Evaluating AI Efficiency in Backend Software Development

A Comparative Analysis Across Frameworks

Arfa Saif Agha

Master's thesis
May 2025
Master of Engineering - Full Stack Software Development

Jyväskylän ammattikorkeakoulu
University of Applied Sciences

Agha, Arfa Saif

Evaluating AI Efficiency in Backend Software Development - A Comparative Analysis Across Frameworks

Jyväskylä: Jamk University of Applied Sciences, May 2025, 86 pages.

Master of Engineering - Full Stack Software Development. Master's thesis.

Permission for open access publication: Yes

Language of publication: English

Abstract

Artificial intelligence (AI) has increasingly been adopted to assist backend software development through code generation, automation, and prompt optimization. The objective of this study was to compare the performance of various AI code generation models—namely ChatGPT, DeepSeek, Gemini, and GitHub Copilot—across multiple backend frameworks including Node.js, Flask, .NET, and Laravel. A standardized Secure File Transfer Protocol (SFTP) application was implemented to evaluate AI-generated code using software engineering benchmarks. These benchmarks included code correctness, runtime efficiency, maintainability, readability, and error rate. In addition, the study applied three prompt optimization strategies—self-refinement loops, few-shot injection, and role-based contextualization—to determine their effect on code quality and reliability. The implementation involved prompt testing under controlled conditions and the use of a custom benchmarking script to quantify performance. Results showed that AI performance varied significantly based on both the underlying framework and prompt strategy. Node.js exhibited the highest compatibility with AI-generated code, while .NET demonstrated superior correctness and maintainability. Prompt optimization strategies were shown to consistently improve output quality, with self-refinement loops providing the most reliable improvements. It was concluded that AI tools can significantly accelerate backend development when guided by structured prompts and evaluated under standardized conditions. These findings support the integration of AI-assisted code generation into backend engineering practices to enhance productivity and reduce technical debt.

Keywords/tags (subjects)

Artificial Intelligence (AI), Backend Development, Code Generation, Prompt Engineering, Large Language Models (LLMs), Node.js, Flask, Laravel, .NET, Technical Debt, Prompt Optimization, Self-Refinement Loop, Few-Shot Injection, Role-Based Contextualization, Software Engineering Benchmarks, MongoDB, SFTP Automation

Contents

List of Figures

List of Tables

1 Introduction

In the modern age of advanced technology, artificial intelligence models like ChatGPT, DeepSeek, GitHub Copilot, and OpenAI Codex have gained a lot of adoption in various industries. Different sectors—medical, engineering, and technical fields—have started adopting AI in their operations to improve efficiency and drive innovation. For example, companies like Johnson & Johnson are launching AI training programs for staff to integrate AI in pharmaceutical development and operations (Business Insider, 2025). Moreover, research organizations like the MIT Jameel Clinic have started developing AI models that can simulate complex physical systems, with implications in drug discovery and the development of medical devices (MIT Jameel Clinic, 2025). Besides industrial use, individuals in daily life are increasingly depending on AI to help them with their daily tasks, thereby making it an indispensable part of modern society. For instance, students have started using AI technologies to aid their academic tasks, editors have started implementing AI based video features, etc. Additionally, AI features in smartphones are used to automatically edit images, suggest replies in messaging apps, and organize daily routines. However, despite its advantages, the application of AI in web development—especially in backend development—is still limited. While AI has the capability to write code snippets, generate functions, and optimize certain lines of code, it lacks a basic understanding of how code must be organized and how applications should be developed within specific constraints. Though AI shows efficiency, it lacks a core understanding of application and performance requirements. In addition to this, it also struggles to identify both critical and niche features, thus requiring human involvement. For instance, studies have shown that a significant percentage of code written by AI tools like GitHub Copilot can have glitches or exploitable design flaws (Pearce et al., 2021).

1.1 Background Theory

The combination of frontend and backend elements form the foundation of full-stack applications. While the frontend specializes in UI and client-side functionality, the backend manages data processing, business logic, user authentication, and communication with databases and external services (Jain et al., 2022). The backend serves as the computational core of full-stack applications and ensures the flow of data between an application's functions runs smoothly and securely across the board. A well-designed backend consists of several layers (server, application logic and database) which work harmoniously together to make it more effective and reliable when working behind the scenes (Ganguly, 2021). The backend is responsible for optimizing API requests, implementing airtight security measures and ensuring a desirable system performance; thus, acting as the backbone of any full-stack architecture (Sharma & Patel, 2020). Backend development uses a wide range of technological frameworks such as Node.js, Django, Flask, and Spring Boot and is supported by a host of various database systems such as MySQL, PostgreSQL, and MongoDB (Moussa, 2021). By adhering to the principles of prudent data handling and having checks in place for authentication, these technologies are both highly useful at retrieving data, and at implementing defenses against any hostile takeover. Backend development is more of a complex process in the sense that it deals with how high traffic apps process and manage user data (Tella, 2020). Furthermore, it keeps the system running full speed by offering mitigation against system failures and high-quality recovery options to ensure fault tolerance. AI is a significant driver of change in the field of backend development that has led to automation in the areas involving strengthening APIs, increasing performance of the application, and database maintenance. Unlike frontend development where AI concentrates on UI and UX, in the backend domain, AI focuses on system and process optimization, error detection and monitoring, and automation in decision-making. Machine learning algorithms have the ability to optimize database queries, predict system failures, and streamline real-time authentication processes, thus improving application efficiency (Brown & Davis, 2023). An added advantage of AI in backend development is automation for tasks such as database indexing, log parsing and anomaly detection (Hernandez & Clarke, 2023). AI also helps in boosting the scalability of backend systems with the help of predictive load balancing and adaptive resource allocation (Zhang et al., 2022).

