المعهد العالي للدراسات التكنولوجيــة في المواصلات بتونس
Institut Supérieur des Etudes Technologiques en Communications de Tunis

Department: STIC

Reference : SR

Applied License in Information and Communication Sciences and Technologies

Supervised Project

LogChat: AI-Driven SIEM

Realized by: Chames Edin Turki & Saleh Eddine Touil

Class: L3 SR B & L3 SR C

Superviser(s): Mr.Mounir Kthiri

Host Company: -

Academic year: 2025-2026

Acknowledgements

We would like to express our sincere gratitude to all those who contributed to the successful completion of this project.

First and foremost, we extend our deepest appreciation to our academic supervisor, **Mr. Mounir Kthiri**, for his invaluable guidance, constructive criticism, and continuous encouragement throughout this project. His expertise in software architecture and security best practices has profoundly influenced our approach to building LogChat.

We are grateful to the **Higher Institute of Technological Studies in Communications of Tunis** for providing the academic framework, technical resources, and conducive learning environment that made this project possible.

Our sincere thanks go to the **members of the jury** for dedicating their valuable time to review and evaluate this work. Their feedback and insights are instrumental in our professional growth.

We acknowledge the contributions of our **classmates and peers** who participated in beta testing the Golang Agent across various Windows and Linux environments. Their feedback helped identify critical edge cases and improve system reliability.

Special recognition goes to the **open-source community**, particularly the teams behind:

- **Next.js** — for the powerful React framework
- **Prisma** — for the elegant ORM solution
- **Ollama** — for democratizing local LLM deployment
- **The Go Team** — for creating a language perfect for systems programming

Finally, we extend our heartfelt thanks to our **families and friends** for their moral support, understanding, and encouragement during the challenging phases of this project.

Saleh Eddine Touil & Chames Edin Turki
Tunis, January 2026

Abstract

The exponential growth of distributed systems, cloud-native architectures, and microservices has resulted in unprecedented volumes of log data, rendering traditional manual security auditing obsolete. Contemporary Security Information and Event Management (SIEM) solutions, while powerful, present significant barriers to adoption: prohibitive licensing costs exceeding $100,000 annually, steep learning curves requiring specialized query languages, and complex multi-node deployment requirements.

LogChat addresses these challenges by democratizing security analytics through the application of Generative Artificial Intelligence. This end-of-studies project presents the design, implementation, and evaluation of an open-source, privacy-first SIEM platform comprising three core components:

1. A **high-performance Golang Agent** (5MB binary) enabling cross-platform log collection from Windows Event Logs, Linux Syslog/Journald, and arbitrary log files — compiled with zero runtime dependencies for seamless deployment.
2. A **Node.js/Express backend** providing RESTful APIs for log ingestion at 1000+ requests/second, real-time threat detection using pattern-based analysis, and a sophisticated RAG (Retrieval Augmented Generation) engine for contextual AI responses.
3. A **Next.js 14 frontend** delivering an intuitive, real-time security dashboard with sub-second update latency via Server-Sent Events (SSE) and a natural language chat interface for log analysis.

The integration of **Ollama** for local LLM inference ensures complete data privacy — sensitive log information never leaves the organization's infrastructure. Deployed via **Docker Compose**, LogChat becomes operational within five minutes, transforming complex security queries into actionable insights without requiring SPL, KQL, or other proprietary languages.

Keywords: SIEM, Log Management, Generative AI, RAG, Golang, Docker, Cybersecurity, NLP, Real-time Analytics, Threat Detection, Local LLM, Privacy-First Architecture.

Résumé

La croissance exponentielle des systèmes distribués et des architectures cloud-native a entraîné des volumes sans précédent de données de journalisation, rendant l'audit de sécurité manuel obsolète. Les solutions SIEM contemporaines présentent des obstacles significatifs: coûts de licence prohibitifs, courbes d'apprentissage abruptes et exigences de déploiement complexes.

LogChat répond à ces défis en démocratisant l'analyse de sécurité grâce à l'Intelligence Artificielle Générative. Ce projet de fin d'études présente une plateforme SIEM open-source axée sur la confidentialité, comprenant:

1. Un **Agent Golang** haute performance pour la collecte de logs multi-plateforme
2. Un **backend Node.js/Express** avec détection de menaces en temps réel et moteur RAG
3. Un **frontend Next.js 14** offrant un tableau de bord interactif et une interface de chat IA

L'intégration d'**Ollama** pour l'inférence LLM locale garantit une confidentialité totale des données. Déployé via **Docker Compose**, LogChat devient opérationnel en cinq minutes.

Mots-clés: SIEM, Gestion des Logs, IA Générative, RAG, Golang, Docker, Cybersécurité, TAL, Analyse en Temps Réel.

Contents

Acknowledgements

Abstract

Résumé

Contents

List of Figures

List of Tables

List of Abbreviations

General Introduction **1**

1 State of the Art **4**

1.1 Introduction 4

1.2 Historical Evolution of Log Management 4

1.2.1 Early Approaches (1990s-2000s) 4

1.2.2 Centralized Logging (2005-2015) 5

1.2.3 Modern AI-Enhanced SIEM (2020-Present) 5

1.3 Market Analysis of Existing Solutions 5

1.3.1 Elastic Stack (ELK) 5

1.3.2 Splunk Enterprise 6

1.3.3 Wazuh 7

1.3.4 Microsoft Sentinel 7

1.3.5 Comparative Analysis 8

1.4 Technology Stack Selection 9

1.4.1 Backend: Node.js with TypeScript 9

1.4.2 Frontend: Next.js 14 with React 18 10

1.4.3 Database: PostgreSQL 16 with Prisma ORM 10
1.4.4 Agent: Golang 11
1.4.5 AI Engine: Ollama with Local LLMs 12
1.5 Development Methodology 13
1.6 Conclusion 14

2 Analysis and Specification **15**
2.1 Introduction 15
2.2 Actors Identification 15
2.3 Functional Requirements 16
2.3.1 Agent Requirements (RF-AGT) 16
2.3.2 Backend Server Requirements (RF-SRV) 17
2.3.3 Frontend Requirements (RF-UI) 17
2.4 Non-Functional Requirements 18
2.5 Use Case Modeling 18
2.5.1 Global Use Case Diagram 18
2.5.2 Detailed Use Case Descriptions 20
2.5.3 Use Case Prioritization Matrix 21
2.6 Behavioral Modeling 21
2.6.1 Sequence Diagram: Authentication Flow 21
2.6.2 Sequence Diagram: RAG Chat Workflow 22
2.6.3 Sequence Diagram: Log Ingestion with Threat Detection 23
2.6.4 Activity Diagram: Threat Detection Engine 24
2.7 Structural Modeling 25
2.7.1 Class Diagram: Domain Model 25
2.7.2 Entity Descriptions 26
2.8 Architectural Design 26
2.8.1 High-Level Architecture 26
2.8.2 Component Architecture 27
2.8.3 Deployment Architecture 28
2.8.4 Data Flow Diagram 29
2.9 Database Design 29
2.9.1 Physical Database Schema 29

2.9.2 Indexing Strategy . . . 30
2.10 Security Architecture . . . 31
2.10.1 Authentication Flow . . . 31
2.10.2 Role-Based Access Control . . . 32
2.11 API Design . . . 32
2.11.1 RESTful Endpoint Summary . . . 32
2.12 Conclusion . . . 32

3 Implementation **33**
3.1 Introduction . . . 33
3.2 Development Environment . . . 33
3.2.1 Hardware Configuration . . . 33
3.2.2 Software Tools . . . 34
3.3 The Golang Agent . . . 34
3.3.1 Project Architecture . . . 34
3.3.2 Collector Interface Pattern . . . 35
3.3.3 Windows Event Log Collection . . . 36
3.3.4 File Tailing Implementation . . . 37
3.3.5 Resilient Sender with Buffering . . . 37
3.3.6 Cross-Compilation Strategy . . . 38
3.4 Backend API Server . . . 38
3.4.1 Project Structure . . . 38
3.4.2 Express Application Setup . . . 39
3.4.3 Authentication Service . . . 40
3.4.4 Threat Detection Engine . . . 40
3.4.5 RAG Chat Implementation . . . 41
3.4.6 Server-Sent Events (SSE) Implementation . . . 41
3.5 Frontend Dashboard . . . 41
3.5.1 Real-time Dashboard Integration . . . 41
3.5.2 Stats Cards Component . . . 42
3.5.3 AI Chat Interface . . . 43
3.6 Database Schema . . . 44
3.7 Conclusion . . . 44

4 Testing and Deployment **45**
4.1 Introduction . . . 45
4.2 Testing Strategy Overview . . . 45
4.3 Unit Testing . . . 47
4.3.1 Threat Detection Engine Tests . . . 47
4.3.2 Authentication Service Tests . . . 48
4.4 Integration Testing . . . 48
4.4.1 API Endpoint Testing . . . 48
4.4.2 SSE Stream Testing . . . 48
4.5 Performance Testing . . . 49
4.5.1 Load Testing Results . . . 49
4.5.2 Performance Benchmarks Summary . . . 49
4.6 Docker Deployment . . . 50
4.6.1 Orchestration with Docker Compose . . . 50
4.6.2 Containerization Strategy . . . 51
4.7 Agent Deployment . . . 52
4.7.1 Automated Installation Scripts . . . 52
4.8 Production Deployment Checklist . . . 52
4.9 Application Screenshots . . . 52
4.10 Conclusion . . . 59

General Conclusion **60**

References **64**

Bibliography **65**

A Source Code Listings **67**
A.1 Golang Agent Code . . . 67
A.2 Backend Code . . . 71
A.3 Frontend Code . . . 78
A.4 Database Schema . . . 81
A.5 Testing and Deployment Code . . . 82

B Glossary **91**

List of Figures

1 Enterprise Log Volume Growth (2015-2025) 1

1.1 Evolution of Log Management Technologies 5
1.2 Elastic Stack (ELK) Architecture 6
1.3 Splunk Enterprise Dashboard Example 7
1.4 SIEM Solutions Radar Comparison 9
1.5 Node.js Event Loop Architecture 10
1.6 PostgreSQL Indexing Strategy for Logs 11
1.7 Binary Size Comparison by Language 12
1.8 Ollama Local LLM Architecture 13
1.9 Scrum Sprint Timeline and Milestones 14

2.1 System Context Diagram with Actors 16
2.2 Global Use Case Diagram 19
2.3 Authentication Sequence Diagram 22
2.4 RAG Chat Workflow Sequence Diagram 23
2.5 Log Ingestion Sequence Diagram 23
2.6 Threat Detection Activity Diagram 24
2.7 Domain Model Class Diagram 25
2.8 Global System Architecture 27
2.9 Component Architecture Diagram 28
2.10 Deployment Architecture Diagram 29
2.11 Data Flow Diagram (DFD Level 1) 29
2.12 PostgreSQL Physical Database Schema 30
2.13 JWT Authentication Architecture 31

3.1 Golang Agent Internal Architecture 35
3.2 Collector Class Hierarchy 36
3.3 Windows Event Log Collection Flow 36
3.4 Sender State Machine 38

3.5 Backend Component Architecture 39
3.6 Threat Detection Pattern Categories 40
3.7 RAG Workflow Detailed Diagram 41
3.8 Dashboard Screenshot 43
3.9 AI Chat Interface Screenshot 44

4.1 Testing Pyramid Strategy 46
4.2 Test Coverage Report 48
4.3 Performance Test Charts 50
4.4 Docker Network Topology 51
4.5 Dashboard Overview 53
4.6 AI Chat Interface 54
4.7 Log Detail View 54
4.8 Admin Dashboard 55
4.9 Login Page Screenshot With Demo Credentials 56
4.10 Admin Log Sources Configuration 56
4.11 Admin AI Settings 57
4.12 Admin Users Management 57
4.13 Admin Analytics Users/Chats/Activity 58
4.14 LogChat Agent On Windows 11 58
4.15 Future Implementation External Alerts Slack/Twilio/SendGrid Mail . . 59

List of Tables

1.1 Elastic Stack Comprehensive Evaluation 6
1.2 Splunk Enterprise Evaluation 6
1.3 Wazuh Platform Evaluation 7
1.4 Microsoft Sentinel Evaluation 7
1.5 Comprehensive SIEM Solution Comparison 8
1.6 Frontend Technology Comparison 10
1.7 Agent Runtime Comparison 12
1.8 Supported LLM Models 13

2.1 System Actors and Responsibilities 15
2.2 Golang Agent Functional Requirements 16
2.3 Backend Server Functional Requirements 17
2.4 Frontend Functional Requirements 17
2.5 Non-Functional Requirements Specification 18
2.6 Use Case UC-01: User Authentication 20
2.7 Use Case UC-04: Chat with AI Assistant 20
2.8 Use Case Prioritization (MoSCoW) 21
2.9 Entity Attribute Specifications 26
2.10 Database Indexing Strategy 30
2.11 RBAC Permission Matrix 32
2.12 API Endpoint Overview 32

3.1 Development Environment Specifications 33
3.2 Development Tools and Versions 34

4.1 Testing Strategy by Component 47
4.2 Threat Detection Test Cases 47
4.3 Performance Benchmark Results 49
4.4 Production Deployment Checklist 52

4.5 Project Objectives Fulfillment Matrix 61
4.6 Technical Challenges and Resolutions 62
4.7 LogChat Development Roadmap 62

List of Abbreviations

Abbreviation	Definition
AI	Artificial Intelligence
API	Application Programming Interface
CRUD	Create, Read, Update, Delete
CSS	Cascading Style Sheets
DB	Database
DFD	Data Flow Diagram
Docker	Container Runtime Platform
ERD	Entity-Relationship Diagram
GDPR	General Data Protection Regulation
HIPAA	Health Insurance Portability and Accountability Act
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HTTP Secure
IDS	Intrusion Detection System
JSON	JavaScript Object Notation
JWT	JSON Web Token
KQL	Kusto Query Language
LLM	Large Language Model
NLP	Natural Language Processing
ORM	Object-Relational Mapping
PCI-DSS	Payment Card Industry Data Security Standard
RBAC	Role-Based Access Control
RAG	Retrieval Augmented Generation
REST	Representational State Transfer
SIEM	Security Information and Event Management
SOC	Security Operations Center
SPL	Search Processing Language (Splunk)
SQL	Structured Query Language
SSE	Server-Sent Events
TLS	Transport Layer Security
UI	User Interface
UML	Unified Modeling Language

Abbreviation	Definition
UUID	Universally Unique Identifier
UX	User Experience
XSS	Cross-Site Scripting
YAML	YAML Ain't Markup Language

General Introduction

Context and Motivation

In the contemporary digital landscape, cybersecurity has emerged as one of the most critical challenges facing organizations worldwide. According to IBM's Cost of a Data Breach Report 2024 [1], the average cost of a data breach reached $4.88 million globally, with organizations taking an average of 277 days to identify and contain a breach. In this context, effective log management and security monitoring have become indispensable components of any organization's security posture.

Application logs constitute the primary source of truth for system health, security posture, and operational intelligence. The average enterprise generates terabytes of log data daily across its infrastructure — from web servers and databases to authentication systems and network devices. Figure 1 illustrates the exponential growth in enterprise log volumes over the past decade.

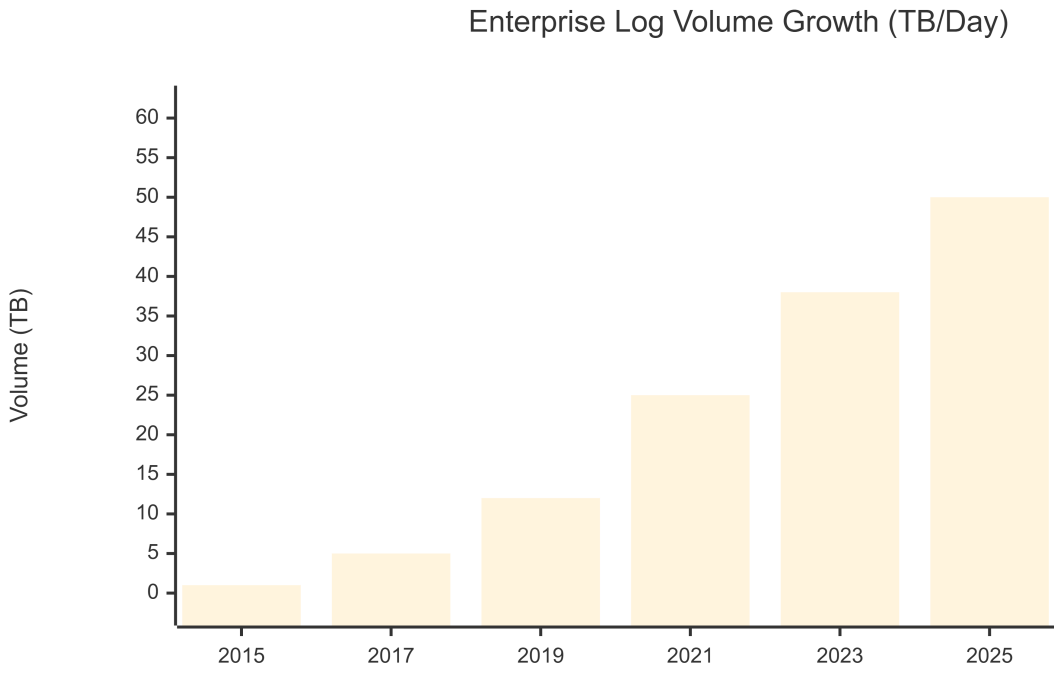

Figure 1: Enterprise Log Volume Growth (2015-2025)

This data, when properly analyzed, reveals critical insights: performance bottle-

necks, security breaches, compliance violations, and emerging threats. However, the sheer velocity, volume, and heterogeneity of logs generated by modern infrastructure create what security professionals term the "data noise" problem.

Problem Statement

Small to Medium Enterprises (SMEs) and resource-constrained security teams face a critical "Security Gap" characterized by three fundamental barriers:

The Three Barriers to Effective Log Management

1. **Commercial Barriers:** Enterprise SIEM solutions like Splunk [4], IBM QRadar, and Microsoft Sentinel command licensing fees ranging from $50,000 to $500,000+ annually, effectively pricing out smaller organizations from effective security monitoring.

2. **Technical Skill Gap:** Effective log analysis requires proficiency in proprietary query languages (SPL for Splunk, KQL for Azure Sentinel, Lucene for Elasticsearch). Junior analysts and developers often lack this specialized knowledge, creating a dependency on expensive senior talent.

3. **Alert Fatigue:** Traditional rule-based detection systems generate excessive false positives, overwhelming security teams and causing genuine threats to be overlooked. Studies indicate that SOC analysts spend up to 45% of their time on false positive investigations.

Project Objectives

LogChat was conceived to bridge this gap by building an open-source, AI-first log management platform with the following objectives:

1. **Unified Collection:** Develop a single-binary agent (Golang) compatible with Windows Event Logs, Linux Syslog/Journald, and file-based logs — deployable across heterogeneous infrastructure without runtime dependencies.

2. **Intelligent Analysis:** Replace complex query languages with Natural Language Chat, powered by local Large Language Models (LLMs) for privacy-preserving inference using the RAG pattern [8].

3. **Real-time Visualization:** Provide an interactive, streaming dashboard for immediate situational awareness with sub-second update latency.

4. **Zero-Configuration Deployment:** Enable complete platform deployment via a single `docker-compose up` command, achieving operational status within five minutes.

5. **Privacy-First Design:** Ensure that sensitive log data never leaves the organization's infrastructure by utilizing local AI inference.

Document Structure

This report is organized into five chapters, each addressing a specific phase of the software development lifecycle:

- **Chapter 1: State of the Art** presents a comprehensive analysis of existing SIEM solutions, evaluates technological alternatives, and provides justification for the selected technology stack.
- **Chapter 2: Analysis & Specification** details the functional and non-functional requirements, presents UML modeling artifacts including use case, sequence, and class diagrams, and describes the architectural design decisions.
- **Chapter 3: Implementation** provides a technical deep-dive into the development of the Golang Agent, Backend API, RAG Engine, and Frontend components with annotated code excerpts.
- **Chapter 4: Testing & Deployment** outlines the quality assurance strategy, describes the Docker-based deployment architecture, and presents performance benchmarks.
- **General Conclusion** summarizes achievements, reflects on lessons learned, and outlines the future roadmap for LogChat development.

Chapter 1

State of the Art

1.1 Introduction

Before embarking on the development of LogChat, a comprehensive analysis of the current log management and SIEM landscape was essential. This chapter surveys existing solutions, evaluates technological alternatives, and provides the rationale for our architectural decisions. The insights gathered during this phase directly influenced the design patterns and technology choices implemented in LogChat.

1.2 Historical Evolution of Log Management

1.2.1 Early Approaches (1990s-2000s)

In the early days of computing, log management was a manual process. System administrators would periodically review text files using command-line tools such as `grep`, `awk`, and `tail`. Figure 1.1 presents the evolution of log management technologies.

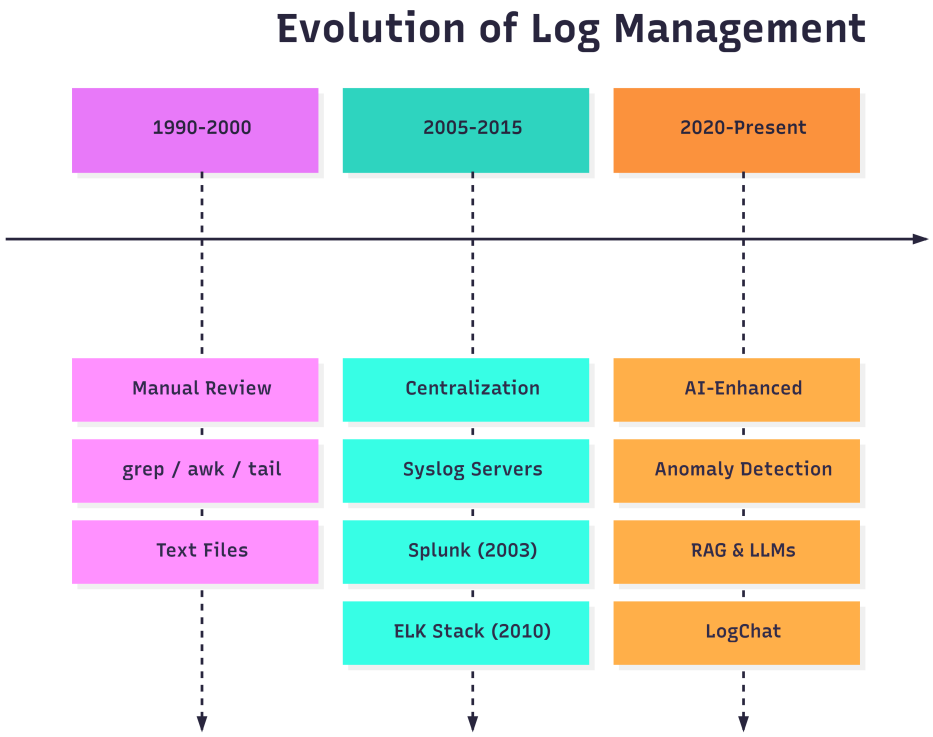

Figure 1.1: Evolution of Log Management Technologies

1.2.2 Centralized Logging (2005-2015)

The emergence of distributed systems necessitated centralized logging solutions. Syslog servers became standard, and tools like Splunk (2003) and the ELK Stack (2010) emerged to address the growing complexity.

1.2.3 Modern AI-Enhanced SIEM (2020-Present)

The current generation of SIEM solutions leverages machine learning for anomaly detection and, increasingly, large language models for natural language querying. LogChat positions itself at the forefront of this evolution.

1.3 Market Analysis of Existing Solutions

1.3.1 Elastic Stack (ELK)

The Elastic Stack — comprising Elasticsearch, Logstash, and Kibana — represents the industry standard for open-source log management [5]. Figure 1.2 illustrates its

architecture.

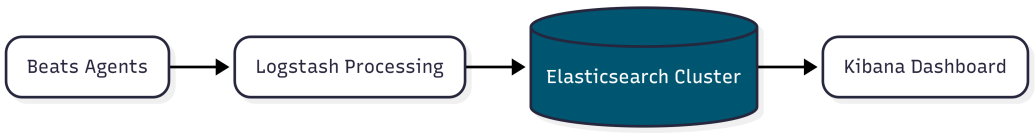

Figure 1.2: Elastic Stack (ELK) Architecture

Table 1.1: Elastic Stack Comprehensive Evaluation

Strengths	Highly scalable horizontal architecture, powerful Kibana visualization, extensive plugin ecosystem, large community support
Weaknesses	Resource-intensive JVM-based architecture, complex cluster management, steep learning curve for Lucene/DSL queries
Hardware Requirements	Minimum 3 nodes, 16GB+ RAM per node, SSD storage recommended
Licensing	Open Source (Basic), Commercial (Platinum: $125/node/month)

1.3.2 Splunk Enterprise

Splunk remains the enterprise leader in SIEM, commanding over 30% market share in the security analytics segment [4].

