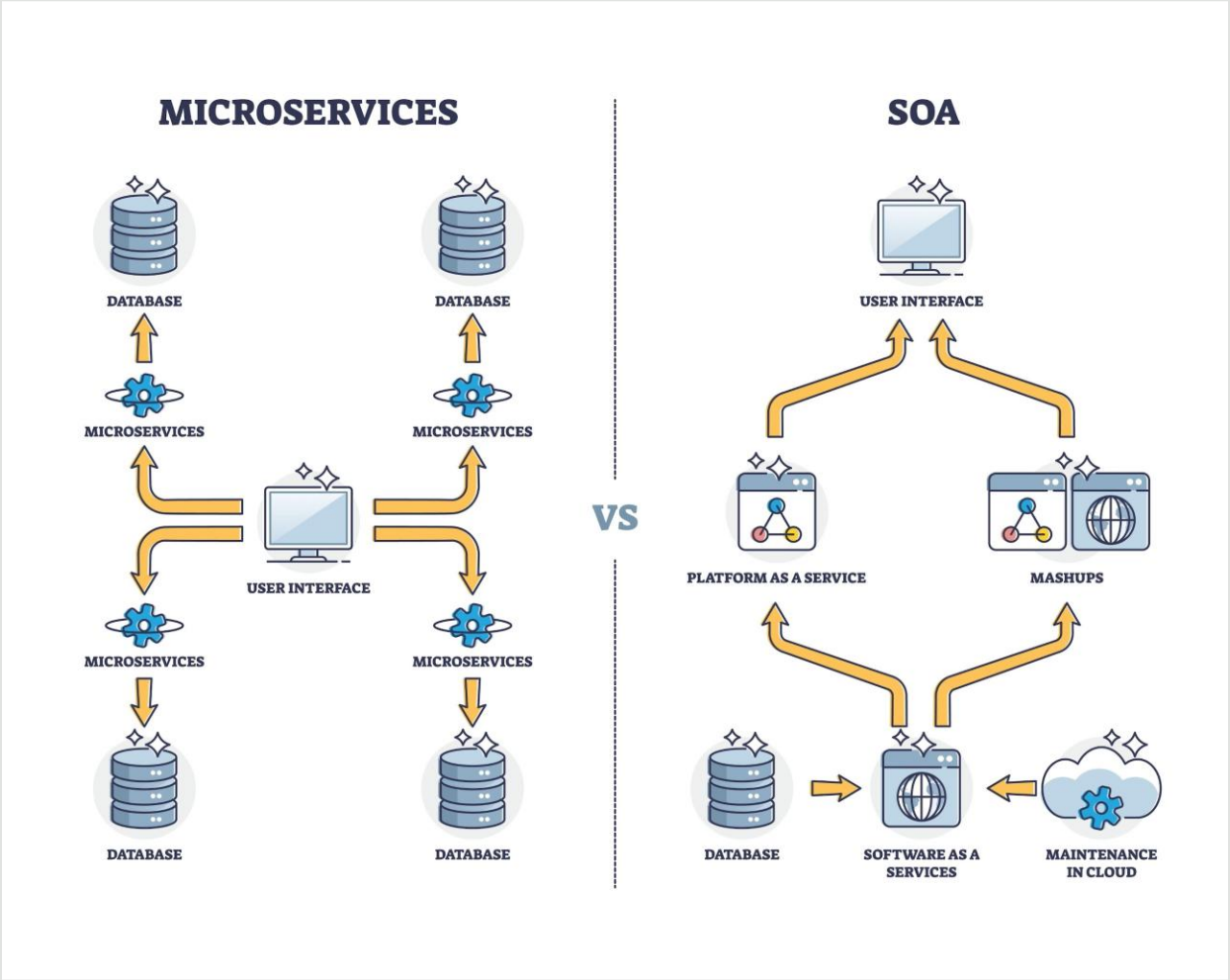
Week	Goal & Focus	Key Tasks	Review & Deliverable
Week 1	**Planning & Compliance**	Finalize the comprehensive **HIPAA/GDPR compliance checklist**. Design the core Patient Portal and Doctor Dashboard UI/UX. Implement database schema focused on **encryption at rest**.	**Security Logic Review:** Sign-off on the data encryption and access control strategy.
Week 2	**Core Platform Development**	Implement a secure **Custom User Model** in Django. Develop appointment scheduling logic and the medical record API. Implement **Secure File Uploads** (including mime type checking and metadata stripping).	**Penetration Test Logic:** Conduct internal testing to attempt bypassing permissions.
Week 3	**AI Integration & Async Processing**	Configure the **Celery worker** to monitor uploaded image queues. Integrate the image processing pipeline: pass the image to a pre-trained CV model (e.g., ResNet/PyTorch for demo) and generate diagnostic heatmaps.	**Queue Performance Review:** Ensure the main web thread maintains responsiveness during peak image processing loads.

