# APILOGGUARD FOR SECURITY EVENTS

Major project submitted in partial fulfillment of the requirement for the award of the degree of

**BACHELOR OF TECHNOLOGY**

**IN**

**COMPUTER SCIENCE AND ENGINEERING**

Under the esteemed guidance of

**A Rahul**
**Assistant Professor**

By

**Abburi Bhavya Sri (21R11A05A5)**
**Kosuru Bharath Kumar (21R11A05C8)**

**Department of Computer Science and Engineering**

**Geethanjali College of Engineering and Technology (Autonomous)**

**Accredited by NAAC with A$^+$ Grade: B.Tech. CSE, EEE, ECE accredited by NBA** Sy. No: 33 & 34, Cheeryal (V), Keesara (M), Medchal District, Telangana – 501301

**MAY – 2025**

**Geethanjali College of Engineering and Technology (Autonomous)**

**Accredited by NAAC with A$^+$ Grade : B.Tech. CSE, EEE, ECE accredited by NBA** Sy. No: 33 & 34, Cheeryal (V), Keesara (M), Medchal District, Telangana – 501301

---

## Department of Computer Science and Engineering

# CERTIFICATE

This is to certify that the B.Tech Major Project report entitled **"APILogGuard For Security Events"** is a bonafide work done by **Abburi Bhavya Sri (21R11A05A5), Kosuru Bharath Kumar (21R11A05C8)** in partial fulfillment of the requirement of the award for the degree of Bachelor of Technology in "**Computer Science and Engineering**" from Jawaharlal Nehru Technological University, Hyderabad during the year 2024-2025.

**A Rahul**
**Assistant Professor**

**HoD – CSE**
**Dr. E. Ravindra**
**Professor**

**External Examiner**

# Geethanjali College of Engineering and Technology (Autonomous)

**Accredited by NAAC with A$^+$ Grade : B.Tech. CSE, EEE, ECE accredited by NBA** Sy. No: 33 & 34, Cheeryal (V), Keesara (M), Medchal District, Telangana – 501301

## Department of Computer Science and Engineering

## DECLARATION BY THE CANDIDATE

We, **Abburi Bhavya Sri, Kosuru Bharath Kumar**, bearing Roll Nos. **21R11A05A5, 21R11A05C8**, hereby declare that the project report entitled **"APILogGuard for Security Events"** is done under the guidance of **Mr. A Rahul**, **Assistant Professor**, Department of Computer Science and Engineering, Geethanjali College of Engineering and Technology, is submitted in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology in Computer Science and Engineering**.

This is a record of bonafide work carried out by me/us and the results embodied in this project have not been reproduced or copied from any source. The results embodied in this project report have not been submitted to any other University or Institute for the award of any other degree or diploma.

**Abburi Bhavya Sri, 21R11A05A5,**

**Kosuru Bharath Kumar, 21R11A05C8,**

**Department of CSE, Geethanjali College of Engineering and Technology,**

**Cheeryal.**

# ACKNOWLEDGEMENT

It is with profound gratitude and sincere appreciation that I extend my heartfelt thanks to all those who have played a significant role in the successful completion of this undergraduate project.

First and foremost, I express my deep sense of respect and gratitude to our **Honourable Chairman, Mr. G. R. Ravinder Reddy**, for his constant encouragement and for nurturing a culture of academic excellence and innovation within the institution.

I would also like to express my sincere thanks to **Dr. S. Udaya Kumar**, **Director,** for his visionary leadership and continued support, which have provided the ideal environment and motivation for academic pursuits such as this project.

My heartfelt appreciation goes to **Dr. K. Sagar**, **Principal,** for his steadfast guidance, infrastructural support, and encouragement that have helped bring this project to successful fruition.

I am deeply grateful to **Dr. E. Ravindra, Professor**, **Head of the Department- CSE**, for his academic leadership, valuable feedback, and continuous support throughout the duration of this project.

I extend my sincere thanks to our **Project Coordinators, Mr. S Durga Prasad, Mr. Krishna Rao**, for their meticulous planning, timely evaluations, and constant coordination that ensured a smooth and structured project development process.

A special note of gratitude is reserved for my project guide, **Mr. A Rahul, Assistant Professor**, whose expert supervision, insightful suggestions, and dedicated mentorship have been instrumental in shaping the direction and outcome of this work.

Lastly, I am ever grateful to my **parents and family** for their unconditional love, encouragement, and moral support. Their faith in me has been my greatest strength throughout this journey.

With genuine appreciation, I acknowledge every individual who has, in one way or another, contributed to the successful completion of this project.

**With warm regards,**
**Abburi Bhavya Sri (21R11A05A5)**
**Kosuru Bharath Kumar (21R11A05C8)**
**Department of Computer Science and Engineering**
**Geethanjali College of Engineering and Technology**

# ABSTRACT

In the evolving landscape of cybersecurity, Application Programming Interfaces (APIs) have become a critical backbone for communication between systems. However, APIs are often left exposed and are vulnerable to various forms of attacks such as brute-force login attempts, Distributed Denial of Service (DDoS) attacks, and anomaly-based intrusions. Existing monitoring tools like Prometheus, Grafana, and ELK Stack, while powerful, are often too heavy, expensive, or complex for small to mid-scale applications.

This project, **APILogGuard**, presents a lightweight, customizable solution for API security monitoring and anomaly detection. The system acts as an intermediary gateway between clients and servers, capturing, logging, and analyzing every incoming API request. It detects anomalous patterns based on predefined thresholds and statistical behaviors and alerts the administrator through a real-time dashboard interface. Administrators can also download comprehensive reports of anomalies and logs in JSON and PDF formats.

The backend is developed using Node.js and Express.js, and the frontend is built with React.js, TypeScript, and Tailwind CSS to ensure a responsive and interactive user experience. A dedicated CLI-based attack simulator has been incorporated to test the resilience of the system against various attack scenarios, including DDoS and brute-force simulations.

**APILogGuard** eliminates the dependency on third-party monitoring platforms by providing an all-in-one, self-hosted security event management system. It is highly extendable for future integration with persistent databases, advanced threat intelligence feeds, and real-time alerting mechanisms. This project is particularly beneficial for startups, educational institutions, and small enterprises seeking an efficient, cost-effective API security solution.

## List of Screen shots

## List of Figures

## List of Tables

# List of Abbreviations

| S No | Abbreviations | Full Form |
|---|---|---|
| 1 | API | Application Programming Interface |
| 2 | CLI | Command Line Interface |
| 3 | DDoS | Distributed Denial of Service |
| 4 | HTPP | Hypertext Transfer Protocol |
| 5 | IP Address | Internet Protocol Address |
| 6 | JSON | JavaScript Object Notation |
| 7 | JWT | JSON Web Token |
| 8 | KPI | Key Performance Indicator |
| 9 | OS | Operating System |
| 10 | PDF | Portable Document Format |
| 11 | SDLC | Software Development Life Cycle |
| 12 | SIEM | Security Information and Event Management |
| 13 | SRS | Software Requirements Specification |
| 14 | SQA | Software Quality Assurance |
| 15 | RAM | Random Access Memory |
| 16 | UI | User Interface |
| 17 | UML | Unified Modeling Language |

# Table of Contents

# 1. Introduction

This chapter introduces the APILogGuard project. It begins by outlining the core motivation and purpose behind the development of the system, followed by a discussion of the problems addressed and the primary objectives aimed to be achieved. It also defines the scope and boundaries of the system, explains the software development methodology adopted, and concludes with an outline of the report structure that follows in subsequent chapters.

## 1.1 Overview of the Project

In the modern era of cloud-native development, APIs serve as the critical communication bridge between systems, applications, and users. However, as APIs expose endpoints to the internet, they become prime targets for various types of attacks, such as brute force attempts, Distributed Denial of Service (DDoS) attacks, and payload manipulation.

The proposed project, titled **APILogGuard**, addresses this critical need by offering a lightweight, customizable security monitoring and anomaly detection system specifically for APIs.

It acts as a transparent gateway layer between client requests and the server, logging every interaction, analysing behaviour patterns, detecting anomalies, and generating real-time insights. Unlike conventional heavyweight platforms such as Prometheus or Grafana, APILogGuard provides a focused, low-resource solution that empowers developers and administrators to protect APIs effectively without incurring high costs or infrastructural complexity.

The system features a responsive dashboard built using React.js and Tailwind CSS, offering administrators intuitive access to traffic statistics, anomalies, and downloadable security reports in both JSON and PDF formats. Additionally, an integrated attack simulation tool enables testing of the system’s resilience under different threat conditions.

## 1.2 Problem Statement

Increased API adoption across industries has simultaneously introduced a surge in API-targeted cyberattacks. Traditional monitoring and security systems, although powerful, are often:

- Expensive to license and maintain.
- Complex to deploy for small and medium-sized applications.
- Rigid in adapting to highly customized API ecosystems.

Small development teams, startups, and educational institutions face significant hurdles in setting up efficient, affordable API monitoring solutions.

Thus, there is a pressing need for a lightweight, modular, extensible system capable of API request logging, threat detection, alerting, and reporting — without overwhelming resource demands or operational complexity.

## 1.3 Objectives of the Project

The primary objectives of **APILogGuard** are:

- To design and implement an API gateway gateway capable of capturing and logging every incoming request.
- To build an anomaly detection engine that identifies critical, suspicious, or benign activities based on real-time traffic analysis.
- To develop a user-friendly, real-time dashboard that visualizes API request statistics and threat intelligence.
- To enable administrators to download security logs and reports in structured formats (JSON, PDF) for auditing and review purposes.
- To eliminate the dependency on external heavyweight monitoring platforms by providing a fully self-hosted, modular solution.
- To create a simulation environment to test system performance under various attack conditions.

## 1.4 Scope of the Project

The scope of this project includes:

- Development of a complete end-to-end API logging and monitoring solution using open-source technologies.
- Real-time monitoring and anomaly detection of API request traffic without persistent database dependency (initially in-memory).
- Generation of security event reports downloadable by administrators.
- Provision of a Command Line Interface (CLI)-based attack simulator to validate system robustness against brute-force and DdoS attacks.
- Scalability for future enhancements, including persistent database integration (MongoDB/PostgreSQL) and real-time alerting (via WebSocket or email).

The current scope limits the system to in-memory log storage without permanent historical records, but provisions are made for database expansion in future versions.

## 1.5 SDLC Model Adopted

For the development of APILogGuard, the **Agile Iterative Model** was adopted.

- Development proceeded through multiple short cycles (iterations).

- Each iteration involved:
  - Planning
  - Design
  - Coding
  - Testing
  - Deployment
  - Review and feedback
- Continuous integration and testing were conducted after every module completion to ensure modular integrity and reduce cumulative errors.
- Requirements and designs were iteratively refined based on interim testing outcomes, ensuring flexibility and adaptability throughout the project lifecycle.

This methodology helped in early detection of issues, minimized risks, and ensured a highly adaptive and incremental development approach suitable for a security-sensitive project.

## 1.6 Organization of the Report

The report is organized into ten chapters as follows:

- Chapter 1: Introduction to the project including overview, objectives, problem statement, and methodology.
- Chapter 2: Literature Survey reviewing existing systems, their limitations, and the rationale for the proposed system.
- Chapter 3: System Analysis covering feasibility study, requirement specifications, and functional analysis.
- Chapter 4: System Design detailing architecture diagrams, data flow models, UML diagrams, and design standards.
- Chapter 5: Implementation explaining the technologies used, module development, code snippets, and coding standards.
- Chapter 6: Testing strategy, unit tests, integration tests, system tests, and bug tracking procedures.
- Chapter 7: Results showcasing output screens, result interpretations, and performance evaluations.
- Chapter 8: Conclusion and Future Scope discussing completed work, limitations, challenges, and future improvements.
- Chapter 9: References to technical papers, articles, and web resources consulted.
- Chapter 10: Appendices including SDLC artifacts, screenshots, reports, certificates, and deployment details.

# 2. Literature Survey

## 2.1 Review of Existing System

Current API security and monitoring solutions often rely on traditional log collection and visualization tools like Prometheus, Grafana, and the ELK Stack (Elasticsearch, Logstash, Kibana). While these platforms provide robust infrastructure for real-time data monitoring and dashboard visualization, they are generally resource-intensive and complex to configure for smaller projects or educational use. Additionally, enterprise SIEM tools like Splunk and IBM Qradar offer deep analytics and security monitoring features but are expensive and require advanced infrastructure. These existing systems are designed for large-scale deployments, making them less practical for lightweight or highly customized API monitoring solutions.

## 2.2 Limitations of Existing Approaches

While existing systems offer comprehensive monitoring capabilities, they come with notable limitations. Most tools have high memory and CPU overheads and require significant setup time and expertise. The licensing costs of premium features can be prohibitive, especially for academic, startup, or low-budget deployments. Furthermore, these systems offer limited flexibility in terms of real-time anomaly detection tailored to specific API patterns. Their generic dashboards often do not support advanced behavioral or statistical threat detection out-of-the-box. Most importantly, these systems typically do not provide customizable, low-latency alerting mechanisms or support simulated threat testing, which are essential for educational projects and agile development environments.

## 2.3 Need for the Proposed System

There is a clear need for a lightweight, customizable, and easily deployable API monitoring and threat detection system. The proposed system, APILogGuard, aims to fill this gap by offering a gateway-based architecture that tracks API requests, detects anomalies like brute force attacks or traffic bursts, and provides real-time insights through a React-based dashboard. Unlike existing solutions, APILogGuard emphasizes low infrastructure overhead, built-in attack simulation, and downloadable log reports in both JSON and PDF formats. Its modular design supports future integration with persistent databases and live alerting systems, making it suitable for both academic demonstrations and practical deployments in small-to-medium applications.

## 2.4 Comparative Study

### 2.4.1 Event-Driven API Gateways: Enabling Real-Time Communication in Modern Microservices Architecture (2024)

This paper explores the implementation of event-driven API gateways to support real-time communication and scalability in distributed systems. The authors highlight the benefits of asynchronous messaging, elasticity, and high performance. Case studies from companies like XYZ Corp and ABC Inc. show how adopting event-driven patterns can significantly reduce downtime and improve integration between microservices. The insights from this paper influenced APILogGuard's approach to building a resilient and real-time responsive gateway component.

### 2.4.2 Designing Robust API Monitoring Solutions (2023)

This research focuses on the architectural and technical challenges of API monitoring. The authors propose a solution called SNIPER that combines Dynamic Binary Instrumentation (DBI) and hardware-assisted virtualization to enable accurate and low-artifact monitoring. It is especially suited for debugging, malware detection, and security auditing. The relevance to APILogGuard lies in understanding how deeper system-level monitoring can be used to enrich API-level analysis and improve visibility under adversarial conditions.

### 2.4.3 WebAPI Evolution Patterns: A Usage-Driven Approach (2023)

This study investigates how real-world API usage patterns, derived from logs, can inform API redesign and optimization. It introduces process mining as a technique for analyzing consumer behavior and recommends changes such as parameter updates or endpoint merging. These principles align with APILogGuard's logging system, where API usage metrics are captured to identify patterns that may indicate misuse, overuse, or anomalies that require intervention or restructuring.

### 2.4.4 Data Visualization and Monitoring with Grafana and Prometheus (2021)

The authors present a detailed guide on deploying Prometheus for metric collection and Grafana for real-time data visualization in IT infrastructures. The paper emphasizes efficient data handling and visualization without replacing legacy tools. Although APILogGuard implements its own dashboard using React and Tailwind CSS, this paper influenced the system's UI design and inspired simplified, lightweight visualization alternatives for performance and anomaly tracking.

### 2.4.5 Design, Monitoring, and Testing of Microservices Systems: The Practitioners' Perspective (2021)

This paper explores the design strategies and monitoring practices employed in microservices development. Based on interviews and surveys with 106 practitioners, it identifies key challenges such as system complexity, decentralized logging, and lack of observability. APILogGuard directly addresses these issues through centralized request logging and simplified anomaly detection that are easy to integrate with microservice-based applications.

### 2.4.6 Log-Based Software Monitoring: A Systematic Mapping Study (2021)

This systematic review highlights the role of log data in software reliability, issue diagnosis, and behavior analysis. It also explores the use of machine learning for contextual log interpretation. This study shaped APILogGuard's design by reinforcing the value of well-structured logs for security use cases and motivating the addition of real-time anomaly classification and simulated alert generation based on log patterns.

### 2.4.7 Tools and Benchmarks for Automated Log Parsing (2019)

This evaluation covers various log parsing algorithms such as Drain, SLCT, and IPLoM, comparing their accuracy and scalability. It underscores the need to convert unstructured logs into structured formats to support analytics and detection. APILogGuard's export-to-JSON feature and plans for database integration align with the goals of structured log parsing to support detailed security audits and downstream ML applications.