In contrast to AI applied on the frontend side of applications where the primary task is to create a user-friendly experience and personalized recommendations or chatbots, AI in the backend offers more operational efficiency and security advantages (Williams, 2022). Frontend AI deals mainly with tasks that seek to enhance the user interface in terms of personalization, recommendations and the information that is quite helpful for the end users. On the other hand, AI in the backend focuses more on streamlined operational efficiency and hardened security. Furthermore, AI has revolutionized backend development via AI-powered code editors such as GitHub Copilot and OpenAI Codex where developers can write machine generated code to add boilerplate code blocks, search code, or write a function quickly. In addition to this, AI-powered automation in DevOps improves the deployment process by allowing developers to automate build pipelines, test workflows, and continuous integration/continuous deployment (CI/CD) practices. With these improvements, adding artificial intelligence to a backend model allows for more scalability, automation, and optimization which can significantly improve existing full-stack development endeavors.

1.2 Research Problem

The fundamental concern in the current development environment is building applications that are highly accurate, secure, scalable, and extensible while minimizing the development time. In addition, the code should be efficient, readable, and resistant to security vulnerabilities. One of the emerging concerns in AI-assisted development is prompt engineering, which directly impacts the quality of AI-generated code. Generic prompts often result in generalized solutions, but real-world development tasks require specific outputs that align with needs of an organization. In such cases, the quality of prompts plays an important role in generating accurate, scalable, and maintainable backend code. This highlights the significant influence that different prompt strategies have on AI-generated backend solutions. Integrating AI-generated code into real-world backend technologies presents a favorable opportunity for enhancing the development efficiency. By adapting and fine-tuning prompt engineering techniques, AI models can generate highly accurate, scalable, and secure code more effectively. However, this exploration produces a lot of important questions such as - reflecting on the accuracy of the code generated by different AI models, optimization of prompts to generate more scalable and maintainable backend code, and how AI can assist in detecting and preventing security vulnerabilities in backend systems.

This thesis has summarized the research problem into three distinct questions –

1. What are effective and objective methods to compare different AI models in terms of accuracy when generating backend code?

2. What impact does prompt engineering have on AI's ability to generate scalable, maintainable backend code and how does it handle the security concerns?

3. How can AI-generated code be optimized to reduce technical debt?

To address these research problems, the thesis proposes a comparative analysis of different AI models and backend frameworks, applied to an SFTP backend application. This evaluation will be based on the established SWE benchmarks, including accuracy, efficiency, scalability, readability, and error handling. Additionally, the research will also look into the role of prompt engineering as a way to enhance the quality of the AI-generated code by experimenting with various prompt strategies. Since the quality of the code is highly influenced by the input prompts, this study aims to demonstrate how different prompt techniques affected backend development outcomes.

1.3 Research Methods

Several important steps were involved in the methodology to fully analyze the chosen methods implemented in the practical applications. Initially, a comprehensive review of relevant literature was done, and this was intended to help in understanding the theory, the advancements, and overall utilization of AI in backend development. This literature review also helped in defining the scope of the thesis, as well as formulating the research questions present in this study. After the review, four backend development frameworks - Express with Node.js, Flask with Python, ASP.NET with C#, and Laravel with PHP - were chosen based on the popularity survey conducted by StackOverflow. An SFTP application was chosen for implementation to facilitate the comparison of AI-generated code using the standardized SWE benchmarks. The research used various AI models - such as ChatGPT, Github Copilot, and Gemini - to generate code specific to the application's requirements. This approach helped to provide a more comprehensive analysis of how different models perform in terms of accuracy, efficiency, and maintainability.