Table 1.2: Splunk Enterprise Evaluation

Strengths	Comprehensive ecosystem, advanced ML-based analytics, excellent enterprise support, extensive app marketplace
Weaknesses	Proprietary platform, extremely expensive pricing model, vendor lock-in concerns
Pricing Model	$1,800+ per GB/day ingested (perpetual), or subscription model
Annual Cost (10GB/day)	Approximately $180,000 - $250,000

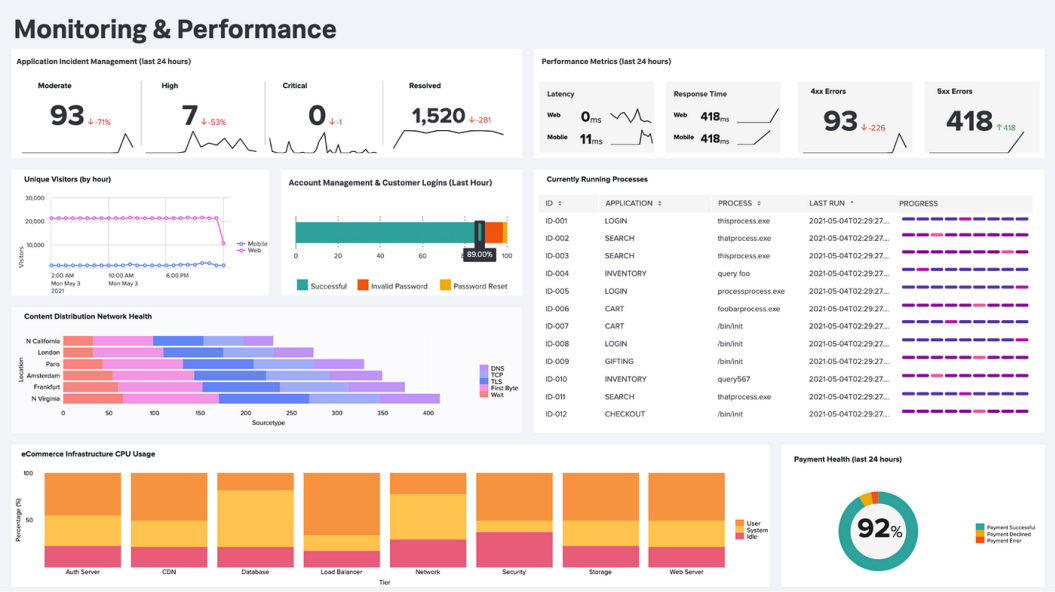

Figure 1.3: Splunk Enterprise Dashboard Example

1.3.3 Wazuh

Wazuh is an open-source security platform focusing on intrusion detection, file integrity monitoring, and compliance.

Table 1.3: Wazuh Platform Evaluation

Strengths	Strong Host-based IDS capabilities, file integrity monitoring, MITRE ATT&CK mapping [3], regulatory compliance modules
Weaknesses	Complex UI/UX, agent-heavy architecture, challenging customization, limited NLP capabilities
Deployment	Requires Elasticsearch backend, complex multi-component setup

1.3.4 Microsoft Sentinel

Microsoft Sentinel provides cloud-native SIEM integrated with the Azure ecosystem.

Table 1.4: Microsoft Sentinel Evaluation

Strengths	Cloud-native scalability, tight Azure integration, SOAR capabilities, KQL query language
Weaknesses	Azure lock-in, complex pricing, limited on-premises support, data residency concerns
Pricing	Pay-per-use: \$2.46/GB ingested + \$0.10/GB retained

1.3.5 Comparative Analysis

Table 1.5 provides a comprehensive comparison of evaluated solutions against LogChat's proposed capabilities.

Table 1.5: Comprehensive SIEM Solution Comparison

Criterion	**ELK**	**Splunk**	**Wazuh**	**Sentinel**	**LogChat**
Open Source	✓	✗	✓	✗	✓
Natural Language Query	✗	Partial	✗	✗	✓
Single-Binary Agent	✗	✗	✗	N/A	✓
Local AI (Privacy)	✗	✗	✗	✗	✓
One-Command Deploy	✗	✗	✗	✗	✓
Real-time Streaming	✓	✓	✓	✓	✓
MITRE ATT&CK Mapping	Plugin	✓	✓	✓	✓
Min. RAM Requirement	16GB	8GB	8GB	N/A	4GB

Figure 1.4 presents a radar chart visualization of the comparison.

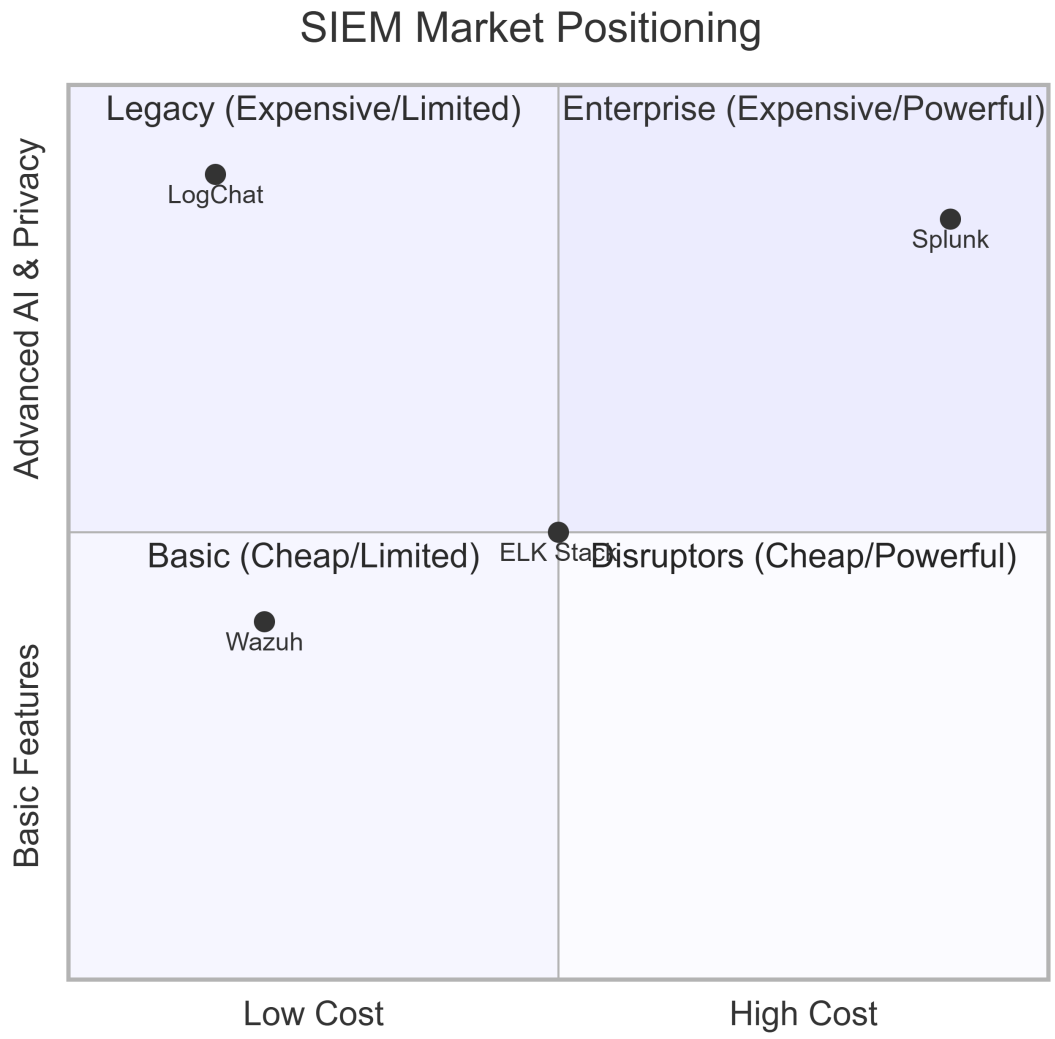

Figure 1.4: SIEM Solutions Radar Comparison

1.4 Technology Stack Selection

Based on the market analysis and project requirements, the following technology stack was selected. Each choice was driven by specific technical and business considerations.

1.4.1 Backend: Node.js with TypeScript

Node.js was selected for the backend API server for the following reasons:

Node.js Selection Rationale

- **Event-Driven Architecture:** Non-blocking I/O model perfectly suited for handling high-concurrency log ingestion and SSE streaming
- **npm Ecosystem:** Access to mature libraries including Express,

Prisma [10], and Zod

- **TypeScript Integration:** Compile-time type safety critical for maintaining data integrity
- **Shared Language:** Same language as frontend reduces context switching and enables code sharing

Figure 1.5 illustrates the Node.js event loop architecture.

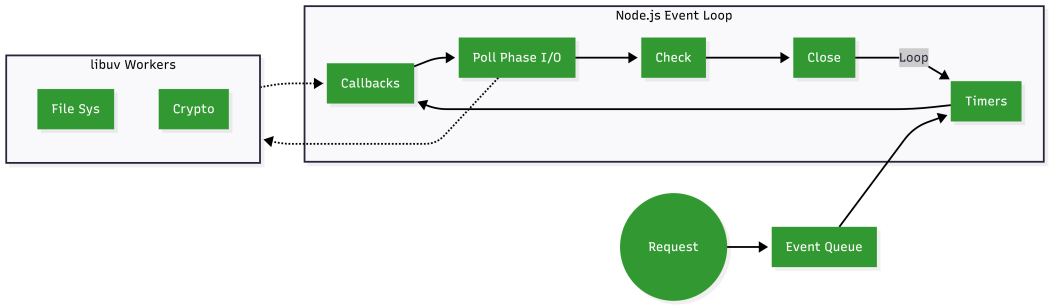

Figure 1.5: Node.js Event Loop Architecture

1.4.2 Frontend: Next.js 14 with React 18

Next.js 14 was chosen for its hybrid rendering capabilities and modern React features [9].

Table 1.6: Frontend Technology Comparison

Feature	Next.js	Remix	Vite+React	Angular
SSR/SSG	✓	✓	Plugin	✓
App Router	✓	✓	✗	✓
Bundle Size	Small	Small	Custom	Large
Learning Curve	Moderate	Moderate	Low	High
Community	Large	Growing	Large	Large

1.4.3 Database: PostgreSQL 16 with Prisma ORM

PostgreSQL was selected as the primary database for its robustness and advanced features [11].

PostgreSQL 16 Key Features

- **JSONB Columns:** Native support for semi-structured log metadata
- **Full-Text Search:** Built-in text search for log message querying
- **Time-Series Indexing:** Efficient B-tree and BRIN indexes for timestamp queries
- **pgvector Compatibility:** Future-proof for semantic search implementation
- **ACID Compliance:** Guaranteed data integrity for audit requirements

Figure 1.6 illustrates the indexing strategy employed for log queries.

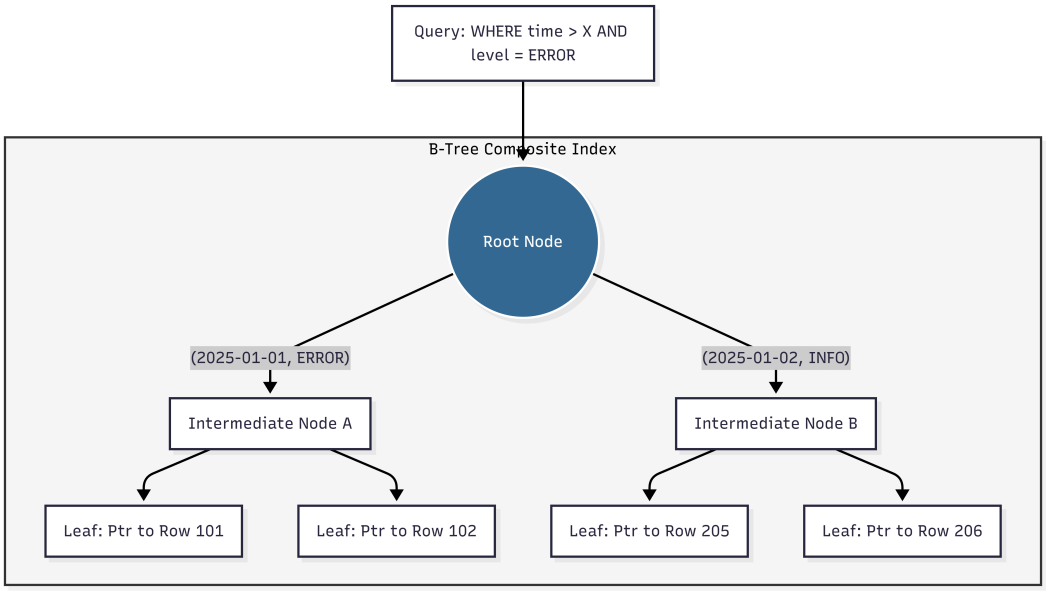

Figure 1.6: PostgreSQL Indexing Strategy for Logs

1.4.4 Agent: Golang

For the distributed log collector, Golang was selected over Python, Java, and Rust [12]:

Table 1.7: Agent Runtime Comparison

Feature	Python	Java	Rust	Go
Single Binary	✗	✗	✓	✓
No Runtime Dependencies	✗	✗	✓	✓
Windows API Access	Complex	Complex	Moderate	✓
Memory Footprint	High	High	Low	Low
Goroutine/Async	asyncio	Threads	async	✓
Cross-Compilation	Complex	Complex	Moderate	Built-in
Learning Curve	Low	Moderate	High	Moderate

Figure 1.7 compares binary sizes across languages for equivalent functionality.

Figure 1.7: Binary Size Comparison by Language

1.4.5 AI Engine: Ollama with Local LLMs

To ensure data privacy, cloud-based AI APIs were ruled out for sensitive environments. Ollama enables running quantized models locally without internet connectivity [6].

Table 1.8: Supported LLM Models

Model	Parameters	VRAM Required	Use Case
Qwen 2.5:0.5b	0.5B	1GB	Fast responses, basic analysis
Qwen 2.5:3b	3B	4GB	Balanced performance
Llama 3.1:8b	8B	8GB	Complex analysis
Mistral 7B	7B	6GB	European compliant

Figure 1.8 shows the Ollama integration architecture.

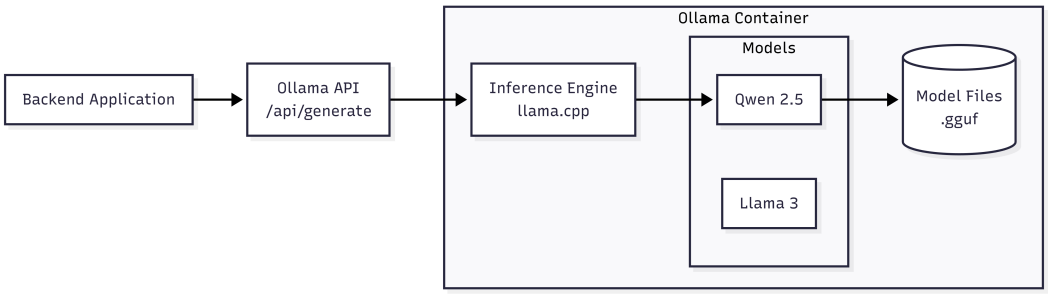

Figure 1.8: Ollama Local LLM Architecture

1.5 Development Methodology

The project followed **Scrum**, an Agile methodology well-suited for iterative development:

- **Sprint Duration:** 1 week
- **Total Sprints:** 12 (3 months)
- **Team Size:** 2 developers
- **Tools:** Git, GitHub, Docker, VS Code

Figure 1.9 presents the sprint timeline and deliverables.

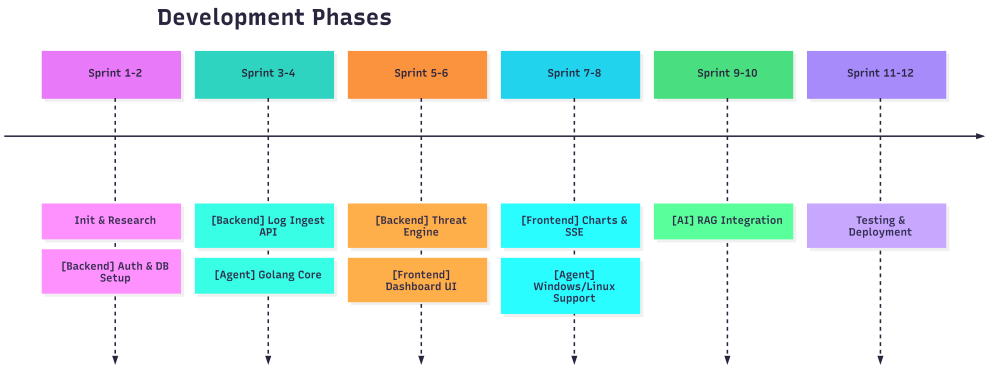

Figure 1.9: Scrum Sprint Timeline and Milestones

1.6 Conclusion

This state-of-the-art analysis confirmed the viability of building a competitive, open-source SIEM alternative. The selected technology stack balances developer productivity, runtime performance, and deployment simplicity. The following chapter details the functional and non-functional requirements derived from this analysis.

Chapter 2

Analysis and Specification

2.1 Introduction

This chapter translates business requirements into formal technical specifications using UML (Unified Modeling Language). We present the actors, use cases, data models, sequence diagrams, and architectural blueprints that guide the implementation phase. The artifacts produced in this chapter serve as the contractual specification between stakeholders.

2.2 Actors Identification

The system interacts with three primary actors, each with distinct roles and permissions:

Table 2.1: System Actors and Responsibilities

Actor	**Type**	**Responsibilities**
Security Analyst	Human (Primary)	Monitor dashboards, investigate alerts, query logs via AI chat, export reports
System Administrator	Human (Primary)	Configure log sources, manage users, define alert rules, review audit trails
LogChat Agent	System (Automated)	Collect logs from sources, transmit to server, buffer during outages

Figure 2.1 presents the context diagram showing actor interactions.

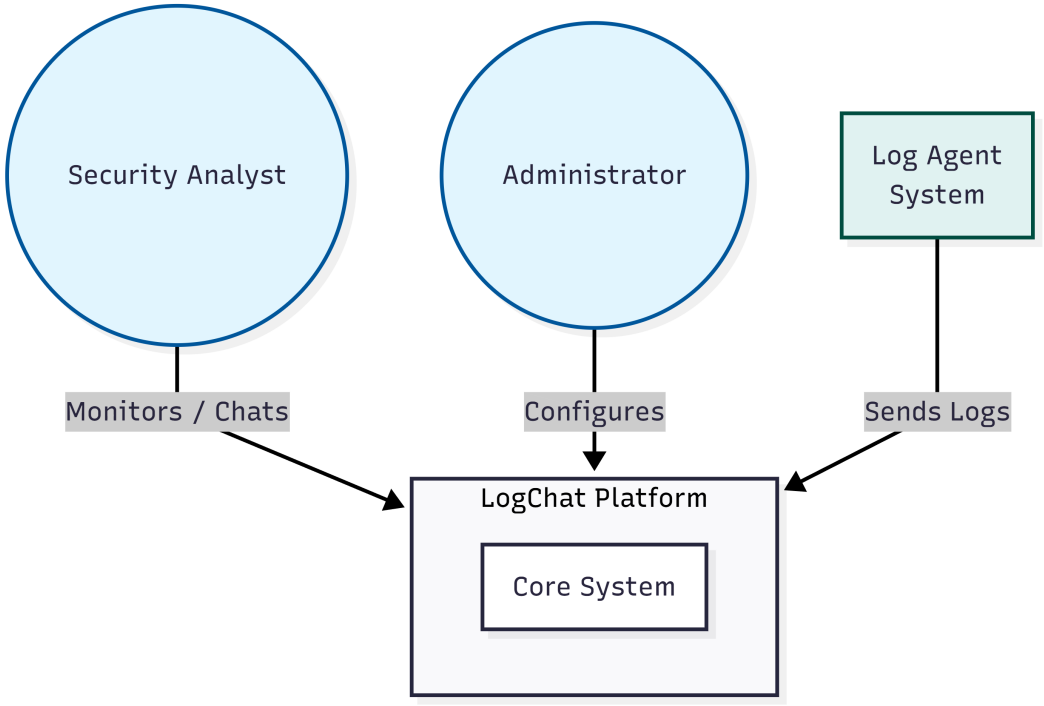

Figure 2.1: System Context Diagram with Actors

2.3 Functional Requirements

2.3.1 Agent Requirements (RF-AGT)

Table 2.2: Golang Agent Functional Requirements

ID	Requirement Description	Priority
RF-AGT-01	Compile to static binaries for Windows (amd64, 386) and Linux (amd64, arm64)	High
RF-AGT-02	Monitor log files using tail-like behavior (follow mode with rotation support)	High
RF-AGT-03	Collect Windows Event Logs from Application, System, Security channels	High
RF-AGT-04	Collect Linux logs via Journald (systemd) and traditional Syslog	High
RF-AGT-05	Buffer logs locally using ring buffer if server is unreachable (resilience)	Medium
RF-AGT-06	Authenticate with server via X-API-Key header	High
RF-AGT-07	Support JSON, regex, and raw log parsing formats	Medium
RF-AGT-08	Provide health endpoint for monitoring agent status	Low
RF-AGT-09	Support hot configuration reload without restart	Low

2.3.2 Backend Server Requirements (RF-SRV)

Table 2.3: Backend Server Functional Requirements

ID	Requirement Description	Priority
RF-SRV-01	Provide REST API endpoints for single and batch log ingestion	High
RF-SRV-02	Implement real-time threat detection using regex pattern matching	High
RF-SRV-03	Support natural language queries via AI chat endpoint with RAG context	High
RF-SRV-04	Provide SSE (Server-Sent Events) for real-time dashboard updates	High
RF-SRV-05	Implement JWT-based authentication with configurable expiry	High
RF-SRV-06	Enforce role-based access control (ADMIN, STAFF, USER)	High
RF-SRV-07	Maintain immutable audit logs for all administrative actions	Medium
RF-SRV-08	Support log export in CSV and JSON formats	Medium
RF-SRV-09	Provide API rate limiting per source	Medium
RF-SRV-10	Support multiple AI providers (Ollama, OpenAI, Anthropic, Gemini)	Medium

2.3.3 Frontend Requirements (RF-UI)

Table 2.4: Frontend Functional Requirements

ID	Requirement Description	Priority
RF-UI-01	Display real-time statistics cards (total logs, errors, warnings, threats)	High
RF-UI-02	Provide interactive time-series chart for log volume visualization	High
RF-UI-03	Implement paginated log table with inline expansion	High
RF-UI-04	Support filtering by time range, log level, service name, and text search	High
RF-UI-05	Provide AI chat interface with markdown rendering	High
RF-UI-06	Display toast notifications for real-time alerts	Medium
RF-UI-07	Implement dark mode toggle	Low
RF-UI-08	Provide responsive design for tablet devices	Medium

2.4 Non-Functional Requirements

Table 2.5: Non-Functional Requirements Specification

Category	**ID**	**Requirement**	**Target**
Performance	RNF-01	Log ingestion throughput	1000+ req/s
Performance	RNF-02	Dashboard update latency	$<$ 500ms
Performance	RNF-03	AI response time (local)	$<$ 5 seconds
Availability	RNF-04	System uptime	99.9%
Security	RNF-05	Password hashing algorithm	bcrypt (cost 12)
Security	RNF-06	API authentication mechanism	JWT (RS256)
Security	RNF-07	TLS version for external APIs	TLS 1.2+
Portability	RNF-08	Deployment method	Docker Compose [13]
Portability	RNF-09	Supported agent platforms	Windows, Linux
Usability	RNF-10	Dashboard responsiveness	Desktop, Tablet
Maintainability	RNF-11	Code documentation coverage	80%+

2.5 Use Case Modeling

2.5.1 Global Use Case Diagram

Figure 2.2 presents the comprehensive use case diagram depicting all actors and their interactions with the LogChat platform.

