System Design:
Aegis Secure

1.0 INTRODUCTION

1.1 Purpose

This System Design Document describes the architectural and system design of the Aegis Secure application. Its purpose is to provide a comprehensive guide for software developers and testers regarding the system's structure, components, interfaces, and data design. It translates the requirements into a detailed technical blueprint necessary for the implementation and maintenance phases. The intended audience includes software engineers, system architects, and project managers involved in the development of the Aegis platform.

1.2 Scope

The Aegis Secure is a secure, asynchronous application service built to protect users from digital fraud. The scope of this software includes the management of user authentication, synchronization of communication channels (Gmail and SMS), and the orchestration of scam detection algorithms.

Goals and Objectives:

- To provide a robust API for the Aegis mobile/web client.
- To ingest and process user messages (SMS and Email) in real-time.
- To identify potential security threats using advanced spam prediction models.
- To alert users of malicious content with a high degree of confidence.

Benefits:

- Enhanced Security: Protects users from phishing and scam attempts.
- Real-time Analysis: Provides immediate feedback on incoming messages.
- Scalability: Built on an asynchronous framework to handle concurrent user requests efficiently.

1.3 Overview

This document is organized into the following sections:

- Section 1 provides an introduction and defines the scope.
- Section 2 offers a high-level system overview.
- Section 3 (to be completed) details the system architecture and decomposition.
- Section 4 (to be completed) describes the data design and dictionary.
- Section 5 (to be completed) covers component design.

1.4 Reference Material

- IEEE Std 1016-2009, *IEEE Recommended Practice for Software Design Descriptions*.
- FastAPI Documentation (https://fastapi.tiangolo.com/).
- Motor (Async Driver for MongoDB) Documentation.
- Project Software Requirements Specification (SWRS).

1.5 Definitions and Acronyms

- API (Application Programming Interface): A set of functions and procedures allowing the creation of applications that access the features or data of an operating system, application, or other service.
- CORS (Cross-Origin Resource Sharing): A mechanism that allows restricted resources on a web page to be requested from another domain outside the domain from which the first resource was served.
- LLM (Large Language Model): AI models used for analyzing message content for scam indicators.
- MongoDB: A source-available cross-platform document-oriented database program.

- OTP (One-Time Password): A password that is valid for only one login session or transaction.
- SWDD: Software Design Document.
- XSS (Cross-Site Scripting): A security vulnerability typically found in web applications; the backend includes sanitation to prevent this.
- SQLi (SQL Injection): A code injection technique; the backend includes regex patterns to detect and block these attacks.

2.0 SYSTEM OVERVIEW

The Aegis Secure serves as the core processing unit for a scam detection application. Its primary function is to ingest user communication data, specifically SMS and Emails, and analyze them to detect potential fraud. The system operates by flagging suspicious messages with a confidence score, enabling the client application to alert the user about the possibility of scamming.

Functionality & Design: The system is designed as a modular RESTful API built using the FastAPI framework, ensuring high performance and asynchronous request handling. It interacts with a MongoDB database for persistent storage of user profiles, messages, and analysis results.

Key functional areas include:

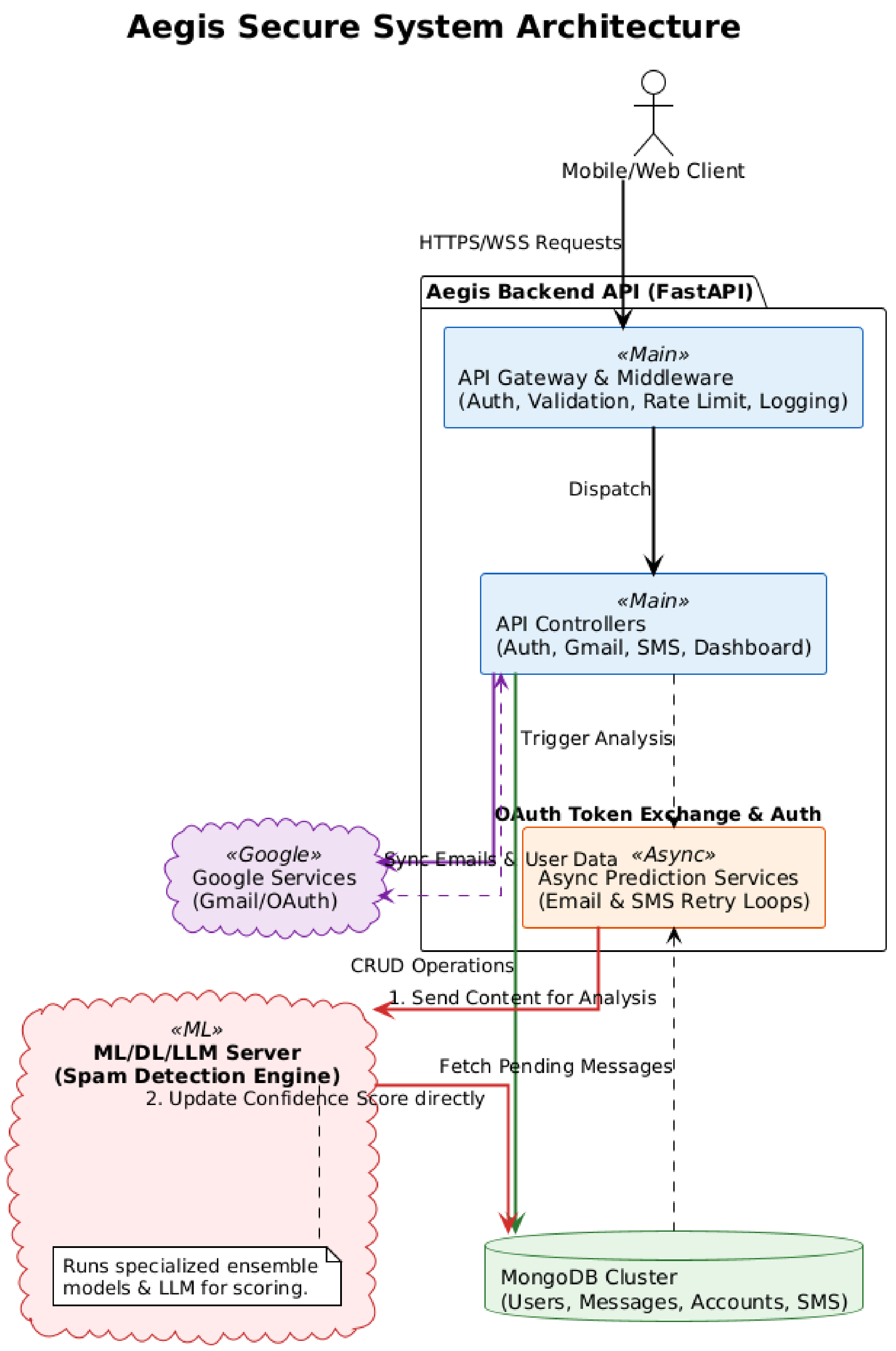
1. Scam Detection Engine: The backend orchestrates the analysis process by calling an external API hosted on a separate server. This server runs a specialized ensemble of Machine Learning models and Large Language Models (LLMs) to evaluate message content. The results are stored in the database and pushed to the User Interface concurrently. Background tasks (retry_email_SpamPrediction, retry_sms_SpamPrediction) ensure robust processing and retries for failed predictions.
2. Authentication & Security: The system implements a secure authentication flow including Registration, Login, OAuth, and OTP verification. It utilizes custom middleware for:
    - Rate Limiting: Prevents abuse by limiting requests per IP, with stricter limits for authentication endpoints.
    - Request Validation: Sanitizes inputs to prevent XSS and SQL Injection attacks using regex pattern matching.
    - Security Headers: Injects standard security headers into responses.