### 2.4.8 What Public Transit API Logs Tell Us About Travel Flows (2016)

This study analyzes public API logs (from the iRail API) to understand commuter behavior using visual tools like Origin-Destination matrices. Although from a different domain, it demonstrates how API logs can uncover patterns in user behavior, informing improvements in system planning and service optimization. Similarly, APILogGuard uses request logs to track anomalies and inform adaptive response measures.

#### 2.4.9 Malware Detection Systems Based on API Log Data Mining (2015)

This paper presents a malware detection model using classification techniques like Naïve Bayes and SVM applied to dynamic API logs. It demonstrates that behavioral patterns in API usage are strong indicators of malicious activity. APILogGuard draws from this principle by detecting brute force and DdoS behaviors through traffic frequency, request types, and source patterns — without using traditional signature-based methods.

## 2.5 Summary

This chapter reviewed various existing systems and research efforts in the field of API monitoring and security. While tools such as Prometheus, Grafana, and ELK Stack offer robust monitoring and visualization capabilities, they often come with high setup complexity, infrastructure overhead, and limited adaptability for smaller or custom use cases. Commercial solutions like Splunk provide advanced analytics but are often inaccessible due to cost and technical requirements. The reviewed literature also highlights the growing emphasis on log analysis, anomaly detection, and the need for transparency in modern systems. To address these limitations, the proposed system, APILogGuard, offers a lightweight, modular, and developer-friendly solution that combines real-time threat detection, in-browser visualization, and report generation. Its flexible architecture makes it well-suited for academic projects, prototype development, and small-to-medium scale API security deployments.

# 3. System Analysis

This chapter outlines the analytical foundations on which the APILogGuard system is designed and developed. It begins with a feasibility study assessing the project's viability in technical, economic, and operational dimensions. It then defines the software requirements based on the system's intended functions, and concludes by identifying the core functional and non-functional expectations that guide the system's design and implementation.

## 3.1 Feasibility Study

To ensure that the proposed system is practical and implementable, a detailed feasibility study was conducted:

- Technical Feasibility: The project is technically feasible due to the use of well-supported open-source technologies. The backend is built using Node.js and Express.js, while the frontend is developed with React.js and Tailwind CSS. These technologies are efficient, lightweight, and suitable for real-time web applications. The project also utilizes PDFKit and Axios for PDF generation and HTTP communication, respectively. No proprietary technologies are required.
- Economic Feasibility: All core components are open-source, which means no licensing or infrastructure costs are incurred. The system can be deployed locally or on a minimal cloud instance, making it highly economical, especially for student projects, startups, and small-scale organizations.
- Operational Feasibility: The system is designed with simplicity and modularity in mind. It requires no complex configuration and can be operated through a straightforward user interface. The attack simulation and anomaly reporting modules are easy to trigger and monitor, making the platform usable even for non-security professionals.
- Time and Cost Estimation: Tasks were divided into iterations covering planning, design, implementation, testing, and documentation. Since development was local and tools were free, the overall cost of execution was negligible.

## 3.2 Software Requirements Specification (SRS)

The Software Requirements Specification describes the behavior, constraints, and interactions of the system being developed.

The following key requirements were identified:

- The system must serve as a gateway to intercept and log all incoming API requests with relevant metadata including IP address, endpoint, method, and timestamp.

- It must analyze real-time traffic to detect patterns indicative of brute-force attacks, DdoS behavior, or other suspicious activities.
- It must allow administrators to monitor live system activity through a web-based dashboard.
- It must offer options to download stored anomaly and log data in structured formats such as PDF and JSON.
- The backend should provide structured RESTful APIs for the frontend to retrieve data periodically.
- The system should simulate attack scenarios through a command-line utility to evaluate performance and resilience.

## 3.3 Functional and Non-Functional Requirements

The following are the specific requirements that drive system implementation and performance.

- Functional Requirements:
    - The system shall log every incoming API request using gateway gateway.
    - The system shall detect and classify anomalies into categories like critical, suspicious, and benign.
    - The system shall expose endpoints for retrieving anomaly statistics and threat indicators.
    - The frontend shall display real-time statistics, including total requests, anomaly counts, and success/error rates.
    - The system shall allow users to download anomaly logs and request history in JSON and PDF format.
    - The attack simulation tool shall support burst, brute force, DdoS, anomaly alert, and threat injection scenarios.
- Non-Functional Requirements:
    - The system shall handle at least 100 concurrent requests without failure.
    - All modules shall be modular, reusable, and scalable for future database or authentication integration.
    - Sensitive routes (such as log download) shall be protected using basic authentication.
    - The user interface shall be responsive and compatible with modern browsers.
    - The system shall log errors and unexpected behaviors for debugging and future improvements.

# 4. System Design

This chapter focuses on the structural and visual representation of the APILogGuard system. It outlines how the various components interact, how data flows through the system, and how security and design principles have been implemented. This phase ensures that the system's functional and non-functional requirements are translated into a clear architectural blueprint. Proper system design not only improves implementation but also helps in long-term scalability, maintainability, and risk mitigation.

## 4.1 System Architecture

The APILogGuard system adopts a modular, layered architecture. Each component is designed to perform a specific task — from request interception to anomaly detection and visualization. The architecture ensures real-time data flow from the backend to the frontend with minimal latency.

At a high level, the client sends API requests, which are intercepted by the Gateway. The gateway logs each request, forwards it to the threat detection engine, and then passes responses back to the client. The analysis results are exposed through secure RESTful APIs, which the frontend dashboard fetches periodically.

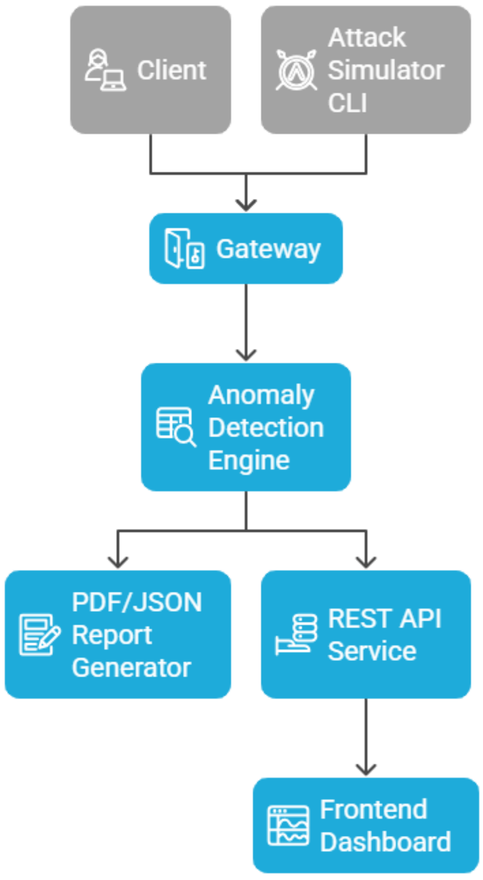


Fig 4.1.1 System Architecture

## 4.2 Data Flow Diagrams

The internal workflow follows a straightforward yet effective structure that ensures every incoming API call is captured, processed, and displayed to the user through the dashboard.

The flow of data:

- The system receives an API request from the client.
- The request is logged and passed through monitoring gateway.
- If a predefined anomaly pattern is matched, it is recorded as an anomaly.
- Gateway statistics (total requests, success rate, error rate) and anomaly data are updated in memory.
- The frontend dashboard periodically polls the backend every 10 seconds to refresh these statistics.
- Admins can trigger a download of logs as JSON or PDF through protected endpoints.

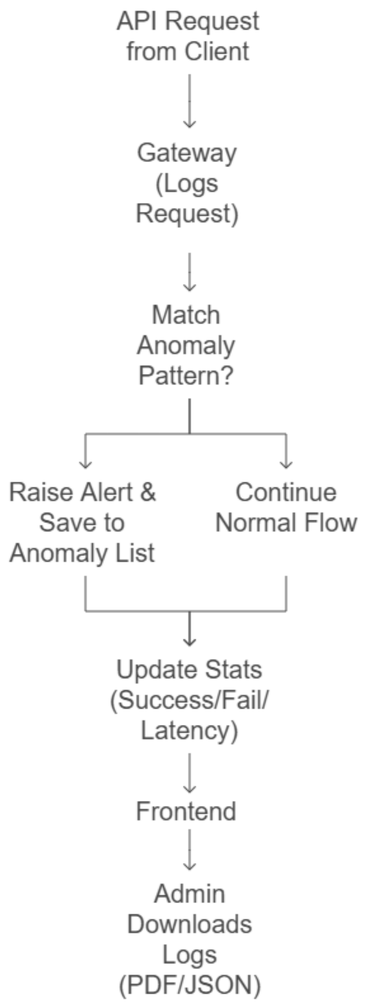

API Request from Client
Gateway (Logs Request)
Match Anomaly Pattern?
Raise Alert & Save to Anomaly List
Continue Normal Flow
Update Stats (Success/Fail/ Latency)
Frontend
Admin Downloads Logs (PDF/JSON)

Fig 4.2.1 Flow Chart

## 4.3 UML Diagrams

This section presents the UML diagrams used to model and explain the design logic and component interactions of the APILogGuard system. These diagrams offer a clear, visual representation of how different modules such as middleware, anomaly detection, and the frontend dashboard coordinate to deliver real-time API monitoring and security insights.

- Use Case Diagram:

  The use case diagram illustrates how the Admin interacts with different components of the APILogGuard system. The Admin can perform several actions such as viewing real-time API statistics, downloading logs in PDF or JSON format, simulating different types of attacks to test system resilience, injecting manual alerts for testing purposes, and reviewing stored threat indicators and anomalies. These interactions highlight the key administrative functionalities made available through the dashboard interface and backend endpoints, ensuring comprehensive monitoring and control over API behavior and security events.

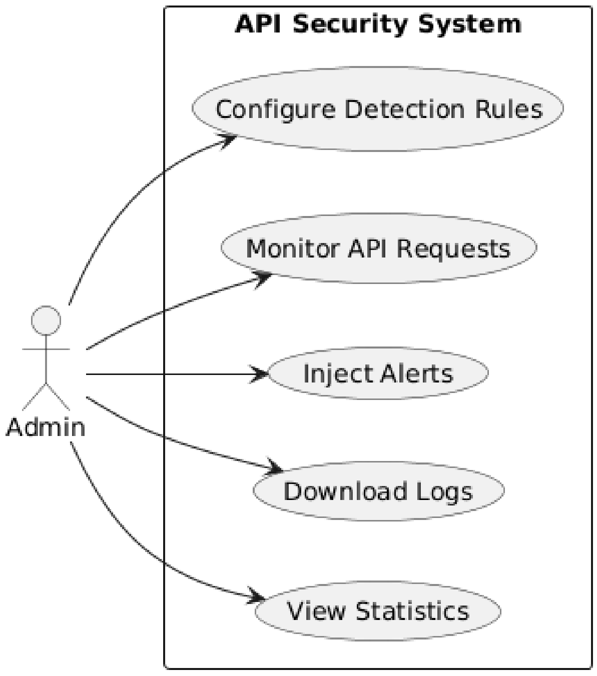


Fig 4.3.1 Use Case Diagram

- Class Diagram:

  The class diagram depicts the structural components of the APILogGuard system and their relationships. The main classes include:

  - APIRequest: Represents each logged API call, storing information such as IP address, method, status, endpoint, and timestamp.
  - AnomalyAlert: Captures details about anomalies detected in request patterns, including severity, type, source, and status.
  - ThreatIndicator: Manages threat intelligence data (e.g., suspicious Ips or payload patterns) and includes attributes like value, description, and severity.

  The relationships between these classes support modular logging, alert generation, and administrative review, forming the backbone of the system's monitoring framework.

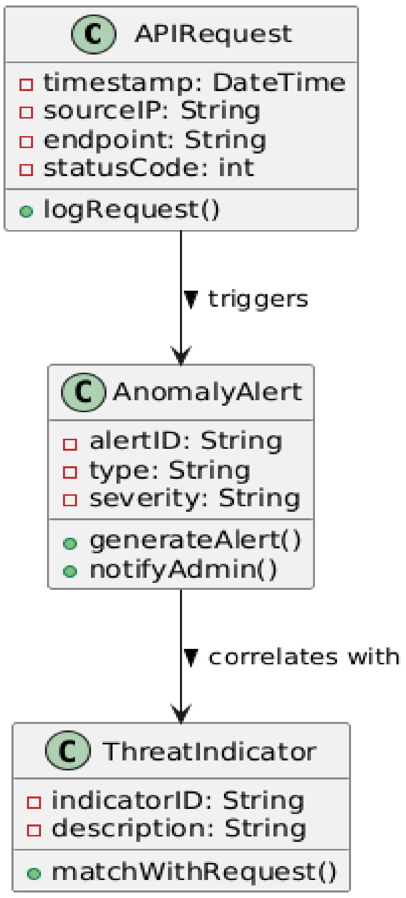


Fig 4.3.2 Class Diagram

- Activity Diagram:

  The activity diagram illustrates the sequence of operations performed by the system when handling an incoming API request. The workflow begins when the system receives a request, which is immediately logged by the middleware. The request is then analyzed by the anomaly detection engine. If an anomaly is identified, an alert is raised; otherwise, the request is forwarded for standard processing. In both cases, API traffic statistics are updated in memory. This data is made available to the frontend dashboard via REST API endpoints, enabling administrators to monitor request activity and anomaly trends in real time. The clear modularity in the activity flow supports scalability and reliability.

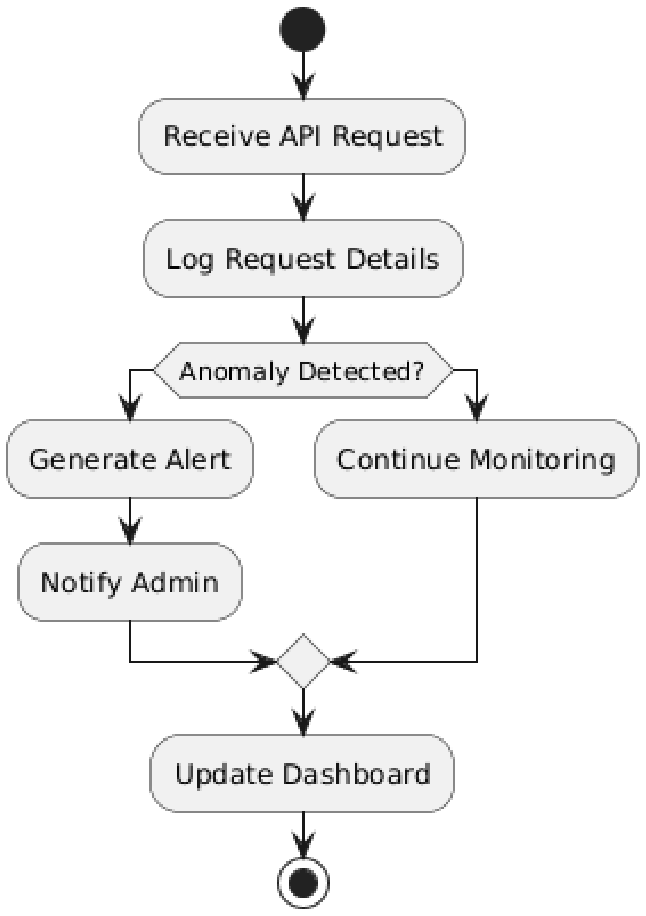


Fig 4.3.3 Activity Diagram

- Sequence Diagram:

  The sequence diagram illustrates the flow of data and operations from the moment an API request is sent by the client to the final rendering of statistics on the dashboard. The process begins when a client issues an API request, which is first intercepted by the Gateway Middleware. This module logs the request details and forwards it to the Anomaly Detector. If suspicious behavior is detected, an alert is generated. The updated statistics and alerts are saved in memory. Meanwhile, the frontend dashboard continuously polls the REST API (every 10 seconds), retrieving the latest request statistics and anomaly reports. This end-to-end process ensures real-time visibility and detection of threats across the API system.