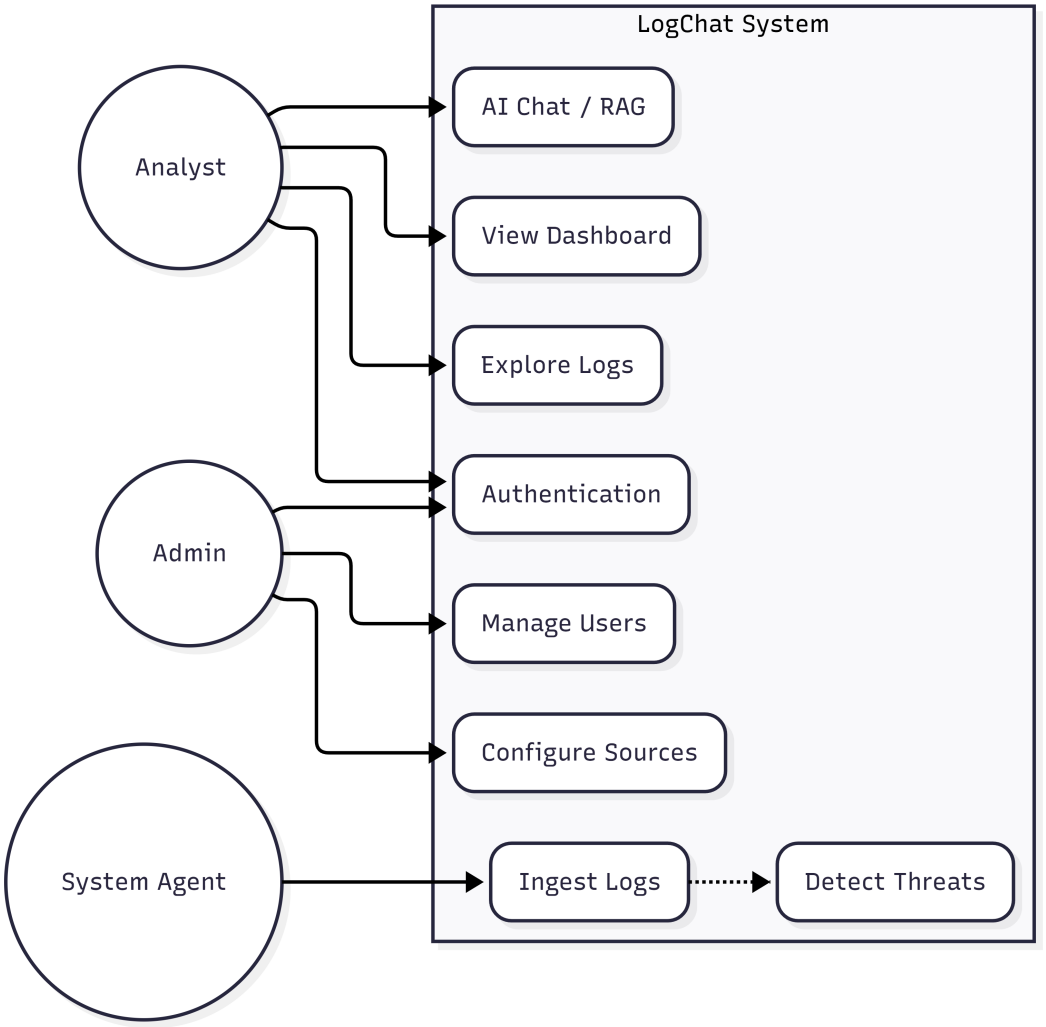

Figure 2.2: Global Use Case Diagram

2.5.2 Detailed Use Case Descriptions

UC-01: User Authentication

Table 2.6: Use Case UC-01: User Authentication

Use Case ID	UC-01
Name	User Authentication
Actor	Security Analyst, Administrator
Precondition	User has valid credentials in the system
Main Flow	1. User navigates to login page 2. User enters email and password 3. System validates credentials 4. System generates JWT token 5. System creates session record 6. User is redirected to dashboard
Alternative Flow	A1. Invalid credentials: Display error, remain on login A2. Account disabled: Display account status message
Postcondition	User is authenticated with valid session

UC-04: Chat with AI Assistant

Table 2.7: Use Case UC-04: Chat with AI Assistant

Use Case ID	UC-04
Name	Chat with AI Assistant
Actor	Security Analyst
Precondition	User is authenticated; AI service is available
Main Flow	1. User navigates to Chat interface 2. User enters natural language query 3. System retrieves relevant logs (RAG) 4. System constructs context-enriched prompt 5. System sends prompt to LLM 6. LLM generates analysis response 7. System displays formatted response
Alternative Flow	A1. AI unavailable: Display offline message with suggestions
Postcondition	Chat message and response are persisted

2.5.3 Use Case Prioritization Matrix

Table 2.8: Use Case Prioritization (MoSCoW)

Use Case	Category	Sprint
UC-01: Authentication	Must Have	1-2
UC-02: View Dashboard	Must Have	3-4
UC-03: Explore Logs	Must Have	3-4
UC-04: AI Chat	Must Have	5-6
UC-10: Ingest Logs	Must Have	2-3
UC-11: Threat Detection	Should Have	4-5
UC-06: Manage Users	Should Have	7-8
UC-05: Export Reports	Could Have	9

2.6 Behavioral Modeling

2.6.1 Sequence Diagram: Authentication Flow

Figure 2.3 details the complete authentication sequence from user input to dashboard redirect.

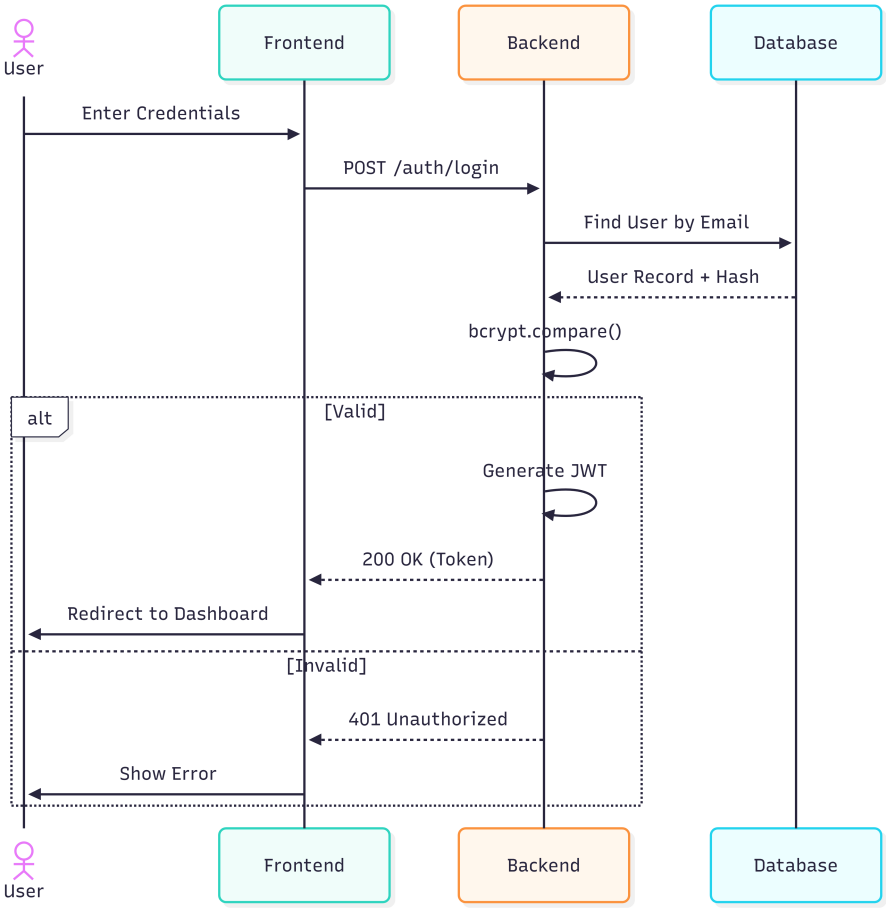

Figure 2.3: Authentication Sequence Diagram

2.6.2 Sequence Diagram: RAG Chat Workflow

Figure 2.4 illustrates the RAG (Retrieval Augmented Generation) process, which is central to LogChat's AI capabilities.

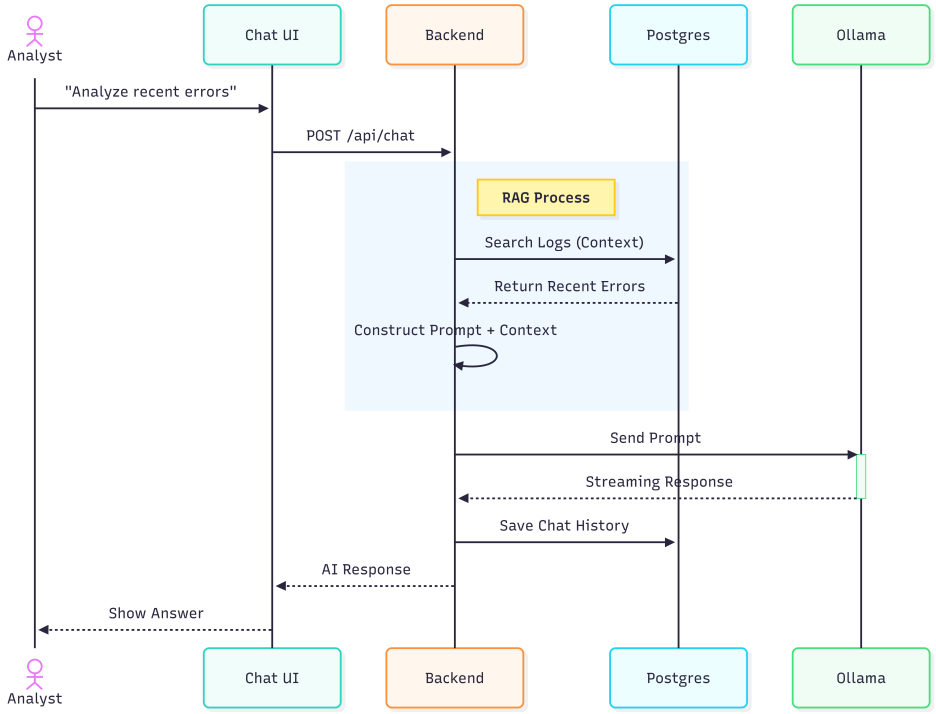

Figure 2.4: RAG Chat Workflow Sequence Diagram

2.6.3 Sequence Diagram: Log Ingestion with Threat Detection

Figure 2.5 presents the complete flow from agent transmission to threat detection and dashboard notification.

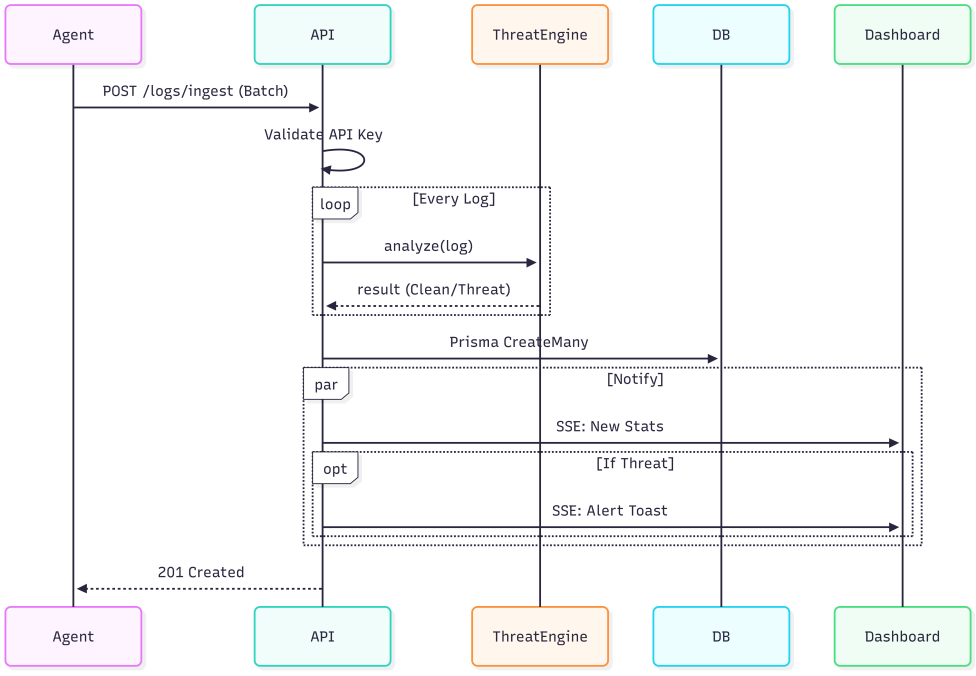

Figure 2.5: Log Ingestion Sequence Diagram

2.6.4 Activity Diagram: Threat Detection Engine

Figure 2.6 models the decision flow within the threat detection engine.

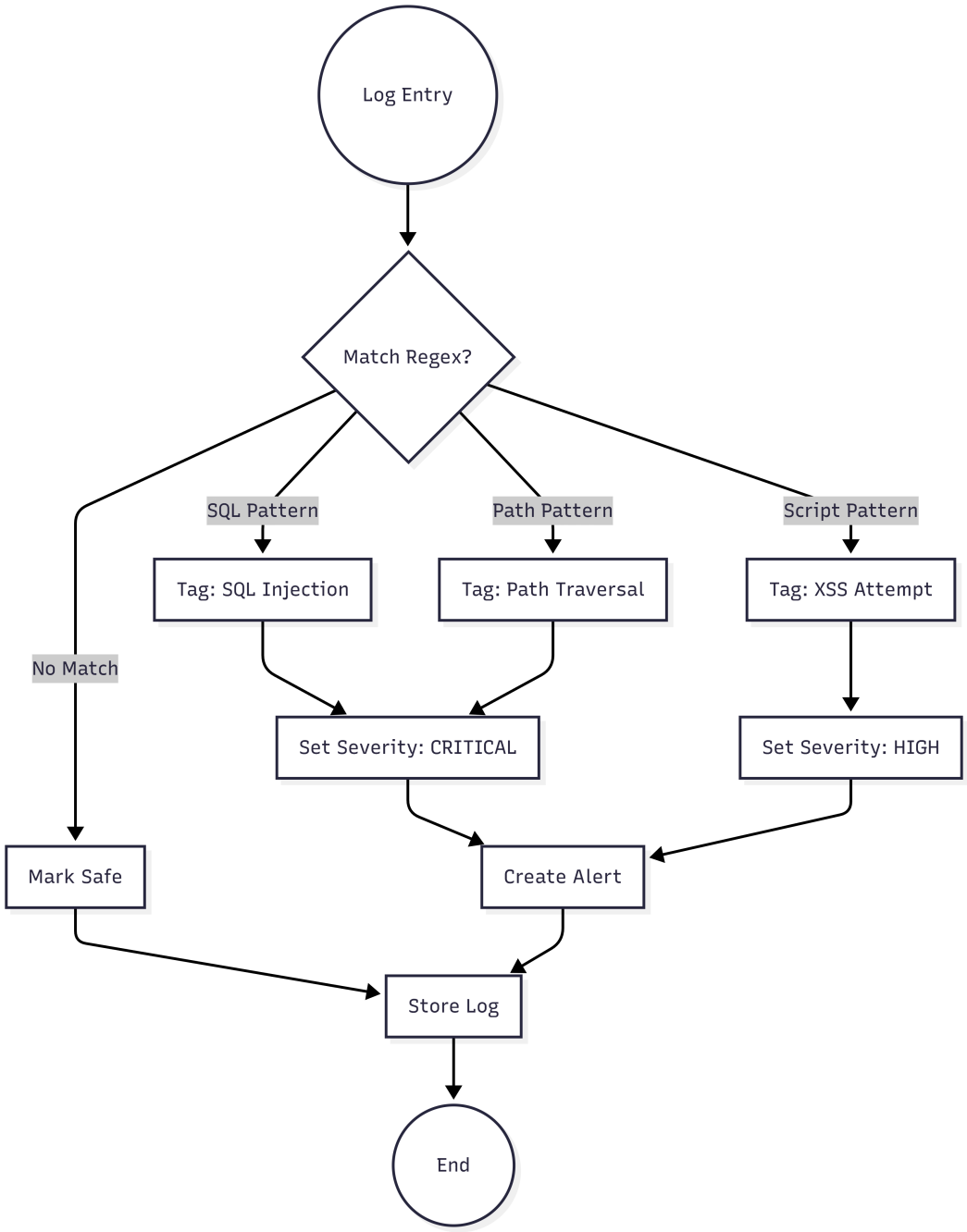

Figure 2.6: Threat Detection Activity Diagram

2.7 Structural Modeling

2.7.1 Class Diagram: Domain Model

Figure 2.7 presents the domain model with all entities, attributes, and relationships.

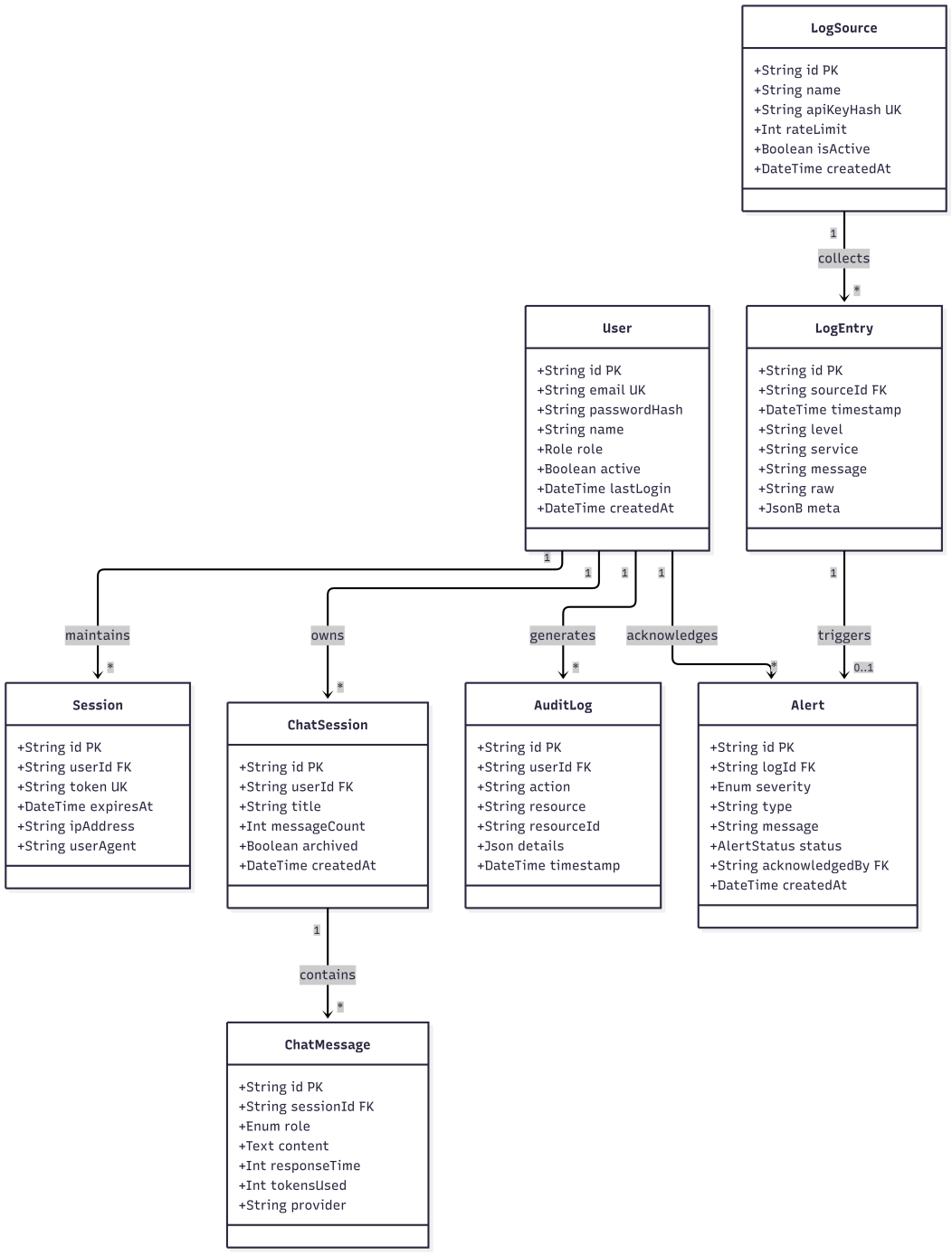

Figure 2.7: Domain Model Class Diagram

2.7.2 Entity Descriptions

Table 2.9: Entity Attribute Specifications

Entity	Key Attributes
User	id (CUID), email (unique), passwordHash, name, role (enum), active, lastLogin
Session	id, token (unique), userId (FK), expiresAt, ipAddress, userAgent
LogEntry	id, timestamp, level (enum), service, message, raw, meta (JSONB)
ChatSession	id, userId (FK), title, messageCount, archived, createdAt
ChatMessage	id, sessionId (FK), role (enum), content, responseTime, tokensUsed
Alert	id, logId (FK), severity (enum), type, message, status, acknowledgedBy
LogSource	id, name, apiKey (unique), apiKeyHash, rateLimit, isActive
AuditLog	id, userId (FK), action, resource, resourceId, details (JSON), timestamp

2.8 Architectural Design

2.8.1 High-Level Architecture

Figure 2.8 presents the global system architecture showing all components and their interactions.

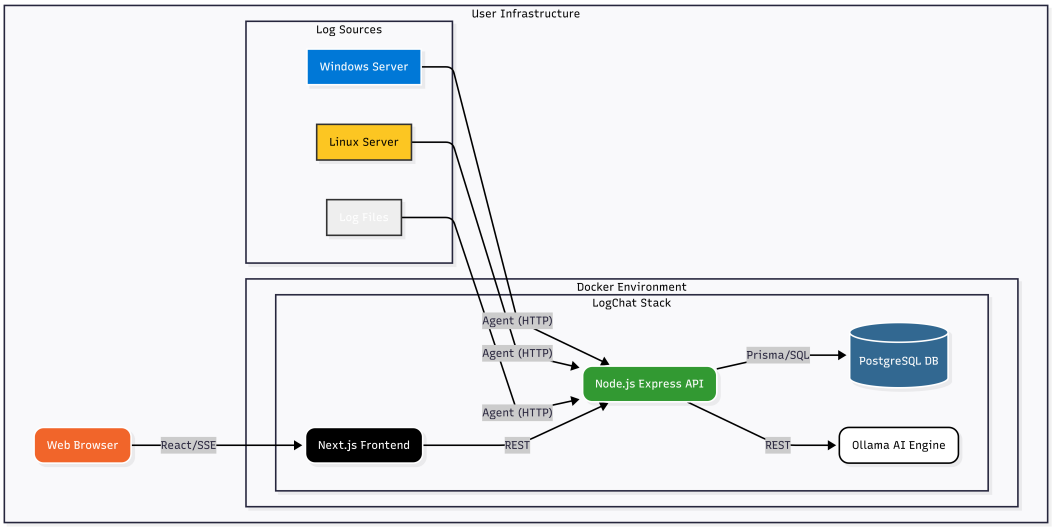

Figure 2.8: Global System Architecture

2.8.2 Component Architecture

Figure 2.9 details the internal structure of each containerized service.

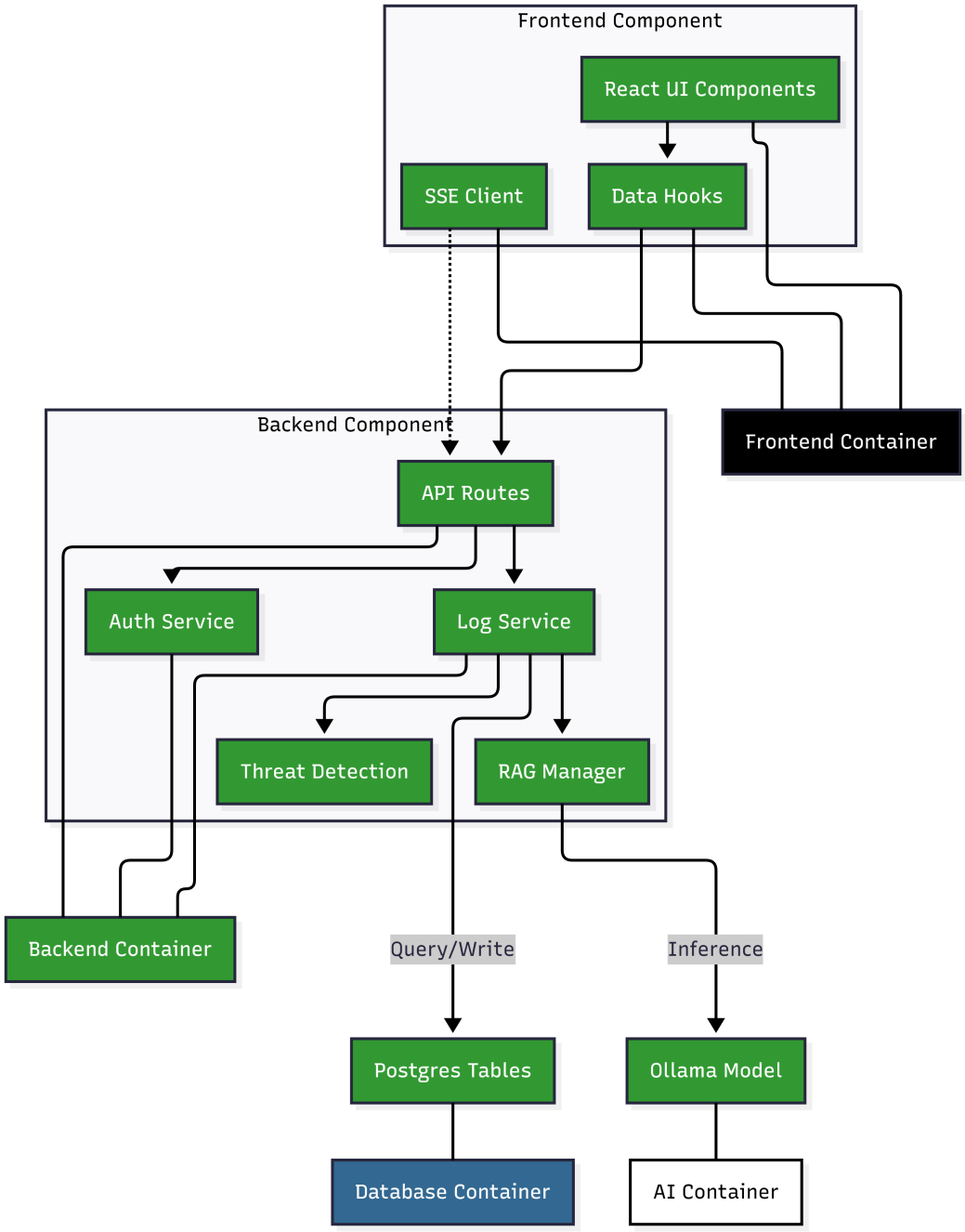

Figure 2.9: Component Architecture Diagram

2.8.3 Deployment Architecture

Figure 2.10 illustrates the production deployment topology.