1.4 Literature Review

The study – *Evaluating Large Language Models Trained on Code* by Mark Chen, Jerry Tworek, Heewoo Jun, and others examines the performance of Codex - a GPT-based AI model fine-tuned on publicly available GitHub code. Using a fine-tuning approach and sampling techniques, the study assessed Codex's ability to generate functionally correct code. The results indicate that Codex outperforms previous models like GPT-3 and GPT-J, by solving 28.8% of problems in the HumanEval dataset. However, the study also highlights limitations, including syntactical errors, undefined function calls, and inherited biases from training data (Chen et al.). The research information provided in the review is significant in understanding the accuracy and limitations of AI-generated backend code.

Another study - *Asleep at the Keyboard? Assessing the Security of AI-Generated Code* by Hammond Pearce, Baleegh Ahmad, Benjamin Tan, and others explores the security risks associated with AI-generated code. Using a security evaluation framework, the study tested AI-generated snippets for vulnerabilities such as SQL injection, hardcoded secrets, and insecure authentication. The findings reveal that Copilot frequently produces insecure code, raising concerns about its reliability in real-world backend applications. The study emphasizes the necessity of AI-assisted security auditing tools to mitigate risks before deployment and is significant in understanding the security implications of AI-generated backend solutions.

Additionally, the study - *Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm* by Laria Reynolds and Kyle McDonell investigates the role of prompt engineering in optimizing AI-generated code. By employing comparative analysis, this study evaluates how different prompt structures influence the accuracy, scalability, and security of AI-generated code. The results show that the structured and detailed prompts significantly improve the performance of different language modes, while vague or generalized prompts lead to incomplete or error-filled outputs. Overall, these studies provide a solid foundation for formulating the core concept and research questions for this thesis.

1.5 Structure of Thesis

The thesis is divided into six chapters, covering the theory, practical implementation, research questions and subsequently the conclusion of the research.

- Chapter 1 serves as the introduction, defining the motivation behind the research and expresses the main research problem and objectives. It provides an initial exploration into the field of AI-code generation and prompt optimization explaining its significance in backend development.
- In chapter 2, the conceptual framework is elaborated. Here, the theoretical foundations and key concepts of AI in backend development are examined. Additionally, it provides a comprehensive overview of AI models, research methodology, and sets up the base for the SFTP application.
- Chapter 3 is dedicated to the implementation, prompt optimization, and testing of the code generated by different AI models using various prompts.
- Chapter 4 delves into the research questions; comparative analysis and consequent optimization strategies used to explore and explain the ideas and information for the said questions.
- Chapter 5 is devoted to the results of different comparative analysis using SWE benchmarks and the results of prompt optimization strategies. It provides a detailed account of the results of the implementation of the application discussed in chapter 3.
- Lastly, chapter 6 summarizes the conclusion and the key findings of this study.

2 Conceptual Layout

2.1 Backend in AI

AI is radically transforming the field of backend development by automating processes, enhancing efficiency and security measures. Traditional backend systems usually rely on manual coding, predefined rule-based logic, and rigid workflows. On the flip side, AI-powered or AI-instructed backend systems utilize machine learning algorithms, NLP, and automation frameworks to streamline operations (Smith, 2023). AI is showing incredible value in use from automated code generation and database optimization to security and smart API management (Jones & Patel, 2022).

Figure 1 represents a sequence diagram. It consists of the comprehensive lifecycle of an AI request in a modern backend system. It begins with a user sending a request to the API, which is then forwarded to the AI model for processing. In this case, the AI model interacts with both the Database and Security layer to fetch information and to ensure compliance and protection respectively. After the processing, the backend API sends back a response to the user.