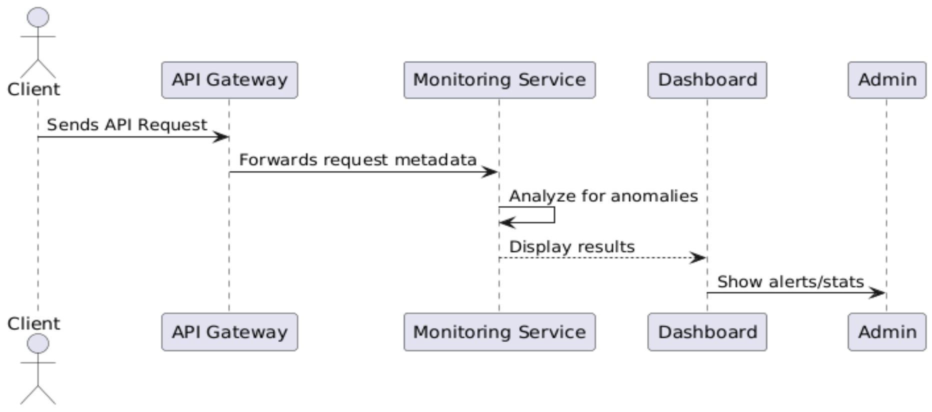


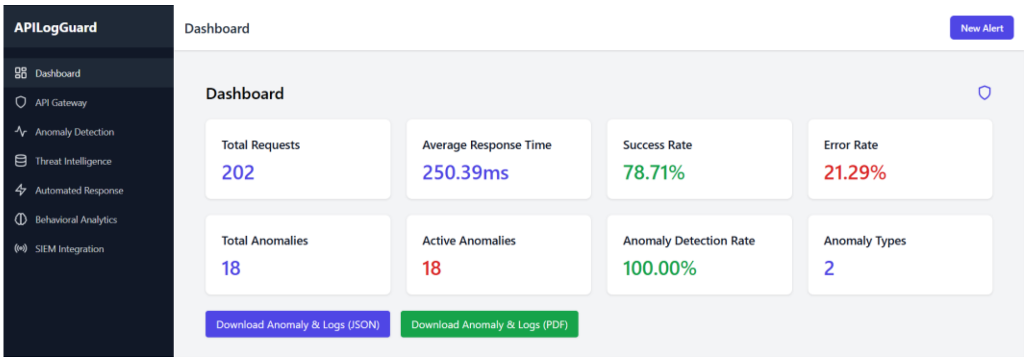
Fig 4.3.4. Sequence Diagram

## 4.4 User Interface Design

The frontend interface was built using React.js and Tailwind CSS. It features a modern, clean, and responsive layout that presents key security statistics in card-style panels. The dashboard includes real-time metrics such as:

- Total API Requests
- Success Rate
- Error Rate
- Average Response Time
- Active Anomalies

User interactions include log downloads, alert visualization, and navigation to sub-pages like Threat Intelligence and SIEM. Visual feedback elements (spinners, error messages, status badges) enhance usability and system clarity.



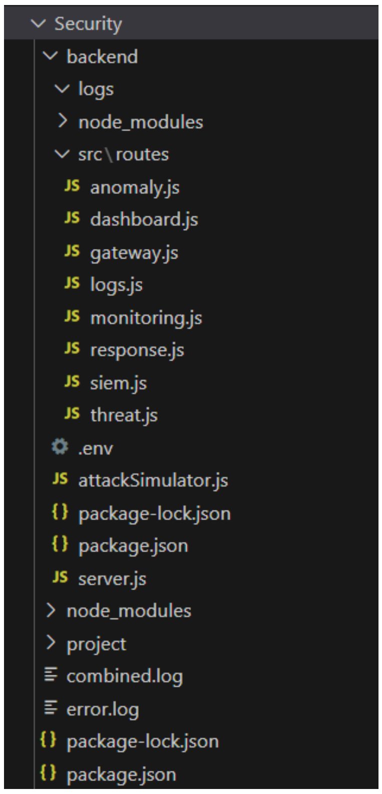
Total Requests 202
Average Response Time 250.39ms
Success Rate 78.71%
Error Rate 21.29%
Total Anomalies 18
Active Anomalies 18
Anomaly Detection Rate 100.00%
Anomaly Types 2

Fig 4.4.1 Dashboard Interface

## 4.5 Design Standards Followed

To ensure code consistency and long-term maintainability, the following design standards were adhered to:

- Frontend Codebase:
  - Airbnb TypeScript Style Guide
  - Component-based UI structure
  - Centralized API service layer (api.ts)
- Backend Codebase:
  - RESTful API design with Express.js best practices
  - Modular routing structure for each system feature
  - Proper use of HTTP status codes and error messages



```
Security
  backend
    logs
      node_modules
      src\routes
        anomaly.js
        dashboard.js
        gateway.js
        logs.js
        monitoring.js
        response.js
        siem.js
        threat.js
      .env
      attackSimulator.js
      package-lock.json
      package.json
      server.js
    node_modules
    project
    combined.log
    error.log
    package-lock.json
    package.json
```

Fig 4.5.1 Project File Structure

## 4.6 Safety & Risk Mitigation Measures

Security and stability were considered throughout system design. The following safeguards were included:

- Basic authentication on log download routes to prevent unauthorized access.
- CORS configuration to prevent cross-origin exploitation during API requests.
- Input sanitization and validation to avoid malformed or malicious data.
- Safe error handling to avoid leaking internal system info to external users.
- System modularization ensures that future upgrades (e.g., JWT auth, MongoDB integration) can be integrated with minimal risk.

# 5. Implementation

The implementation phase involves translating the design and analysis into executable modules using the selected technology stack. Each module in the project was developed and tested independently, and later integrated to form a fully functional security monitoring platform. This chapter outlines the technologies used, the implementation of key modules, integration strategy, and coding best practices followed throughout development.

## 5.1 Technology Stack

- React.js: Used for building the dashboard and user interface components in a modular and responsive manner.
- Node.js: A JavaScript runtime used to build the server-side logic and handle API requests.
- Express.js: A fast and minimalist web framework used for building RESTful API routes and applying gateway functions.
- PDFKit: A server-side PDF generation library used to create downloadable log and anomaly reports in PDF format.
- dotenv: A configuration tool used to manage environment variables securely (e.g., port number, authentication credentials).
- Visual Studio Code (VSCode): The primary integrated development environment (IDE) used for both frontend and backend development.
- npm: Node Package Manager used for installing and managing dependencies for the project.
- CLI-based Attack Simulator: A custom tool built with Node.js to simulate real-world API threats like DdoS, brute force, and anomaly injection.

## 5.2 Module-wise Implementation

The APILogGuard system is composed of multiple independent yet collaborative modules, each of which is designed to handle a specific aspect of API monitoring and threat detection. These modules collectively contribute to building a lightweight, scalable, and effective API security solution. The key modules are:

- Gateway: This module acts as the central gateway for all incoming API traffic. It intercepts each request and logs essential metadata such as the request method, endpoint, IP address, status code, and latency. These logs are used for both live monitoring and post-analysis. The gateway ensures that each request is processed in real-time before being passed to the backend service.
- Anomaly Detection Engine: This component is responsible for analyzing traffic patterns and detecting behavioral anomalies. It identifies events such as repeated failed logins (brute-force), high-frequency requests

(DdoS), and suspicious usage patterns. Anomalies are classified as critical, suspicious, or benign and stored temporarily for review via the dashboard.

- Dashboard Statistics Aggregator: This backend module compiles real-time statistics based on incoming requests and anomalies. It calculates total API calls, average latency, success/error rates, and the volume and severity of detected anomalies. The frontend dashboard polls this data periodically to provide up-to-date visual insights to the administrator.
- Log Reporting and Download Module: This module allows administrators to export API logs and anomaly data in both JSON and PDF formats. It uses a PDF generation library to create structured, timestamped reports. Access to these downloadable files is protected via Basic Authentication to prevent unauthorized exposure of security-sensitive data.
- Attack Simulation Module: A command-line interface (CLI) based tool built using Node.js that simulates different types of API-based attacks. These include:
  - Burst traffic (multiple rapid GET requests)
  - Brute-force login attempts
  - DdoS-like concurrent flooding
  - Manual anomaly injections
  - Fake threat indicator insertions

    This module is used to validate the robustness and responsiveness of the system under various stress conditions.

## 5.3 Code Integration Strategy

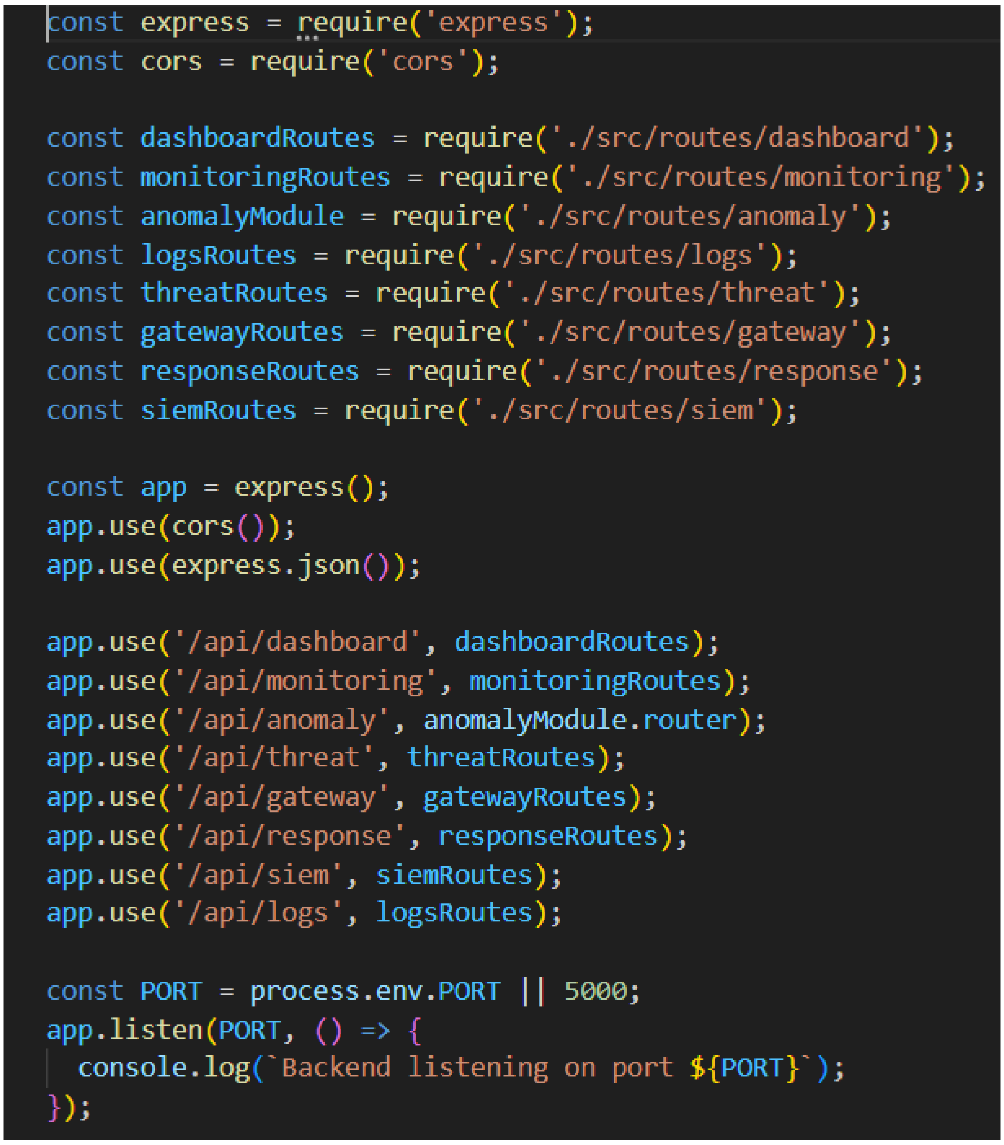
The system is designed using a modular, component-based approach where each functionality—such as request logging, anomaly detection, alert generation, and frontend interaction—is encapsulated in its own file or route. This separation of concerns improves maintainability, makes debugging easier, and enables future enhancements without disrupting core functionality.

On the backend, Express.js routes are grouped by domain (e.g., /gateway, /anomaly, /logs, /threat) to isolate responsibilities. The gateway, intercepting every API request and forwarding structured data to the anomaly detection engine. All logging, threat processing, and response modules are independent, allowing developers to modify one feature without affecting others.

The frontend communicates with the backend through a centralized API service file (api.ts), where each endpoint call is defined using Axios. React components utilize useEffect hooks for periodic polling and useState hooks for storing and displaying API data. This architecture ensures that the dashboard updates live without page reloads.

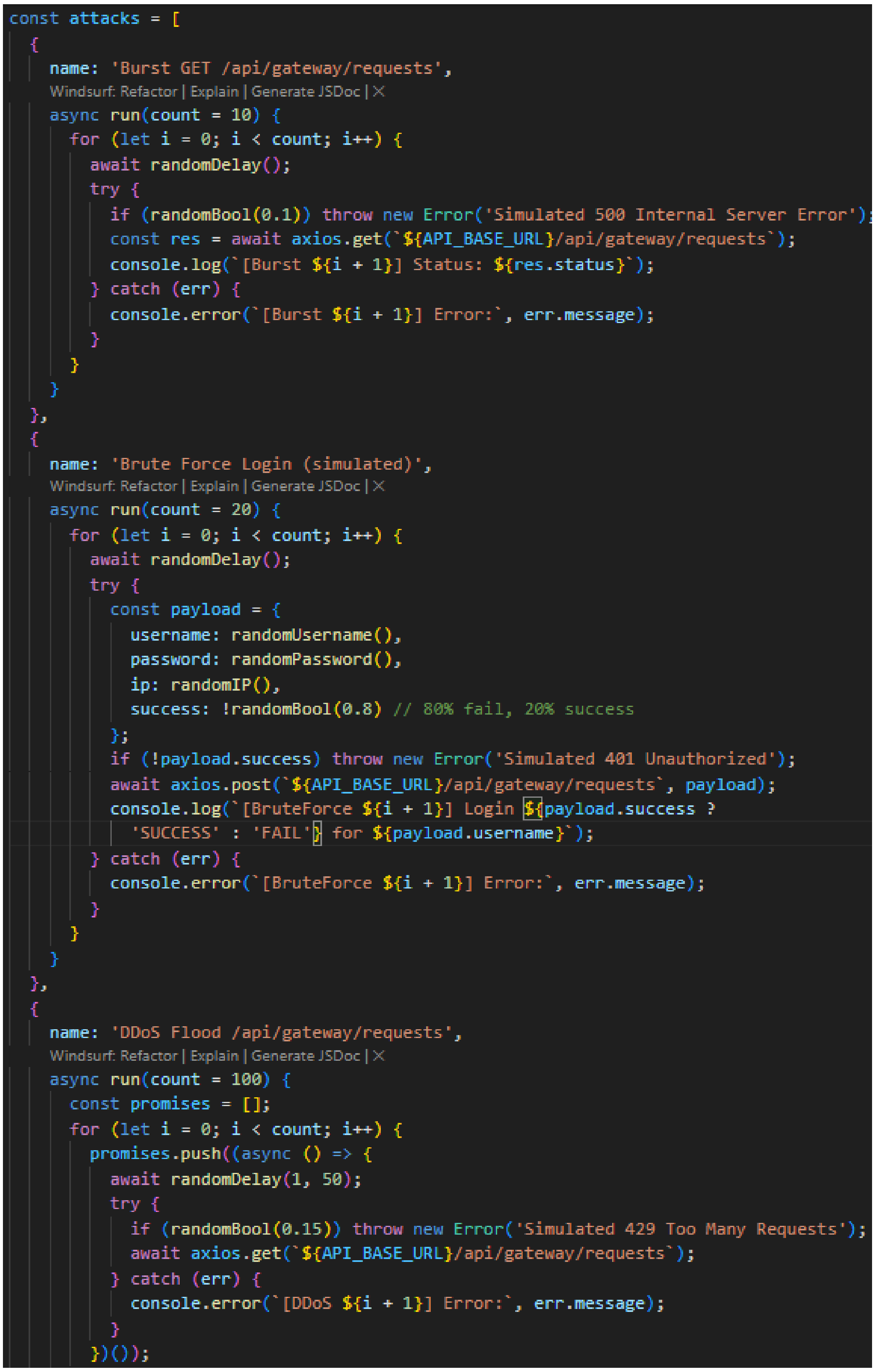
Authentication is integrated into the log reporting module, where sensitive routes (such as /logs/download and /logs/download/pdf) are protected using basic authentication headers.