3. Data Management: The system manages distinct collections for Users (auth_db), Emails (Mails_db), and SMS (Sms_db). It utilizes a DatabaseManager with retry logic to ensure connection stability.
4. Logging & Monitoring: A comprehensive logging system captures request lifecycles, database operations, security events, and errors, utilizing color-coded console output and file logging for traceability.

Context: The backend acts as the bridge between the user-facing client (Mobile/Web), the data persistence layer (MongoDB), and the intelligent analysis layer (External ML/LLM API). It handles the synchronization of data from Gmail and local SMS inboxes, ensuring the analysis engine has access to the latest user communications.

3.0 System High Level View

3.1)System Architecture:


Aegis Secure System Architecture
Mobile/Web Client
HTTPS/WSS Requests
Aegis Backend API (FastAPI)
«Main»
API Gateway & Middleware
(Auth, Validation, Rate Limit, Logging)
Dispatch
«Main»
API Controllers
(Auth, Gmail, SMS, Dashboard)
Trigger Analysis
OAuth Token Exchange & Auth
«Google»
Google Services
(Gmail/OAuth)
Sync Emails & User Data
«Async»
Async Prediction Services
(Email & SMS Retry Loops)
CRUD Operations
1. Send Content for Analysis
«ML»
ML/DL/LLM Server
(Spam Detection Engine)
Fetch Pending Messages
2. Update Confidence Score directly
Runs specialized ensemble models & LLM for scoring.
MongoDB Cluster
(Users, Messages, Accounts, SMS)

3.2) Decomposition Description (Aegis Secure)

The Aegis Backend is decomposed into four primary functional subsystems. This decomposition follows a layered architecture pattern, separating concerns between request handling, business logic, security, and data persistence.

1. Interface & Routing Subsystem

- Description**:** This subsystem is the entry point for all client interactions. It routes HTTP requests to the appropriate controllers based on the URL path.
- Key Components:
    - Auth Router (/auth)**:** Handles registration, login, and token generation.
    - Gmail/SMS Routers: Manage the synchronization of user messages.
    - OAuth Router: Manages the handshake with Google Services.

2. Middleware & Security Subsystem

- Description: This subsystem intercepts every incoming request before it reaches the routing logic. It enforces security policies and ensures data integrity.
- Key Components:
    - RateLimitMiddleware: Controls traffic flow to prevent abuse (DoS attacks).
    - RequestValidationMiddleware: Sanitizes inputs against XSS and SQL injection.
    - SecurityHeadersMiddleware: Injects HTTP security headers (HSTS, X-Frame-Options).