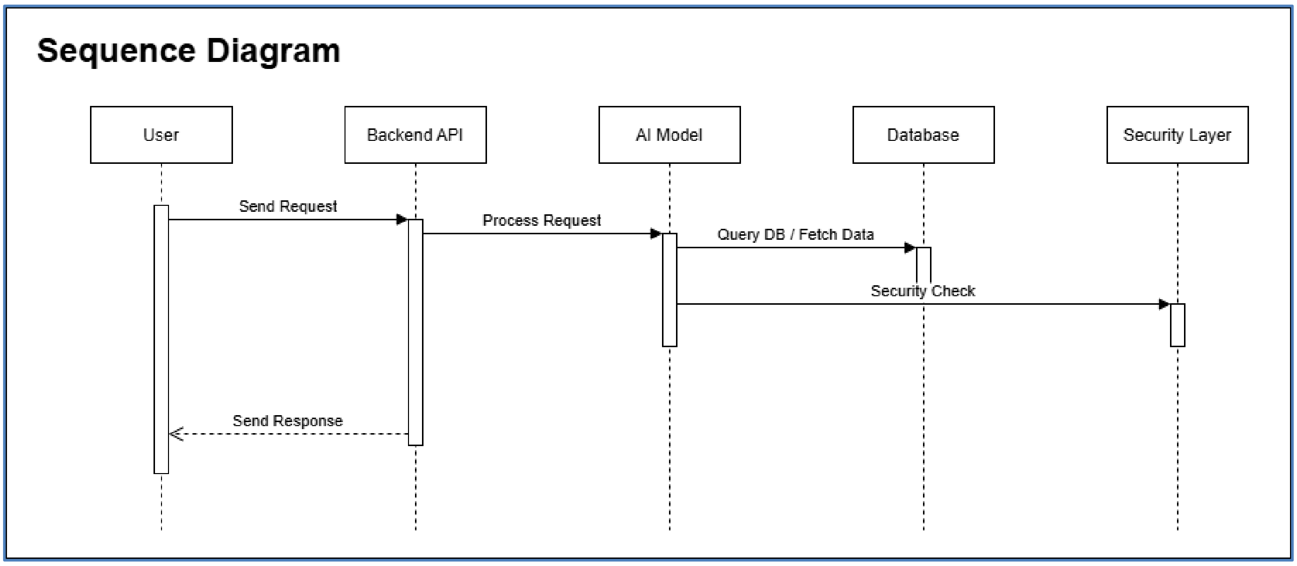


Figure 1 - Backend Sequence Diagram

One of the most significant benefits of AI backend development is that it can automate time-consuming and repetitive tasks while providing code suggestions, identifying syntax errors, and optimizing queries (OpenAI, 2021). For example, AI-based backend systems can use historical data to predict usage spikes and dynamically redistribute resources to prevent server overload (Taylor & Wong, 2023).

Another important area where artificial intelligence contributes to backend development relates to security. Traditional security measures rely on rule-based intrusion detection systems, which are not enough to successfully identify emerging threats (Miller & Zhao, 2021). AI-based threat detection models can analyze network traffic, detect unusual patterns, and prevent cyberattacks in real time (Brown & Singh, 2022). AI also helps in automating vulnerability assessments by scanning backend code for security loopholes, such as SQL injection vulnerabilities or misconfigured authentication mechanisms (Yoon & Lee, 2023). Figure 2 displays the utilization of AI implemented within the architecture of a backend application. As the user sends the request to the backend, it is handled via a multitude of layers, each handling and overseeing a specific part of the request. Additionally, it also displays different methods on how AI can be utilized within different layers. For instance, after the request is sent, the API gateway layer can utilize AI for log detection within the authentication and authorization functions. Moreover, AI can also prove to be helpful within the request transformation from one format to the other. Similarly other layers such as - processing, database, security, and logging layers can utilize AI to the full capabilities for more efficient and effective backend processing.