## 5.4 Sample Code Snippets



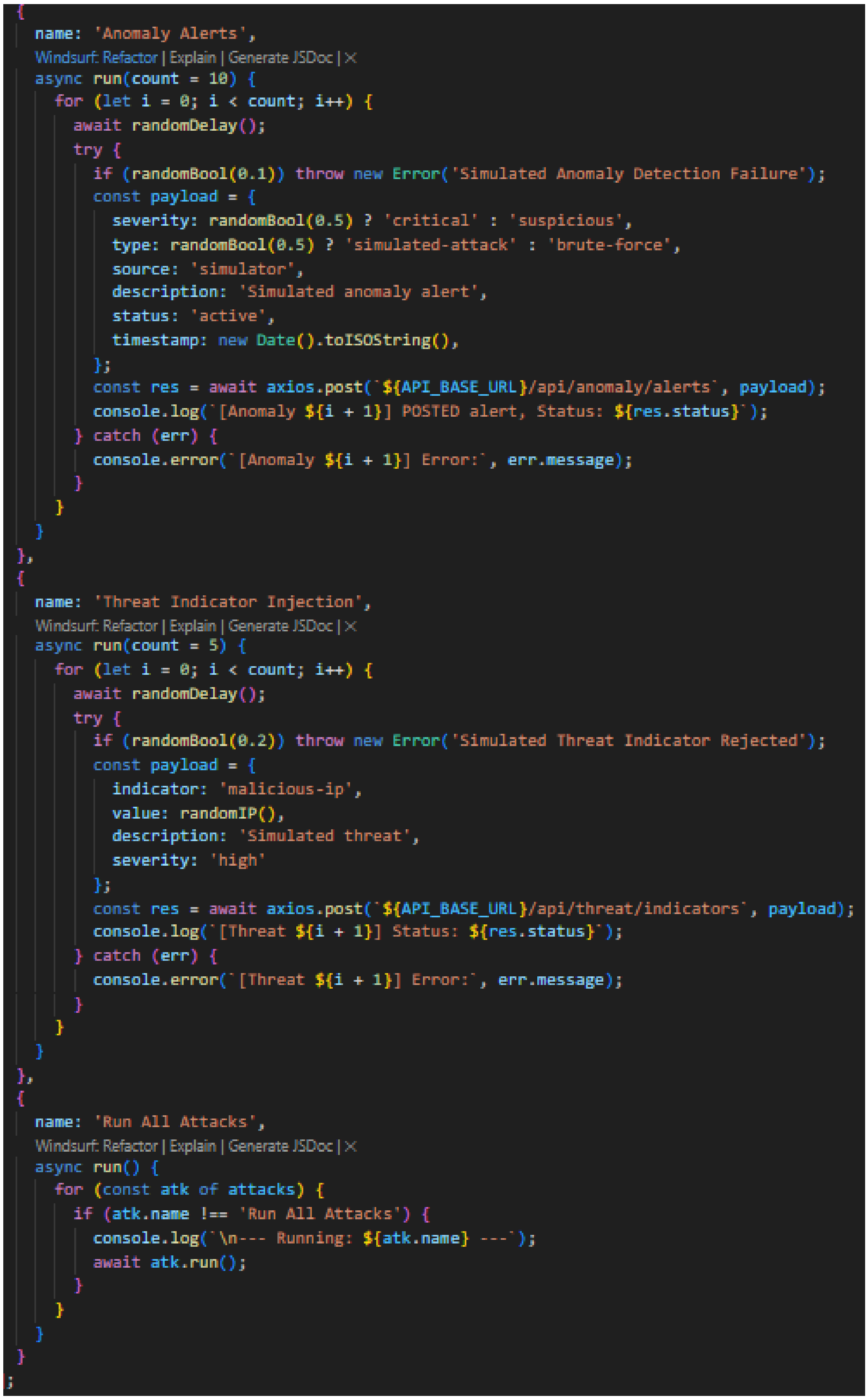
```
const express = require('express');
const cors = require('cors');

const dashboardRoutes = require('./src/routes/dashboard');
const monitoringRoutes = require('./src/routes/monitoring');
const anomalyModule = require('./src/routes/anomaly');
const logsRoutes = require('./src/routes/logs');
const threatRoutes = require('./src/routes/threat');
const gatewayRoutes = require('./src/routes/gateway');
const responseRoutes = require('./src/routes/response');
const siemRoutes = require('./src/routes/siem');

const app = express();
app.use(cors());
app.use(express.json());

app.use('/api/dashboard', dashboardRoutes);
app.use('/api/monitoring', monitoringRoutes);
app.use('/api/anomaly', anomalyModule.router);
app.use('/api/threat', threatRoutes);
app.use('/api/gateway', gatewayRoutes);
app.use('/api/response', responseRoutes);
app.use('/api/siem', siemRoutes);
app.use('/api/logs', logsRoutes);

const PORT = process.env.PORT || 5000;
app.listen(PORT, () => {
  console.log(`Backend listening on port ${PORT}`);
});
```

Fig 5.4.1 Server.js



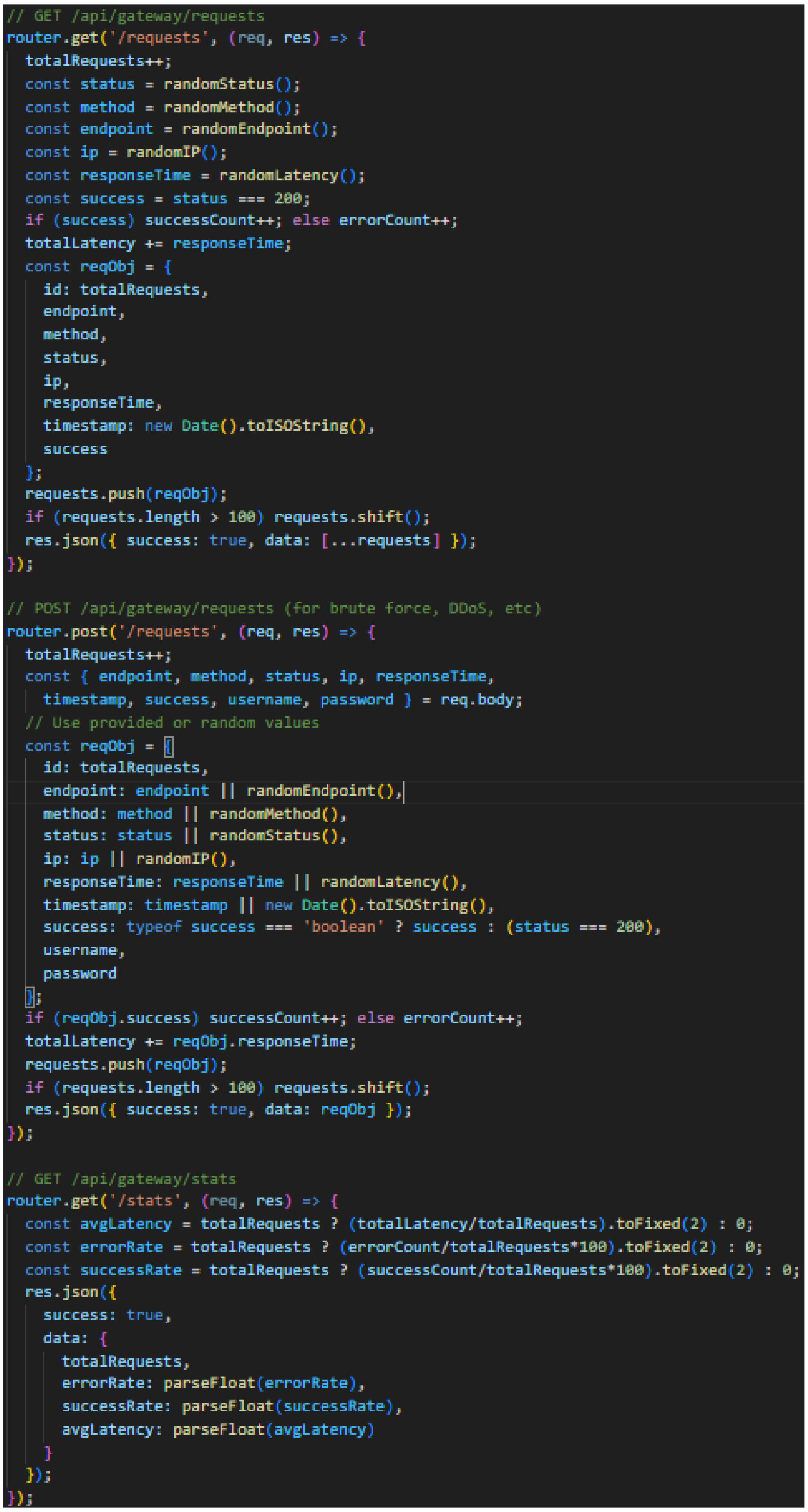
```javascript
const attacks = [
  {
    name: 'Burst GET /api/gateway/requests',
    Windsurf: Refactor | Explain | Generate JSDoc | ×
    async run(count = 10) {
      for (let i = 0; i < count; i++) {
        await randomDelay();
        try {
          if (randomBool(0.1)) throw new Error('Simulated 500 Internal Server Error');
          const res = await axios.get(`${API_BASE_URL}/api/gateway/requests`);
          console.log(`[Burst ${i + 1}] Status: ${res.status}`);
        } catch (err) {
          console.error(`[Burst ${i + 1}] Error:`, err.message);
        }
      }
    }
  },
  {
    name: 'Brute Force Login (simulated)',
    Windsurf: Refactor | Explain | Generate JSDoc | ×
    async run(count = 20) {
      for (let i = 0; i < count; i++) {
        await randomDelay();
        try {
          const payload = {
            username: randomUsername(),
            password: randomPassword(),
            ip: randomIP(),
            success: !randomBool(0.8) // 80% fail, 20% success
          };
          if (!payload.success) throw new Error('Simulated 401 Unauthorized');
          await axios.post(`${API_BASE_URL}/api/gateway/requests`, payload);
          console.log(`[BruteForce ${i + 1}] Login ${payload.success ?
            'SUCCESS' : 'FAIL'} for ${payload.username}`);
        } catch (err) {
          console.error(`[BruteForce ${i + 1}] Error:`, err.message);
        }
      }
    }
  },
  {
    name: 'DDoS Flood /api/gateway/requests',
    Windsurf: Refactor | Explain | Generate JSDoc | ×
    async run(count = 100) {
      const promises = [];
      for (let i = 0; i < count; i++) {
        promises.push((async () => {
          await randomDelay(1, 50);
          try {
            if (randomBool(0.15)) throw new Error('Simulated 429 Too Many Requests');
            await axios.get(`${API_BASE_URL}/api/gateway/requests`);
          } catch (err) {
            console.error(`[DDoS ${i + 1}] Error:`, err.message);
          }
        })());
```

Fig 5.4.2 attackStimulator.js



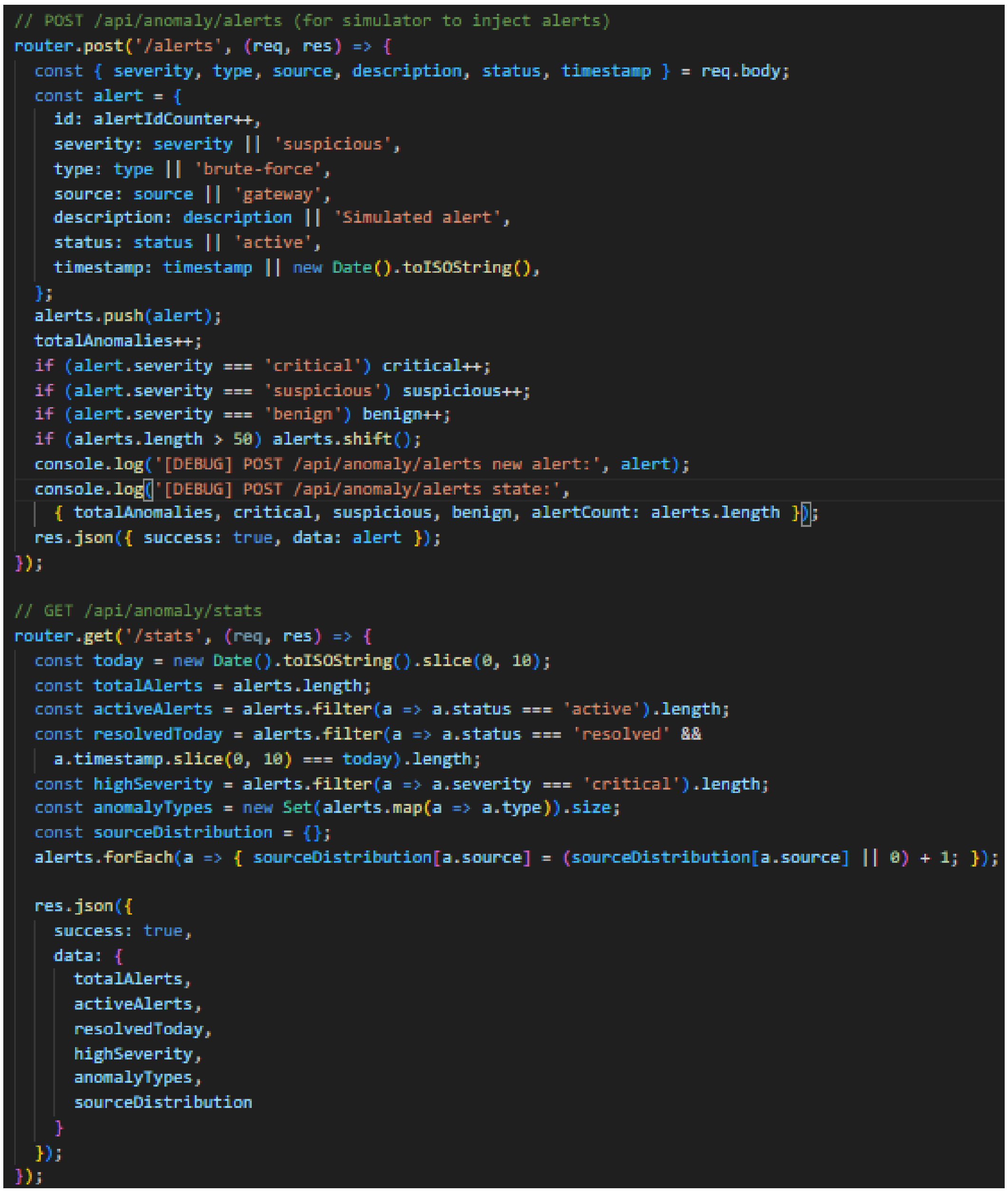
```
{
  name: 'Anomaly Alerts',
  Windsurf: Refactor | Explain | Generate JSDoc | ×
  async run(count = 10) {
    for (let i = 0; i < count; i++) {
      await randomDelay();
      try {
        if (randomBool(0.1)) throw new Error('Simulated Anomaly Detection Failure');
        const payload = {
          severity: randomBool(0.5) ? 'critical' : 'suspicious',
          type: randomBool(0.5) ? 'simulated-attack' : 'brute-force',
          source: 'simulator',
          description: 'Simulated anomaly alert',
          status: 'active',
          timestamp: new Date().toISOString(),
        };
        const res = await axios.post(`${API_BASE_URL}/api/anomaly/alerts`, payload);
        console.log(`[Anomaly ${i + 1}] POSTED alert, Status: ${res.status}`);
      } catch (err) {
        console.error(`[Anomaly ${i + 1}] Error:`, err.message);
      }
    }
  }
},
{
  name: 'Threat Indicator Injection',
  Windsurf: Refactor | Explain | Generate JSDoc | ×
  async run(count = 5) {
    for (let i = 0; i < count; i++) {
      await randomDelay();
      try {
        if (randomBool(0.2)) throw new Error('Simulated Threat Indicator Rejected');
        const payload = {
          indicator: 'malicious-ip',
          value: randomIP(),
          description: 'Simulated threat',
          severity: 'high'
        };
        const res = await axios.post(`${API_BASE_URL}/api/threat/indicators`, payload);
        console.log(`[Threat ${i + 1}] Status: ${res.status}`);
      } catch (err) {
        console.error(`[Threat ${i + 1}] Error:`, err.message);
      }
    }
  }
},
{
  name: 'Run All Attacks',
  Windsurf: Refactor | Explain | Generate JSDoc | ×
  async run() {
    for (const atk of attacks) {
      if (atk.name !== 'Run All Attacks') {
        console.log(`\n--- Running: ${atk.name} ---`);
        await atk.run();
      }
    }
  }
}
;
```