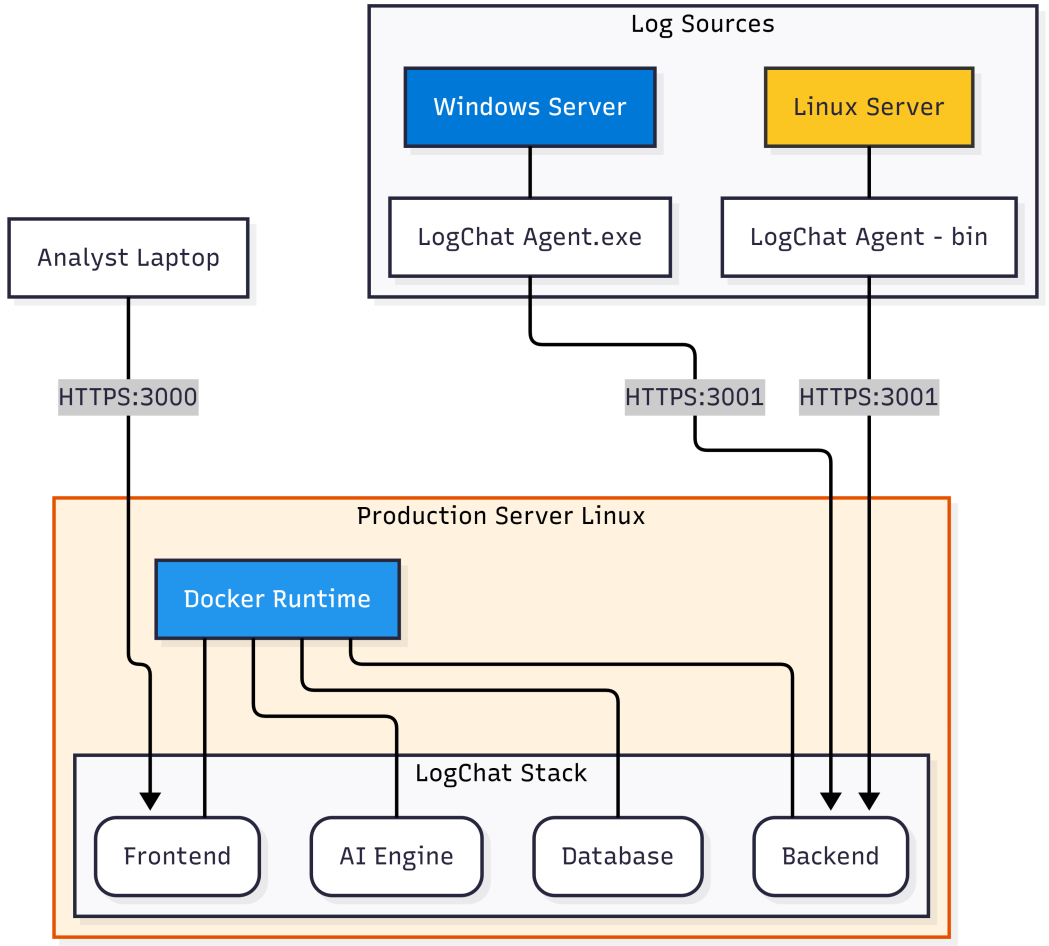

Figure 2.10: Deployment Architecture Diagram

2.8.4 Data Flow Diagram

Figure 2.11 presents the end-to-end data flow from log sources to end users.

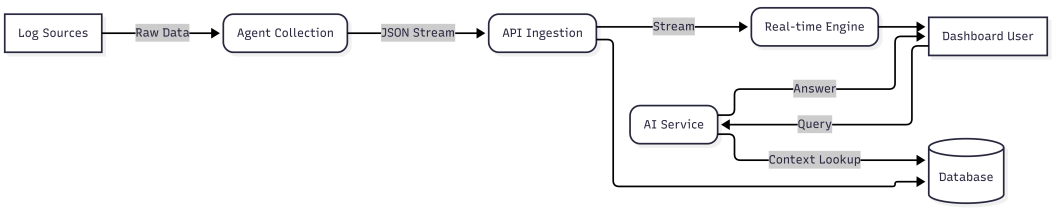

Figure 2.11: Data Flow Diagram (DFD Level 1)

2.9 Database Design

2.9.1 Physical Database Schema

Figure 2.12 presents the physical database schema with all tables and relationships.

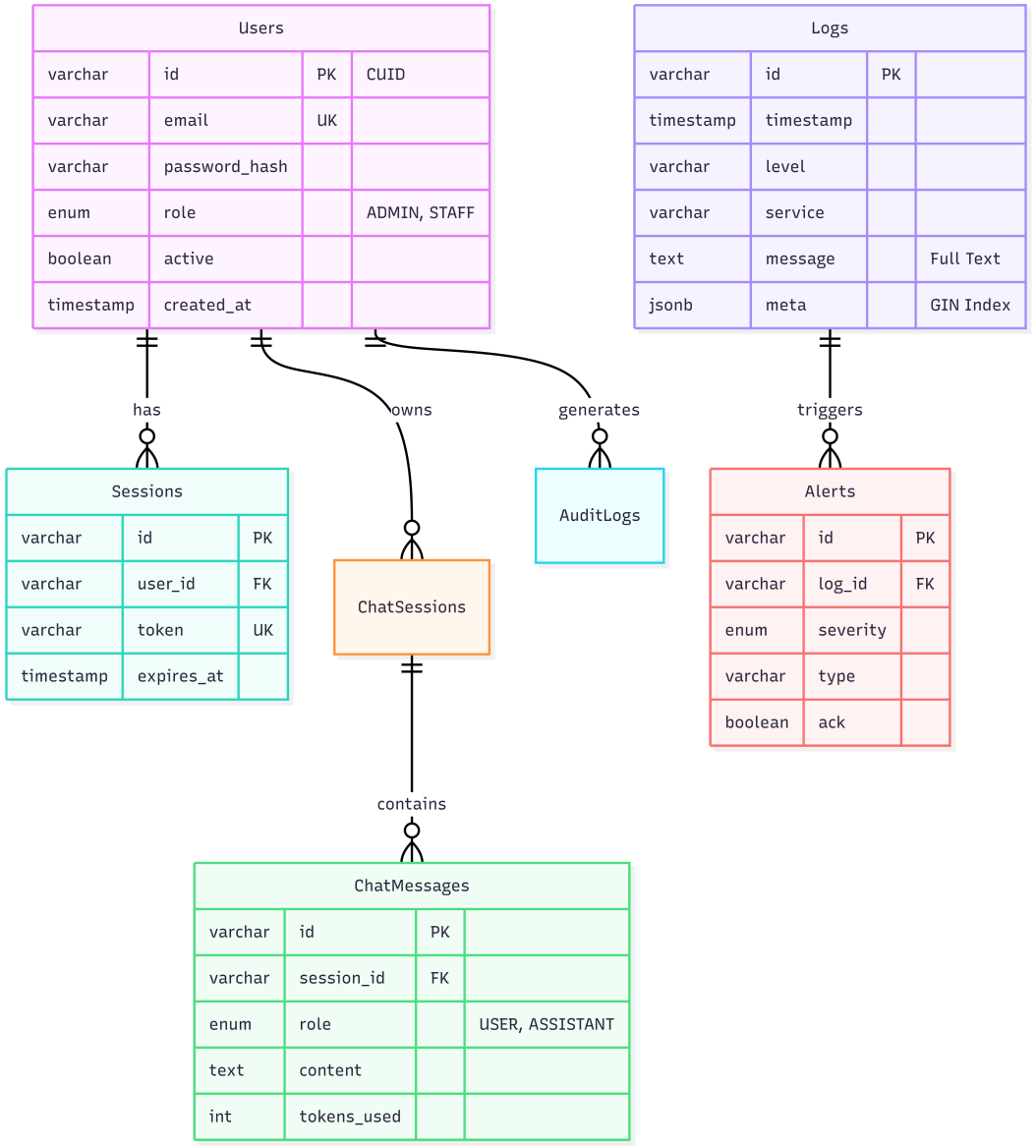

Figure 2.12: PostgreSQL Physical Database Schema

2.9.2 Indexing Strategy

Table 2.10: Database Indexing Strategy

Table	Index	Purpose
logs	idx_logs_timestamp	Time-range queries
logs	idx_logs_level	Filter by severity
logs	idx_logs_service	Filter by service
logs	idx_logs_composite	Combined filtering
users	idx_users_email	Unique login lookup
sessions	idx_sessions_token	Token validation
alerts	idx_alerts_status	Dashboard filtering

2.10 Security Architecture

2.10.1 Authentication Flow

Figure 2.13 illustrates the JWT-based authentication architecture.

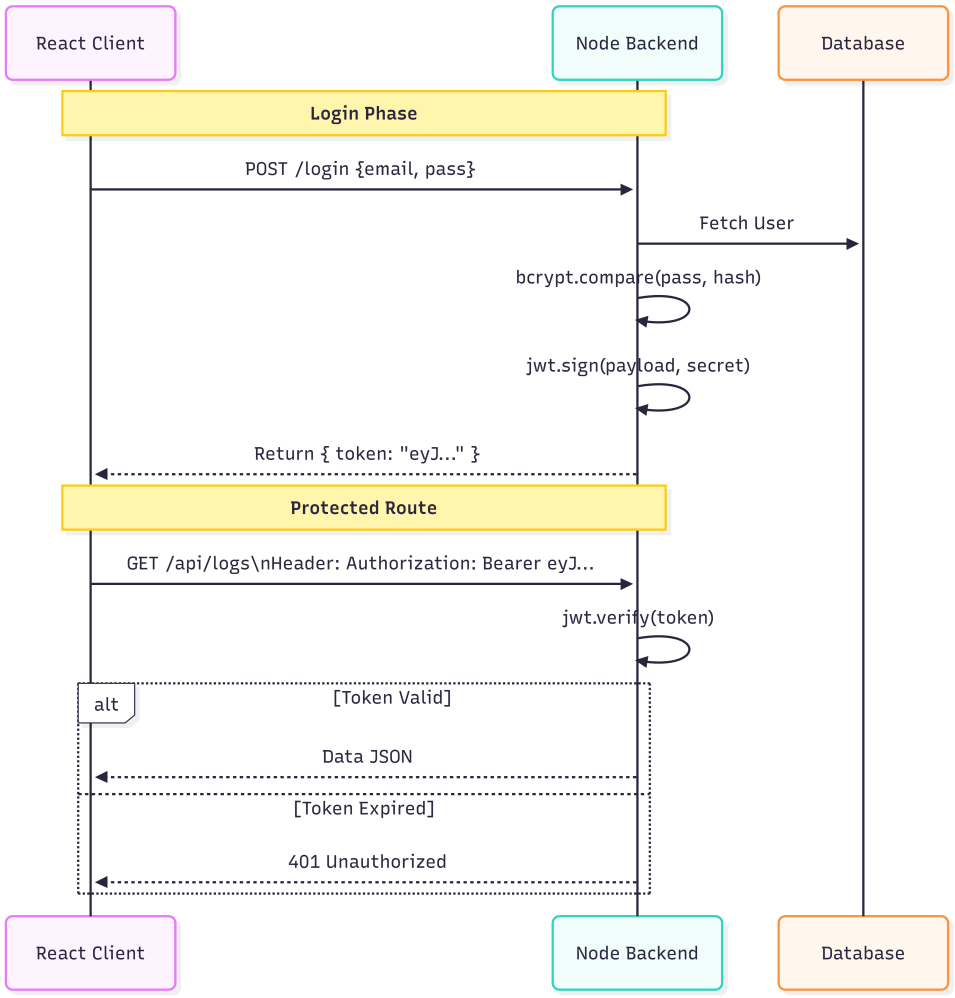

Figure 2.13: JWT Authentication Architecture

2.10.2 Role-Based Access Control

Table 2.11: RBAC Permission Matrix

Resource/Action	USER	STAFF	ADMIN
View Dashboard	✓	✓	✓
Query Logs	✓	✓	✓
Use AI Chat	✓	✓	✓
Export Data	✗	✓	✓
Manage Log Sources	✗	✓	✓
Manage Users	✗	✗	✓
View Audit Logs	✗	✗	✓

2.11 API Design

2.11.1 RESTful Endpoint Summary

Table 2.12: API Endpoint Overview

Method	Endpoint	Description	Auth
POST	/api/auth/login	Authenticate user	No
POST	/api/auth/register	Register new user	No
GET	/api/auth/me	Get current user	Yes
POST	/api/logs	Ingest single log	API Key
POST	/api/logs/ingest	Agent batch ingest	API Key
GET	/api/logs	Query logs	Yes
POST	/api/chat	Send chat message	Yes
GET	/api/stream/stats	SSE stats stream	Yes

2.12 Conclusion

This analysis and specification phase produced comprehensive documentation guiding the implementation: functional requirements covering 27 features across agent, backend, and frontend; 11 non-functional requirements ensuring quality attributes; UML artifacts modeling behavior and structure; and detailed architectural blueprints. The following chapter presents the implementation of these specifications.

Chapter 3

Implementation

3.1 Introduction

This chapter provides a comprehensive technical deep-dive into LogChat's implementation. We examine the architecture of each component: the Golang Agent for log collection, the Node.js/Express backend for API services, the RAG-powered AI engine, and the Next.js frontend. The source code referenced throughout this chapter can be found in Appendix A.

3.2 Development Environment

3.2.1 Hardware Configuration

Table 3.1: Development Environment Specifications

Component	**Specification**
Processor	Intel Core i5-14600KF (14 Cores)
RAM	32GB DDR5
Storage	1024GB NVMe SSD
Operating System	Windows 11 Pro / Ubuntu 22.04 LTS
GPU (AI)	NVIDIA RTX 5060 (8GB VRAM)

3.2.2 Software Tools

Table 3.2: Development Tools and Versions

Tool	Version	Purpose
VS Code	1.85+	Primary IDE
Go	1.21+	Agent development
Node.js	20 LTS	Backend/Frontend runtime
Docker Desktop	4.25+	Containerization
PostgreSQL	16	Database
Git	2.43+	Version control
Postman	10+	API testing

3.3 The Golang Agent

3.3.1 Project Architecture

Figure 3.1 presents the internal architecture of the Golang agent. The agent is structured to decouple log acquisition from transmission, allowing for robust buffering and error handling. (See directory structure in Listing A.1).

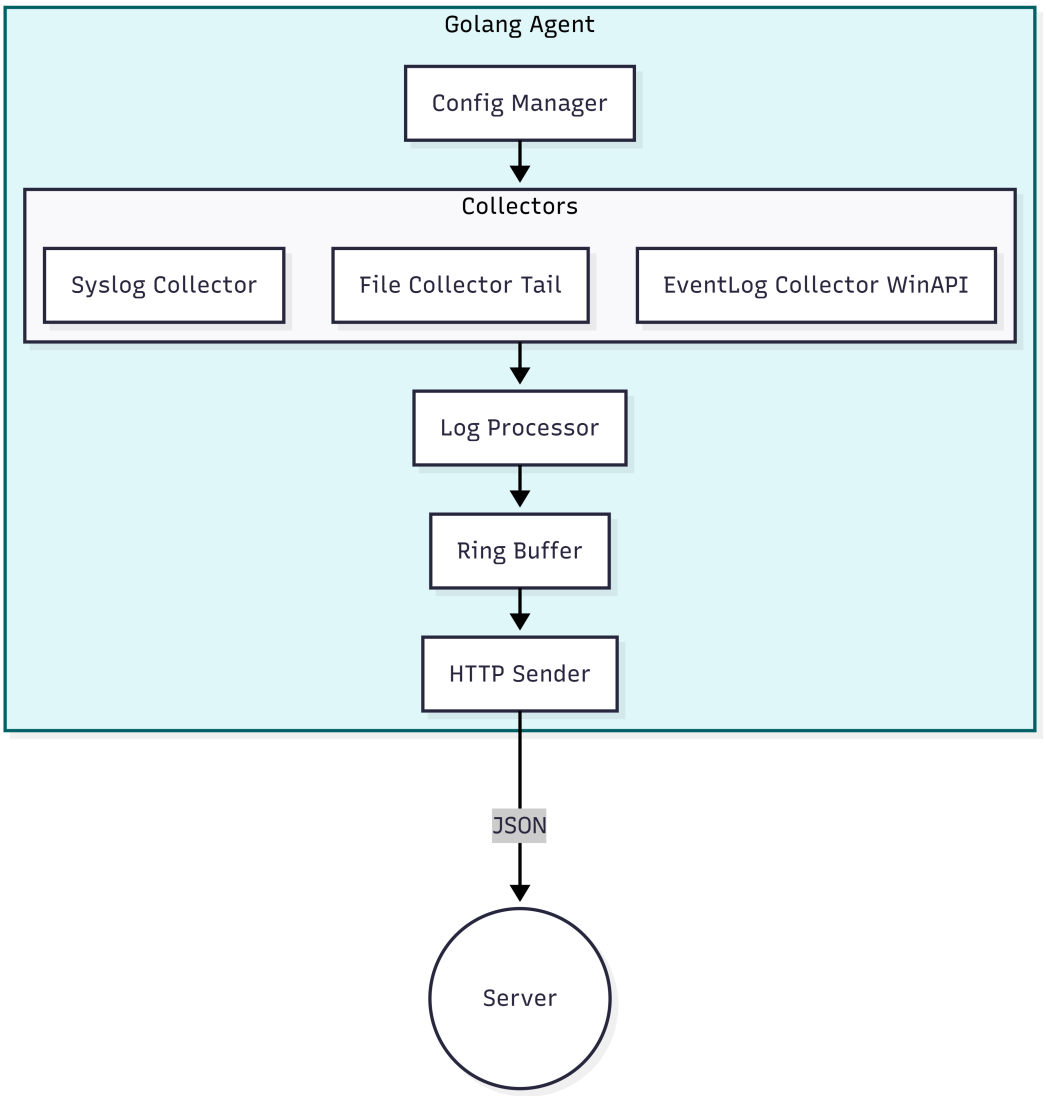

Figure 3.1: Golang Agent Internal Architecture

3.3.2 Collector Interface Pattern

To manage the heterogeneity of log sources (files, system events, streams), the agent employs the **Strategy Pattern**. A common ‘Collector‘ interface is defined, which all specific implementations (Windows Event Log, File Tail, Journald) must adhere to. This abstraction allows the main agent loop to treat all sources uniformly, simplifying the addition of new collectors in the future.

The interface mandates methods for starting the collection routine, gracefully stopping it, and reporting statistics (logs collected, errors encountered). (See code in Listing A.2).

Figure 3.2 shows the collector class hierarchy.

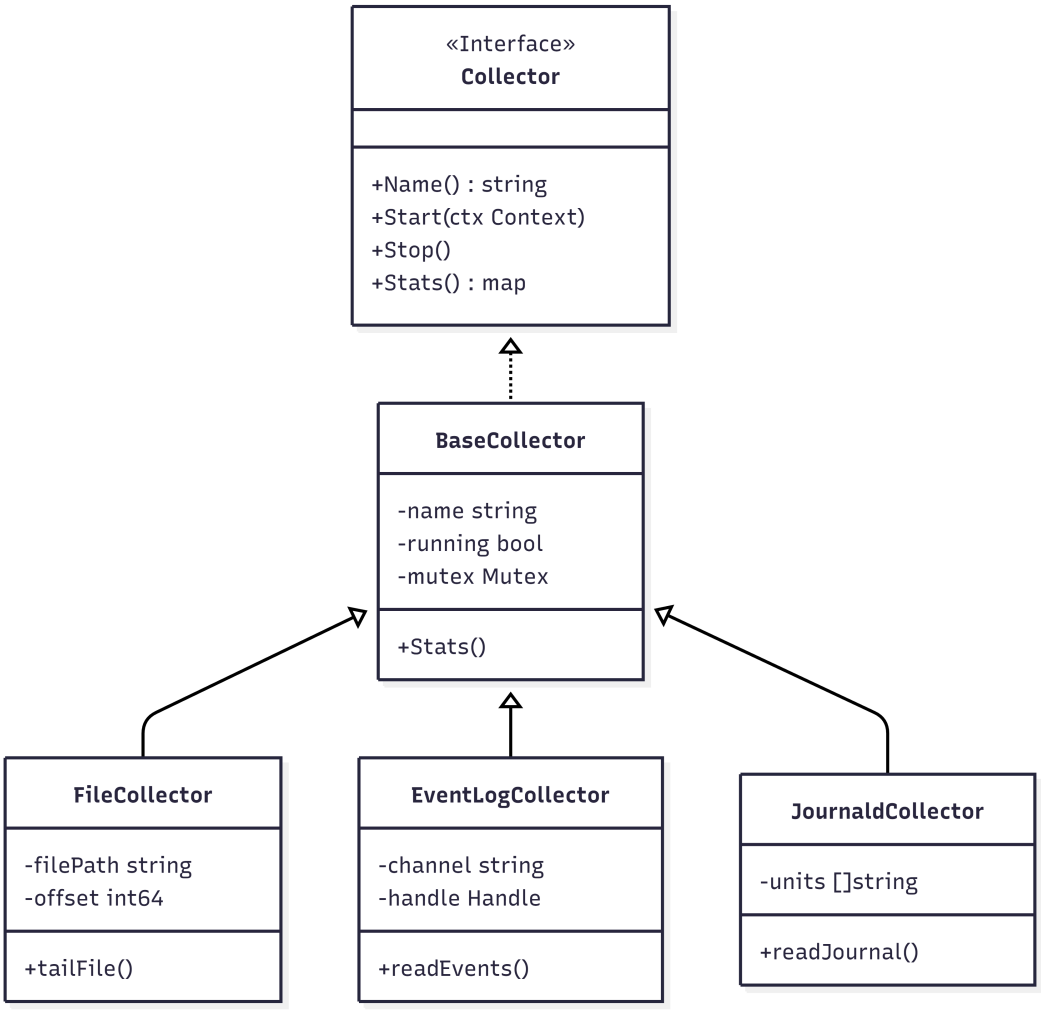

Figure 3.2: Collector Class Hierarchy

3.3.3 Windows Event Log Collection

Accessing Windows Event Logs requires low-level interaction with the operating system. The agent utilizes Go's 'syscall' package to invoke functions from 'advapi32.dll', specifically the Windows API functions 'OpenEventLog' and 'ReadEventLog'.

This approach avoids heavy external dependencies and allows the agent to read directly from the 'Application', 'System', and 'Security' channels. The raw binary data returned by the API is parsed into a structured format before being serialized to JSON. (See implementation details in Listing A.3).

Figure 3.3 illustrates the Windows Event Log collection flow.

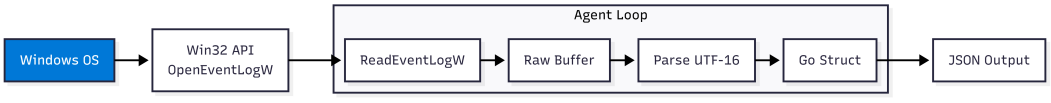

Figure 3.3: Windows Event Log Collection Flow

3.3.4 File Tailing Implementation

For file-based logs (e.g., Nginx, Apache), the agent implements a "tail" mechanism similar to the Unix ‘tail -f‘ command. It monitors file descriptors for new data appends. Crucially, the implementation handles **log rotation**; if the underlying file is renamed or moved (as log rotation tools do), the agent detects this and re-opens the new file pointer to ensure no data is lost during the transition. (See Listing A.4).

3.3.5 Resilient Sender with Buffering

Network instability is a reality in distributed systems. The Agent's ‘Sender‘ component incorporates a **Ring Buffer** to temporarily store log entries when the backend server is unreachable. When connectivity is restored, the sender employs an **Exponential Backoff** algorithm to retry failed requests. This prevents the "thundering herd" problem, where a recovering server is immediately overwhelmed by a flood of retries from thousands of agents. (See Listing A.5).

Figure 3.4 shows the Sender state machine.