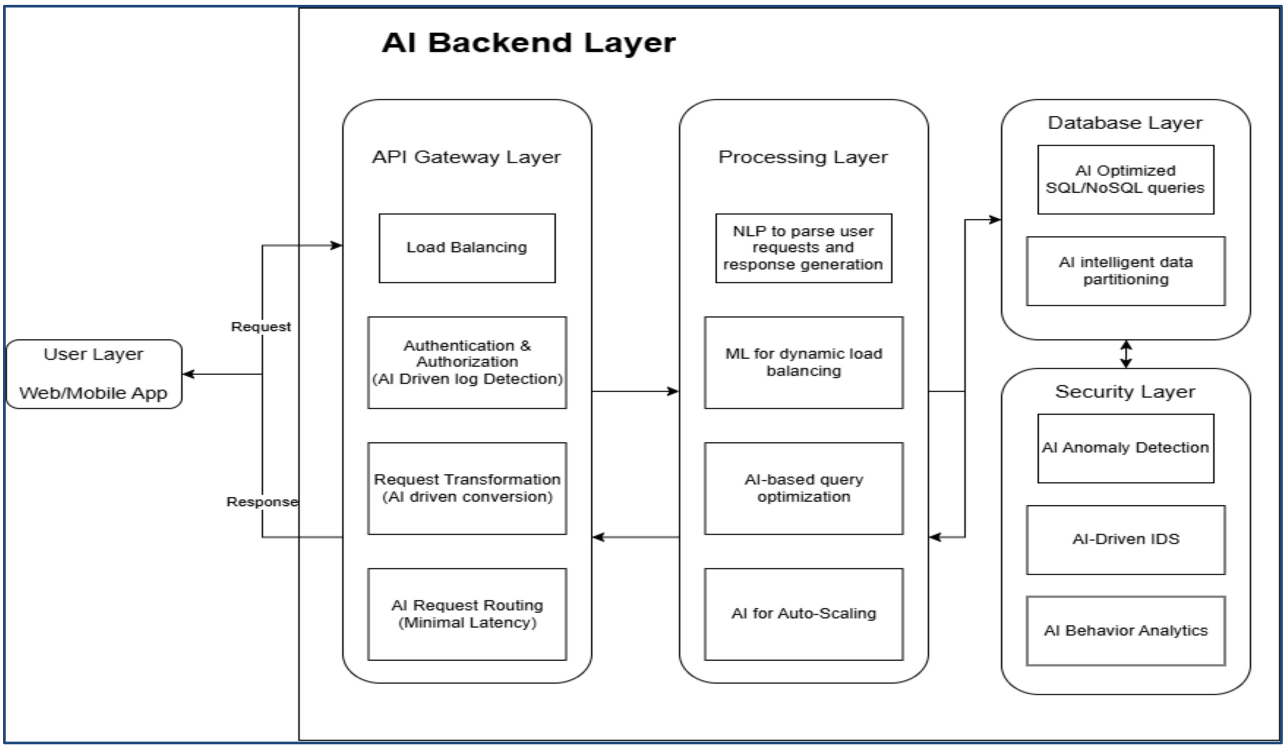


Figure 2 - AI Backend Layer

Overall, the integration of AI into backend development can enhance performance, security, and scalability while reducing manual effort. As AI continues to evolve, its role in backend systems will expand, enabling more autonomous, self-optimizing backend architectures (Miller et al., 2024).

2.2 AI Engines Overview

For the code generation and its subsequent automation, many AI models and providers such as ChatGpt, Deepseek, and Perplexity can be utilized by developers. These AI engines significantly enhance productivity, debugging efficiency, and code optimization. However, this study will utilize the code generated by the publicly available AI engines - ChatGpt, Gemini, DeepSeek, and Github Copilot. These AI models/tools were chosen based on the most popular AI surveys conducted by Stanford University (Stanford University, 2024) and Orca Security (Bar Kaduri & Stansfield, 2024).

2.2.1 ChatGPT

ChatGPT, developed by OpenAI, operates on the GPT-4 architecture, which utilizes a transformer-based deep learning model trained on an extensive dataset of publicly available code, licensed data, and OpenAI-partnered resources. Unlike conventional rule-based code generators, ChatGPT incorporates self-attention mechanisms and dynamic token prediction, allowing it to understand and generate multi-line backend code with context preservation. This is useful in backend systems that require modular code structures, middleware integration, and API endpoint generation (Brown et al., 2020). The transformer model processes text by computing probability distributions over possible token sequences, optimizing for logical consistency and syntactical correctness in code generation (Vaswani et al., 2017). For back-end programming, ChatGPT accelerates coding workflows for developers through context-aware code block suggestions, refactoring of current backend logic, and API service error debugging. As opposed to traditional auto-completion tools, ChatGPT uses reinforcement learning through human feedback (RLHF) to align its generated responses with real-world software engineering trends (Christiano et al., 2017). However, since ChatGPT is trained on fixed data sets, it is not in real-time flexible to the evolution of frameworks and libraries and therefore requires developers to test generated code manually for compatibility with backend technologies like Node.js, Django, or Spring Boot (OpenAI, 2023).

2.2.2 DeepSeek AI

DeepSeek AI is optimally suited to backend engineering automation using an adaptive iterative reinforcement learning approach. Different from static models of backend code generation, DeepSeek uses tree-based code generation approaches wherein it recursively generates program structures hierarchically such that function calls, class dependencies, and API request flows are always optimized (Li et al., 2024). This method drastically improves backend API development by making sure that generated functions are industry-compliant, remain modular, and are not redundant.

DeepSeek AI is optimally suited to backend engineering automation using an adaptive iterative reinforcement learning approach. Different from static models of backend code generation, DeepSeek uses tree-based code generation approaches wherein it recursively generates program structures hierarchically such that function calls, class dependencies, and API request flows are always optimized (Li et al., 2024). This method improves backend API development by making sure that generated functions remain modular, and are not redundant. It is particularly useful for backend developers utilizing RESTful and GraphQL APIs. In that particular case, it aids in the automatic mapping of data models, enhances the structure of request payloads, and creates database schema migrations. DeepSeek also provides comprehensive static analysis, which enables the prediction of logical errors or costly query executions prior to deployment (Zhao et al., 2024). It varies from other models in a way that it emphasizes structure optimization rather than just ensuring syntactical correctness.

2.2.3 Gemini

Google's Gemini AI model stands out from other code generation platforms with the use of a multimodal learning framework. This distinctive feature allows for the analysis of diverse data types, such as text, images, audio, and other structured collections of data. Gemini's operational architecture takes advantage of the advanced computational power of Google’s TPU v5 hardware accelerators, which support efficient training of the model according to the computational requirements inherent in artificial intelligence. As a result, this architectural setup allows for the concurrent execution of code-related operations through the use of backend microservices (Google DeepMind, 2024). In addition, the model has been carefully optimized for integration with Google Cloud Functions, Firebase, and Kubernetes, thus ensuring its compatibility with serverless infrastructures and containerized backend systems.