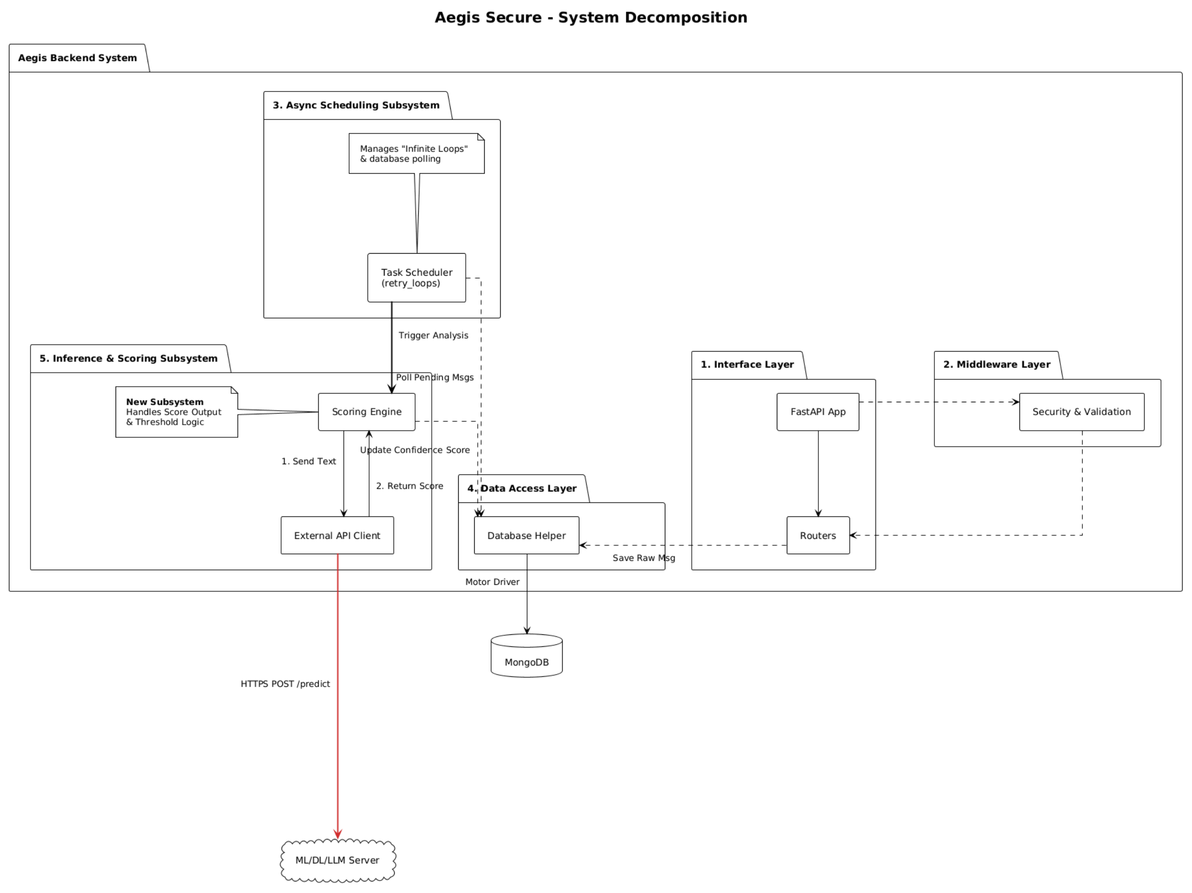
3. Asynchronous Analysis Subsystem (Background Services)

- Description: This is the core "intelligence" layer. It operates independently of the request-response cycle to ensure the user interface remains responsive. It continuously monitors the database for unanalyzed messages.
- Key Components:
    - retry_email_SpamPrediction: A continuous loop processing new emails.
    - retry_sms_SpamPrediction: A continuous loop processing new SMS.
    - External Integration: Manages the secure connection to the ML/DL/LLM Server for analysis.

4. Data Access Subsystem

- Description: This subsystem abstracts the complexity of the MongoDB driver (motor). It handles connection pooling, retry logic for failed queries, and CRUD operations.
- Key Components:
    - DatabaseManager: Manages the lifecycle (connect/disconnect) of the database connection.
    - DatabaseHelper: Provides wrapper functions (find_one, insert_one) with built-in error handling and logging.

System Decomposition Diagram:



3.3 Design Rationale

The architectural decisions for the Aegis Backend were driven by the need for high concurrency, low latency, and fault tolerance when handling real-time communication data.

- Asynchronous Non-Blocking I/O: We selected FastAPI combined with the Motor (MongoDB) driver to fully leverage Python's asyncio capabilities. This allows the server to handle multiple concurrent data synchronization requests (Email/SMS ingestion) without blocking the main execution thread, ensuring the user interface remains responsive.
- Decoupled Inference Logic: The computationally intensive task of Spam Prediction (involving ML/LLM analysis) was deliberately isolated into asynchronous background services (retry_loops). This

design ensures that the latency of the external ML inference engine never delays the immediate HTTP response to the client, allowing data ingestion and analysis to proceed at different speeds.