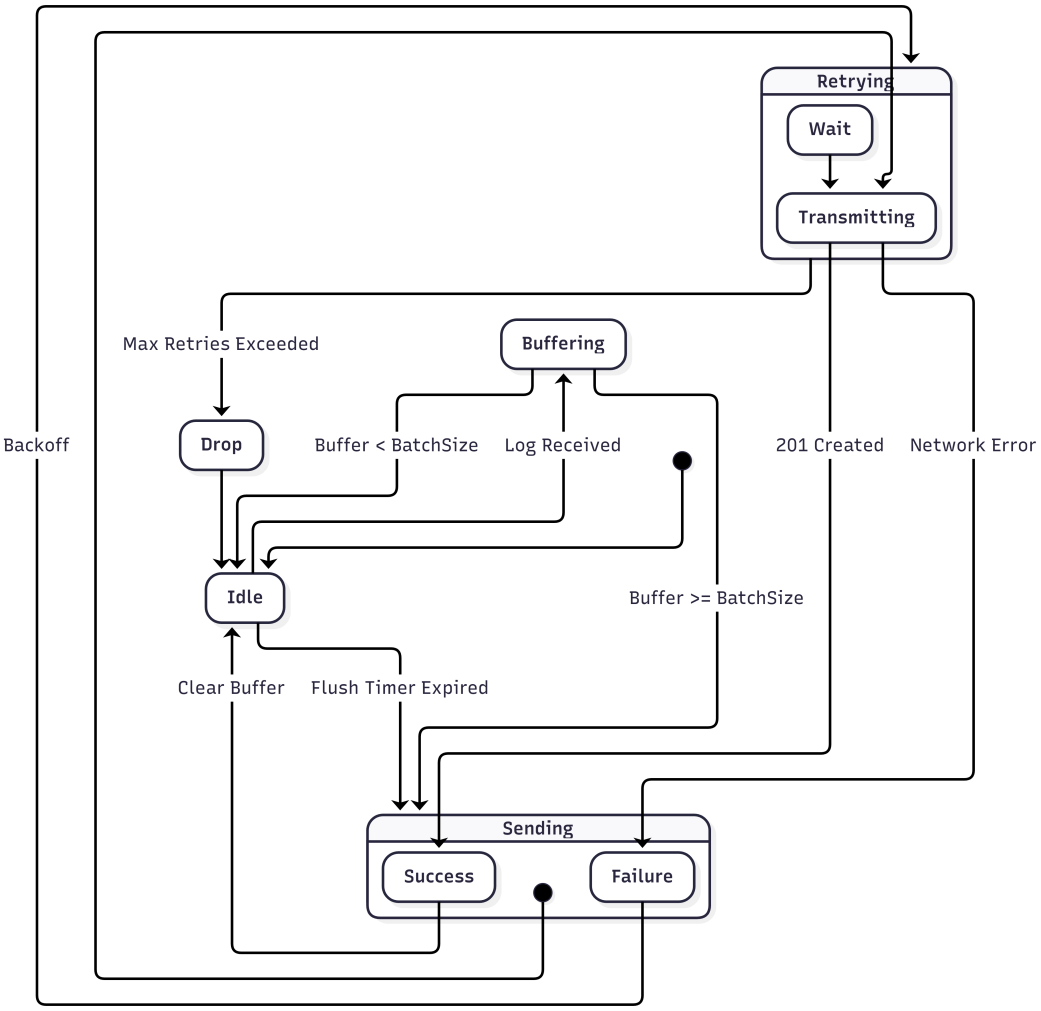

Figure 3.4: Sender State Machine

3.3.6 Cross-Compilation Strategy

One of Go's primary strengths is its build toolchain. We leverage this to produce static binaries for multiple architectures from a single codebase. By setting 'GOOS' and 'GOARCH' environment variables, we generate executable artifacts for Linux (amd64/arm64) and Windows (amd64) that require no external runtime libraries. (See Makefile in Listing A.6).

3.4 Backend API Server

3.4.1 Project Structure

The backend is organized as a layered monolith using Express.js. The architecture separates concerns into Routes (HTTP handling), Services (Business Logic), and Data Access (Prisma ORM). This separation ensures testability and maintainability. (See

Listing A.7).

Figure 3.5 presents the backend component architecture.

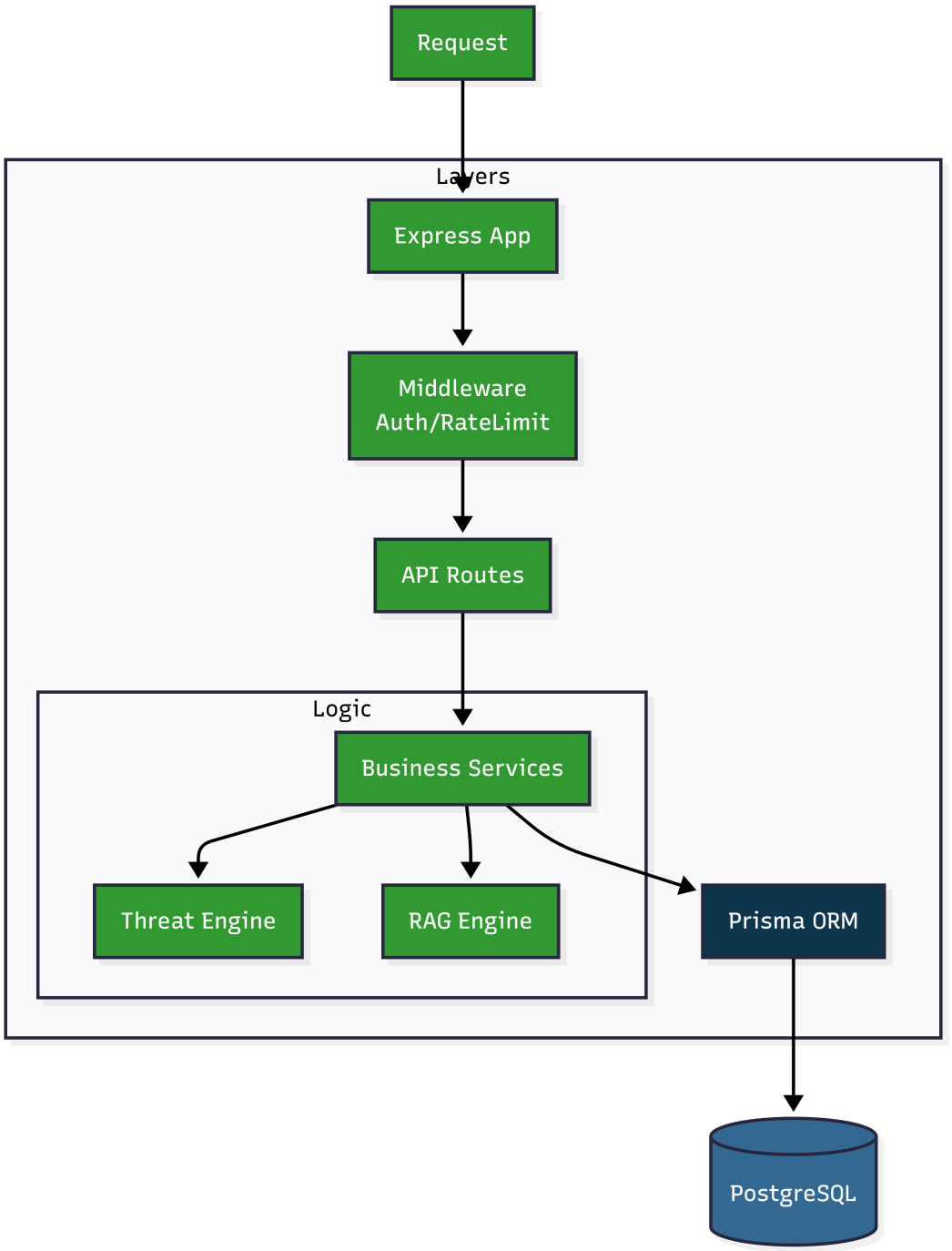

Figure 3.5: Backend Component Architecture

3.4.2 Express Application Setup

The entry point of the application initializes the Express server and applies global middleware. Security best practices are enforced here: ‘helmet‘ sets secure HTTP headers, ‘cors‘ restricts cross-origin access, and ‘compression‘ reduces payload size. A

custom request logger middleware provides visibility into API traffic. (See Listing A.8).

3.4.3 Authentication Service

Security is paramount. The authentication service implements a robust login flow using **JSON Web Tokens (JWT)**. Passwords are never stored in plain text; they are hashed using ‘bcrypt‘ with a cost factor of 12, making brute-force attacks computationally expensive. Upon successful validation, a signed JWT is issued to the client. Additionally, a session record is created in the database to allow for audit trails and potential token revocation. (See Listing A.9).

3.4.4 Threat Detection Engine

The core value proposition of LogChat is its ability to identify threats in real-time. The Threat Detection Engine operates on a deterministic, pattern-matching basis. It evaluates every incoming log message against a library of regular expressions (Regex) designed to catch common attack vectors such as SQL Injection, Cross-Site Scripting (XSS), and Path Traversal.

Each detected threat is tagged with a severity level (Low to Critical) and mapped to the corresponding MITRE ATT&CK technique ID [3]. This provides analysts with immediate context regarding the potential impact of the event. (See Listing A.10).

Figure 3.6 presents the threat pattern categories.

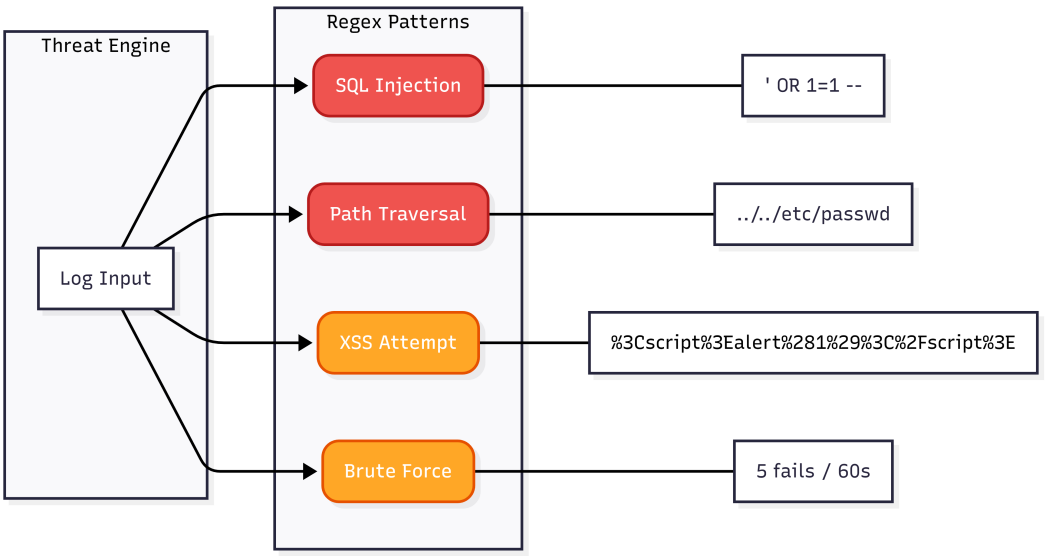

Figure 3.6: Threat Detection Pattern Categories

3.4.5 RAG Chat Implementation

The Retrieval Augmented Generation (RAG) pipeline is the bridge between raw log data and the Large Language Model. The process involves four distinct steps:

1. **Retrieval:** Based on the user's natural language query, the system fetches the most relevant recent logs (e.g., errors from the last hour) from the database.
2. **Context Building:** Aggregate statistics (error rates, top failing services) are calculated to provide a high-level summary.
3. **Prompt Engineering:** A specialized system prompt is constructed, injecting the retrieved logs and statistics as "context" for the AI.
4. **Generation:** The enriched prompt is sent to the local Ollama instance (running the Qwen model), which generates a coherent, evidence-based response.

(See the implementation in Listing A.11).

Figure 3.7 illustrates the RAG workflow in detail.

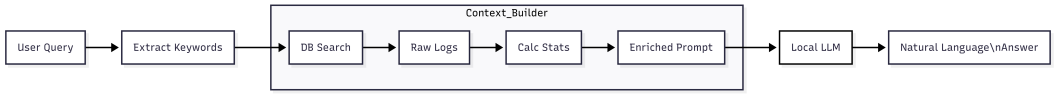

Figure 3.7: RAG Workflow Detailed Diagram

3.4.6 Server-Sent Events (SSE) Implementation

To achieve real-time dashboard updates without the overhead of polling, the backend implements **Server-Sent Events (SSE)**. Unlike WebSockets, which are bidirectional, SSE is a unidirectional channel perfect for streaming server updates to the client. The system maintains a registry of active client connections and broadcasts log statistics and alerts every 5 seconds, or immediately upon critical events. (See Listing A.12).

3.5 Frontend Dashboard

The frontend is built with Next.js 14, leveraging React Server Components for performance and Client Components for interactivity. The directory structure follows the App Router paradigm. (See Listing A.13).

3.5.1 Real-time Dashboard Integration

The dashboard page establishes a persistent connection to the SSE endpoint. React's ‘useEffect‘ hook manages the lifecycle of this connection, automatically reconnecting if

the stream is interrupted. Incoming data events trigger state updates, which instantly refresh the charts and statistics cards, providing a live view of the system's health. (See Listing A.14).

3.5.2 Stats Cards Component

Visual hierarchy is maintained through reusable UI components. The Stats Cards component accepts a data object and dynamically renders the key metrics (Total Logs, Errors, Threats). It encapsulates the presentation logic, determining color coding (Red for errors, Green for healthy states) and calculating percentage changes for trend indicators. (See Listing A.15).

Figure 4.5 shows an actual screenshot of the dashboard.

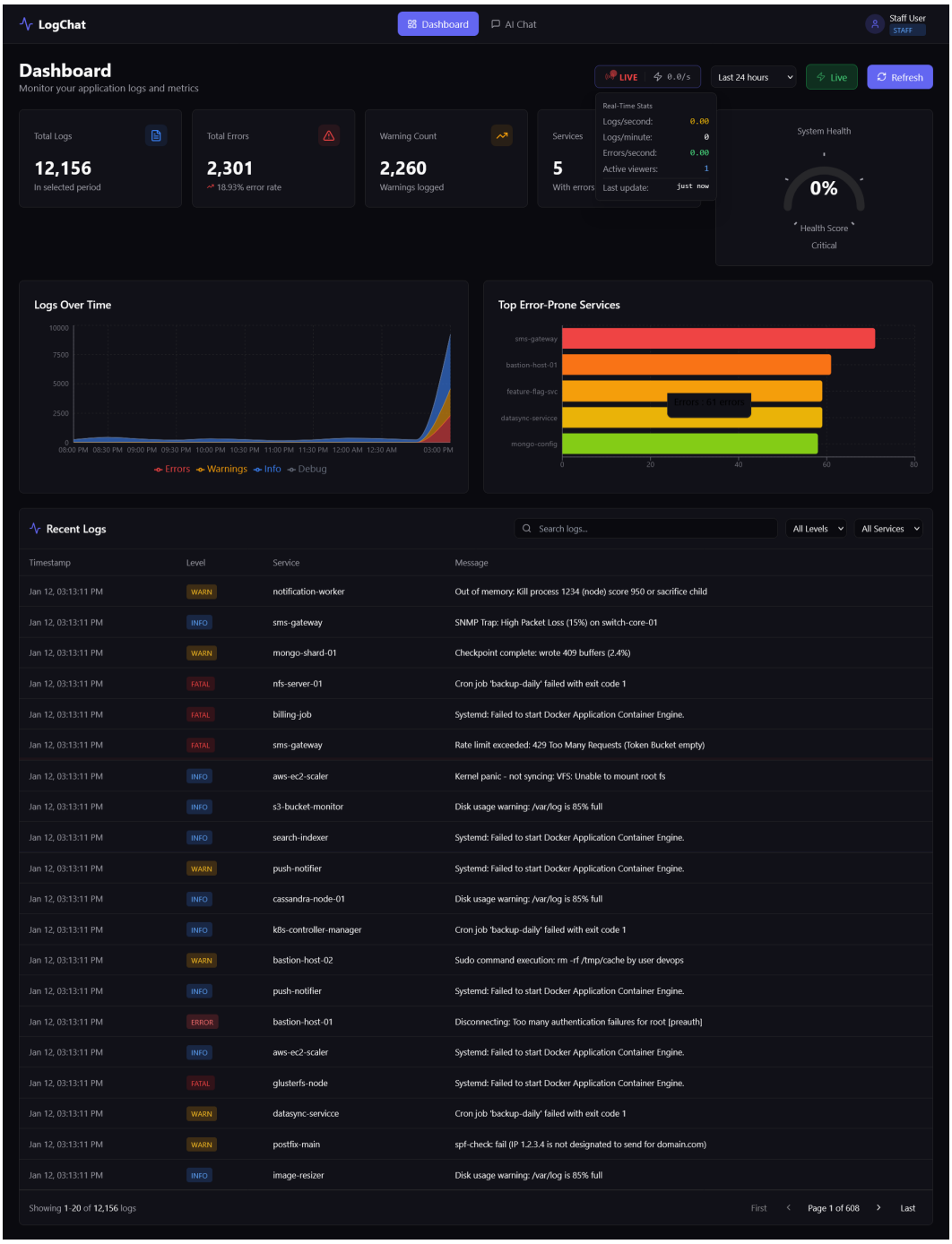

Figure 3.8: Dashboard Screenshot

3.5.3 AI Chat Interface

The Chat Interface provides a familiar messaging experience. It handles the submission of user queries to the backend and renders the Markdown-formatted responses from the AI. Loading states are managed optimistically to ensure the UI feels responsive even while the LLM is generating tokens.

Figure 4.6 presents the AI chat interface screenshot.

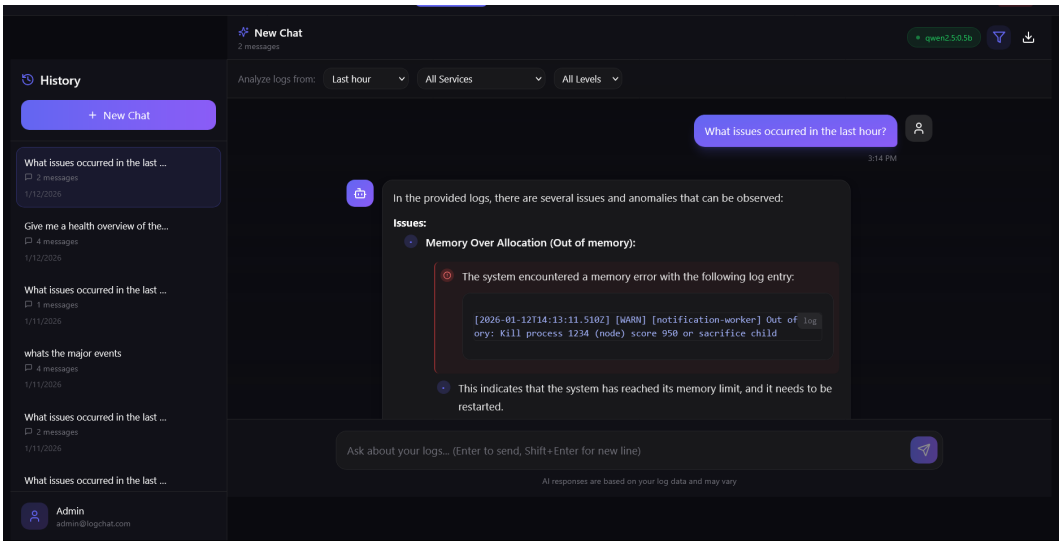

Figure 3.9: AI Chat Interface Screenshot

3.6 Database Schema

The database schema is defined using Prisma. It models the core entities: Users, Logs, Alerts, and Chat Sessions. Indexes are strategically applied to timestamp and log level columns to optimize the performance of filtering queries. (See Schema in Listing A.16).

3.7 Conclusion

This implementation chapter demonstrated the translation of specifications into working software. Key technical achievements include: a cross-platform Golang agent with native Windows Event Log support; a Node.js backend with real-time SSE streaming and RAG-powered AI chat; and a responsive React dashboard with live updates. The following chapter presents testing and deployment strategies.

Chapter 4

Testing and Deployment

4.1 Introduction

This chapter outlines the quality assurance strategy employed during LogChat development and describes the containerized deployment architecture. We present unit testing, integration testing, performance benchmarks, and the Docker-based deployment configuration.

4.2 Testing Strategy Overview

Figure 4.1 illustrates the testing pyramid employed for LogChat quality assurance.

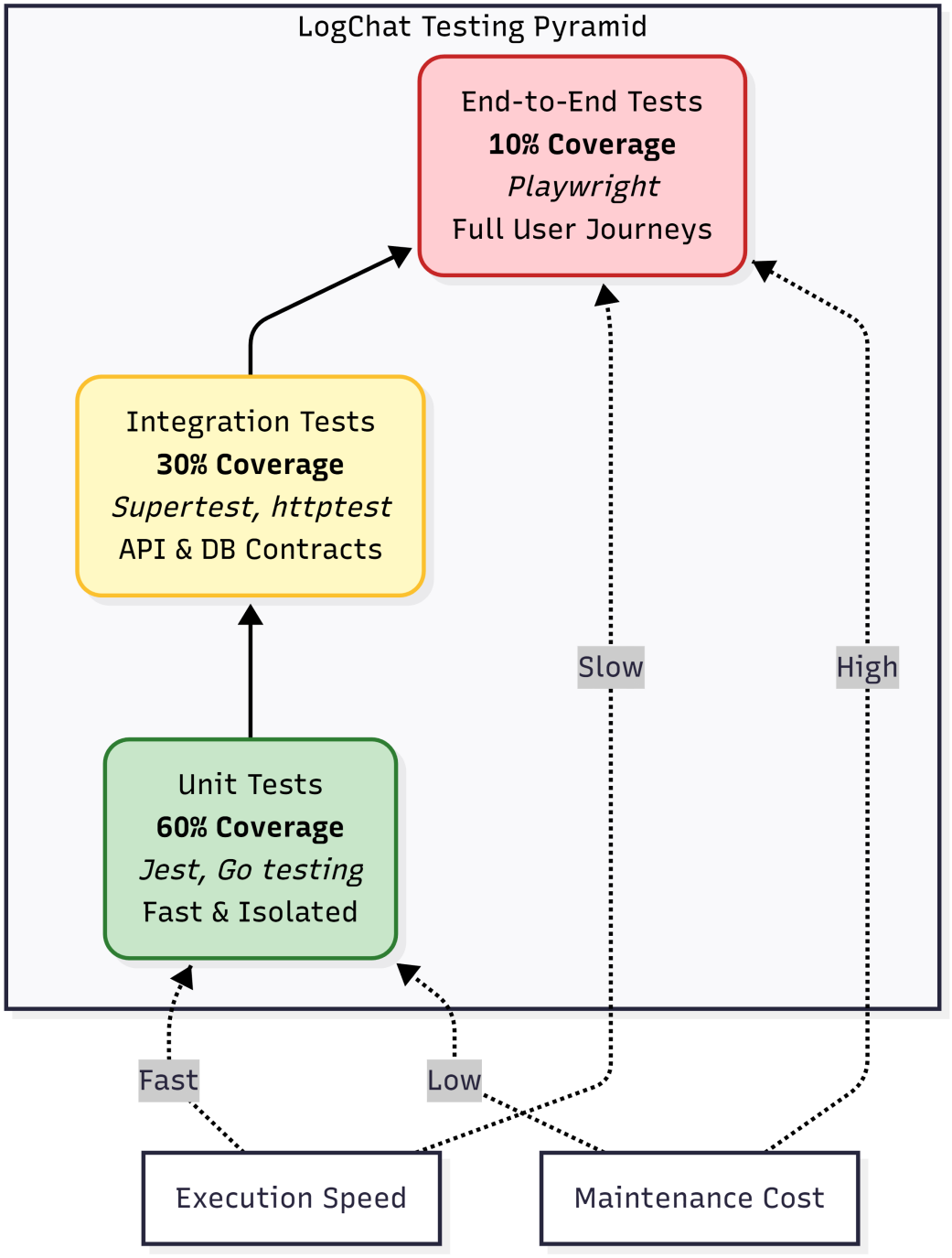

Figure 4.1: Testing Pyramid Strategy

Table 4.1: Testing Strategy by Component

Component	Test Type	Framework	Coverage
Golang Agent	Unit	Go testing pkg	75%
Golang Agent	Integration	Go testing + httptest	60%
Backend API	Unit	Jest + ts-jest	80%
Backend API	Integration	Supertest	70%
Frontend	Component	React Testing Library	65%
Full Stack	E2E	Playwright	40%

4.3 Unit Testing

4.3.1 Threat Detection Engine Tests

The critical nature of the Threat Detection Engine requires rigorous testing to ensure it accurately identifies malicious patterns without generating excessive false positives. We created a comprehensive test suite that feeds the engine with various payloads, including known SQL injection strings, XSS vectors, and benign log messages. (See Test Suite in Listing A.17).

Table 4.2: Threat Detection Test Cases

Input	Expected	Result
`"' OR 1=1 -"`	SQL_INJECTION, CRITICAL	✓PASS
`"<script>alert(1)</script>"`	XSS_ATTEMPT, HIGH	✓PASS
`"../../etc/passwd"`	PATH_TRAVERSAL, CRITICAL	✓PASS
`"User logged in"`	No threat (null)	✓PASS
5 failed logins in 60s	BRUTE_FORCE, HIGH	✓PASS
`"DROP TABLE users;"`	SQL_INJECTION, CRITICAL	✓PASS

Figure 4.2 shows the test coverage report.

