# Building Knowledge Explorer: An Educational Web Application Integrating Cloud Deployment, AI Summarization, and Security Best Practices

## Introduction

In pursuit of combining my passion for education, technology, and cybersecurity, I developed Knowledge Explorer — a cloud-hosted web application that generates multi-level educational content from Wikipedia using AI-powered summarization. The goal of this project was to create an interactive learning platform while applying the theoretical knowledge gained during my preparation for the CompTIA Security+ and Network+ certifications. Through this project, I brought together a diverse set of technologies including a React frontend, Flask backend, PostgreSQL database, Redis caching, and AWS cloud services. This allowed me to gain practical experience in secure architecture design, network configuration, and scalable cloud deployment.

Throughout the development of Knowledge Explorer, I engaged with nearly every aspect of the technology stack: from frontend and backend development to API integration, caching strategies, debugging, database management, cloud networking, and cybersecurity practices. This essay outlines the objectives, system design, implementation process, security approach, penetration testing, challenges faced, and future roadmap of the project.

## Project Goals and Objectives

The concept for Knowledge Explorer grew out of my personal experience as a curious learner. I have often found myself navigating through Wikipedia, following interconnected articles to expand my understanding of unfamiliar topics. Recognizing the value of accessible and layered information, I aimed to create a platform that presents material at varying levels of complexity. This approach accommodates different users — from beginners to advanced learners — and encourages exploration by offering related topics for deeper engagement.

The primary objectives of this project were twofold:

1. **Educational Utility:**  
   Develop an application that allows users to explore topics in depth, with AI-generated summaries tailored to basic, intermediate, and advanced levels. Additionally, the application suggests related topics to promote continued learning.
2. **Practical Application of Skills:**  
   Design and deploy a functional, secure web application that integrates cloud services, web development practices, database management, and cybersecurity principles. This objective aligned closely with my preparation for CompTIA certifications, offering a practical setting to apply theoretical concepts.

## System Design and Implementation

To achieve these objectives, I adopted a multi-tier architecture designed for scalability, performance, and security. I built and integrated each component iteratively, testing as I went to ensure the system remained stable and responsive.

### Frontend Development

The frontend is built using React, structured as a single-page application (SPA) for a smooth user experience. It accepts user queries and displays AI-generated summaries and related topics. To serve the frontend and handle routing efficiently, I configured Nginx as both a web server and reverse proxy, forwarding API requests securely to the backend. All communication between frontend and backend is encrypted via HTTPS, with SSL certificates provided by Let’s Encrypt and managed using Certbot for automatic renewal.

### Backend and API Integration

The backend, developed in Flask, acts as the central orchestrator. It receives requests from the frontend, retrieves Wikipedia content, processes it, interacts with the AI summarization API, and returns structured responses. I established RESTful API endpoints, beginning with basic health checks and advancing to topic query handling. Testing endpoints with curl and browser tools helped verify response structures early.

For retrieving Wikipedia data, I built a dedicated function using the MediaWiki API, ensuring proper compliance with API standards by including correct headers and enabling gzip compression to reduce payload size. To responsibly manage API usage, I introduced short delays between requests.

### AI Summarization Integration

A core feature of Knowledge Explorer is its ability to deliver summaries at varying levels of complexity. I integrated the Anthropic Claude API to analyze Wikipedia content and produce summaries categorized into beginner, intermediate, and advanced levels. Additionally, the AI suggests related topics to build out learning paths.

I selected Anthropic Claude specifically because it is one of the more cost-effective models available. Since the application provides pre-sourced, structured Wikipedia text for summarization, the task is less computationally demanding than generating original content from scratch. This meant I did not need to rely on a more expensive, higher-capacity model. Claude strikes a balance between affordability and quality, making it well-suited for the summarization workload of Knowledge Explorer.

To protect API keys and maintain security, all credentials are stored in environment variables and kept server-side. I also implemented responsible usage practices, such as rate limiting and error handling, to manage load and maintain service stability.

### Data Management

Initially, I used SQLite for local development due to its simplicity, which allowed me to test database interactions quickly. For production, I transitioned to PostgreSQL, hosted on AWS RDS, for its scalability and robustness.

I chose a relational database model because it naturally fit the structure of Knowledge Explorer, where articles, summaries, and related topics interconnect. Using SQLAlchemy ORM, I safeguarded database operations with parameterized queries to protect against SQL injection attacks.

### Caching Strategy

To enhance performance, I implemented Redis, hosted via AWS ElastiCache, as a caching layer. Redis stores frequently accessed Wikipedia results and AI-generated summaries, which significantly improves response times and reduces strain on the database and external APIs.

From the beginning, I prioritized Redis over simple in-memory caching solutions to ensure production readiness. This decision paid off by maintaining application responsiveness even under increasing traffic.

### Containerization and Deployment

To ensure portability and consistency across environments, I containerized the entire application using Docker. Both the frontend and backend run in separate containers, orchestrated by Docker Compose. This setup allowed me to build, test, and deploy the application reliably across development and production stages.