Week 4	**Polish & Deployment**	Finalize SSL/TLS setup and hardening. Deploy to a production-like environment (AWS EC2/DigitalOcean). Configure **S3 buckets with private ACLs** for secure, compliant image storage.	Final walkthrough and sign-off on the end-to-end "Doctor Workflow."

-----4. Project 3: E-Commerce - Microservices Inventory & Recommendation System

Project Title: Scalable Marketplace with Recommendation Engine
Product Brand Name: "ShopSphere"

Use Case (Production): Creating a multi-vendor marketplace capable of handling millions of products and high traffic during peak seasons. The system's core value proposition is its ability to deliver personalized product recommendations based on collaborative filtering and real-time user browsing history, ensuring high conversion rates.



Feature Type	Detail	Implementation Focus
Basic Functions	Product Search, Shopping Cart, Checkout Flow (Stripe Integration), Order History Tracking.	Standard E-Commerce functionality.
Deep (Production) - Search	**ElasticSearch Integration:** Replacing standard relational database text search with a distributed search engine for fast, fuzzy matching across the massive product catalog.	Integration using `Elasticsearch-DSL`.
Deep (Production) - Scale	**Event-Driven Inventory:** Utilizing a message broker (RabbitMQ) to manage system-wide updates: an order event triggers parallel updates to inventory, vendor notification, and cache invalidation.	Decoupling services using RabbitMQ and asynchronous listeners.
Deep (Production) - Intelligence	**Recommendation Service:** A dedicated microservice using a **Collaborative Filtering** approach ("Users who bought X also bought Y") to provide personalized suggestions.	Isolated, stateless microservice for machine learning inference.

Implementation Details:

- **Architecture: Microservices.** Service A (Auth/User - Flask), Service B (Product/Search - FastAPI), Service C (Orders - Django). The choice of framework is based on the service's primary function.
- **Diagram:** A detailed C4 Model diagram illustrating the API Gateway and inter-service communication paths (REST/gRPC).
- **Resources:** `Elasticsearch-DSL`, `Stripe-Python`, `RabbitMQ` official client.

Week	Goal & Focus	Key Tasks	Review & Deliverable
Week 1	**Microservice Planning**	Formal definition of service boundaries and responsibilities. Define the **central API Gateway** routing logic. Decide on inter-service communication protocol (initial REST, with consideration for gRPC).	**Interface Contract Verification:** Ensure seamless, secure communication and data exchange between all three services.
Week 2	**Catalog & Search (Heavy Lifting)**	Implement the data ingestion pipeline to scrape/import large volumes of dummy product data. Index all data into **ElasticSearch**. Build the search API with robust filtering capabilities.	**Search Latency Test:** Search queries must return results in **under 100ms** for a simulated dataset of 1 million records.
Week 3	**Orders & Recommendations**	Implement transient Cart logic using **Redis (Session Storage)**. Build the core Order processing service. Implement a simple **Matrix Factorization** algorithm for initial recommendation generation.	**Concurrency/Race Condition Test:** Simulate simultaneous purchases of the last inventory item to test the integrity of the order and inventory service.