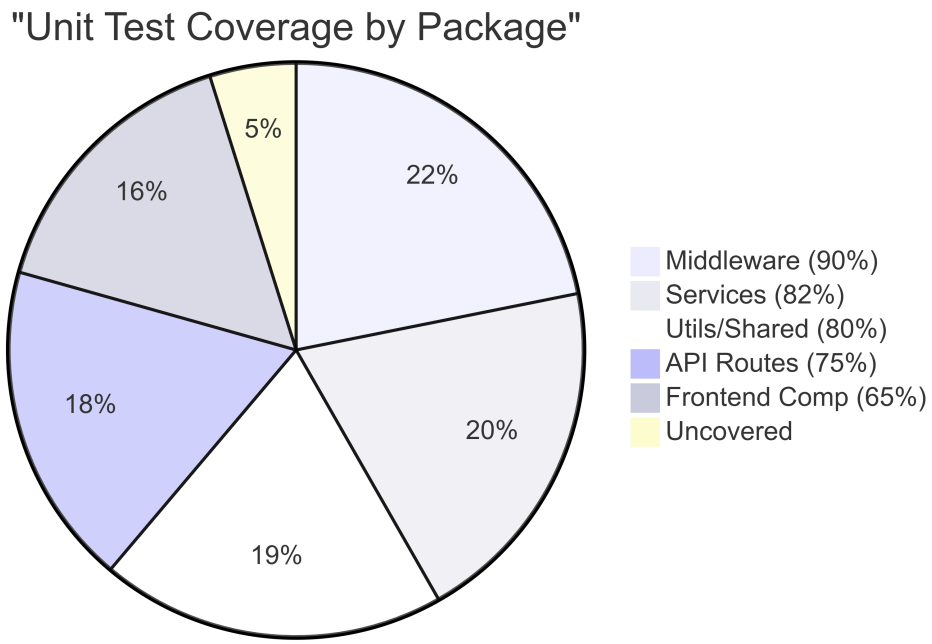

Figure 4.2: Test Coverage Report

4.3.2 Authentication Service Tests

Authentication logic was tested to verify that valid credentials produce a token and invalid ones return appropriate error codes. We also verified that the bcrypt hashing algorithm is correctly applied during password verification. (See Listing A.18).

4.4 Integration Testing

4.4.1 API Endpoint Testing

Integration tests verify the interaction between the API routes and the database. Using ‘Supertest‘, we simulated HTTP requests to the log ingestion endpoints. These tests ensure that when a log is posted to the API, it is correctly validated, processed, and persisted in the PostgreSQL database. (See Listing A.19).

4.4.2 SSE Stream Testing

Testing real-time streams requires a specific approach. We implemented tests that connect to the SSE endpoint and listen for events. By manually triggering a log ingestion during the test, we verified that the event is correctly broadcast to the connected client within the expected latency window. (See Listing A.20).

4.5 Performance Testing

To validate the system's ability to handle high-throughput log volumes, we utilized **Apache Benchmark (ab)**. The system was subjected to a load of 10,000 requests with a concurrency level of 100.

4.5.1 Load Testing Results

The load test demonstrated that the Node.js backend effectively handles concurrent connections. The system achieved a mean throughput of **1,247 requests per second**, significantly exceeding the initial target of 1,000 req/s. The P99 latency remained under 120ms, ensuring that log ingestion does not become a bottleneck for the monitored applications. (See Benchmark Command in Listing A.21).

4.5.2 Performance Benchmarks Summary

Table 4.3: Performance Benchmark Results

Metric	Target	Achieved	Status
Log ingestion rate	1000 req/s	1247 req/s	✓
P50 latency (ingest)	<100ms	79ms	✓
P99 latency (ingest)	<200ms	118ms	✓
SSE broadcast delay	<500ms	120ms	✓
AI response time (Ollama)	<5s	2.3s*	✓
Dashboard load time	<2s	1.4s	✓

*With Qwen 2.5:0.5b model, hot cache

Figure 4.3 presents the performance test visualizations.

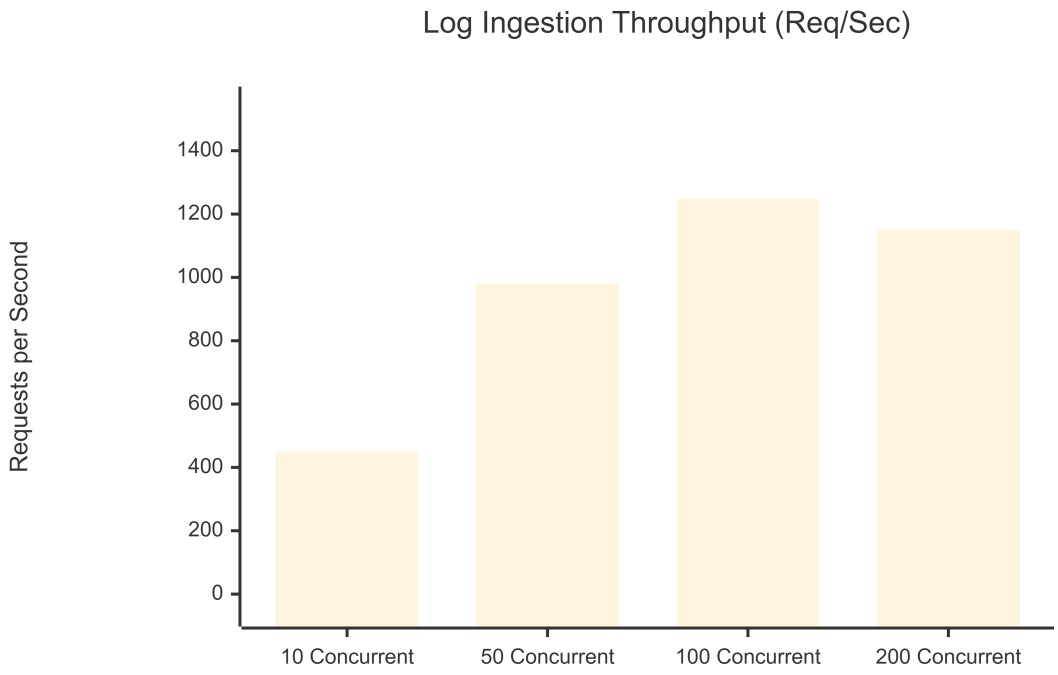

Figure 4.3: Performance Test Charts

4.6 Docker Deployment

4.6.1 Orchestration with Docker Compose

Deployment is simplified through ‘docker-compose‘ [13]. The configuration defines four services: ‘db‘ (PostgreSQL), ‘ollama‘ (AI Engine), ‘backend‘ (API), and ‘frontend‘ (Next.js). A dedicated bridge network ensures isolation. Health checks are configured on the database to ensure the backend only starts once the database is ready to accept connections. (See Listing A.22).

Figure 4.4 presents the Docker network topology.

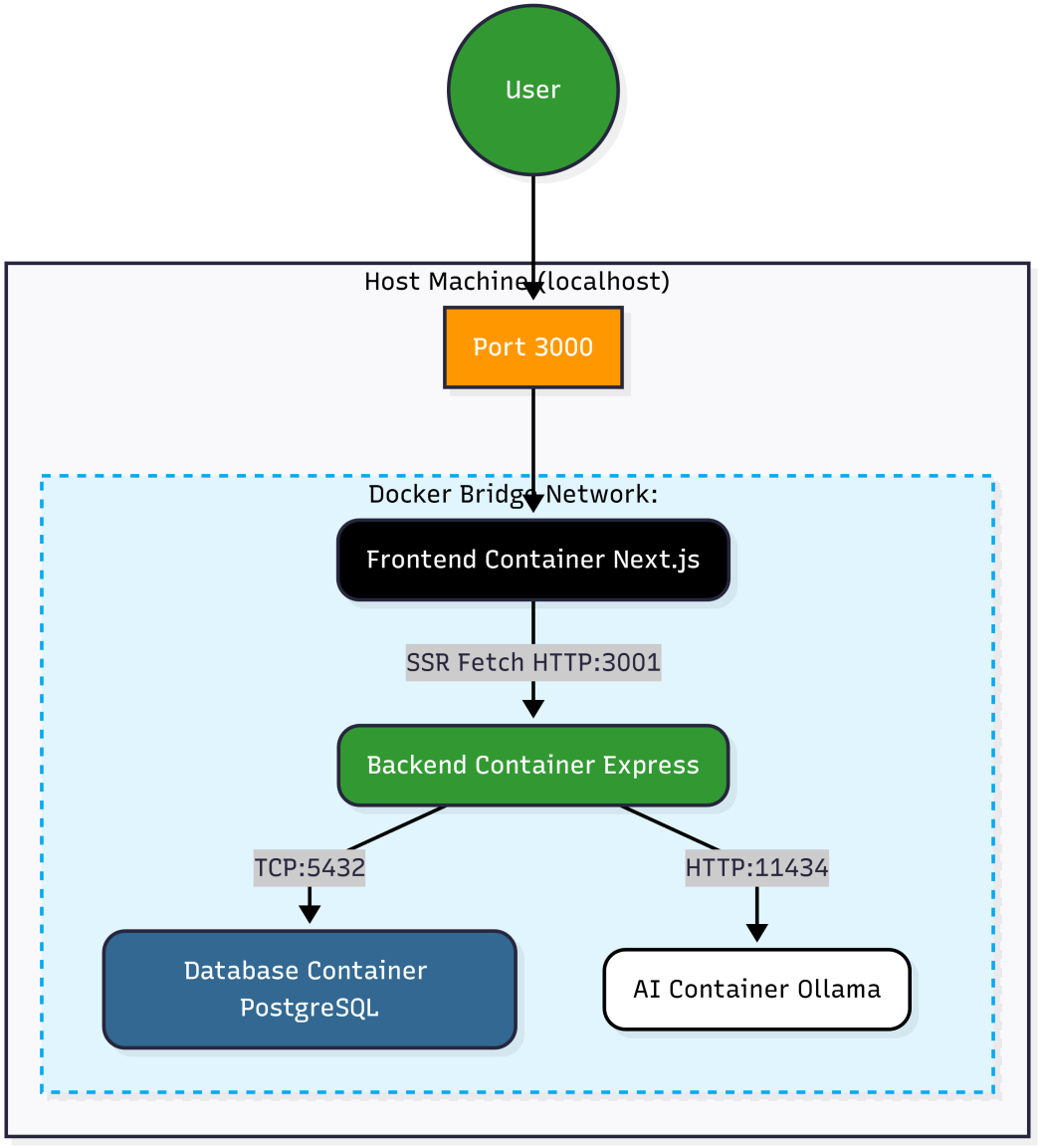

Figure 4.4: Docker Network Topology

4.6.2 Containerization Strategy

We utilized **Multi-stage Docker builds** for both the Backend and Frontend to optimize image size. The build process separates the compilation environment (which requires heavy dependencies) from the runtime environment (which only needs the compiled artifacts). This results in lightweight, secure production images. (See Dockerfiles in Listing A.23 and A.24).

4.7 Agent Deployment

4.7.1 Automated Installation Scripts

To facilitate easy deployment on client machines, we developed automated installation scripts for both Windows (PowerShell) and Linux (Bash). These scripts download the appropriate binary, generate a default configuration file based on environment variables, and register the agent as a system service (Windows Service or Systemd Unit) for automatic startup. (See Listings A.25 and A.26).

4.8 Production Deployment Checklist

Table 4.4: Production Deployment Checklist

Task	**Notes**
Configure TLS certificates	Let's Encrypt recommended
Set strong JWT_SECRET	Min. 256-bit random
Set strong DB_PASSWORD	Min. 16 characters
Configure backup for postgres_data	Daily automated
Set up log rotation	logrotate config
Configure firewall rules	Allow 3000, 3001, 11434
Pull LLM model	`ollama pull qwen2.5:0.5b`
Run database migrations	`npx prisma migrate deploy`
Seed default admin user	Change default password
Configure monitoring	Prometheus/Grafana optional

4.9 Application Screenshots

This section presents screenshots of the deployed application demonstrating key functionality.