One of the key strengths of Gemini is its flexibility with regard to model training, which allows it to continuously improve its programming capabilities through the use of federated learning techniques. This feature allows developers who work with Google services to update the model without violating data privacy (Bonawitz et al., 2019). Compared to ChatGPT, which generates standalone scripts to use in backend development, Gemini incorporates real-time API security scanning, improves the effectiveness of query execution in backend databases, and provides AI-augmented bug detection as part of DevOps processes (Google AI, 2024). However, the heavy dependence on Google's infrastructure inherent with Gemini can lead to compatibility issues for developers who use AWS Lambda or Azure Functions when seeking to deploy AI-written backend code.

2.2.4 Github Copilot

GitHub Copilot works together with OpenAI Codex, a specialized version of GPT that has been carefully fine-tuned for code-generation applications. One of the distinguishing features of Copilot compared to other AI-powered coding tools is its deep integration into software development environments such as Visual Studio Code, JetBrains, and Neovim, where it provides AI-powered real-time autocompletion suggestions and advanced code suggestions (Chen et al., 2021). Codex has been trained on a large corpus of billions of lines of open-source code, which allows it to recognize common backend patterns, known best practices, and standard formats for API requests. Unlike ChatGPT and Gemini, which require users to provide explicit input for backend logic generation, Copilot performs a thorough contextual analysis of the entire project. This ability enables the creation of code recommendations real-time based on previously written fragments of code, in addition to evaluations of the overall architectural context of the project. Consequently, Copilot demonstrates high competence in performing backend operations related to boilerplate code development, API endpoint configuration, and security setup (Ziegler et al., 2023). Nevertheless, through the large use of open-source codebases in Copilot, there is a probable chance of generating insecure or outdated code snippets; therefore, developers need to apply manual checks before deployment to production environments of AI-coded backend code.

Table 1 portrays a comparison among different AI models in backend development.

Table 1 - AI Model Comparison

AI Model	Training Focus	Best For	Cloud/IDE Integration	Strengths
ChatGPT	General-purpose AI trained on diverse datasets	Full-stack development, chatbot APIs	OpenAI API	Human-like explanations, supports multiple languages
Gemini	Optimized for multimodal AI tasks	Google Cloud, Firebase, Kubernetes	Google Cloud AI	Cloud-based AI debugging, vulnerability detection
DeepSeek	Backend automation with reinforcement learning	API generation, backend services	Docker, Kubernetes	Code structuring, optimized performance
GitHub Copilot	Code-first approach using GitHub data	IDE-based code completion	VS Code, JetBrains, Neovim	Best for inline suggestions, refactoring, unit test generation

2.3 Performance Metrics

The foundation of this study follows a hybrid methodology that blends the conceptual research with comparative technical assessment. This dual approach was selected to explore not only the theoretical information of AI on backend development but also its practical effectiveness when applied to real-world scenarios. Instead of relying solely on academic narratives or isolated case studies, this research bridges both domains – bringing AI-generated code into active software development environments to measure its actual effectiveness, accuracy, and overall benefits. To ensure consistency in comparison, a common application type was defined across all chosen frameworks – SFTP (Secure File Transfer Protocol) server. This application was specifically chosen as it consists of moderately complex backend logic, secure handling of user input, session control, encryption mechanisms, and file operations. This complexity provides an astute way to benchmark the code generated by the AI models.

To evaluate the quality of code produced, a set of carefully chosen software engineering benchmarks was applied—

a. **Code Correctness -** The criteria that was used in assessing if the code generated by AI has effectively met the SFTP application's functional requirements. The main goal of this evaluation was to find if there were any logical mistakes, missing implementations, or output variations that would jeopardize their intended functioning.

b. **Runtime Efficiency -** Integrated scripts and API testing tools such as Postman were used in order to determine how well the code that was generated was executed. In order to provide information about the adaptability and viability of the code for production environments, this metric helps to measure the code's responsiveness, the time taken for its execution, and the number of resources that has been consumed.

c. **Maintainability**: This criterion was looked into through a thorough examination of the modular design of the code, compliance to the commonly used coding practices, as well as the removal of redundant or duplicate elements in the code. In regards to the collaborative development, the code has to be maintained so it can be used in facilitating updates, troubleshooting, and long-term flexibility.

d. **Readability**: The consciousness of the structure of the code, which includes appropriate indentation and comments that have clear insights, as well as the clarity and the simplicity of the logic, were used to evaluate the readability. Furthermore, increased understanding is facilitated by clear readability, helping in making the maintenance and collaborative development in the future easier.

e. **Error Rate and Debugging Complexity**: The criteria assessed the number of times human involvement was needed and adjustments were needed in order to make the code created by AI completely functional. It additionally also took into consideration the complexity of finding and fixing flaws, which demonstrated the output's dependability and resilience.

SFTP application provides a moderate complexity and is sufficiently rich in backend logic to test state management, real-time event handling, and compliance with security practices.