Fig 5.4.3 attackStimulator.js



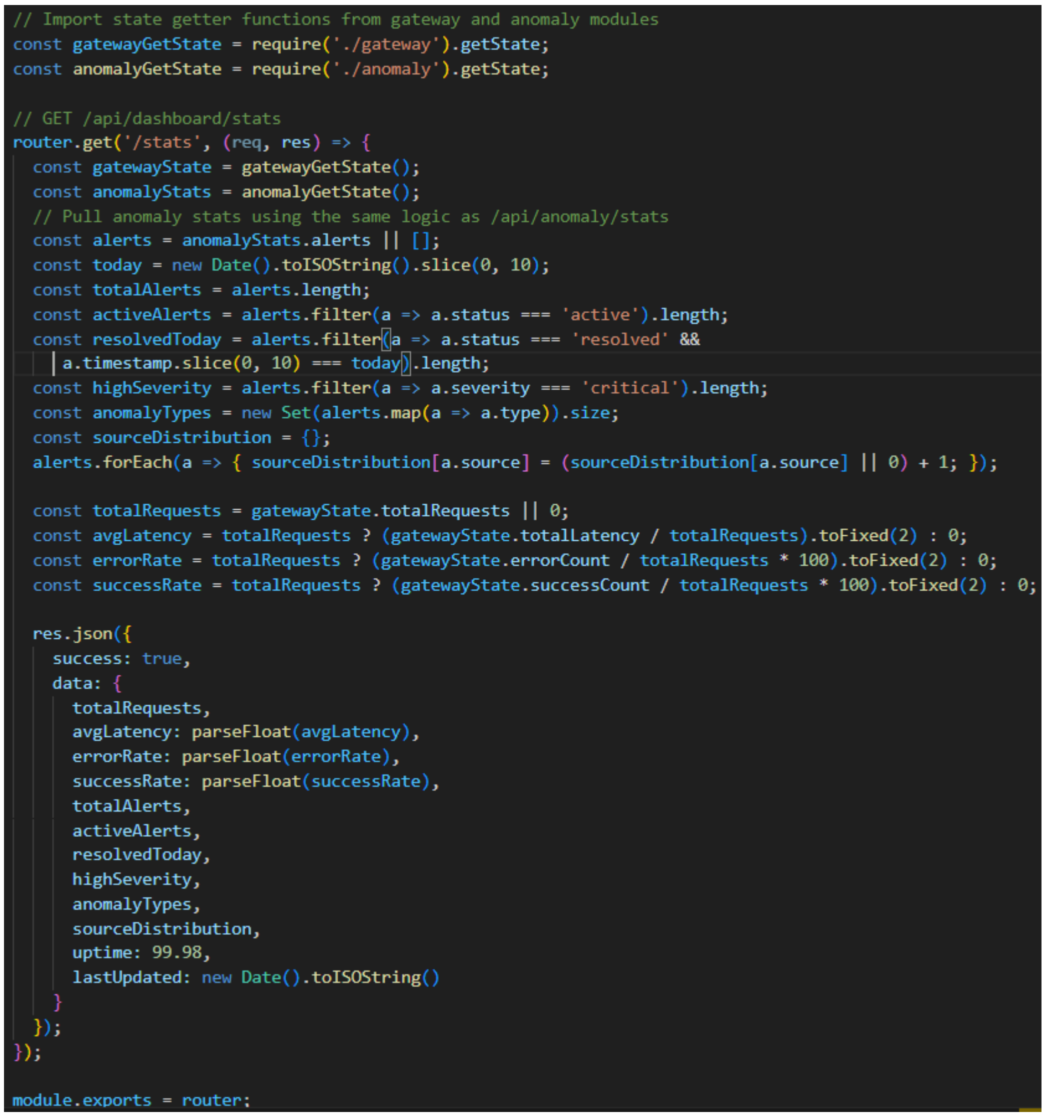
```
// GET /api/gateway/requests
router.get('/requests', (req, res) => {
  totalRequests++;
  const status = randomStatus();
  const method = randomMethod();
  const endpoint = randomEndpoint();
  const ip = randomIP();
  const responseTime = randomLatency();
  const success = status === 200;
  if (success) successCount++; else errorCount++;
  totalLatency += responseTime;
  const reqObj = {
    id: totalRequests,
    endpoint,
    method,
    status,
    ip,
    responseTime,
    timestamp: new Date().toISOString(),
    success
  };
  requests.push(reqObj);
  if (requests.length > 100) requests.shift();
  res.json({ success: true, data: [...requests] });
});

// POST /api/gateway/requests (for brute force, DDoS, etc)
router.post('/requests', (req, res) => {
  totalRequests++;
  const { endpoint, method, status, ip, responseTime,
    timestamp, success, username, password } = req.body;
  // Use provided or random values
  const reqObj = {
    id: totalRequests,
    endpoint: endpoint || randomEndpoint(),
    method: method || randomMethod(),
    status: status || randomStatus(),
    ip: ip || randomIP(),
    responseTime: responseTime || randomLatency(),
    timestamp: timestamp || new Date().toISOString(),
    success: typeof success === 'boolean' ? success : (status === 200),
    username,
    password
  };
  if (reqObj.success) successCount++; else errorCount++;
  totalLatency += reqObj.responseTime;
  requests.push(reqObj);
  if (requests.length > 100) requests.shift();
  res.json({ success: true, data: reqObj });
});

// GET /api/gateway/stats
router.get('/stats', (req, res) => {
  const avgLatency = totalRequests ? (totalLatency/totalRequests).toFixed(2) : 0;
  const errorRate = totalRequests ? (errorCount/totalRequests*100).toFixed(2) : 0;
  const successRate = totalRequests ? (successCount/totalRequests*100).toFixed(2) : 0;
  res.json({
    success: true,
    data: {
      totalRequests,
      errorRate: parseFloat(errorRate),
      successRate: parseFloat(successRate),
      avgLatency: parseFloat(avgLatency)
    }
  });
});
```

Fig 5.4.4 gateway.js



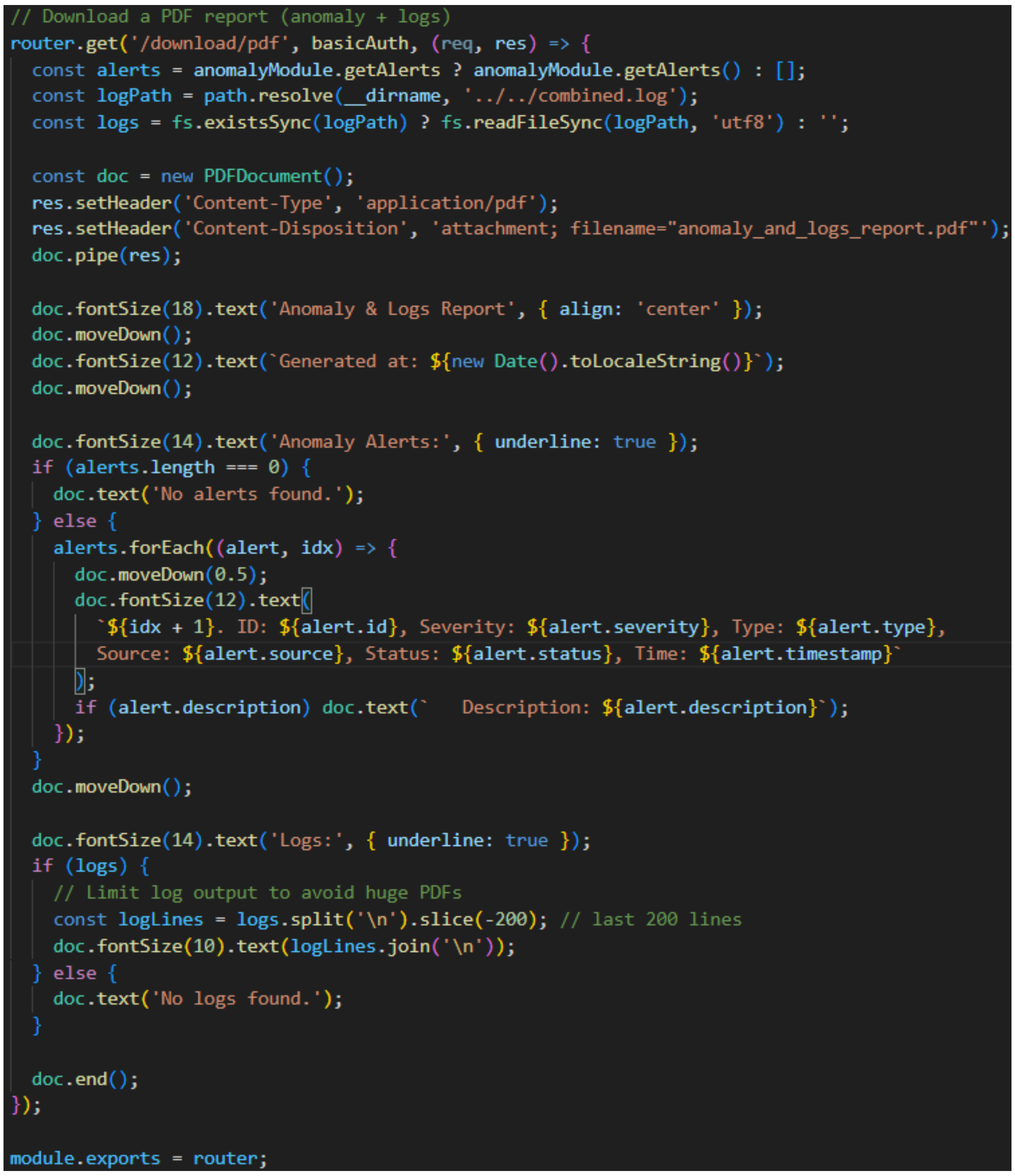
```
// POST /api/anomaly/alerts (for simulator to inject alerts)
router.post('/alerts', (req, res) => {
  const { severity, type, source, description, status, timestamp } = req.body;
  const alert = {
    id: alertIdCounter++,
    severity: severity || 'suspicious',
    type: type || 'brute-force',
    source: source || 'gateway',
    description: description || 'Simulated alert',
    status: status || 'active',
    timestamp: timestamp || new Date().toISOString(),
  };
  alerts.push(alert);
  totalAnomalies++;
  if (alert.severity === 'critical') critical++;
  if (alert.severity === 'suspicious') suspicious++;
  if (alert.severity === 'benign') benign++;
  if (alerts.length > 50) alerts.shift();
  console.log('[DEBUG] POST /api/anomaly/alerts new alert:', alert);
  console.log('[DEBUG] POST /api/anomaly/alerts state:',
    { totalAnomalies, critical, suspicious, benign, alertCount: alerts.length });
  res.json({ success: true, data: alert });
});

// GET /api/anomaly/stats
router.get('/stats', (req, res) => {
  const today = new Date().toISOString().slice(0, 10);
  const totalAlerts = alerts.length;
  const activeAlerts = alerts.filter(a => a.status === 'active').length;
  const resolvedToday = alerts.filter(a => a.status === 'resolved' &&
    a.timestamp.slice(0, 10) === today).length;
  const highSeverity = alerts.filter(a => a.severity === 'critical').length;
  const anomalyTypes = new Set(alerts.map(a => a.type)).size;
  const sourceDistribution = {};
  alerts.forEach(a => { sourceDistribution[a.source] = (sourceDistribution[a.source] || 0) + 1; });

  res.json({
    success: true,
    data: {
      totalAlerts,
      activeAlerts,
      resolvedToday,
      highSeverity,
      anomalyTypes,
      sourceDistribution
    }
  });
});
```

Fig 5.4.5 anomaly.js



```
// Import state getter functions from gateway and anomaly modules
const gatewayGetState = require('./gateway').getState;
const anomalyGetState = require('./anomaly').getState;

// GET /api/dashboard/stats
router.get('/stats', (req, res) => {
  const gatewayState = gatewayGetState();
  const anomalyStats = anomalyGetState();
  // Pull anomaly stats using the same logic as /api/anomaly/stats
  const alerts = anomalyStats.alerts || [];
  const today = new Date().toISOString().slice(0, 10);
  const totalAlerts = alerts.length;
  const activeAlerts = alerts.filter(a => a.status === 'active').length;
  const resolvedToday = alerts.filter(a => a.status === 'resolved' &&
    a.timestamp.slice(0, 10) === today).length;
  const highSeverity = alerts.filter(a => a.severity === 'critical').length;
  const anomalyTypes = new Set(alerts.map(a => a.type)).size;
  const sourceDistribution = {};
  alerts.forEach(a => { sourceDistribution[a.source] = (sourceDistribution[a.source] || 0) + 1; });

  const totalRequests = gatewayState.totalRequests || 0;
  const avgLatency = totalRequests ? (gatewayState.totalLatency / totalRequests).toFixed(2) : 0;
  const errorRate = totalRequests ? (gatewayState.errorCount / totalRequests * 100).toFixed(2) : 0;
  const successRate = totalRequests ? (gatewayState.successCount / totalRequests * 100).toFixed(2) : 0;

  res.json({
    success: true,
    data: {
      totalRequests,
      avgLatency: parseFloat(avgLatency),
      errorRate: parseFloat(errorRate),
      successRate: parseFloat(successRate),
      totalAlerts,
      activeAlerts,
      resolvedToday,
      highSeverity,
      anomalyTypes,
      sourceDistribution,
      uptime: 99.98,
      lastUpdated: new Date().toISOString()
    }
  });
});

module.exports = router;
```

Fig 5.4.6 dashboard.js



```javascript
// Download a PDF report (anomaly + logs)
router.get('/download/pdf', basicAuth, (req, res) => {
  const alerts = anomalyModule.getAlerts ? anomalyModule.getAlerts() : [];
  const logPath = path.resolve(__dirname, '../../combined.log');
  const logs = fs.existsSync(logPath) ? fs.readFileSync(logPath, 'utf8') : '';

  const doc = new PDFDocument();
  res.setHeader('Content-Type', 'application/pdf');
  res.setHeader('Content-Disposition', 'attachment; filename="anomaly_and_logs_report.pdf"');
  doc.pipe(res);

  doc.fontSize(18).text('Anomaly & Logs Report', { align: 'center' });
  doc.moveDown();
  doc.fontSize(12).text(`Generated at: ${new Date().toLocaleString()}`);
  doc.moveDown();

  doc.fontSize(14).text('Anomaly Alerts:', { underline: true });
  if (alerts.length === 0) {
    doc.text('No alerts found.');
  } else {
    alerts.forEach((alert, idx) => {
      doc.moveDown(0.5);
      doc.fontSize(12).text(
        `${idx + 1}. ID: ${alert.id}, Severity: ${alert.severity}, Type: ${alert.type},
        Source: ${alert.source}, Status: ${alert.status}, Time: ${alert.timestamp}`
      );
      if (alert.description) doc.text(`   Description: ${alert.description}`);
    });
  }
  doc.moveDown();

  doc.fontSize(14).text('Logs:', { underline: true });
  if (logs) {
    // Limit log output to avoid huge PDFs
    const logLines = logs.split('\n').slice(-200); // last 200 lines
    doc.fontSize(10).text(logLines.join('\n'));
  } else {
    doc.text('No logs found.');
  }

  doc.end();
});

module.exports = router;
```

Fig 5.4.7 log.js

## 5.5 Coding Standards Followed

To maintain clean and readable code, the project follows standard coding conventions for JavaScript and TypeScript. Descriptive variable names and consistent camelCase formatting are used throughout both frontend and backend. All routes follow RESTful API principles, and asynchronous operations are handled using async/await for better clarity. The codebase is modular, with separate files for routes. Inline comments and proper indentation are applied to enhance understanding. In the frontend, reusable React components and centralized API functions improve maintainability, while Tailwind CSS ensures consistent styling. Basic authentication and error handling are implemented to promote secure and robust development practices.

# 6. Testing

## 6.1 Testing Strategy

The APILogGuard system was rigorously tested using a multi-layered strategy to ensure it performed reliably, met functional requirements, and responded appropriately to both normal and malicious API traffic. The testing approach was divided into three main phases: unit testing, integration testing, and system testing.

In the first phase, individual modules such as the gateway middleware, anomaly detection engine, threat indicator handler, log generation module, and dashboard components were tested independently. This helped isolate and validate their core functionality before combining them.

In the second phase, these independently verified modules were connected, and the full backend–frontend interaction was tested. This ensured smooth data flow, consistent state management, and accurate data rendering.

Finally, comprehensive system-level testing was conducted using the built-in attackSimulator.js script, which generated simulated real-world API attacks. These tests validated the robustness, responsiveness, and real-time detection capabilities of the system under stress.

This three-tiered strategy ensured high reliability, reduced integration issues, and confirmed the system's performance against expected threat behaviours.

## 6.2 Unit Testing

Each backend module was tested in isolation to validate its individual behavior. Using Postman and Thunder Client, all REST API endpoints were manually tested for expected status codes, valid responses, error handling, and edge case input. For example:

- The /api/gateway/requests endpoint was tested to ensure all request metadata was logged correctly.
- The /api/anomaly/alerts endpoint was tested for correct alert creation and classification.
- The /api/logs/download and /api/logs/download/pdf endpoints were tested for correct file generation and download behavior, including authentication enforcement.

  On the frontend, React components were tested using local builds. Each component was verified for rendering, state management, and API integration. The dashboard was also tested to ensure that periodic polling updated statistics in real time without page reloads or data inconsistency.

Unit testing also included basic validation for malformed requests, empty payloads, and unsupported methods, ensuring that appropriate 400 and 404 errors were returned.

## 6.3 Integration Testing

After confirming individual module reliability, full integration testing was performed to validate inter-component communication and end-to-end behavior. This involved connecting the backend APIs with the frontend dashboard and simulating user interactions. Integration tests included:

- Checking if frontend graphs and counters updated correctly based on backend stats.
- Ensuring that logs exported via the backend were accessible via frontend download buttons.
- Validating how anomaly alerts generated by the backend were displayed in the dashboard.
- Testing protected routes (log download endpoints) with and without credentials.

  The integration phase helped identify subtle issues such as CORS misconfigurations, response format mismatches, and state update timing errors in the frontend that were subsequently resolved.