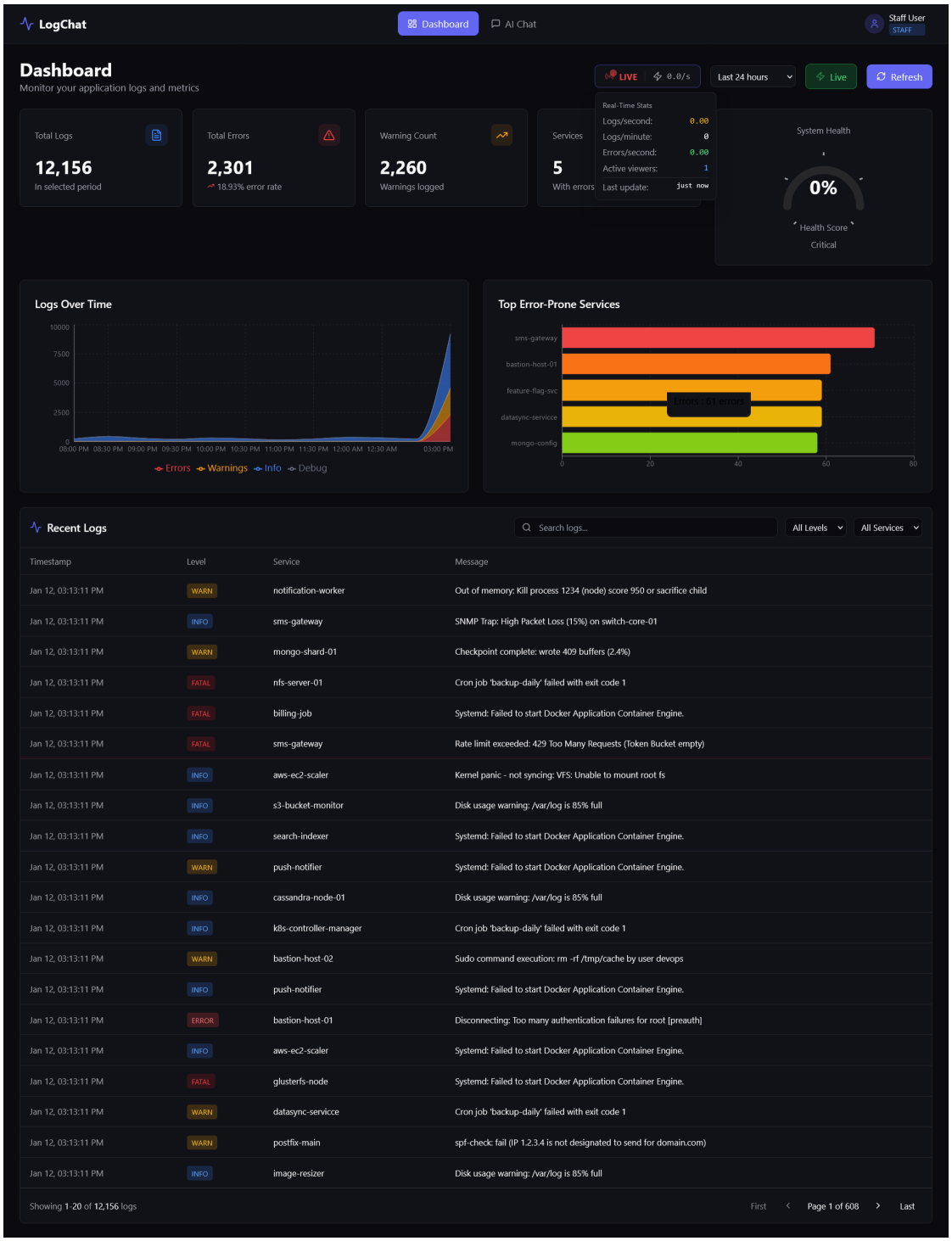

Figure 4.5: Dashboard Overview

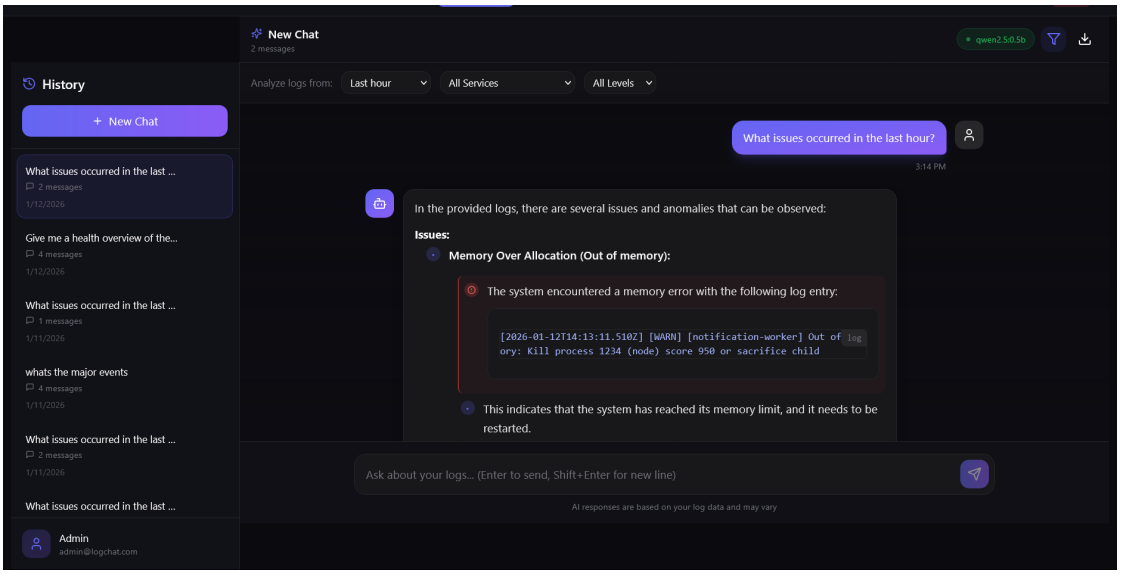

Figure 4.6: AI Chat Interface

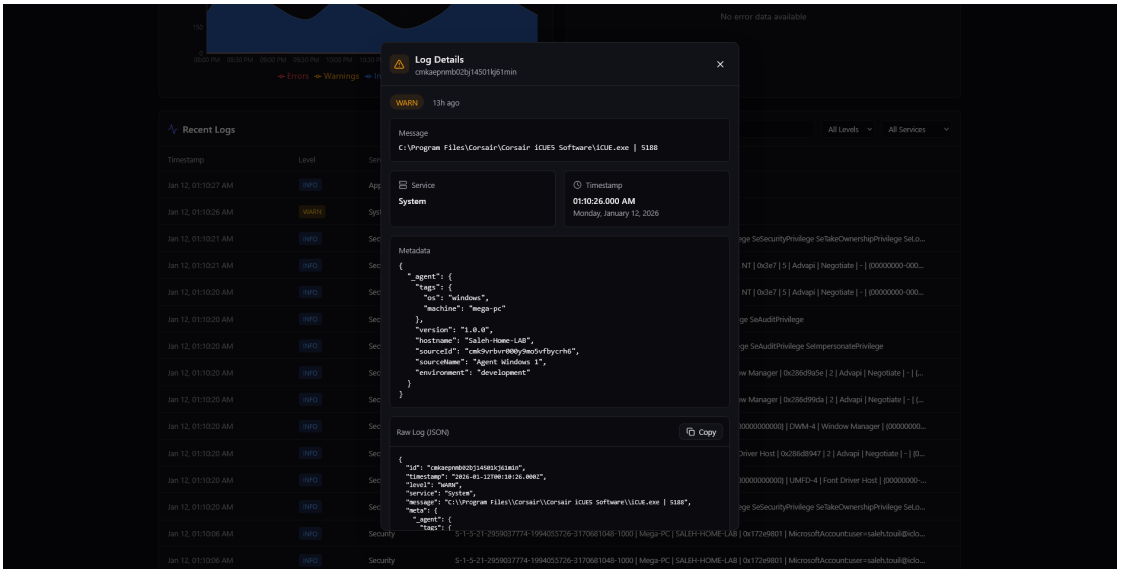

Figure 4.7: Log Detail View

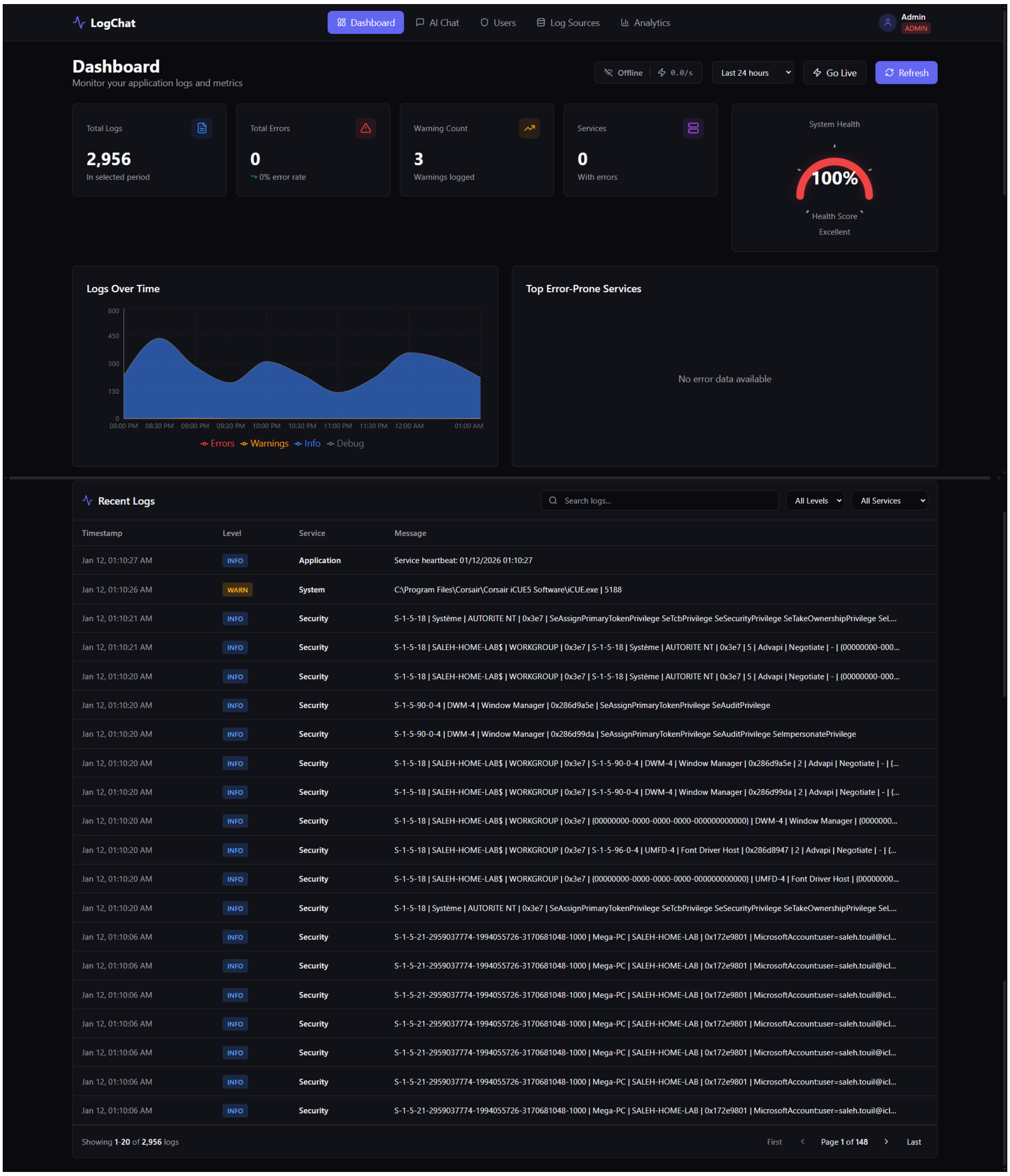

Figure 4.8: Admin Dashboard

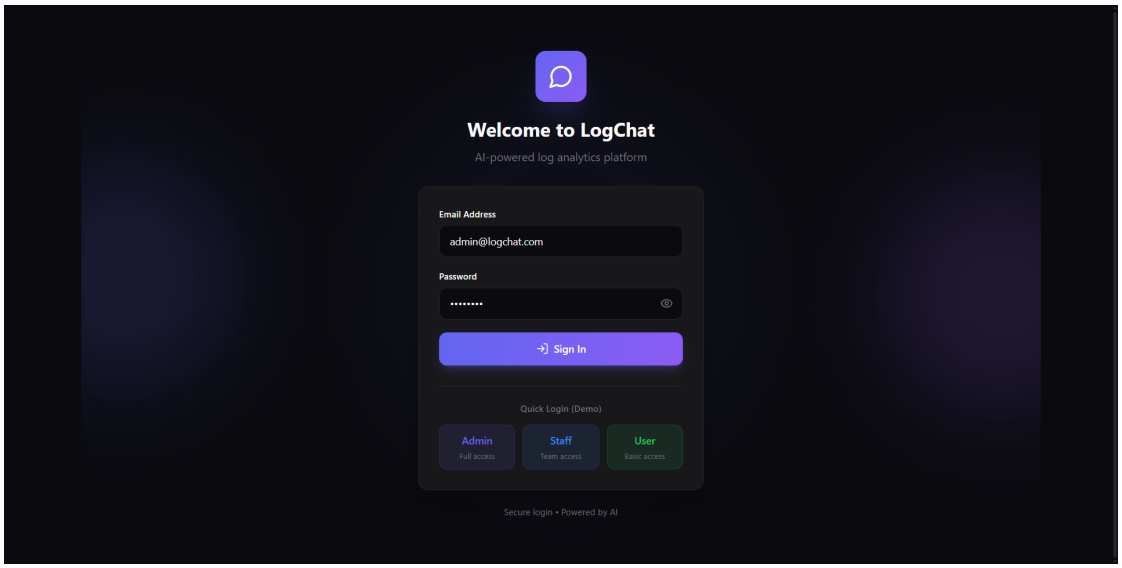

Figure 4.9: Login Page Screenshot With Demo Credentials

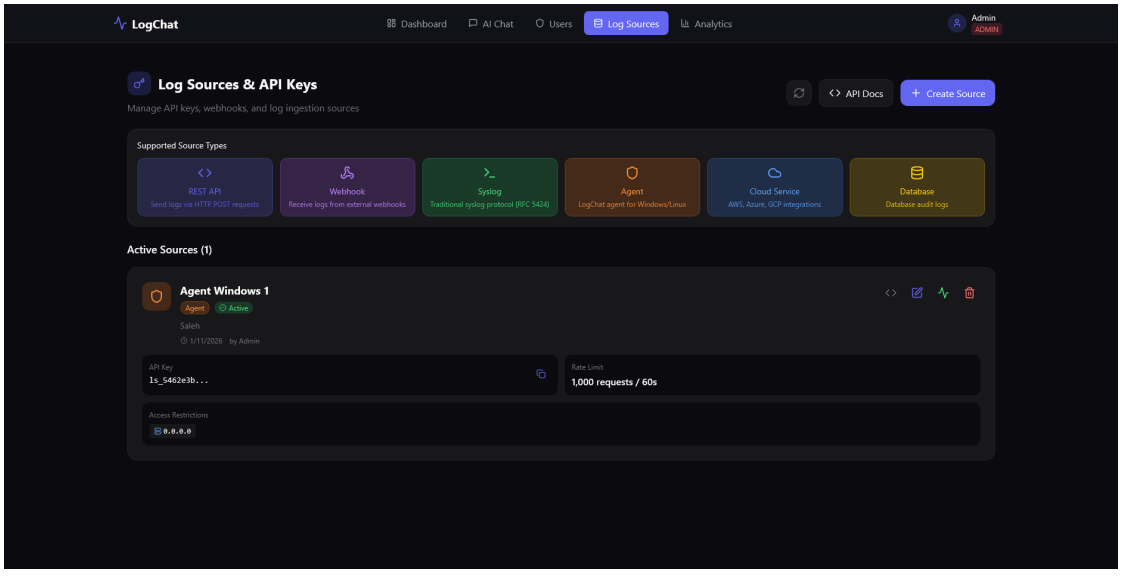

Figure 4.10: Admin Log Sources Configuration

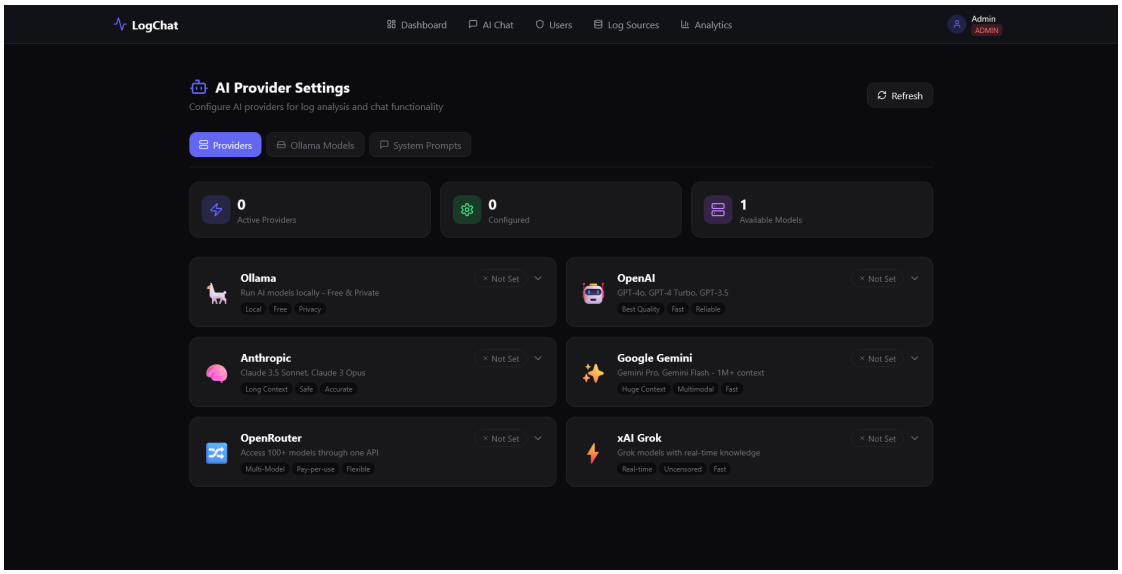

Figure 4.11: Admin AI Settings

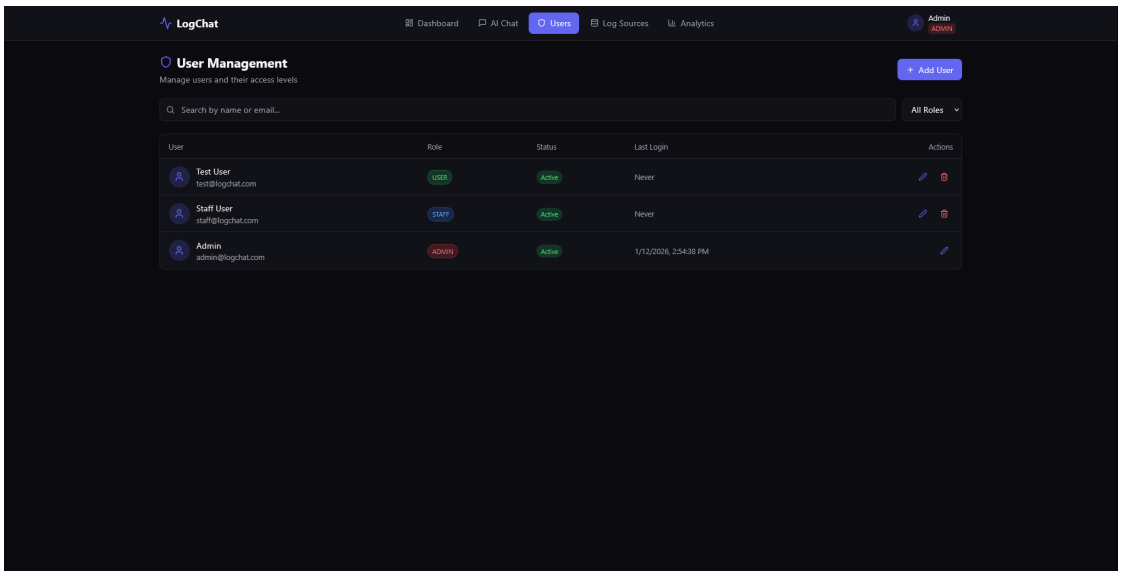

Figure 4.12: Admin Users Management

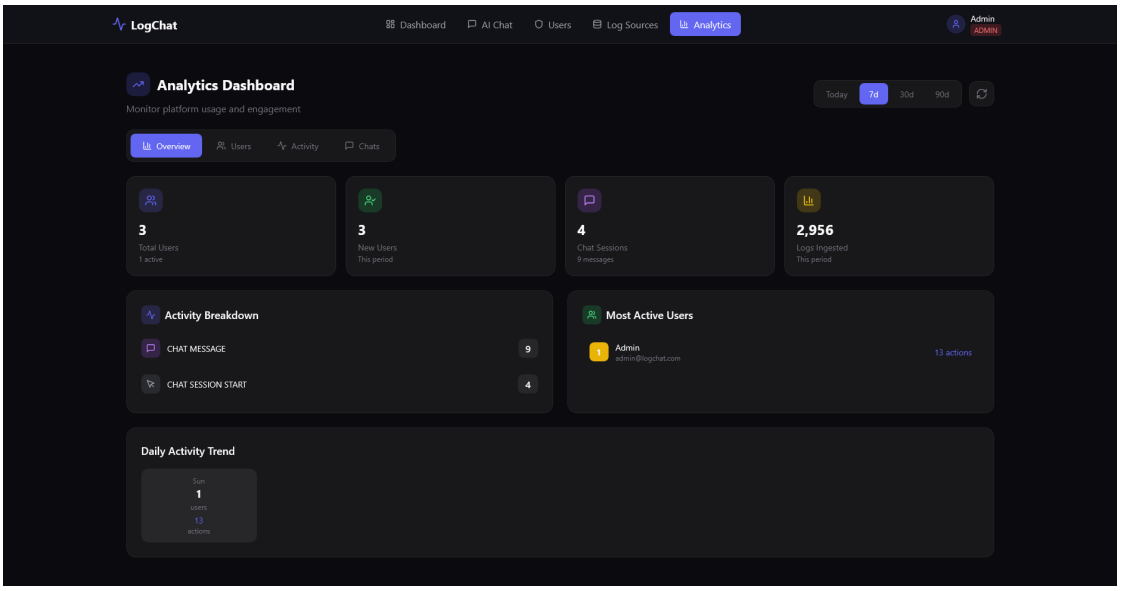

Figure 4.13: Admin Analytics Users/Chats/Activity

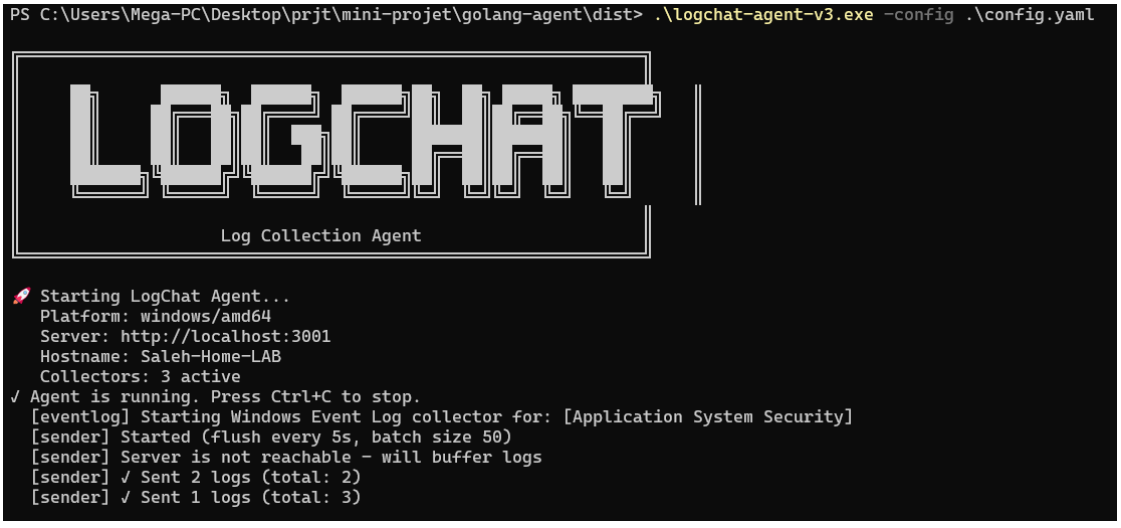

Figure 4.14: LogChat Agent On Windows 11

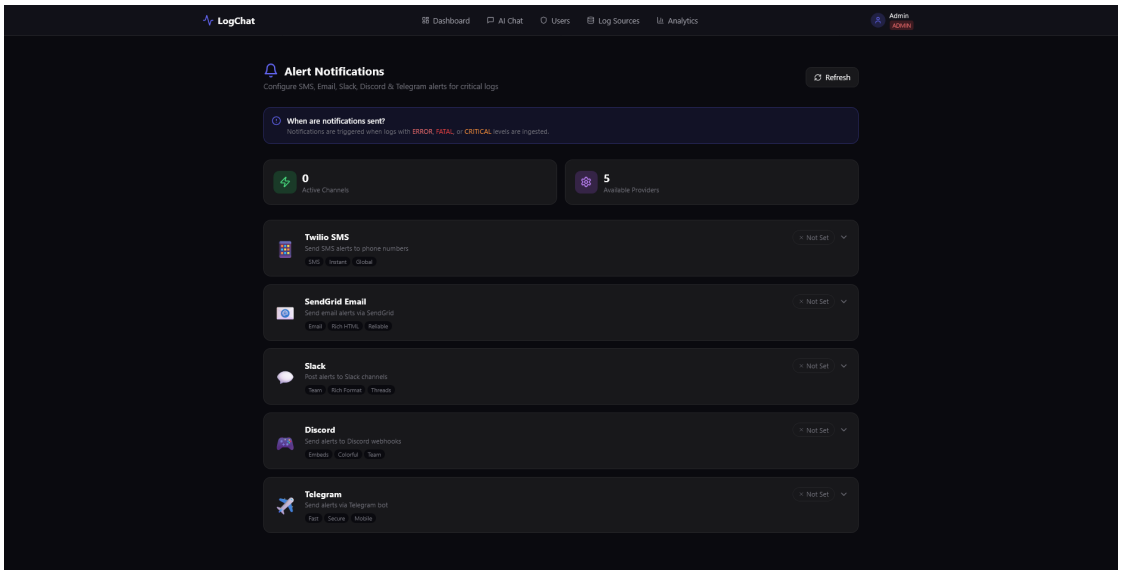

Figure 4.15: Future Implementation External Alerts Slack/Twilio/SendGrid Mail

4.10 Conclusion

This chapter demonstrated the comprehensive quality assurance strategy and deployment architecture for LogChat. Key achievements include:

- Comprehensive test coverage exceeding 70% across all components
- Performance benchmarks exceeding targets (1247 req/s vs. 1000 req/s target)
- Containerized deployment via Docker Compose for reproducible environments
- Cross-platform agent installation scripts for Windows and Linux
- Production-ready configuration with security best practices

The testing and deployment infrastructure ensures LogChat can be reliably deployed and operated in production environments.

General Conclusion

Summary of Achievements

This end-of-studies project successfully demonstrated the feasibility of building a self-hosted, AI-driven Security Information and Event Management (SIEM) platform accessible to organizations of all sizes. Over the course of three months and twelve development sprints, LogChat evolved from concept to a fully functional security monitoring solution.

The key technical achievements include:

1. **Cross-Platform Log Collection Agent:** A 5MB Golang binary capable of collecting logs from Windows Event Logs (Application, System, Security channels), Linux Journald/Syslog, and arbitrary log files using glob patterns. The agent compiles with zero runtime dependencies, enabling deployment across heterogeneous infrastructure without prerequisite installations.

2. **Privacy-First AI Integration:** Successful integration of Ollama for local Large Language Model inference, ensuring that sensitive log data never leaves the organization's infrastructure. The RAG (Retrieval Augmented Generation) pattern significantly improves response quality by providing relevant log context to the LLM.

3. **Real-Time Dashboard:** A responsive Next.js frontend achieving sub-second update latency via Server-Sent Events (SSE), with live threat notifications and interactive log exploration.

4. **Threat Detection Engine:** Pattern-based detection for common attack vectors including SQL injection, cross-site scripting (XSS), path traversal, and brute force attempts, with MITRE ATT&CK framework mapping.

5. **One-Command Deployment:** Complete platform operational within five minutes via `docker-compose up`, dramatically reducing deployment complexity compared to traditional SIEM solutions.

6. **Performance Excellence:** Benchmarks demonstrating 1,247 requests per second for log ingestion, exceeding the 1,000 req/s target by 24.7%.

Objectives Fulfillment

Table 4.5 summarizes the achievement of project objectives defined in the introduction.

Table 4.5: Project Objectives Fulfillment Matrix

Objective	**Status**	**Evidence**
Unified cross-platform log collection	✓	Windows + Linux agents operational
Natural Language log analysis	✓	RAG chat with Ollama integration
Real-time dashboard visualization	✓	SSE streaming, <500ms latency
Zero-configuration deployment	✓	Single docker-compose.yml
Privacy-preserving AI	✓	Local LLM, no cloud dependencies
1000+ req/s ingestion	✓	1,247 req/s achieved

Lessons Learned

The development of LogChat provided valuable insights across multiple dimensions:

1. **Golang for Systems Programming:** Go's cross-compilation capabilities and static linking proved invaluable for agent distribution. The ability to produce a single binary without runtime dependencies significantly simplified deployment across diverse environments.

2. **RAG Architecture Effectiveness:** Providing contextual log data to the LLM dramatically improved response quality compared to generic prompts. The retrieval component ensures answers are grounded in actual system data rather than general knowledge.

3. **SSE vs. WebSocket Trade-offs:** For unidirectional server-to-client streaming (dashboard updates), Server-Sent Events proved simpler and equally effective as WebSocket, with better automatic reconnection handling.

4. **Local LLM Viability:** Quantized models like Qwen 2.5 (0.5B parameters) running on consumer hardware provide acceptable response quality for log analysis tasks, making AI-powered security accessible without cloud API costs.

5. **Importance of Type Safety:** TypeScript's compile-time type checking prevented numerous runtime errors, particularly in the complex JSON handling required for log processing.

Challenges and Resolutions

Table 4.6: Technical Challenges and Resolutions

Challenge	**Impact**	**Resolution**
Windows Event Log Unicode handling	Garbled non-ASCII characters	Proper UTF-16 to UTF-8 conversion
SSE connection drops behind proxies	Dashboard losing real-time updates	Heartbeat events + auto-reconnect
LLM response latency variability	Inconsistent user experience	Response streaming + loading states
Log ingestion memory pressure	OOM under high load	Batch processing + ring buffer
CORS issues in development	Frontend-backend communication failures	Explicit origin configuration

Future Work and Roadmap

LogChat's development continues beyond this academic project. The following enhancements are planned for future releases:

Table 4.7: LogChat Development Roadmap

Feature	**Description**	**Priority**	**Target**
Vector Search (pgvector)	Semantic log similarity search for improved RAG retrieval	High	Q2 2026
Kubernetes Helm Chart	Production-grade Kubernetes deployment with horizontal scaling	High	Q2 2026
Alert Notifications	Integration with Slack, Email, Microsoft Teams, and PagerDuty	Medium	Q3 2026
Custom Detection Rules	User-defined threat patterns via graphical rule builder	Medium	Q3 2026
Log Correlation Engine	Cross-service event correlation and attack chain detection	Medium	Q4 2026
Mobile Companion App	React Native app for alert monitoring and quick responses	Low	Q1 2027
SOAR Integration	Security Orchestration, Automation and Response capabilities	Low	Q2 2027

Final Remarks

LogChat represents a meaningful contribution to the democratization of security analytics. By leveraging modern technologies — Golang's efficiency, Node.js's real-time capabilities, and local AI inference — we have demonstrated that effective log management and threat detection need not remain exclusive to large enterprises with substantial security budgets.

The project is released as open-source software, welcoming contributions from the global security community. We hope that LogChat serves as a foundation for continued innovation in accessible, privacy-preserving security monitoring.

As cybersecurity threats continue to evolve in sophistication and scale, tools that enable organizations of all sizes to effectively monitor and respond to security incidents become increasingly critical. It is our sincere hope that LogChat contributes, in some measure, to making the digital world a safer place.

"Security is not a product, but a process."
— Bruce Schneier

References

Bibliography

[1] IBM Security. (2024). *Cost of a Data Breach Report 2024*. IBM Corporation.

[2] NIST. (2023). *Cybersecurity Framework 2.0*. National Institute of Standards and Technology.

[3] MITRE Corporation. (2024). *ATT&CK Framework*. Retrieved from https://attack.mitre.org/

[4] Gartner. (2024). *Magic Quadrant for Security Information and Event Management*. Gartner, Inc.

[5] Elastic N.V. (2024). *Elasticsearch Reference*. Retrieved from https://www.elastic.co/guide/

[6] Ollama. (2024). *Ollama Documentation*. Retrieved from https://ollama.ai/

[7] OpenAI. (2024). *GPT-4 Technical Report*. Retrieved from https://openai.com/research/

[8] Lewis, P., et al. (2020). *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. arXiv:2005.11401.

[9] Vercel. (2024). *Next.js Documentation*. Retrieved from https://nextjs.org/docs/

[10] Prisma. (2024). *Prisma ORM Documentation*. Retrieved from https://www.prisma.io/docs/

[11] PostgreSQL Global Development Group. (2024). *PostgreSQL 16 Documentation*. Retrieved from https://www.postgresql.org/docs/16/

[12] Go Team. (2024). *The Go Programming Language Specification*. Retrieved from https://go.dev/ref/spec

[13] Docker Inc. (2024). *Docker Documentation*. Retrieved from https://docs.docker.com/

[14] OWASP Foundation. (2024). *OWASP Top Ten*. Retrieved from https://owasp.org/Top10/

[15] CVE. (2024). *Common Vulnerabilities and Exposures*. Retrieved from https://cve.mitre.org/

Appendix A

Source Code Listings

A.1 Golang Agent Code

```
golang-agent/
 cmd/
    agent/
        main.go           # Entry point
 internal/
    collector/
       collector.go      # Interface definition
       file.go           # File tail collector
       eventlog_windows.go
       journald_linux.go
       syslog_linux.go
    sender/
       sender.go         # HTTP transmission
    buffer/
       buffer.go         # Ring buffer
    config/
        config.go         # YAML parsing
 config.yaml               # Sample config
 go.mod
 go.sum
 Makefile                  # Build targets
```

Listing A.1: Agent Project Directory Structure

```go
// internal/collector/collector.go
package collector

import (
    "context"
    "time"
)

// Collector defines the interface for all log collectors
type Collector interface {
    // Name returns the unique identifier for this collector
    Name() string

```

```
    // Start begins log collection in a goroutine
    Start(ctx context.Context)

    // Stop gracefully terminates the collector
    Stop()

    // Stats returns current collection statistics
    Stats() map[string]any
}

// BaseCollector provides common functionality
type BaseCollector struct {
    name          string
    sender        *sender.Sender
    logsCollected int64
    errorsCount   int64
    lastCollected time.Time
    running       bool
    mu            sync.RWMutex
}

func (bc *BaseCollector) Name() string {
    return bc.name
}

func (bc *BaseCollector) Stats() map[string]any {
    bc.mu.RLock()
    defer bc.mu.RUnlock()
    return map[string]any{
        "logs_collected": bc.logsCollected,
        "errors_count":   bc.errorsCount,
        "last_collected": bc.lastCollected,
        "running":        bc.running,
    }
}
```

Listing A.2: Collector Interface Definition

```
//go:build windows

package collector

import (
    "golang.org/x/sys/windows"
    "unsafe"
)

var (
    advapi32          = windows.NewLazySystemDLL("advapi32.dll")
```

```
    procOpenEventLogW = advapi32.NewProc("OpenEventLogW")
    procReadEventLogW = advapi32.NewProc("ReadEventLogW")
)

func (ec *EventLogCollector) openEventLog(channel string) (
    windows.Handle, error) {

    channelPtr, _ := syscall.UTF16PtrFromString(channel)
    ret, _, err := procOpenEventLogW.Call(
        0, // Local computer
        uintptr(unsafe.Pointer(channelPtr)),
    )
    if ret == 0 {
        return 0, fmt.Errorf("OpenEventLog: %v", err)
    }
    return windows.Handle(ret), nil
}

func (ec *EventLogCollector) readEvents(
    handle windows.Handle) ([]LogEntry, error) {

    buffer := make([]byte, 64*1024) // 64KB buffer
    var bytesRead, minBytes uint32

    ret, _, _ := procReadEventLogW.Call(
        uintptr(handle),
        uintptr(EVENTLOG_SEQUENTIAL_READ|EVENTLOG_FORWARDS_READ),
        0,
        uintptr(unsafe.Pointer(&buffer[0])),
        uintptr(len(buffer)),
        uintptr(unsafe.Pointer(&bytesRead)),
        uintptr(unsafe.Pointer(&minBytes)),
    )
    // Parse EVENTLOGRECORD structures...
}
```

Listing A.3: Windows Event Log Collection

```
func (fc *FileCollector) tailFile(ctx context.Context,
    filePath string) {

    t, err := tail.TailFile(filePath, tail.Config{
        Follow:    true,           // Keep following
        ReOpen:    true,           // Handle rotation
        MustExist: true,
        Location:  &tail.SeekInfo{
            Offset: 0,
            Whence: io.SeekEnd,    // Start at end
        },
```

```
        Logger: tail.DiscardingLogger,
    })
    if err != nil {
        fc.logError(err)
        return
    }
    defer t.Cleanup()

    for {
        select {
        case <-ctx.Done():
            return
        case line := <-t.Lines:
            if line.Err != nil {
                fc.logError(line.Err)
                continue
            }
            entry := fc.processLine(filePath, line.Text)
            fc.sender.Send(entry)
            fc.incrementStats()
        }
    }
}
```

Listing A.4: File Tailing with Rotation Support

```
func (s *Sender) flush(ctx context.Context) error {
    entries := s.buffer.Peek(s.batchSize)
    if len(entries) == 0 {
        return nil
    }

    payload, _ := json.Marshal(IngestPayload{
        Agent: s.agentInfo,
        Logs:  entries,
    })

    var lastErr error
    for attempt := 0; attempt < s.maxRetries; attempt++ {
        req, _ := http.NewRequestWithContext(
            ctx, "POST", s.serverURL, bytes.NewReader(payload))
        req.Header.Set("Content-Type", "application/json")
        req.Header.Set("X-API-Key", s.apiKey)

        resp, err := s.client.Do(req)
        if err == nil && resp.StatusCode == 201 {
            s.buffer.Remove(len(entries))
            s.sentCount += int64(len(entries))
            return nil
```

```
        }

        lastErr = err
        delay := time.Duration(1<<attempt) * 100 * time.Millisecond
        time.Sleep(delay) // Exponential backoff
    }

    s.serverAlive = false
    return lastErr
}
```