The production environment runs on an AWS EC2 instance within a Virtual Private Cloud (VPC). I applied strict AWS Security Group rules to control network access:

* **Port 22 (SSH):** Restricted to my administrative IP.
* **Port 443 (HTTPS):** Open for web traffic.
* **Ports 5432 (Postgres) and 6379 (Redis):** Scoped to internal services only.

This layered approach to deployment and network security ensured that only essential services remained exposed, reducing the potential attack surface.

### Testing and Troubleshooting

Throughout development, I continuously tested components in isolation and as part of the integrated system. Using logging, debug statements, and system monitoring, I verified that data flowed correctly between the frontend, backend, database, caching layer, and external APIs.

Troubleshooting involved monitoring application logs to confirm whether responses were served from cache or required fresh API calls, validating database queries, and ensuring API rate limits were respected.

## Security Considerations

Security was integrated into every stage of development. I evaluated the system from both defensive and offensive perspectives to anticipate vulnerabilities and strengthen defenses.

### Defensive Measures

I employed multiple layers of protection:

* All data exchanges use HTTPS encryption.
* Sensitive credentials are managed through environment variables.
* AWS Security Groups tightly control network traffic.
* SQLAlchemy ORM enforces safe, parameterized queries.
* Structured error handling prevents exposure of sensitive internal details.

Caching and rate limiting further protect the system from abuse and reduce unnecessary load on external APIs.

### Offensive Considerations

Viewing the system from an attacker’s perspective, I identified public-facing endpoints as potential vectors for abuse. To mitigate this, I implemented monitoring and logging to track unusual activity. I also plan to add further protections, including:

* A Web Application Firewall (WAF).
* More granular throttling controls.
* API key rotation and centralized secrets management using AWS Secrets Manager.

These enhancements will strengthen the application’s resilience against attacks and improve operational security.

## Penetration Testing and Security Assessment

To validate the security posture of Knowledge Explorer, I conducted a structured black-box web application penetration test. The assessment targeted the public-facing web application at [**www.knowledge-explorer-app.com**](http://www.knowledge-explorer-app.com), focusing on API enumeration, abuse potential, input handling, rate limiting, and external API integrations.

### Summary of Findings

The application demonstrated a strong baseline of security for a new deployment. No critical injection vulnerabilities or authentication weaknesses were identified, and out-of-band testing confirmed that the application resists server-side request forgery (SSRF) and external callouts.

However, several operational risks were uncovered:

* **Lack of Rate Limiting (High Risk):** The absence of application-layer rate limiting allows attackers to flood the backend with unlimited requests, risking service degradation and increased operational costs.
* **External API Resource Exhaustion (High Risk):** Uncached queries to external APIs are significantly slower than cached responses. Attackers could exploit this by continually submitting new, uncached requests, exhausting API quotas and backend resources.
* **Source Map Exposure (Medium Risk):** JavaScript source maps were found exposed in the production environment. While no secrets were discovered, the exposure aids reverse engineering efforts.

### Recommendations

To address these findings, I plan to:

* Implement application and infrastructure-level rate limiting, including IP- and user-based controls.
* Enhance caching strategies, including negative caching for invalid or empty responses.
* Remove source maps from production and update build pipelines to prevent future exposure.
* Deploy alerting and monitoring for abnormal API usage patterns.

These improvements align with my broader security roadmap and will strengthen both application resilience and cost control.

## Challenges and Solutions

Developing in a virtualized environment using WSL presented early obstacles, particularly with managing dependencies and isolated environments. Working through these challenges improved my ability to configure complex development environments.

Ensuring API compliance was another critical task. By thoroughly studying the MediaWiki API documentation and implementing best practices, I maintained reliable integration and avoided service interruptions.

Containerizing the application with Docker was a new skillset for me. Writing and testing Dockerfiles, configuring Docker Compose, and setting up Nginx as a reverse proxy expanded my practical knowledge of containerized deployments and cloud infrastructure.

## Lessons Learned and Future Roadmap

This project reinforced the importance of early planning and iterative development. Establishing caching strategies, API compliance, and modular architecture from the outset prevented significant rework later. Testing each component individually before integration ensured smooth system behavior and reliable performance.

Following the penetration test, I refined my roadmap to include:

* **CI/CD Pipeline:** Automate testing and deployment with GitHub Actions.
* **Enhanced Security Measures:** Deploy advanced rate limiting, WAF integration, and API usage monitoring.
* **Detection and Response:** Simulate attacks to test detection accuracy and response workflows.
* **Secrets Management:** Implement AWS Secrets Manager for secure, centralized credential storage.
* **Build Hygiene:** Ensure production deployments exclude development artifacts like source maps.

These steps will continue evolving Knowledge Explorer into a robust, secure platform for both educational use and ongoing cybersecurity practice.

## Conclusion

Knowledge Explorer merges my interests in education, web development, and cybersecurity into a functional, real-world application. By generating multi-level summaries and curated learning paths from Wikipedia data, it empowers users to explore topics at their own pace.

Beyond its educational value, the project demonstrates my ability to design, build, test, and secure a modern web application. Through this process, I gained hands-on experience in cloud deployment, API integration, application security, penetration testing, and containerization. As I continue to build upon this foundation, Knowledge Explorer will remain both a useful learning tool and an active demonstration of my technical growth.