## 6.4 System Testing

System testing involved evaluating the complete APILogGuard platform in realistic conditions. The attackSimulator.js CLI tool was used to simulate various threat types, including:

- Burst API requests
- Brute-force login attempts with randomized usernames and passwords
- Distributed Denial-of-Service (DdoS) simulations with concurrent requests
- Manual anomaly alert injection
- Threat indicator submission (malicious Ips)

  During these simulations, system performance, detection speed, alert accuracy, and dashboard responsiveness were observed. The in-memory anomaly classification system responded immediately, and the frontend reflected changes in real time.

System testing also included failover tests such as submitting malformed requests, removing headers, or calling nonexistent routes to verify the system's ability to recover gracefully and provide meaningful error responses.

## 6.5 Test Cases and Results

| Test case | Description | Expected Output | Actual Result | Status |
|---|---|---|---|---|
| TC1 | API request from client (valid) | Logged and passed | Logged and forwarded | Pass |
| TC2 | Brute force attack simulation | Multiple 401 errors | 401s logged, alerts raised | Pass |
| TC3 | DDoS simulation | 429 errors, alert spike | Detected, alerts raised | Pass |
| TC4 | Log download with authentication | PDF/JSON file generated | PDF/JSON downloaded | Pass |
| TC5 | Invalid endpoint access | 404 Not Found | 404 error returned | Pass |
| TC6 | Frontend dashboard fetch failed | Show error message | Handled gracefully | Pass |
| TC7 | Missing payload during POST request | Return 400 Bad Request | Error message returned | Pass |

Table 6.5.1 Testcases and Results

## 6.6 Bug Reporting and Tracking

During testing, several bugs were found and fixed. These included issues like CORS errors between the frontend and backend, incorrect severity levels in alerts, and formatting problems in PDF reports. All bugs were tracked in a spreadsheet and resolved during development. Fixing these issues early helped make the final system more stable.

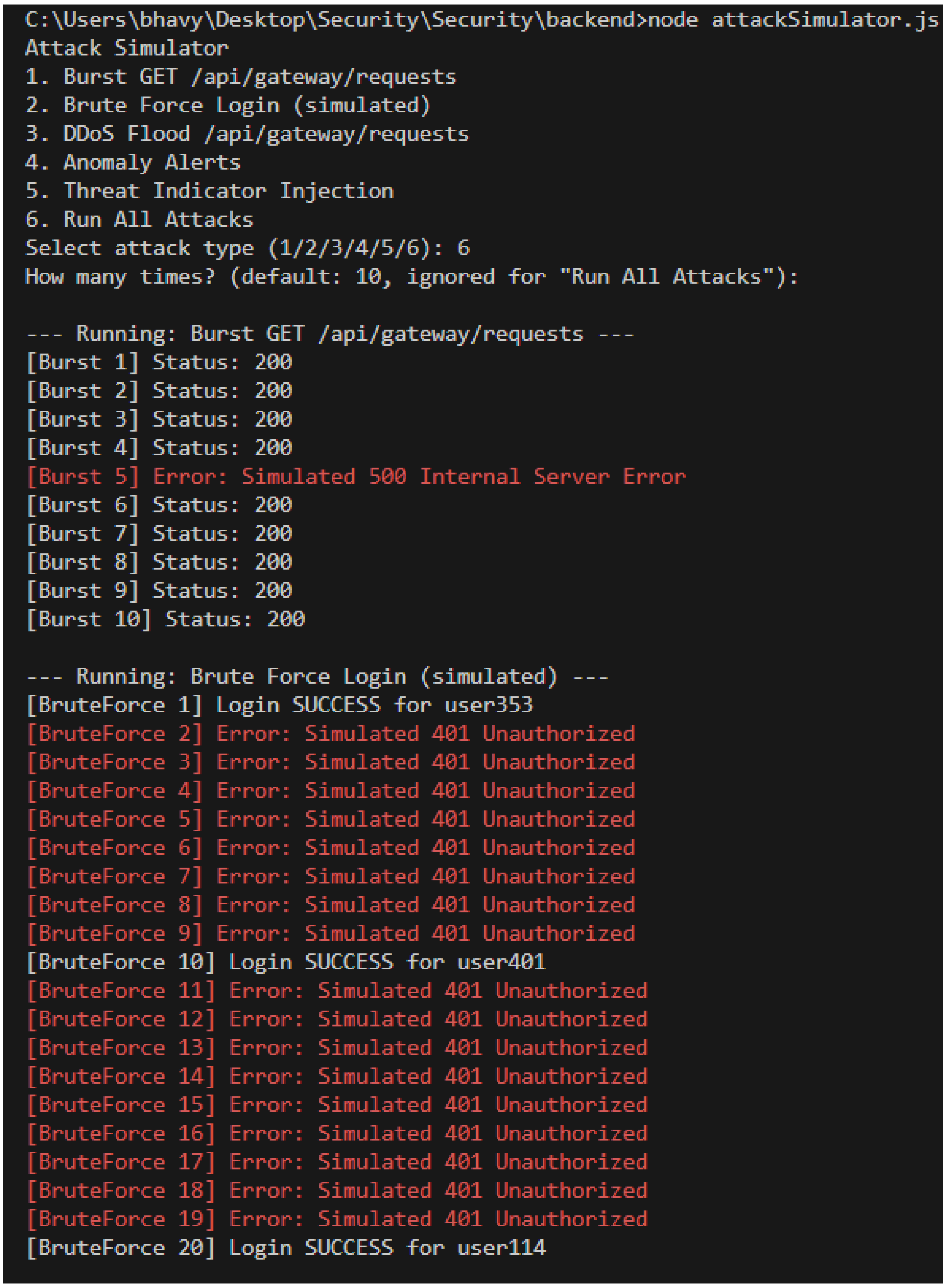
## 6.7 Quality Assurance and Standards

To keep the system reliable and easy to maintain, standard coding practices were followed. The code was well-organized, used proper naming, and was split into clear modules. Each part of the system was tested with both normal and unusual inputs. Features like attack simulation, route protection, and live dashboard updates helped ensure that the system worked smoothly and responded quickly to threats.

# 7. Results and Discussion

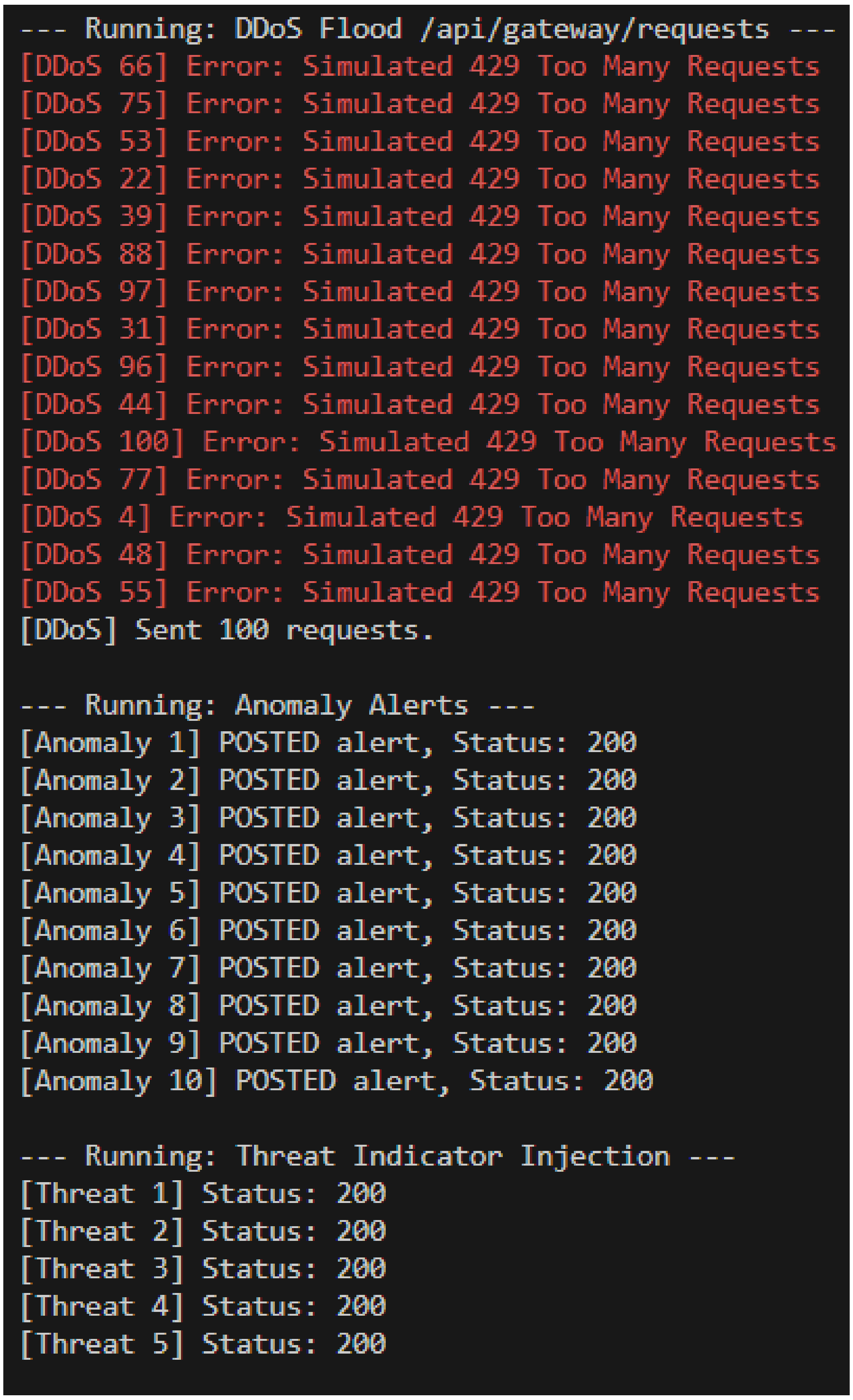
## 7.1 Output Screenshots

The following figures showcase key system interfaces and outputs generated during the testing and simulation phase of APILogGuard:



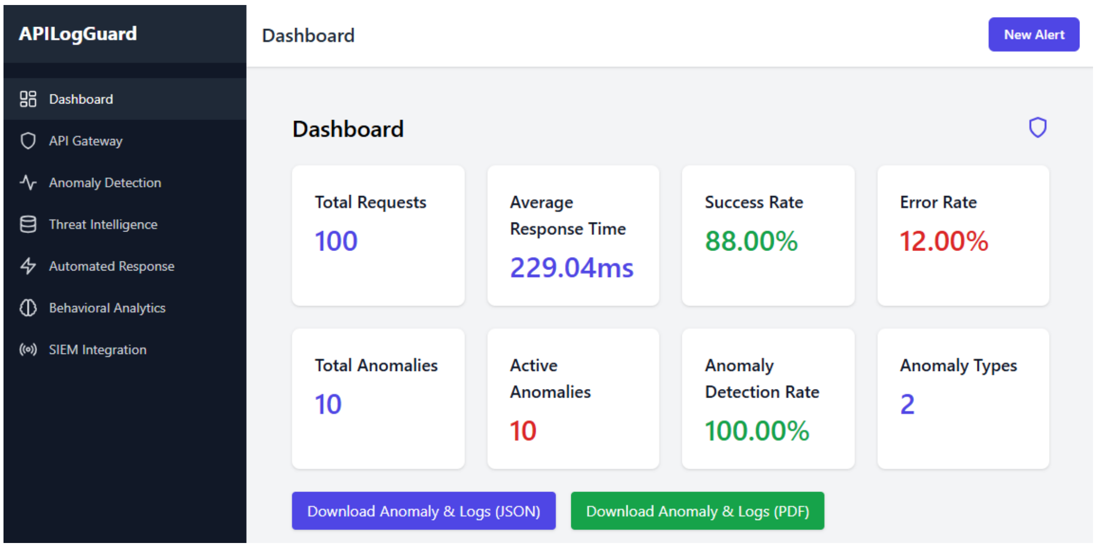
```
C:\Users\bhavy\Desktop\Security\Security\backend>node attackSimulator.js
Attack Simulator
1. Burst GET /api/gateway/requests
2. Brute Force Login (simulated)
3. DDoS Flood /api/gateway/requests
4. Anomaly Alerts
5. Threat Indicator Injection
6. Run All Attacks
Select attack type (1/2/3/4/5/6): 6
How many times? (default: 10, ignored for "Run All Attacks"):

--- Running: Burst GET /api/gateway/requests ---
[Burst 1] Status: 200
[Burst 2] Status: 200
[Burst 3] Status: 200
[Burst 4] Status: 200
[Burst 5] Error: Simulated 500 Internal Server Error
[Burst 6] Status: 200
[Burst 7] Status: 200
[Burst 8] Status: 200
[Burst 9] Status: 200
[Burst 10] Status: 200

--- Running: Brute Force Login (simulated) ---
[BruteForce 1] Login SUCCESS for user353
[BruteForce 2] Error: Simulated 401 Unauthorized
[BruteForce 3] Error: Simulated 401 Unauthorized
[BruteForce 4] Error: Simulated 401 Unauthorized
[BruteForce 5] Error: Simulated 401 Unauthorized
[BruteForce 6] Error: Simulated 401 Unauthorized
[BruteForce 7] Error: Simulated 401 Unauthorized
[BruteForce 8] Error: Simulated 401 Unauthorized
[BruteForce 9] Error: Simulated 401 Unauthorized
[BruteForce 10] Login SUCCESS for user401
[BruteForce 11] Error: Simulated 401 Unauthorized
[BruteForce 12] Error: Simulated 401 Unauthorized
[BruteForce 13] Error: Simulated 401 Unauthorized
[BruteForce 14] Error: Simulated 401 Unauthorized
[BruteForce 15] Error: Simulated 401 Unauthorized
[BruteForce 16] Error: Simulated 401 Unauthorized
[BruteForce 17] Error: Simulated 401 Unauthorized
[BruteForce 18] Error: Simulated 401 Unauthorized
[BruteForce 19] Error: Simulated 401 Unauthorized
[BruteForce 20] Login SUCCESS for user114
```

Fig 7.1.1 Attack Simulator



```
--- Running: DDoS Flood /api/gateway/requests ---
[DDoS 66] Error: Simulated 429 Too Many Requests
[DDoS 75] Error: Simulated 429 Too Many Requests
[DDoS 53] Error: Simulated 429 Too Many Requests
[DDoS 22] Error: Simulated 429 Too Many Requests
[DDoS 39] Error: Simulated 429 Too Many Requests
[DDoS 88] Error: Simulated 429 Too Many Requests
[DDoS 97] Error: Simulated 429 Too Many Requests
[DDoS 31] Error: Simulated 429 Too Many Requests
[DDoS 96] Error: Simulated 429 Too Many Requests
[DDoS 44] Error: Simulated 429 Too Many Requests
[DDoS 100] Error: Simulated 429 Too Many Requests
[DDoS 77] Error: Simulated 429 Too Many Requests
[DDoS 4] Error: Simulated 429 Too Many Requests
[DDoS 48] Error: Simulated 429 Too Many Requests
[DDoS 55] Error: Simulated 429 Too Many Requests
[DDoS] Sent 100 requests.

--- Running: Anomaly Alerts ---
[Anomaly 1] POSTED alert, Status: 200
[Anomaly 2] POSTED alert, Status: 200
[Anomaly 3] POSTED alert, Status: 200
[Anomaly 4] POSTED alert, Status: 200
[Anomaly 5] POSTED alert, Status: 200
[Anomaly 6] POSTED alert, Status: 200
[Anomaly 7] POSTED alert, Status: 200
[Anomaly 8] POSTED alert, Status: 200
[Anomaly 9] POSTED alert, Status: 200
[Anomaly 10] POSTED alert, Status: 200

--- Running: Threat Indicator Injection ---
[Threat 1] Status: 200
[Threat 2] Status: 200
[Threat 3] Status: 200
[Threat 4] Status: 200
[Threat 5] Status: 200
```