- Resiliency via Decorators: We implemented a Retry Pattern using decorators (@with_retry) for all database and external API interactions. This logic ensures the system automatically recovers from transient network failures or database timeouts without crashing or requiring manual user intervention.
- Middleware-Driven Security: Security concerns (Rate Limiting, XSS/SQLi Sanitization) were abstracted into a Middleware Layer rather than being hardcoded into individual endpoints. This enforces a consistent global security policy and reduces the risk of developers accidentally omitting checks in new routes.
- Flexible Data Schema: MongoDB was chosen over relational databases to accommodate the unstructured and variable nature of Email and SMS metadata, allowing for rapid iteration of the data model without complex migrations.

4.0 DATA DESIGN

4.1 Data Description

The Aegis Backend utilizes a NoSQL Document Store (MongoDB) for data persistence, chosen for its flexibility in handling unstructured message data (Emails and SMS) and its high write throughput.

The system interacts with the database asynchronously using the Motor driver (motor.motor_asyncio), ensuring that database I/O operations do not block the main application thread.

The data is organized into three distinct logical databases to separate concerns:

1. Auth DB (auth_db): Stores user identity and authentication data.
2. Mails DB (Mails_db): Stores synchronized Gmail data and account linkages.
3. SMS DB (Sms_db): Stores synchronized SMS messages and metadata.

4.2 Data Dictionary

Below is the list of major system entities, database collections, and data models used in the system.

A. Database Collections

Entity Name	Collection Name	Database	Description
Users	users	auth_db	Stores registered user profiles and credentials.
OTPs	otps	auth_db	Temporary storage for One-Time Passwords used in verification.
Accounts	accounts	mail_db	Links user profiles to external accounts (e.g., Gmail Oauth tokens).
Messages	messages	mail_db	Stores synchronized email content and their analysis status.
Avatars	avatars	mail_db	Stores user or sender avatar images.
SMS Messages	sms_messages	sms_db	Stores synchronized SMS content and their hash for deduplication.

B. Data Models (Application Objects) These models define the structure of data exchanged via the API.

- RegisterRequest:
    - *Attributes:* name (str), email (str), password (str).
    - *Usage:* Payload for user registration.
- LoginRequest:
    - *Attributes:* email (str), password (str).
    - *Usage:* Payload for user authentication.
- SmsMessage:
    - *Attributes:* address (str), body (str), date_ms (int), type (str).
    - *Usage:* Represents a single SMS synced from the client.
- SpamRequest:
    - *Attributes:* sender (str), subject (str), text (str).
    - *Usage:* Payload sent to the inference engine for spam prediction.
- UserResponse:
    - *Attributes:* name (str), email (str), user_id (str).
    - *Usage:* Standardized user profile response object.

5.0 COMPONENT DESIGN

This section details the internal logic of key system components, using Pseudocode (PDL) to describe the algorithms used for critical functions.

5.1 Database Utility Component (Db_utils.py)

This component handles the resilience of the data layer. It ensures that temporary network glitches do not crash the application.

- Component: DatabaseHelper / with_retry decorator
- Purpose: To execute database operations with automatic retry logic upon connection failure.

```
FUNCTION with_retry(max_retries, delay, operation):
    SET last_exception = NULL


    FOR attempt FROM 0 TO max_retries:
        TRY:
            result = EXECUTE operation()
            RETURN result
        CATCH ConnectionFailure AS error:
            SET last_exception = error
            IF attempt < (max_retries - 1):
                LOG "Connection failed, retrying in {delay} seconds"
                WAIT for delay seconds
            ELSE:
                LOG "Operation failed after max attempts"
        CATCH AnyOtherException:
```

```
        RAISE exception immediately (do not retry)


    RAISE DatabaseError("Operation failed: " + last_exception)

END FUNCTION
```

5.2 Rate Limiting Component (middleware.py)

This component protects the API from abuse (DoS attacks) by tracking the frequency of requests from client IPs.