These benchmarks were selected because they reflect real-world developer priorities. Together, these provide a comprehensive view of both the technical performance and practical usability of AI-generated backend applications. Through the layered methodology, this research evaluates not just what AI can generate but also how it aligns with real-world development standards and how different prompt-optimizing strategies can affect the outcome as seen in table 2.

Table 2 - Code Metrics

Metric	Definition	Weight (%)	Importance
Code Correctness	Logical validity and functional accuracy	20%	Validates core AI output functionality
Runtime Efficiency	Execution time, latency, memory/cpu usage	20%	Essential for backend performance
Maintainability	Modularity and code reusability	20%	Reduces long-term technical debt
Readability	Indentation, meaningful variable names	20%	Enables collaborative work and quick onboarding
Error Rate	Error Rate Evaluation	20%	Measures practical usability and reliability of AI-generated code

This approach is technically viable due to its flexibility to meet all the requirements –

- **Standardization:** Using a fixed application context (SFTP) removes variability across test cases.
- **Multi-dimensional assessment:** Evaluates code not just for logic, but also for real-world constraints developers face.
- **Reproducibility:** Benchmarks like runtime and maintainability can be quantitatively measured using tools (e.g., linters, profilers, Postman API testing).
- **Alignment with industry metrics:** These criteria align with key principles in software engineering (e.g., ISO/IEC 25010 standard).
- **Prompt strategy control:** Comparing results across prompt variants (role-based, few-shot, refinement-loop) allows causal inference on what works best.

2.4 Base of the Application

As discussed before, the most logical route was to create an SFTP-based application specifically designed to facilitate the standardized and controlled assessment of AI-generated backend code. This application was designed to include a range of common backend features replicated from real-world applications in order to have a nominal complexity sufficient to challenge AI code generation tools with different benchmarks. This application has a common base which includes the default modules and functionalities such as – user authentication and authorization, SFTP integration, data storage using MongoDB, file storage with Cloudinary integration, role-based access, and RESTful API design as described in the table 3.

Table 3 - SFTP Application Components

Component	Usage	Importance
User Authentication & JWT	Login/Signup & Token-based sessions	Security & Access-Control.
SFTP File Transfer	Upload or Download files securely	Real-world file handling & Async Operations
MongoDB (Database)	Stores users & file metadata	Scalable, flexible data storage
Cloudinary (File Storage)	Host uploaded media	Media optimization, external API use
Role-Based Access Control	Admin/user permission separation	User privilege enforcement
RESTful API Endpoints	Route structure for all operations	Modularity, standard backend design

Additional functionalities—such as event triggers, scheduled uploads, and click events— were incorporated into the application. These features were implemented by generating corresponding code snippets using various AI models and integrating them into the selected backend frameworks. Their inclusion was the main key to allowing for broader evaluation of how well the AI tools handle event-driven programming and asynchronous task scheduling, akin to the real-world applications. The base of the application follows a clear structure developed within any framework, as seen in figure 3.