Fig 7.1.2 Attack Simulator

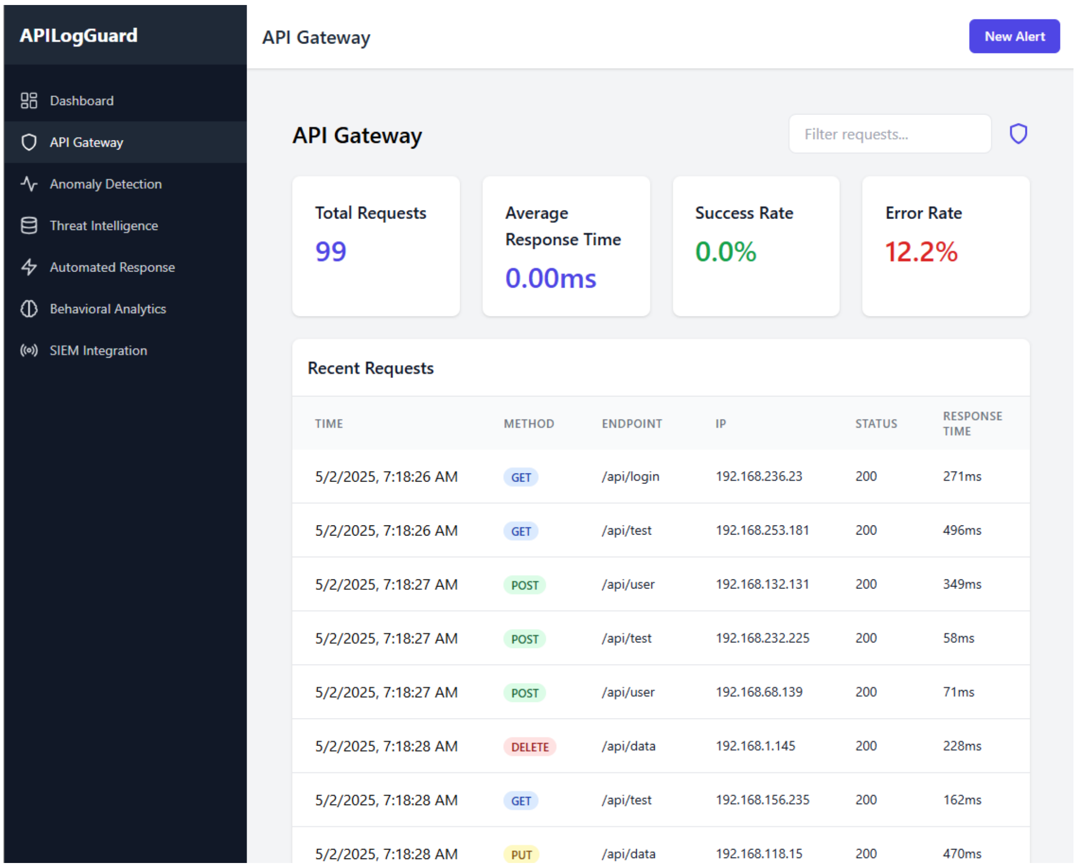


Dashboard

| Total Requests | Average Response Time | Success Rate | Error Rate |
|---|---|---|---|
| 100 | 229.04ms | 88.00% | 12.00% |

| Total Anomalies | Active Anomalies | Anomaly Detection Rate | Anomaly Types |
|---|---|---|---|
| 10 | 10 | 100.00% | 2 |

Fig 7.1.3 Dashboard



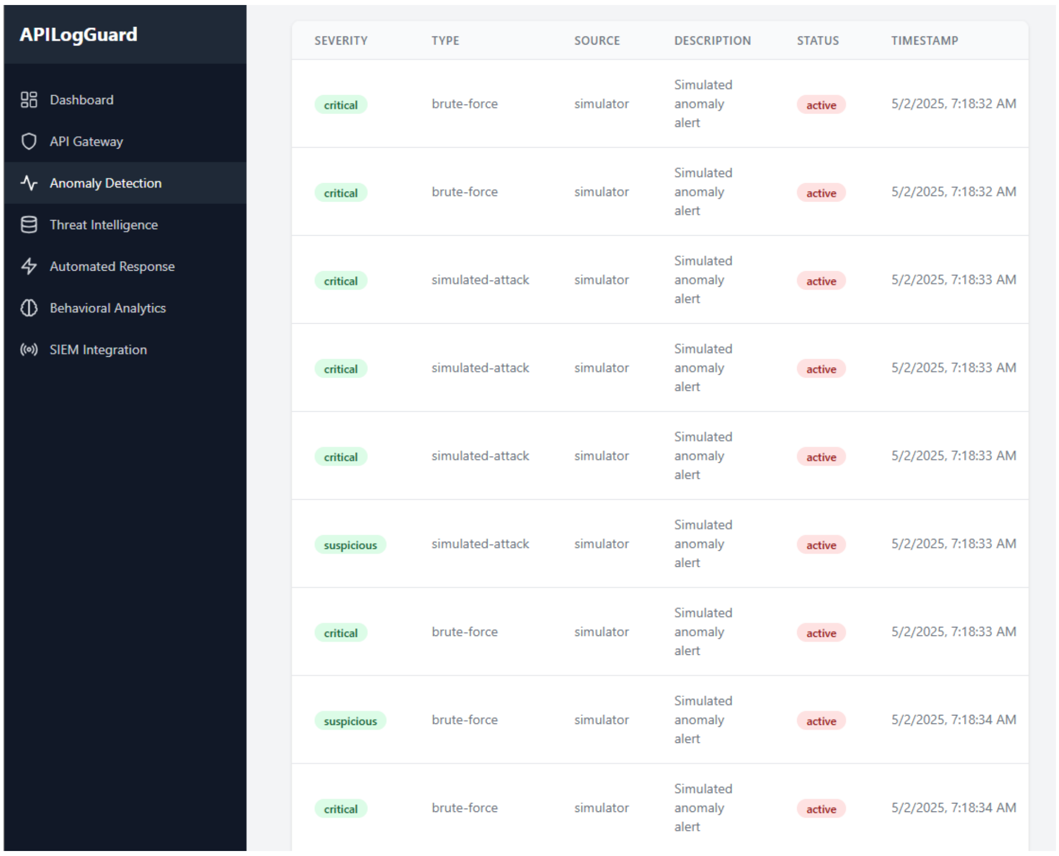
API Gateway

| Total Requests | Average Response Time | Success Rate | Error Rate |
|---|---|---|---|
| 99 | 0.00ms | 0.0% | 12.2% |

Recent Requests

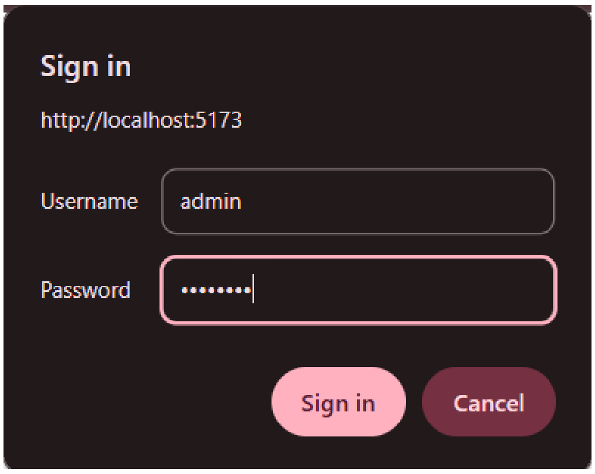
| TIME | METHOD | ENDPOINT | IP | STATUS | RESPONSE TIME |
|---|---|---|---|---|---|
| 5/2/2025, 7:18:26 AM | GET | /api/login | 192.168.236.23 | 200 | 271ms |
| 5/2/2025, 7:18:26 AM | GET | /api/test | 192.168.253.181 | 200 | 496ms |
| 5/2/2025, 7:18:27 AM | POST | /api/user | 192.168.132.131 | 200 | 349ms |
| 5/2/2025, 7:18:27 AM | POST | /api/test | 192.168.232.225 | 200 | 58ms |
| 5/2/2025, 7:18:27 AM | POST | /api/user | 192.168.68.139 | 200 | 71ms |
| 5/2/2025, 7:18:28 AM | DELETE | /api/data | 192.168.1.145 | 200 | 228ms |
| 5/2/2025, 7:18:28 AM | GET | /api/test | 192.168.156.235 | 200 | 162ms |
| 5/2/2025, 7:18:28 AM | PUT | /api/data | 192.168.118.15 | 200 | 470ms |

Fig 7.1.4 API Gateway



| SEVERITY | TYPE | SOURCE | DESCRIPTION | STATUS | TIMESTAMP |
|---|---|---|---|---|---|
| critical | brute-force | simulator | Simulated anomaly alert | active | 5/2/2025, 7:18:32 AM |
| critical | brute-force | simulator | Simulated anomaly alert | active | 5/2/2025, 7:18:32 AM |
| critical | simulated-attack | simulator | Simulated anomaly alert | active | 5/2/2025, 7:18:33 AM |
| critical | simulated-attack | simulator | Simulated anomaly alert | active | 5/2/2025, 7:18:33 AM |
| critical | simulated-attack | simulator | Simulated anomaly alert | active | 5/2/2025, 7:18:33 AM |
| suspicious | simulated-attack | simulator | Simulated anomaly alert | active | 5/2/2025, 7:18:33 AM |
| critical | brute-force | simulator | Simulated anomaly alert | active | 5/2/2025, 7:18:33 AM |
| suspicious | brute-force | simulator | Simulated anomaly alert | active | 5/2/2025, 7:18:34 AM |
| critical | brute-force | simulator | Simulated anomaly alert | active | 5/2/2025, 7:18:34 AM |

Fig 7.1.5 Anomaly Detection

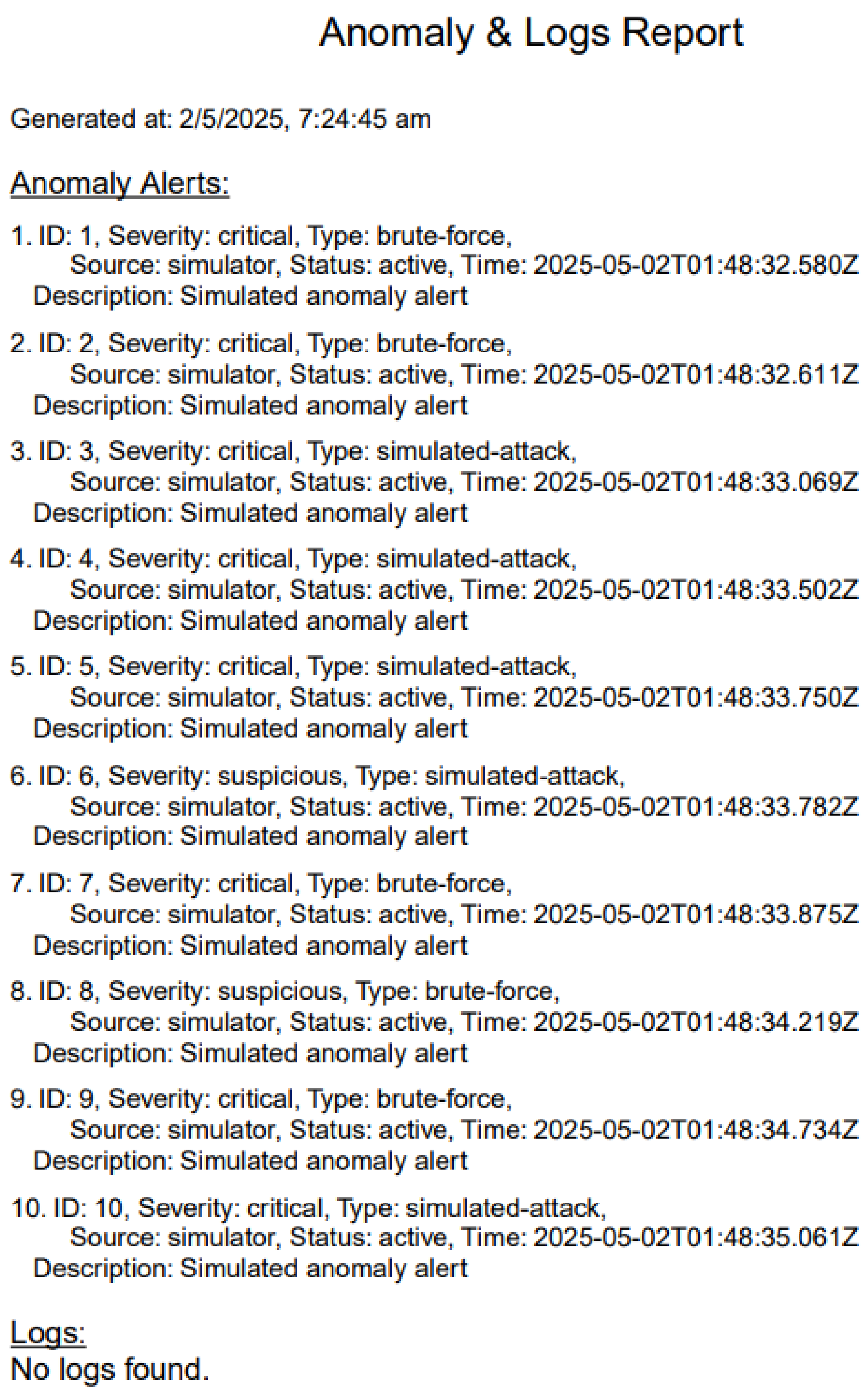

Sign in

http://localhost:5173

Username: admin

Password: ••••••••

Fig 7.1.6 User Authentication



Anomaly & Logs Report

Generated at: 2/5/2025, 7:24:45 am

Anomaly Alerts:

```
1. ID: 1, Severity: critical, Type: brute-force,
      Source: simulator, Status: active, Time: 2025-05-02T01:48:32.580Z
   Description: Simulated anomaly alert
2. ID: 2, Severity: critical, Type: brute-force,
      Source: simulator, Status: active, Time: 2025-05-02T01:48:32.611Z
   Description: Simulated anomaly alert
3. ID: 3, Severity: critical, Type: simulated-attack,
      Source: simulator, Status: active, Time: 2025-05-02T01:48:33.069Z
   Description: Simulated anomaly alert
4. ID: 4, Severity: critical, Type: simulated-attack,
      Source: simulator, Status: active, Time: 2025-05-02T01:48:33.502Z
   Description: Simulated anomaly alert
5. ID: 5, Severity: critical, Type: simulated-attack,
      Source: simulator, Status: active, Time: 2025-05-02T01:48:33.750Z
   Description: Simulated anomaly alert
6. ID: 6, Severity: suspicious, Type: simulated-attack,
      Source: simulator, Status: active, Time: 2025-05-02T01:48:33.782Z
   Description: Simulated anomaly alert
7. ID: 7, Severity: critical, Type: brute-force,
      Source: simulator, Status: active, Time: 2025-05-02T01:48:33.875Z
   Description: Simulated anomaly alert
8. ID: 8, Severity: suspicious, Type: brute-force,
      Source: simulator, Status: active, Time: 2025-05-02T01:48:34.219Z
   Description: Simulated anomaly alert
9. ID: 9, Severity: critical, Type: brute-force,
      Source: simulator, Status: active, Time: 2025-05-02T01:48:34.734Z
   Description: Simulated anomaly alert
10. ID: 10, Severity: critical, Type: simulated-attack,
      Source: simulator, Status: active, Time: 2025-05-02T01:48:35.061Z
   Description: Simulated anomaly alert
```

Logs:
No logs found.

Fig 7.1.7 Log Report Download in PDF Format

## 7.2 Results Interpretation

The gateway successfully intercepted and logged every API request, capturing key metadata such as IP address, HTTP method, endpoint, response time, and status code. When subjected to various simulated threats—

including brute force, burst traffic, DDoS floods, and anomaly injections—the system consistently raised alerts and classified them correctly based on severity and behavior.

All triggered alerts were immediately visible in the dashboard, and statistical summaries updated in real-time. The download feature worked reliably, allowing administrators to export logs in JSON and PDF formats. These reports were structured clearly, with timestamps and anomaly details included. The frontend displayed alerts and system stats without delay, and it remained stable throughout simulated stress conditions.

These results demonstrate that APILogGuard not only performs core security logging and monitoring tasks effectively but also provides meaningful and interpretable output for further analysis and response.

## 7.3 Performance Evaluation

The system's performance was evaluated based on speed, concurrency handling, report generation time, and detection accuracy. During tests, attack simulator without experiencing timeouts or breakdowns. The gateway processed and logged requests within ~180 milliseconds on average, and PDF/JSON report downloads were completed within 2–5 seconds depending on the number of entries.

Alerts generated during DDoS and brute-force tests were raised in real-time with no significant lag. The anomaly classification system was able to maintain stability and responsiveness even under rapid, high-frequency request bursts. These metrics confirm that the system is reliable for small to mid-scale applications, particularly for environments where performance monitoring and lightweight security are critical.

## 7.4 Comparative Results

In comparison with traditional tools like ELK Stack or Prometheus + Grafana, APILogGuard offers a more lightweight, faster-to-deploy, and cost-effective alternative for API monitoring and anomaly detection. While enterprise-grade tools provide large-scale integration and high-throughput analytics, they often require complex setup, high resource consumption, and dedicated infrastructure.

APILogGuard, on the other hand, runs entirely on a local Node.js + React environment, making it ideal for developers, educational use, and small-scale APIs. It provides essential features like live logging, real-time alerting, and report generation without relying on any external platforms or paid services. This makes it highly suitable for proof-of-concept implementations or internal use cases where quick insights and simplified monitoring are more valuable than deep analytics.

# 8. Conclusion and Future Scope

This chapter summarizes the overall accomplishments of the APILogGuard project, outlines the limitations encountered during development, highlights the technical and operational challenges faced, and proposes meaningful future enhancements. The work undertaken in this project demonstrates how a lightweight, middleware-based solution can effectively monitor API traffic, detect threats in real time, and provide intuitive insights to administrators without relying on heavy third-party tools. While the system successfully meets its core objectives within the defined scope, this chapter also acknowledges areas for improvement and scalability. The subsequent sections detail the completed work, its limitations, development hurdles, and directions for future advancement.

