

# Detailed Design Specification: OpsPulse AI

Version: 1.0 (Hackathon MVP)

Based on: OpsPulse AI SRS v1.0 1

## **1. Executive Summary**

OpsPulse AI is a middleware intelligence layer designed to reduce Mean Time To Resolution (MTTR)<sup>2222</sup>. It ingests raw logs, uses NLP to classify them into distinct "event types," detects statistical anomalies using tumbling windows, and automatically retrieves remediation steps via RAG (Retrieval-Augmented Generation)<sup>3333</sup>.

## 2. High-Level Architecture

The system follows a uni-directional stream-processing pipeline. For the hackathon, we will use a **Microservices** approach to separate the "fast" stream processing from the "heavy" AI processing.

## 2.1 Core Modules

1. **Log Ingestion Service:** Buffers high-velocity logs<sup>4</sup>.
  2. **Stream Processor (The Brain):** Handles parsing, NLP classification, and windowing<sup>5</sup>.
  3. **RAG Service:** Indices runbooks and retrieves solutions upon anomaly triggers<sup>6666</sup>.
  4. **Presentation Layer:** A real-time Dashboard and Alert dispatcher<sup>7777</sup>.

### **3. Component Design & Implementation Logic**

### **3.1 Module A: Ingestion & PII Masking**

- **Goal:** Accept logs via TCP/UDP or HTTP, sanitize them, and push to the message broker.

- **Hackathon Stack:** FastAPI (HTTP endpoint) or Python Socket -> Kafka (or Redpanda for lower overhead).
- **Key Logic:**
  1. **Schema Enforcement:** Ensure `Timestamp`, `Source_IP`, `Log_Level`, and `Message` exist<sup>8</sup>.
  2. **PII Masking (Regex):** Before pushing to Kafka, scan `Message` for credit card/password patterns and replace with `[MASKED]`<sup>9</sup>.
  3. **Serialization:** Convert to JSON and push to Kafka Topic: `raw_logs`.

### 3.2 Module B: NLP Classification & Stream Processing

- **Goal:** Normalize raw text into "Templates" and detect anomalies<sup>10</sup>.
- **Hackathon Stack:** Python Consumer (using confluent-kafka) + Scikit-Learn (for lightweight isolation forest or Z-score) + SentenceTransformers (for local embeddings).
- **Logic Flow:**
  1. **Template Extraction (NLP):**
    - Instead of calling an expensive LLM for every log, use a lightweight embedding model (e.g., all-MiniLM-L6-v2) to vectorise the log message.
    - Cluster these vectors to assign a `Class_ID` (e.g., Cluster 1 = "DB Connection Fail")<sup>11</sup>.
  2. **Tumbling Window Aggregation:**
    - Group logs by `Class_ID`.
    - Count occurrences in fixed 1-minute windows<sup>12</sup>.
  3. **Anomaly Detection:**
    - **Spike Detection:** Calculate Z-Score. If  $Z > 3\sigma$ , flag as Spike<sup>13</sup>.
    - **Drop Detection:** If a known `Class_ID` (like "Heartbeat") count == 0, flag as Drop<sup>14</sup>.

### 3.3 Module C: RAG (Retrieval-Augmented Generation)

- **Goal:** Fetch the correct runbook snippet when an anomaly is flagged<sup>15</sup>.
- **Hackathon Stack:** LangChain + ChromaDB (Local Vector Store) + OpenAI API (or Ollama for local LLM).
- **Workflows:**
  1. **Ingestion (Pre-computation):**
    - Upload PDF/MD runbooks<sup>16</sup>.
    - Chunk text and store embeddings in ChromaDB<sup>17</sup>.
  2. **Retrieval (Real-time):**
    - Triggered *only* when `Anomaly == True`.

- Query ChromaDB using the Log Template (not the specific timestamped log) as the key<sup>18</sup>.
- Retrieve the top 3 chunks (Remediation steps)<sup>19</sup>.