- Component: RateLimiter
- Purpose: To decide whether to allow or block a request based on recent activity.

```
FUNCTION is_rate_limited(client_identifier, max_requests,
window_seconds):

    current_time = GET_CURRENT_TIMESTAMP()


    # 1. Cleanup Phase

    IF (current_time - last_cleanup_time) > cleanup_interval:

        FOR each identifier IN request_history:

            REMOVE timestamps older than 1 hour


    # 2. Check Logic

    cutoff_time = current_time - window_seconds

    client_history = GET history for client_identifier


    # Filter history to keep only requests within the current window
    valid_requests = FILTER client_history WHERE timestamp >
cutoff_time


    IF COUNT(valid_requests) >= max_requests:
        RETURN True (Block Request)
    ELSE:
        ADD current_time TO client_history
        RETURN False (Allow Request)
END FUNCTION
```

5.3 Security Validation Component (validators.py)

This component enforces security policies on user inputs to prevent weak credentials and injection attacks.

- Component: PasswordValidator
- Purpose: To validate that a password meets complexity requirements before storage.

```
FUNCTION validate_password(password):
    IF LENGTH(password) < 8:
        RETURN False, "Password too short"


    IF NOT CONTAINS(password, Uppercase_Letter):
        RETURN False, "Missing uppercase"


    IF NOT CONTAINS(password, Lowercase_Letter):
        RETURN False, "Missing lowercase"
    IF NOT CONTAINS(password, Digit):

        RETURN False, "Missing digit"


    IF NOT CONTAINS(password, Special_Character):

        RETURN False, "Missing special char"


    common_passwords = ["password", "12345678", "admin",
"qwerty"]

    IF LOWERCASE(password) IN common_passwords:

        RETURN False, "Password is too common"


    RETURN True, "Valid"

END FUNCTION
```

5.4 Application Lifecycle Component (main.py)

This component manages the startup sequence, ensuring all resources are ready before the API accepts traffic.

- Component: lifespan (FastAPI Context Manager)
- Purpose: To initialize background services and database indexes.

```
FUNCTION lifespan(application):

    PRINT "Starting Aegis Backend..."


    # 1. Start Background Services

    IF not reloader_process:
        START_TASK(retry_email_SpamPrediction)
        START_TASK(retry_sms_SpamPrediction)

    # 2. Database Initialization
    CREATE_INDEX(Users_Collection, "email", unique=True)
    CREATE_INDEX(Accounts_Collection, "user_id")
    CREATE_INDEX(Messages_Collection, "timestamp", descending)
    # ... (other indexes)

    PRINT "All indexes created"

    YIELD (Server runs and accepts requests here)

    PRINT "Shutting down Aegis Backend..."
END FUNCTION
```

6.0 Current System Stack

- Programming Language
    - Python (3.x): Evidenced by the use of Python syntax, type hinting (e.g., List[str], Optional[Dict]), and standard library imports like asyncio and typing.
- Framework
    - FastAPI: The core web framework used for the API, chosen for its asynchronous capabilities and performance.
    - Starlette: Used implicitly as the base for FastAPI and explicitly for middleware components like BaseHTTPMiddleware and CORSMiddleware.
- Database
    - MongoDB: A NoSQL document database used for storing users, messages, and logs. It is referenced via the MONGO_URI environment variable.
    - Motor: The asynchronous Python driver (motor.motor_asyncio) used to interact with MongoDB without blocking the main event loop.
- Cloud Services & Integration**s**
    - Google Cloud Platform (GCP): The system integrates with Google Services for Gmail synchronization and OAuth authentication (evidenced by gmail.router and Oauth.router).
    - External ML/LLM Inference Server: A separate server hosting the "specially trained ensemble of models and LLM" used for spam prediction, which the backend communicates with asynchronously.
- External Dependencies (Key Libraries)
    - fastapi: Web framework.
    - uvicorn: ASGI server (implied by the UVICORN_RELOADER environment check).
    - pydantic: Data validation and settings management (e.g., BaseModel, EmailStr).
    - python-dotenv: Environment variable management (load_dotenv).
    - motor: Async MongoDB driver.
- DevOps Pipeline

- *Not Defined in Source:* The provided code files do not contain configuration files (such as Dockerfile, Jenkinsfile, or GitHub Actions workflows) required to document the CI/CD pipeline.

