I. Chosen Software Architecture Style

-> Microservices Architecture

A. Justification (Granularity & Structure)

<> Why Microservices?

- System is divided into **independent services**
- Each service has **single responsibility**
- Services communicate via **REST APIs**
- Each service can run on a **separate port**
- Services can be deployed independently

<> Component Granularity

The system is decomposed into the following **fine-grained services**:

- **Log Ingestion Service**
- **Log Processing Service**
- **Detection Service**
- **Incident Management Service**
- **Response Automation Engine**
- **Response Executor**
- **Notification Service**
- **Audit Service**

Each of the above services:

- Has its own responsibility
- Has its own API endpoint
- Can scale independently
- Can fail independently

<> Why It Is NOT Monolithic

- Logic is **not in a single application**
- Services are loosely coupled
- Clear separation of concerns
- Independent runtime processes

<> Why It Is NOT Pure Layered Architecture

- Components are deployed as **independent services**
- Communication happens over HTTP
- Not just logical layers in one application

<> Architectural Diagram (Conceptual)

Client

↓

Log Ingestion Service

↓

Log Processing Service

↓

Detection Service

↓

Incident Management Service

↓

Response Automation Engine

↓

Response Executor

↓

Notification / Audit Services

<> Granularity Justification

- **Coarse-grained at service level**
    - Each service handles a full functional responsibility.
- **Fine-grained at domain level**
    - Classes like LogEvent, Incident, Playbook define internal structure.

This aligns with **Microservices Architecture**, where:

- Each business capability is an independent service.
- Services communicate over lightweight protocols.

II) Justification for Choosing Microservices Architecture

<> 1. Scalability

- Each service (Ingestion, Processing, Detection, etc.) can scale independently.
- High-load components (e.g., Detection Service) can be scaled without affecting other services.
- Suitable for handling increasing log volume in real-time systems.

<> 2. Maintainability

- Clear separation of responsibilities (Single Responsibility Principle).
- Smaller codebases per service → easier debugging and testing.
- New features (e.g., new detection rule) can be added without modifying unrelated components.
- Independent updates reduce system-wide impact.

<> 3. Performance Optimization

- Services can be optimized individually.
- Processing-heavy components (e.g., anomaly detection) can use dedicated resources.

- Fault isolation ensures one slow service does not crash the entire system.

<> 4. Fault Tolerance & Reliability

- Failure in one service (e.g., Notification Service) does not bring down the whole system.
- Supports independent restart and recovery.
- Improves overall system availability.

<> 5. Flexibility & Extensibility

- New modules (e.g., new actuator or detection algorithm) can be added as separate services.
- Supports future integration with external systems (cloud tools, etc.).
- Allows technology flexibility for different services if needed.