Listing A.5: Sender with Retry Logic

```
BINARY := logchat-agent
VERSION := 1.0.0
LDFLAGS := -s -w -X main.Version=$(VERSION)

.PHONY: build-all
build-all: build-linux-amd64 build-linux-arm64 \
           build-windows-amd64 build-darwin-amd64

build-linux-amd64:
        GOOS=linux GOARCH=amd64 go build \
            -ldflags="$(LDFLAGS)" \
            -o dist/$(BINARY)-linux-amd64 ./cmd/agent

build-windows-amd64:
        GOOS=windows GOARCH=amd64 go build \
            -ldflags="$(LDFLAGS)" \
            -o dist/$(BINARY)-windows-amd64.exe ./cmd/agent
```

Listing A.6: Makefile for Cross-Compilation

A.2 Backend Code

```
backend/
 src/
    index.ts            # Entry point
    routes/
       auth.ts         # Authentication
       logs.ts         # Log ingestion/query
       chat.ts         # AI chat
       stream.ts       # SSE endpoints
    services/
       auth.ts         # Auth business logic
       logs.ts         # Log operations
       chat.ts         # Chat session management
```

```
        ollama.ts       # Ollama client
        aiManager.ts    # Multi-provider adapter
        threatDetection.ts
    middleware/
         auth.ts         # JWT validation
         rateLimit.ts
 prisma/
    schema.prisma       # Database schema
 package.json
 tsconfig.json
 Dockerfile
```

Listing A.7: Backend Project Structure

```
// src/index.ts
import express from 'express';
import cors from 'cors';
import helmet from 'helmet';
import compression from 'compression';

import { authRouter } from './routes/auth.js';
import { logsRouter } from './routes/logs.js';
import { chatRouter } from './routes/chat.js';
import { streamRouter } from './routes/stream.js';
import { errorHandler } from './middleware/error.js';
import { requestLogger } from './middleware/logging.js';

const app = express();
const PORT = process.env.PORT || 3001;

// Security middleware
app.use(helmet());
app.use(cors({
    origin: process.env.FRONTEND_URL || 'http://localhost:3000',
    credentials: true,
}));
app.use(compression());
app.use(express.json({ limit: '10mb' }));

// Request logging
app.use(requestLogger);

// Health check
app.get('/health', (req, res) => {
    res.json({ status: 'healthy', timestamp: new Date() });
});

// API Routes
app.use('/api/auth', authRouter);
```

```
app.use('/api/logs', logsRouter);
app.use('/api/chat', chatRouter);
app.use('/api/stream', streamRouter);

// Error handling
app.use(errorHandler);

// Start server
app.listen(PORT, () => {
    console.log(`LogChat Backend running on port ${PORT}`);
});
```

Listing A.8: Main Application Entry Point

```
// src/services/auth.ts
import bcrypt from 'bcryptjs';
import jwt from 'jsonwebtoken';
import { prisma } from '../lib/prisma.js';

const JWT_SECRET = process.env.JWT_SECRET!;
const JWT_EXPIRY = '7d';
const BCRYPT_ROUNDS = 12;

export async function loginUser(
    email: string,
    password: string
): Promise<AuthResult> {
    // Find user by email
    const user = await prisma.user.findUnique({
        where: { email }
    });

    if (!user || !user.active) {
        return {
            success: false,
            error: 'Invalid credentials'
        };
    }

    // Verify password with bcrypt
    const isValid = await bcrypt.compare(
        password,
        user.password
    );
    if (!isValid) {
        return {
            success: false,
            error: 'Invalid credentials'
        };
```

```
    }

    // Generate JWT token
    const token = jwt.sign(
        {
            userId: user.id,
            role: user.role,
            email: user.email
        },
        JWT_SECRET,
        { expiresIn: JWT_EXPIRY }
    );

    // Create session record for audit
    await prisma.session.create({
        data: {
            token: hashToken(token),
            userId: user.id,
            expiresAt: new Date(Date.now() + 7*24*60*60*1000),
        }
    });

    // Update last login timestamp
    await prisma.user.update({
        where: { id: user.id },
        data: { lastLogin: new Date() }
    });

    return {
        success: true,
        user: sanitizeUser(user),
        token
    };
}
```

Listing A.9: JWT Authentication Service

```
// src/services/threatDetection.ts

interface ThreatResult {
    detected: boolean;
    type?: string;
    severity?: 'LOW' | 'MEDIUM' | 'HIGH' | 'CRITICAL';
    description?: string;
    mitre?: string;
}

const THREAT_PATTERNS = [
    {
```

```
        name: 'SQL_INJECTION',
        pattern: /('|--|;|OR\s+1\s*=\s*1|UNION\s+SELECT)/i,
        severity: 'CRITICAL' as const,
        mitre: 'T1190',
        description: 'Potential SQL injection attempt'
    },
    {
        name: 'XSS_ATTEMPT',
        pattern: /<script|javascript:|onerror\s*=|onload\s*=/i,
        severity: 'HIGH' as const,
        mitre: 'T1059.007',
        description: 'Cross-site scripting attempt'
    },
    {
        name: 'PATH_TRAVERSAL',
        pattern: /\.\.\/(\.\.\|etc\/passwd|boot\.ini)/i,
        severity: 'CRITICAL' as const,
        mitre: 'T1083',
        description: 'Directory traversal attempt'
    },
];

export function analyzeThreat(log: LogEntry): ThreatResult {
    const content = `${log.message} ${JSON.stringify(log.meta)}`;

    for (const threat of THREAT_PATTERNS) {
        if (threat.pattern.test(content)) {
            return {
                detected: true,
                type: threat.name,
                severity: threat.severity,
                description: threat.description,
                mitre: threat.mitre,
            };
        }
    }

    return { detected: false };
}
```

Listing A.10: Pattern-Based Threat Detection

```
// src/routes/chat.ts

router.post('/', authMiddleware, async (req, res) => {
    const { message, sessionId, filters } = req.body;
    const userId = req.user.id;

    // 1. RETRIEVAL: Get relevant logs for context
```

```
    const logs = await queryLogs({
        level: ['ERROR', 'WARN'],
        startTime: subHours(new Date(), filters?.hours || 1),
        limit: 100,
        search: extractKeywords(message),
    });

    // 2. Get aggregate statistics
    const stats = await getStatsForLLM(60);

    // 3. Build context-enriched prompt
    const context = buildLogContext(logs, stats);
    const systemPrompt = `You␣are␣LogChat,␣an␣AI␣security␣analyst.
Analyze␣the␣following␣log␣data␣and␣answer␣the␣user's␣question.
Be␣specific,␣cite␣log␣entries,␣and␣suggest␣remediation␣steps.

LOG␣CONTEXT:
${context}

STATISTICS:
-␣Total␣logs␣analyzed:␣${stats.totalLogs}
-␣Error␣rate:␣${stats.errorRate}%
-␣Top␣error␣services:␣${stats.topServices.join(',␣')}
`;

    // 4. GENERATION: Call LLM
    const startTime = Date.now();
    const response = await aiManager.chat({
        messages: [
            { role: 'system', content: systemPrompt },
            { role: 'user', content: message }
        ],
        provider: 'ollama',
        model: 'qwen2.5:0.5b',
    });
    const responseTime = Date.now() - startTime;

    // 5. Persist chat message
    await prisma.chatMessage.create({
        data: {
            sessionId,
            role: 'assistant',
            content: response.content,
            responseTime,
            tokensUsed: response.usage?.totalTokens,
            provider: response.provider,
        }
    });
```

```
56
57      res.json({
58          success: true,
59          response: response.content,
60          metadata: {
61              logsAnalyzed: logs.length,
62              responseTime,
63              provider: response.provider,
64          }
65      });
66  });
```

Listing A.11: RAG-Powered Chat Route

```
1   // src/routes/stream.ts
2
3   const clients = new Map<number, Response>();
4
5   router.get('/stats', authMiddleware, (req, res) => {
6       // Set SSE headers
7       res.setHeader('Content-Type', 'text/event-stream');
8       res.setHeader('Cache-Control', 'no-cache');
9       res.setHeader('Connection', 'keep-alive');
10      res.setHeader('X-Accel-Buffering', 'no');
11
12      const clientId = Date.now();
13      clients.set(clientId, res);
14
15      // Send initial connection event
16      res.write(`event: connected\ndata: ${JSON.stringify({
17  id: clientId,
18  timestamp: new Date(),
19  })}\n\n`);
20
21      // Periodic stats broadcast
22      const interval = setInterval(async () => {
23          const stats = await getRealtimeStats();
24          res.write(`event: stats\ndata: ${JSON.stringify(stats)}\n\n`);
25      }, 5000);
26
27      // Cleanup on disconnect
28      req.on('close', () => {
29          clearInterval(interval);
30          clients.delete(clientId);
31      });
32  });
33
34  // Broadcast function for log events
35  export function broadcastLog(log: LogEntry) {
```

```
    const data = JSON.stringify({ event: 'log', log });
    clients.forEach((client) => {
        client.write(`event: log\ndata: ${data}\n\n`);
    });
}
```

Listing A.12: SSE Stream for Real-time Updates

A.3 Frontend Code

```
frontend/
 app/
    layout.tsx          # Root layout
    page.tsx            # Landing page
    dashboard/
       page.tsx        # Main dashboard
    chat/
       page.tsx        # AI chat interface
    login/
       page.tsx        # Authentication
    admin/
        page.tsx        # Admin panel
 components/
    Navbar.tsx
    StatsCards.tsx
    LogsChart.tsx
    LogTable.tsx
    ChatInterface.tsx
 lib/
    api.ts
    auth.tsx            # Auth context
    utils.ts
 styles/
     globals.css
```

Listing A.13: Frontend Project Structure

```
// app/dashboard/page.tsx
'use client';

export default function DashboardPage() {
    const [stats, setStats] = useState<DashboardStats | null>(null);
    const [logs, setLogs] = useState<LogEntry[]>([]);
    const [connected, setConnected] = useState(false);

    // SSE connection
    useEffect(() => {
```

```
        const eventSource = new EventSource(
            `${API_URL}/api/stream/stats`,
            { withCredentials: true }
        );

        eventSource.addEventListener('connected', () => {
            setConnected(true);
        });

        eventSource.addEventListener('stats', (event) => {
            const data = JSON.parse(event.data);
            setStats(data);
        });

        eventSource.addEventListener('log', (event) => {
            const { log } = JSON.parse(event.data);
            setLogs(prev => [log, ...prev.slice(0, 99)]);

            // Toast notification for errors
            if (log.level === 'ERROR') {
                toast.error(log.message.slice(0, 100));
            }
        });

        return () => eventSource.close();
    }, []);

    return (
        <div className="dashboard-container">
            <LiveIndicator connected={connected} />
            <StatsCards stats={stats} />
            <div className="charts-row">
                <LogsChart data={stats?.timeline} />
                <ServicesChart data={stats?.topServices} />
            </div>
            <LogTable logs={logs} />
        </div>
    );
}
```

Listing A.14: Dashboard SSE Integration

```
// components/StatsCards.tsx

interface StatsCardsProps {
    stats: DashboardStats | null;
}

export function StatsCards({ stats }: StatsCardsProps) {
```

```
    const cards = [
        {
            title: 'Total Logs',
            value: stats?.totalLogs ?? 0,
            change: stats?.logsChange ?? 0,
            icon: Database,
            color: 'blue',
        },
        {
            title: 'Errors',
            value: stats?.errorCount ?? 0,
            change: stats?.errorsChange ?? 0,
            icon: AlertTriangle,
            color: 'red',
        },
        {
            title: 'Warnings',
            value: stats?.warnCount ?? 0,
            change: stats?.warningsChange ?? 0,
            icon: AlertCircle,
            color: 'yellow',
        },
        {
            title: 'Threats',
            value: stats?.threatCount ?? 0,
            change: stats?.threatsChange ?? 0,
            icon: Shield,
            color: 'purple',
        },
    ];

    return (
        <div className="stats-grid">
            {cards.map((card) => (
                <div key={card.title}
                     className={`stat-card stat-${card.color}`}>
                    <card.icon className="stat-icon" />
                    <div className="stat-content">
                        <span className="stat-value">
                            {formatNumber(card.value)}
                        </span>
                        <span className="stat-title">{card.title}</span>
                        <ChangeIndicator value={card.change} />
                    </div>
                </div>
            ))}
        </div>
    );
```

```
}
```

Listing A.15: Stats Cards Component

A.4 Database Schema

```
// prisma/schema.prisma

model User {
  id        String   @id @default(cuid())
  email     String   @unique
  password  String
  name      String
  role      Role     @default(USER)
  active    Boolean  @default(true)
  createdAt DateTime @default(now())
  lastLogin DateTime?

  sessions     Session[]
  chatSessions ChatSession[]
  auditLogs    AuditLog[]
}

model Log {
  id        String   @id @default(cuid())
  timestamp DateTime
  level     String
  service   String
  message   String   @db.Text
  raw       String   @db.Text
  meta      Json?
  sourceId  String?
  createdAt DateTime @default(now())

  @@index([timestamp])
  @@index([level])
  @@index([service])
}

model Alert {
  id             String        @id @default(cuid())
  logId          String?
  severity       AlertSeverity
  type           String
  message        String
  status         AlertStatus   @default(NEW)
  isAcknowledged Boolean       @default(false)
```

```
  createdAt      DateTime      @default(now())

  @@index([status])
  @@index([createdAt])
}
```

Listing A.16: Prisma Schema (Excerpt)

A.5 Testing and Deployment Code

```
// src/services/__tests__/threatDetection.test.ts

describe('ThreatDetectionEngine', () => {
    describe('SQL Injection Detection', () => {
        test('detects OR 1=1 pattern', () => {
            const log = createLog("' OR 1=1 --");
            const result = analyzeThreat(log);

            expect(result.detected).toBe(true);
            expect(result.type).toBe('SQL_INJECTION');
            expect(result.severity).toBe('CRITICAL');
        });

        test('detects UNION SELECT', () => {
            const log = createLog("UNION SELECT * FROM users");
            const result = analyzeThreat(log);

            expect(result.detected).toBe(true);
            expect(result.mitre).toBe('T1190');
        });
    });

    describe('XSS Detection', () => {
        test('detects script tag', () => {
            const log = createLog("<script>alert('xss')</script>");
            const result = analyzeThreat(log);

            expect(result.detected).toBe(true);
            expect(result.type).toBe('XSS_ATTEMPT');
            expect(result.severity).toBe('HIGH');
        });
    });

    describe('False Positive Prevention', () => {
        test('ignores normal log messages', () => {
            const log = createLog("User successfully logged in");
            const result = analyzeThreat(log);
```

```
38
39              expect(result.detected).toBe(false);
40          });
41      });
42  });
```

Listing A.17: Threat Detection Unit Test

```
1  describe('AuthService', () => {
2      describe('loginUser', () => {
3          test('succeeds␣with␣valid␣credentials', async () => {
4              const result = await loginUser(
5                  'admin@logchat.io',
6                  'password123'
7              );
8
9              expect(result.success).toBe(true);
10             expect(result.token).toBeDefined();
11             expect(result.user.role).toBe('ADMIN');
12         });
13
14         test('fails␣with␣wrong␣password', async () => {
15             const result = await loginUser(
16                 'admin@logchat.io',
17                 'wrongpassword'
18             );
19
20             expect(result.success).toBe(false);
21             expect(result.error).toBe('Invalid␣credentials');
22         });
23
24         test('fails␣for␣disabled␣account', async () => {
25             const result = await loginUser(
26                 'disabled@test.com',
27                 'password123'
28             );
29
30             expect(result.success).toBe(false);
31         });
32     });
33
34     describe('password␣hashing', () => {
35         test('uses␣bcrypt␣with␣cost␣factor␣12', async () => {
36             const hash = await hashPassword('test123');
37
38             expect(hash).toMatch(/^\$2[aby]?\$12\$/);
39         });
40     });
41 });
```

Listing A.18: Authentication Unit Tests

```
// tests/integration/api.test.ts

describe('Logs API', () => {
    let authToken: string;

    beforeAll(async () => {
        const res = await request(app)
            .post('/api/auth/login')
            .send({
                email: 'test@logchat.io',
                password: 'test123'
            });
        authToken = res.body.token;
    });

    describe('POST /api/logs/ingest', () => {
        test('ingests batch of logs', async () => {
            const response = await request(app)
                .post('/api/logs/ingest')
                .set('X-API-Key', 'test-api-key')
                .send({
                    agent: { hostname: 'test-server' },
                    logs: [
                        { level: 'INFO', message: 'Test log 1' },
                        { level: 'ERROR', message: 'Test error' },
                    ]
                });

            expect(response.status).toBe(201);
            expect(response.body.ingested).toBe(2);
        });

        test('rejects invalid API key', async () => {
            const response = await request(app)
                .post('/api/logs/ingest')
                .set('X-API-Key', 'invalid-key')
                .send({ logs: [] });

            expect(response.status).toBe(401);
        });
    });

    describe('GET /api/logs', () => {
        test('returns filtered logs', async () => {
            const response = await request(app)
```

```
                .get('/api/logs')
                .set('Authorization', `Bearer ${authToken}`)
                .query({ level: 'ERROR', limit: 10 });

            expect(response.status).toBe(200);
            expect(Array.isArray(response.body.logs)).toBe(true);
            response.body.logs.forEach((log: any) => {
                expect(log.level).toBe('ERROR');
            });
        });
    });
});
```

Listing A.19: API Integration Tests

```
describe('SSE Stream', () => {
    test('receives real-time log events', (done) => {
        const eventSource = new EventSource(
            'http://localhost:3001/api/stream/stats'
        );

        eventSource.addEventListener('connected', () => {
            // Trigger a log event
            ingestTestLog({ level: 'ERROR', message: 'Test' });
        });

        eventSource.addEventListener('log', (event) => {
            const data = JSON.parse(event.data);
            expect(data.log.level).toBe('ERROR');
            eventSource.close();
            done();
        });

        setTimeout(() => {
            eventSource.close();
            done(new Error('Timeout waiting for SSE event'));
        }, 5000);
    });
});
```

Listing A.20: SSE Integration Test

```
# Log ingestion endpoint
$ ab -n 10000 -c 100 -p payload.json -T application/json \
     -H "X-API-Key: test-key" \
     http://localhost:3001/api/logs/ingest
```

Listing A.21: Load Test Commands

```yaml
version: '3.8'

services:
  # PostgreSQL Database
  db:
    image: postgres:16-alpine
    restart: unless-stopped
    environment:
      POSTGRES_USER: logchat
      POSTGRES_PASSWORD: ${DB_PASSWORD:-logchat123}
      POSTGRES_DB: logchat
    volumes:
      - postgres_data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD", "pg_isready", "-U", "logchat"]
      interval: 5s
      timeout: 5s
      retries: 5
    networks:
      - logchat-net

  # Ollama AI Service
  ollama:
    image: ollama/ollama:latest
    restart: unless-stopped
    volumes:
      - ollama_data:/root/.ollama
    ports:
      - "11434:11434"
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: 1
              capabilities: [gpu]
    networks:
      - logchat-net

  # Backend API
  backend:
    build:
      context: ./backend
      dockerfile: Dockerfile
    restart: unless-stopped
    environment:
      NODE_ENV: production
      DATABASE_URL: postgresql://logchat:${DB_PASSWORD}@db:5432/logchat
```

```
      JWT_SECRET: ${JWT_SECRET}
      OLLAMA_URL: http://ollama:11434
    ports:
      - "3001:3001"
    depends_on:
      db:
        condition: service_healthy
    networks:
      - logchat-net

  # Frontend
  frontend:
    build:
      context: ./frontend
      dockerfile: Dockerfile
      args:
        NEXT_PUBLIC_API_URL: http://localhost:3001
    restart: unless-stopped
    environment:
      NODE_ENV: production
    ports:
      - "3000:3000"
    depends_on:
      - backend
    networks:
      - logchat-net

volumes:
  postgres_data:
  ollama_data:

networks:
  logchat-net:
    driver: bridge
```

Listing A.22: Complete docker-compose.yml

```
# Build stage
FROM node:20-alpine AS builder
WORKDIR /app

COPY package*.json ./
RUN npm ci

COPY . .
RUN npm run build

# Production stage
FROM node:20-alpine AS production
```

```
WORKDIR /app

RUN addgroup -g 1001 nodejs && \
    adduser -u 1001 -G nodejs -s /bin/sh -D nodejs

COPY --from=builder /app/dist ./dist
COPY --from=builder /app/node_modules ./node_modules
COPY --from=builder /app/package*.json ./
COPY --from=builder /app/prisma ./prisma

RUN npx prisma generate

USER nodejs
EXPOSE 3001
CMD ["node", "dist/index.js"]
```

Listing A.23: Backend Multi-stage Dockerfile

```
FROM node:20-alpine AS builder
WORKDIR /app

ARG NEXT_PUBLIC_API_URL
ENV NEXT_PUBLIC_API_URL=$NEXT_PUBLIC_API_URL

COPY package*.json ./
RUN npm ci

COPY . .
RUN npm run build

FROM node:20-alpine AS production
WORKDIR /app

ENV NODE_ENV=production

COPY --from=builder /app/.next/standalone ./
COPY --from=builder /app/.next/static ./.next/static
COPY --from=builder /app/public ./public

EXPOSE 3000
CMD ["node", "server.js"]
```

Listing A.24: Frontend Next.js Dockerfile

```
# Download agent
$AgentURL = "https://releases.logchat.io/v1.0.0/logchat-agent-windows-amd64.exe"
$InstallPath = "C:\Program Files\LogChat"

New-Item -ItemType Directory -Force -Path $InstallPath
```

```
Invoke-WebRequest -Uri $AgentURL -OutFile "$InstallPath\logchat-agent.exe"

# Create configuration
@"
server:
  url: "https://logchat.example.com:3001"
  api_key: "$env:LOGCHAT_API_KEY"

collectors:
  eventlog:
    enabled: true
    channels:
      - Application
      - System
      - Security
"@ | Out-File -FilePath "$InstallPath\config.yaml" -Encoding UTF8

# Install as Windows Service
New-Service -Name "LogChatAgent" `
    -BinaryPathName "$InstallPath\logchat-agent.exe --config $InstallPath\config.
    yaml" `
    -DisplayName "LogChat Log Agent" `
    -Description "Collects and forwards logs to LogChat server" `
    -StartupType Automatic

# Start the service
Start-Service LogChatAgent
```

Listing A.25: Windows Agent Installation Script

```
#!/bin/bash
set -e

VERSION="1.0.0"
INSTALL_DIR="/opt/logchat"
CONFIG_DIR="/etc/logchat"

# Download binary
curl -sSL "https://releases.logchat.io/v${VERSION}/logchat-agent-linux-amd64" \
    -o /tmp/logchat-agent
chmod +x /tmp/logchat-agent
sudo mv /tmp/logchat-agent "${INSTALL_DIR}/logchat-agent"

# Create configuration
sudo mkdir -p "${CONFIG_DIR}"
sudo tee "${CONFIG_DIR}/config.yaml" > /dev/null <<EOF
server:
  url: "https://logchat.example.com:3001"
  api_key: "${LOGCHAT_API_KEY}"
```

```
20
21 collectors:
22   files:
23     - paths:
24         - /var/log/syslog
25         - /var/log/auth.log
26         - /var/log/nginx/*.log
27       service: "linux-server"
28   journald:
29     enabled: true
30     units:
31       - docker
32       - nginx
33       - sshd
34 EOF
35
36 # Create systemd service
37 sudo tee /etc/systemd/system/logchat-agent.service > /dev/null <<EOF
38 [Unit]
39 Description=LogChat Log Collection Agent
40 After=network.target
41
42 [Service]
43 Type=simple
44 ExecStart=${INSTALL_DIR}/logchat-agent --config ${CONFIG_DIR}/config.yaml
45 Restart=always
46 RestartSec=5
47
48 [Install]
49 WantedBy=multi-user.target
50 EOF
51
52 # Enable and start
53 sudo systemctl daemon-reload
54 sudo systemctl enable logchat-agent
55 sudo systemctl start logchat-agent
```

Listing A.26: Linux Agent Installation Script

Appendix B

Glossary

Term	Definition
Attack Vector	A path or means by which an attacker can gain access to a system
bcrypt	A password hashing function designed for secure password storage
CUID	Collision-resistant Unique IDentifier, an alternative to UUID
Event Loop	Node.js's mechanism for executing non-blocking I/O operations
Goroutine	A lightweight thread managed by the Go runtime
JWT	JSON Web Token, a compact means of representing claims between parties
LLM	Large Language Model, an AI model trained on vast text data
MITRE ATT&CK	A knowledge base of adversary tactics and techniques
ORM	Object-Relational Mapping, a technique for converting data between systems
RAG	Retrieval Augmented Generation, combining search with AI generation
RBAC	Role-Based Access Control, restricting access based on user roles
SIEM	Security Information and Event Management, a security monitoring solution
SOC	Security Operations Center, a facility for monitoring security
SSE	Server-Sent Events, a standard for server-to-client streaming
Tail	Following a file as new content is appended (like `tail -f`)