### 3.4 Module D: Alerting & Dashboard

- **Goal:** Visualize health and send notifications.
- **Hackathon Stack:** Streamlit (Python-only UI).
- **UI Layout:**
  - **Top:** Metrics Ticker (Total Logs, MTTR, Active Anomalies).
  - **Middle:** Real-time Line Chart (Log volume per minute).
  - **Bottom:** "Active Incidents" Table.
    - Columns: Time, Error Class, Recommended Fix (Expandable), Link to Runbook<sup>20</sup>.
  - **Backend Action:** Send JSON payload to a Slack Webhook<sup>21</sup>.

## 4. Data Models (Schema Design)

### 4.1 Log Event (Kafka Message)

```
JSON
{
  "timestamp": "2023-10-27T10:00:00Z",
  "source_ip": "192.168.1.5",
  "log_level": "ERROR",
  "raw_message": "Connection timeout to DB-01",
  "masked_message": "Connection timeout to DB-01",
  "template_id": "ERR_DB_TIMEOUT"
}
```

### 4.2 Runbook Vector Document

```
JSON
{
  "id": "runbook_db_01_chunk_4",
  "embedding": [0.12, -0.05, ...],
  "metadata": {
    "filename": "Database_Recovery.pdf",
    "page_number": 3,
    "content": "If DB timeout occurs, restart service via: sudo systemctl restart postgresql"
  }
}
```

---

## 5. Technology Stack Selection (Hackathon Optimized)

Component	Technology Choice	Rationale
Language	Python 3.10+	Native support for AI/ML and rapid prototyping.
Message Broker	Redpanda (Docker)	Kafka-compatible but lighter and faster to set up.
Vector DB	ChromaDB	Open-source, runs locally, easy Python integration <sup>23</sup> .
LLM / NLP	OpenAI gpt-3.5-turbo	Cheap, fast, reliable for the RAG summarization <sup>24</sup> .
Embeddings	SentenceTransformers	Free local embedding (huggingface) to save API costs.
Dashboard	Streamlit	Zero HTML/CSS required; builds dashboard in minutes <sup>25</sup> .
Orchestration	Docker Compose	Spin up the whole stack with one command.

---

## 6. Implementation Roadmap (24-Hour Plan)

### 1. Hours 0-4: The Pipeline

- Set up Docker Compose (Redpanda, ChromaDB).
- Write `producer.py` (reads a dummy log file and pushes to Kafka)<sup>26</sup>.

### 2. Hours 4-10: The Intelligence

- Write `processor.py`. Implement the Windowing logic manually (Python list buffers) or using `Pandas`.
- Implement basic "Spike Detection" (if count > threshold)<sup>27</sup>.

### 3. Hours 10-16: The RAG

- Build the `indexer.py` to ingest 3 sample PDF runbooks.
- Connect the processor to query ChromaDB when a spike is detected<sup>28</sup>.

### 4. Hours 16-20: The UI

- Build Streamlit app to read from a "Processed" topic or local SQLite DB.
- Display the alerts and the retrieved runbook text<sup>29</sup>.

### 5. Hours 20-24: Polish

- Refine the "Mock Data" to ensure it triggers the demo scenarios perfectly.
- Add PII masking logic<sup>30</sup>.

## 7. Future/Production Considerations (Post-Hackathon)

- **Scalability:** Replace Python consumer with Apache Flink for true distributed windowing<sup>31</sup>.
- **Storage:** Implement a Data Lake (S3) for long-term log retention (Archival)<sup>32</sup>.
- **Auto-Remediation:** Allow the system to trigger Ansible/Terraform scripts directly<sup>33</sup>.

### Suggested Next Step

The SRS specifically asks: "Would you like me to create a Python simulation code for the 'Tumbling Window' statistical analysis part of this SRS?"<sup>34343434</sup>

Given that this is the core logic for the anomaly detection engine, would you like me to generate that simulation code now so you can use it for the "Processing Layer"?