Week 4	**Integration & Container Orchestration**	Create the complete **Docker Compose** file for local staging. Develop initial **Kubernetes** deployment files (Deployment, Service, Ingress objects). Configure Load Balancing.	**Disaster Recovery Test:** Manually kill the Order service container to confirm that the API Gateway and other services fail gracefully and the site remains partially operational.

-----5. Project 4: Logistics - IoT Fleet Management & Predictive Maintenance

Project Title: Real-Time Fleet Tracking & Analytics
Product Brand Name: "LogiTrack 360"

Use Case (Production): A major logistics firm requires a system to track 500+ delivery trucks in real-time. The system must ingest high-frequency GPS and engine telemetry data, provide immediate location updates via a dashboard, trigger alerts for **geofencing violations**, and use engine data to predict maintenance needs.

Feature Type	Detail	Implementation Focus
Basic Functions	Driver App API (GPS payload submission), Admin Dashboard Map View, Historical Trip Route Logging.	Standard geo-location API design.
Deep (Production) - Real-Time	**WebSockets:** Implementation of a real-time, bi-directional communication channel to instantly push truck coordinate updates to the Admin Dashboard (e.g., Leaflet/React Frontend).	Leveraging FastAPI's native WebSocket support.
Deep (Production) - Scale	**Time-Series Data:** Efficient storage and querying of massive volumes of high-frequency GPS and sensor data points.	Utilizing **TimescaleDB** (a PostgreSQL extension) optimized for temporal data.

Deep (Production) - Geospatial	**Geospatial Queries:** Ability to execute complex queries like *"Find all trucks currently within a 5km radius of Warehouse A"* or *"Determine if a truck is inside Route Polygon X."*	Integration with `PostGIS` and `GeoAlchemy2`.

Implementation Details:

- **Architecture:** FastAPI (For high-concurrency WebSocket and API support) + TimescaleDB (Time-Series Data) + Minimal Frontend (Focus remains on the Backend API and data pipeline).
- **Resources:** FastAPI WebSockets documentation; `GeoAlchemy2` for geospatial abstractions.

Week	Goal & Focus	Key Tasks	Review & Deliverable
Week 1	**IoT Protocol & Schema**	Design the high-frequency MQTT/WebSocket payload structure for sensor data. Setup and configure the **TimescaleDB** extension. Plan the intricate **Geofencing** polygon logic.	**Database Indexing Review:** Verify geospatial indexing strategy for optimal query speed.
Week 2	**Real-Time Communication Layer**	Implement the core **FastAPI WebSocket endpoint**. Develop logic to handle connection drops, reconnections, and graceful degradation. Simulate load by pushing data from **500 concurrent "trucks"** every 5 seconds.	**WebSocket Stability Test:** Sustained test of stability and latency under high concurrent load.

Week 3	**Analytics & Alerts**	Fully implement the complex **Geofencing logic** (Point-in-Polygon checks). Build the **"Maintenance Prediction"** background job (e.g., *"If EngineTemp > 100C for 5 consecutive minutes, trigger High-Priority Alert"*).	**Alert Delivery Latency Check:** Ensure alerts are delivered to the notification system/dashboard within a critical timeframe.
Week 4	**Finalization**	Connect the optimized Backend APIs to a simple, functional Map frontend (e.g., using Leaflet). Optimize historical query performance (e.g., plotting a truck's 7-day path). Complete API documentation.	**System Reliability Test:** Mandated 24-hour continuous uptime test with simulated data traffic.

-----Note from Management:

All submission links will be activated on the portal at **5:00 PM on the Friday of Week 4** for each respective project. A strict adherence to the deadline is mandatory; late submissions will be immediately disqualified from deployment to the staging environment.

Good luck, and demonstrate the engineering excellence expected of the Python Elite Track.

Zaalima Development
Building the Future, One Line at a Time.