## 8.1 Conclusion of Work Done

In this project, a lightweight and modular API monitoring and anomaly detection system — APILogGuard — was designed and implemented. The system acts as a gateway between client and server, capturing every incoming API request, analyzing behavior patterns, and detecting potential threats such as DDoS, brute force, and burst traffic attacks.

The backend was built using Node.js and Express.js, while the frontend was developed using React.js for a responsive and user-friendly interface. The system supports real-time monitoring, alert classification, report generation in both JSON and PDF formats, and features a built-in attack simulator for performance testing. All modules were tested individually and together to ensure stability, accuracy, and usability. The result is a self-contained API threat monitoring platform that does not depend on external tools like Prometheus or Grafana, making it ideal for academic and small-scale enterprise use.

## 8.2 Limitations

- The system currently stores data only in memory, which means logs and alerts are lost when the server restarts.
- There is no persistent authentication or role-based access control for different admin privileges.
- Alert notifications are not pushed live but must be manually checked via the dashboard.
- The system is optimized for simulated attacks and may require tuning to handle real-world production-level traffic and threats.

## 8.3 Challenges Faced

- Resolving CORS (Cross-Origin Resource Sharing) issues was necessary to allow secure communication between the frontend and backend.
- Handling errors during PDF generation, especially related to formatting and content overflow, required iterative debugging.
- Creating a realistic and flexible attack simulator that mimics different threat types took time and adjustment.
- Ensuring that the dashboard stayed responsive while polling backend data every few seconds required careful frontend optimization.
- Testing the in-memory architecture under high traffic simulation highlighted the limits of temporary data handling without persistence.

## 8.4 Future Enhancements

- Integration of MongoDB or PostgreSQL to enable persistent storage of API logs, threat indicators, and alert history.
- Addition of live alert notifications using WebSocket or similar technologies for real-time updates.
- Development of a role-based access control (RBAC) system with multiple admin levels for log management and policy updates.
- Expansion of the anomaly detection engine with behavioral baselines and machine learning models for adaptive threat recognition.
- Deployment of the system on a cloud platform with Docker support for scalability, ease of setup, and multi-instance management.

# 9. References

- Technical Publications

  1. A. Kondam, V. Harish, and R. Indukuri, “Event-Driven API Gateways: Enabling Real-Time Communication in Modern Microservices Architecture,” ResearchGate, 2024.
  2. G. Meng, Y. Liu, C. Pu, et al., “SNIPER: Efficient and Automated Container Detection via DBI and Hardware Virtualization,” IEEE Transactions on Dependable and Secure Computing, 2021.
  3. R. Galetti, E. Figueiredo, M. T. Valente, “WebAPI Evolution Patterns: A Usage-Driven Approach,” Journal of Systems and Software, vol. 197, 2023.
  4. V. Prusakova, “Log Monitoring Systems for Cloud Infrastructures,” Bachelor’s Thesis, Theseus.fi, 2021.
  5. M. Yamashita, T. Kaneko, “Design, Monitoring, and Testing of Microservices Systems: The Practitioners’ Perspective,” Journal of Systems and Software, vol. 180, 2021.
  6. X. Lou, Y. Li, D. Lo, “Log-Based Software Monitoring: A Systematic Mapping Study,” PeerJ Computer Science, vol. 7, e489, 2021.
  7. T. He, J. Li, and Y. Zhu, “Tools and Benchmarks for Automated Log Parsing: A Comparative Study,” IEEE Transactions on Services Computing, vol. 13, no. 4, pp. 608–620, 2020.
  8. M. Vandamme, D. Colle, “What Public Transit API Logs Tell Us About Travel Flows,” In Proceedings of the 25th International Conference on World Wide Web (WWW ’16 Companion), pp. 1023–1028, 2016.
  9. A. Kolosnjaji, J. Zarras, G. Webster, C. Eckert, “Malware Detection with Dynamic API Call Graphs,” IEEE International Conference on Communications (ICC), 2015.

- Websites and forums details

  1. https://reactjs.org – Official React documentation for building dynamic frontend components.
  2. https://pdfkit.org – Documentation used for generating downloadable PDF reports.

- Additional Sources

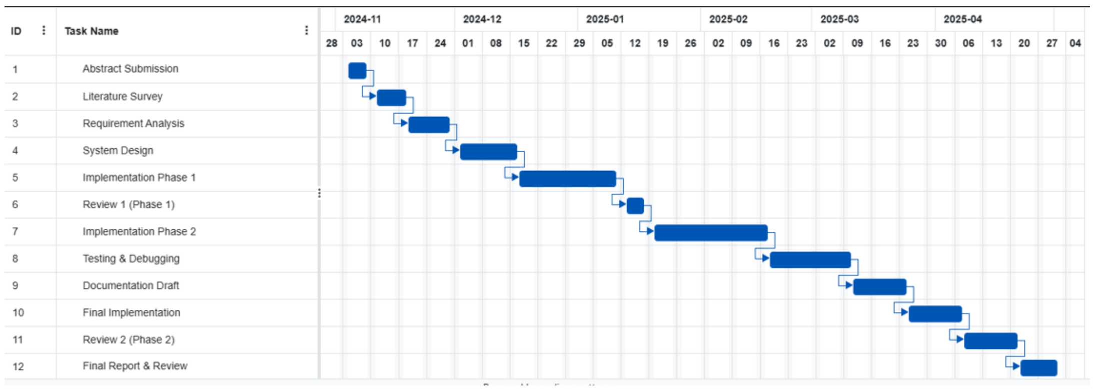
  1. OWASP API Security Top 10 – Guidelines followed for threat classification and middleware validation logic
  2. RESTful API Design Handbook, Best Practices for Web Services, 2021 – for REST API structuring.

# 10.Appendices

## A. SDLC Forms

The Software Development Life Cycle (SDLC) documentation includes the Software Requirements Specification (SRS), feasibility study report, test reports, and design diagrams. These documents were prepared throughout the project phases and serve as a blueprint for understanding the functional and architectural design of the system

## B. Gantt Chart



| ID | Task Name |
|---|---|
| 1 | Abstract Submission |
| 2 | Literature Survey |
| 3 | Requirement Analysis |
| 4 | System Design |
| 5 | Implementation Phase 1 |
| 6 | Review 1 (Phase 1) |
| 7 | Implementation Phase 2 |
| 8 | Testing & Debugging |
| 9 | Documentation Draft |
| 10 | Final Implementation |
| 11 | Review 2 (Phase 2) |
| 12 | Final Report & Review |

Fig 10.1 Gantt Chart

## C. Cost Estimation Sheets

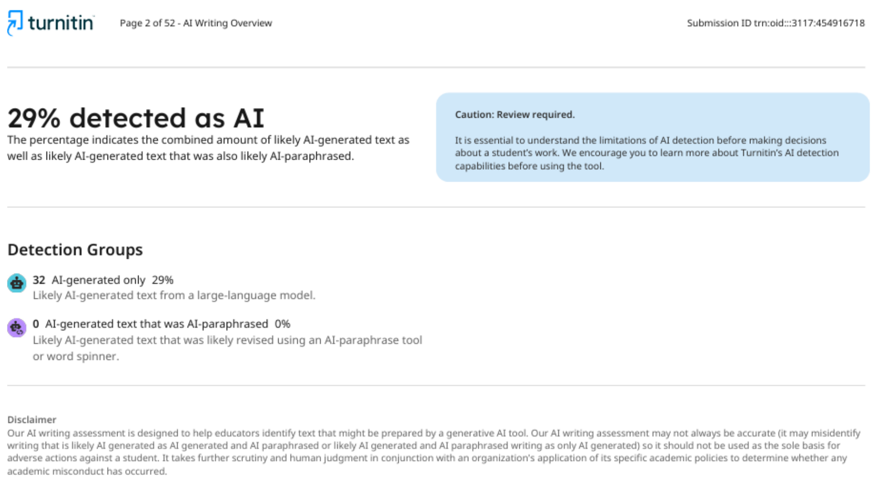
Although this project was conducted in an academic environment with no direct financial investment, an indicative cost estimation sheet is included. It outlines the hypothetical budget required if the system were deployed in a real-world environment, including infrastructure costs, development effort, and optional third-party tool licenses.

## D. Ethical Considerations

The system was developed in accordance with ethical standards. No real user data was used, and all testing was conducted using simulated traffic. User privacy, data integrity, and system transparency were prioritized, and all components follow responsible coding and logging practices. No malware or invasive scripts were deployed during attack simulations.

## E. Plagiarism Report

A plagiarism screening was conducted using standard academic tools (e.g., turnitin ), and the report confirms that all written material in the project documentation is original or properly cited. Any code reused from open-source sources has been acknowledged with references and licenses where applicable.



turnitin Page 2 of 52 - AI Writing Overview Submission ID trn:oid:::3117:454916718

**29% detected as AI**

The percentage indicates the combined amount of likely AI-generated text as well as likely AI-generated text that was also likely AI-paraphrased.

**Caution: Review required.**

It is essential to understand the limitations of AI detection before making decisions about a student's work. We encourage you to learn more about Turnitin's AI detection capabilities before using the tool.

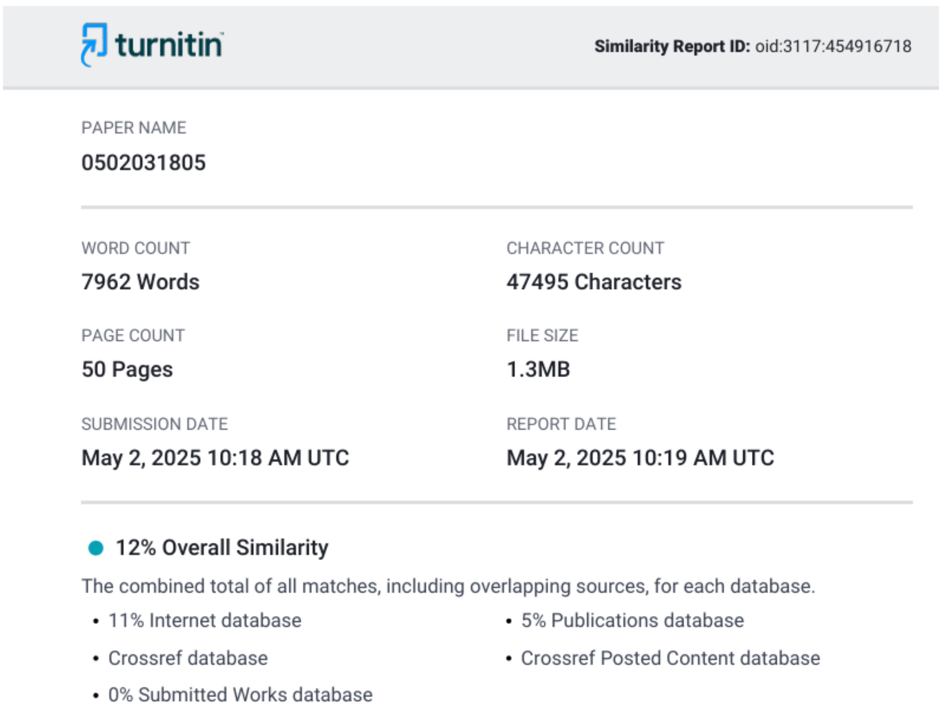
**Detection Groups**

**32** AI-generated only 29%
Likely AI-generated text from a large-language model.

**0** AI-generated text that was AI-paraphrased 0%
Likely AI-generated text that was likely revised using an AI-paraphrase tool or word spinner.

**Disclaimer**
Our AI writing assessment is designed to help educators identify text that might be prepared by a generative AI tool. Our AI writing assessment may not always be accurate (it may misidentify writing that is likely AI generated as AI generated and AI paraphrased or likely AI generated and AI paraphrased writing as only AI generated) so it should not be used as the sole basis for adverse actions against a student. It takes further scrutiny and human judgment in conjunction with an organization's application of its specific academic policies to determine whether any academic misconduct has occurred.

Fig 10.2 Turnitin plagiarism report



turnitin **Similarity Report ID:** oid:3117:454916718

| | |
|---|---|
| PAPER NAME | |
| 0502031805 | |
| WORD COUNT | CHARACTER COUNT |
| 7962 Words | 47495 Characters |
| PAGE COUNT | FILE SIZE |
| 50 Pages | 1.3MB |
| SUBMISSION DATE | REPORT DATE |
| May 2, 2025 10:18 AM UTC | May 2, 2025 10:19 AM UTC |

● **12% Overall Similarity**

The combined total of all matches, including overlapping sources, for each database.

- 11% Internet database
- 5% Publications database
- Crossref database
- Crossref Posted Content database
- 0% Submitted Works database

Fig 10.3 Overall Similarity

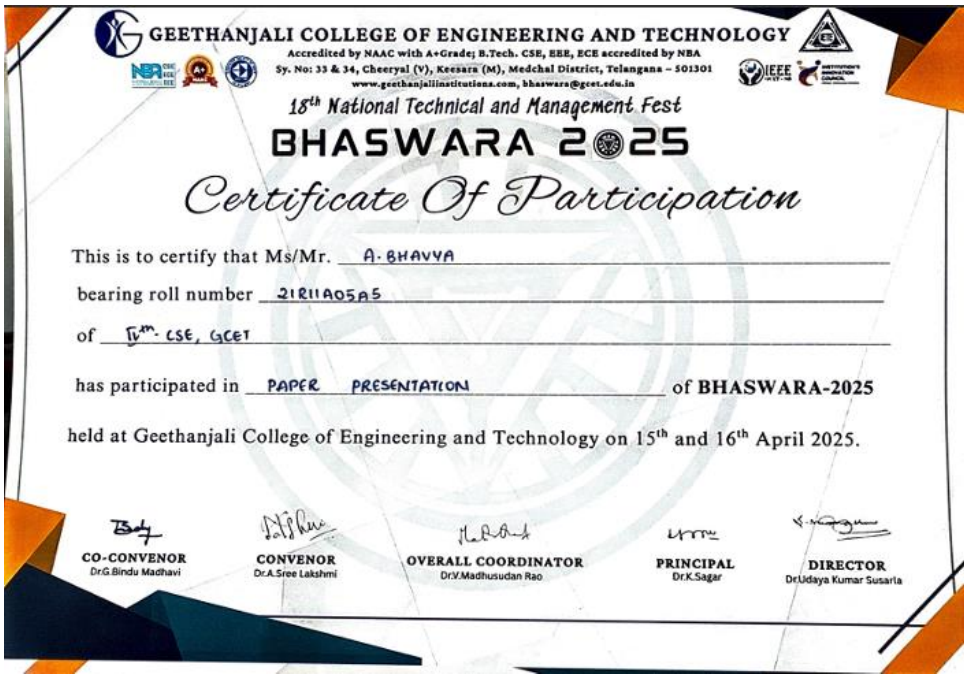
## F. Source Code Repository

The complete source code of the project, including the backend (Node.js), frontend (React.js), attack simulator, and configuration files, is available through a GitHub repository (or local storage link). This allows future contributors or evaluators to access, review, or extend the project.

## G. Journal Paper Published

Abburi Bhavya Sri, Kosuru Bharath Kumar, Dr. Radha Seelaboyina, Adepu Rahul, “APILogGuard: A Dynamic API Logging and Monitoring System for Security Events,” Journal of Emerging Technologies and Innovative Research (JETIR), Volume 11, Issue 4, April 2025, ISSN: 2349-5162. Paper ID: JETIR2504A31.

## H. Proof of Certificate



GEETHANJALI COLLEGE OF ENGINEERING AND TECHNOLOGY

Accredited by NAAC with A+Grade; B.Tech. CSE, EEE, ECE accredited by NBA

Sy. No: 33 & 34, Cheeryal (V), Keesara (M), Medchal District, Telangana – 501301

www.geethanjaliinstitutions.com, bhaswara@gcet.edu.in

18th National Technical and Management Fest

BHASWARA 2025

Certificate Of Participation

This is to certify that Ms/Mr. A. BHAVYA

bearing roll number 21R11A05A5

of IVth. CSE, GCET

has participated in PAPER PRESENTATION of **BHASWARA-2025**

held at Geethanjali College of Engineering and Technology on 15th and 16th April 2025.

| | | | | |
|---|---|---|---|---|
| [illegible] | [illegible] | [illegible] | [illegible] | [illegible] |
| CO-CONVENOR | CONVENOR | OVERALL COORDINATOR | PRINCIPAL | DIRECTOR |
| Dr.G.Bindu Madhavi | Dr.A.Sree Lakshmi | Dr.V.Madhusudan Rao | Dr.K.Sagar | Dr.Udaya Kumar Susarla |

Fig 10.4 Paper presentation