7.0 Security Design

This section outlines the mechanisms used to protect user data and ensure system integrity.

1. Authentication The system uses a multi-layered approach featuring JWT-based email/password login, Google OAuth 2.0 integration, and OTP verification (MFA) for critical actions like registration and password resets.

2. Authorization Access control is enforced by scoping all database queries to the specific user_id, ensuring strict resource isolation. Service-level permission checks prevent unauthorized access to restricted endpoints.

3. Encryption & Integrity Data in transit is secured via enforced TLS/HTTPS headers. Data integrity is maintained through strict password complexity rules (PasswordValidator) and content hashing for duplicate detection.

4. Input Validation The system employs a "Defense in Depth" strategy using Pydantic for schema enforcement, regex-based injection filtering, and a dedicated TextSanitizer to strip malicious HTML/JS tags to prevent XSS.

5. Known Security Gaps

- In-Memory Rate Limiting: Current implementation is stateful and non-distributed; requires migration to Redis.
- Injection Mismatch**:** Middleware filters for SQL injection patterns, leaving the MongoDB (NoSQL) database potentially vulnerable to specific NoSQL vectors.
- CSRF & XSS: Lack of explicit CSRF tokens and reliance on brittle regex for XSS filtering should be addressed with robust frontend frameworks or dedicated libraries.

8.0 Performance & Scalability

This section documents the current system's performance characteristics and scalability strategies based on the implementation.

- Response Times**:** The system utilizes FastAPI and Motor (asynchronous MongoDB driver) to ensure non-blocking I/O. This design minimizes latency for I/O-bound operations (like database queries), allowing the server to handle concurrent requests efficiently. However, the overall end-to-end response time for scam detection is decoupled from the user request to avoid blocking.
- Peak Load: Traffic is managed via a custom RateLimitMiddleware that enforces request caps (e.g., RATE_LIMIT_PER_MINUTE) to prevent system overload during usage spikes. Authentication endpoints have stricter limits (RATE_LIMIT_LOGIN_PER_HOUR) to protect against brute-force attacks.
- Bottlenecks:
    - Major Bottleneck (ML/LLM API**):** The external ML/LLM service is the primary bottleneck due to its non-commercial nature. It has strict rate limits and token restrictions, which severely limits the throughput of the retry_loops background services. This can lead to a growing backlog of unanalyzed messages during high traffic.
    - In-Memory Rate Limiting: The current RateLimiter uses a local Python dictionary (defaultdict), which is not thread-safe across multiple worker processes and prevents horizontal scaling.
- Caching**:** No explicit caching layer (e.g., Redis) is implemented. Repeated database queries for the same user data or static resources will hit the MongoDB instance every time, adding unnecessary load.
- Logging**:** A comprehensive, asynchronous-compatible logging system is implemented via logger.py. It captures request duration, database operation timing (log_database_operation), and external API latency (log_external_api_call), which is crucial for monitoring the latency introduced by the slow ML API.
- Horizontal/Vertical Scaling:
    - Vertical Scaling**:** The application can scale vertically by increasing CPU/RAM to handle more concurrent connections.
    - Horizontal Scaling**:** While the stateless JWT authentication allows for adding more servers, the in-memory rate limiter

breaks horizontal scalability. Since rate limits are stored locally in RAM, they would not be shared across different server instances, effectively rendering the rate limits inaccurate in a distributed setup.