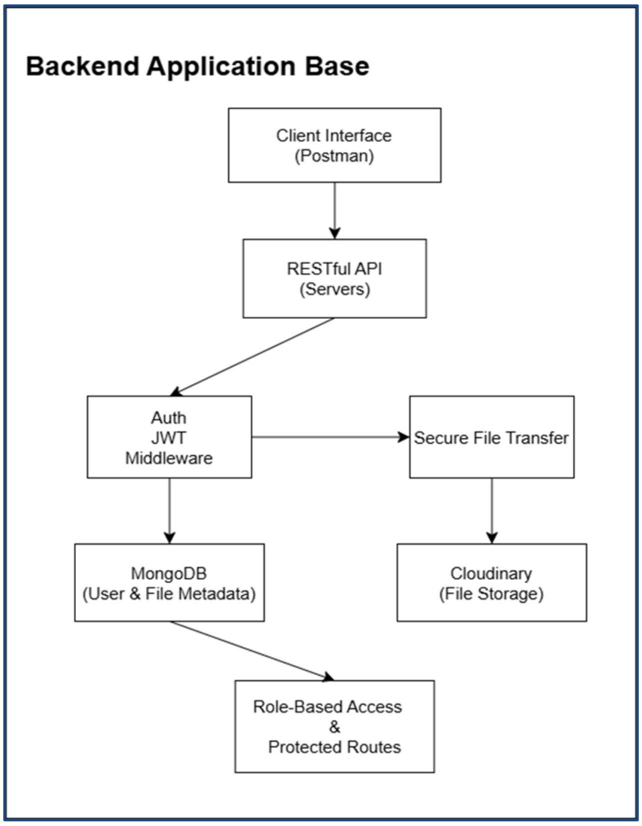


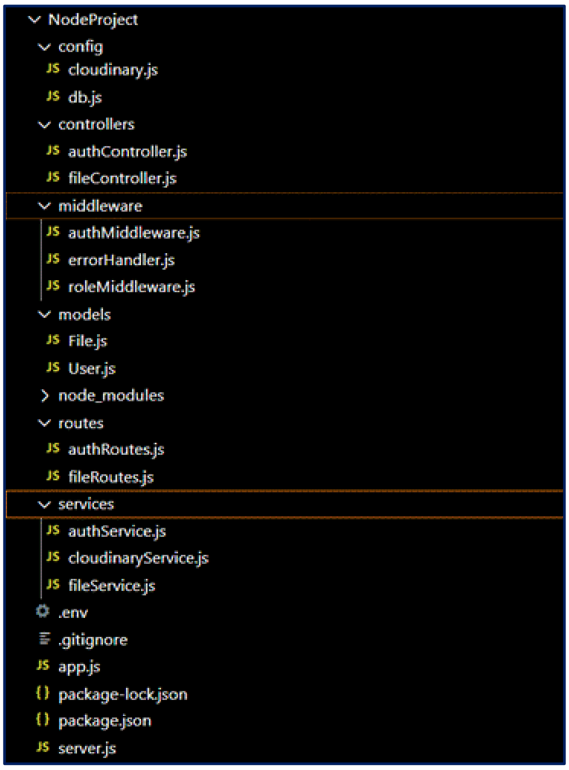
Figure 3 - Backend Application Base

3 Implementation

3.1 Frameworks

3.1.1 Node JS (Express)

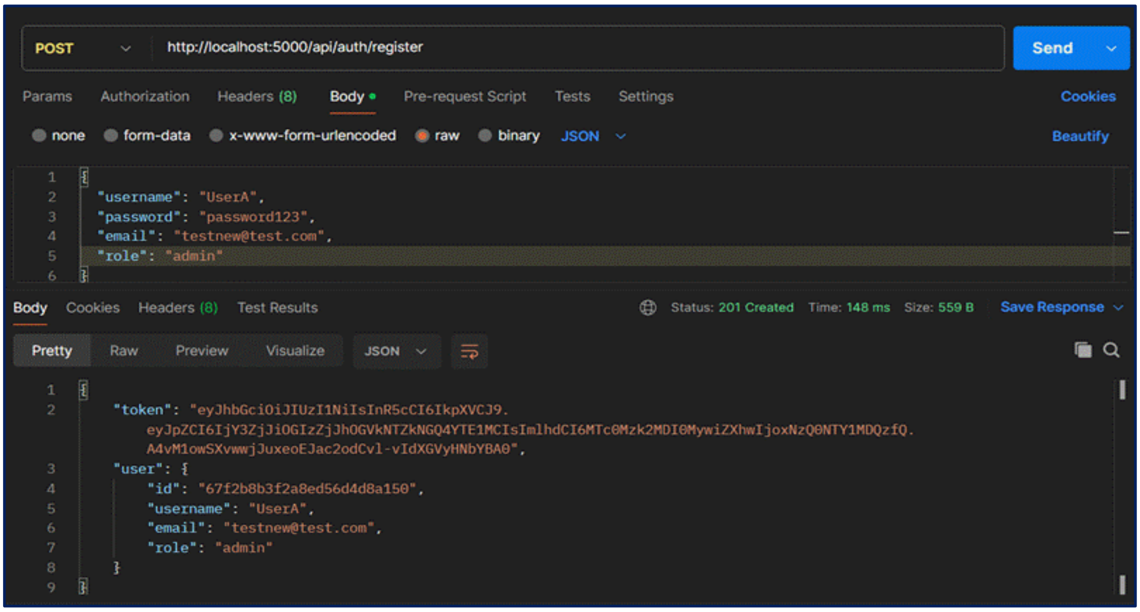
The first framework used to create the application was Node.js with Express. Due to its non-blocking architecture and an accessible ecosystem of npm packages, it helped in the rapid development of complex features such as authentication, cloud integration, and SFTP support with minimal redundant code. Additionally, due to its high flexibility, it allowed an easy integration of MongoDB for the database and Cloudinary for cloud storage. Also, Node.js operates on a single-threaded, event-driven architecture, which makes it suitable for developing applications that require real-time processing and the handling of multiple concurrent operations. In the context of this SFTP application, the ability to handle multiple file uploads and downloads simultaneously without blocking the server is crucial for maintaining performance. Figure 4 displays the architecture of the application.



```
NodeProject
  config
    cloudinary.js
    db.js
  controllers
    authController.js
    fileController.js
  middleware
    authMiddleware.js
    errorHandler.js
    roleMiddleware.js
  models
    File.js
    User.js
  node_modules
  routes
    authRoutes.js
    fileRoutes.js
  services
    authService.js
    cloudinaryService.js
    fileService.js
  .env
  .gitignore
  app.js
  package-lock.json
  package.json
  server.js
```

Figure 4 - Node App Structure

Figures 5 and 6 displays the current registration and login mechanism within the application and also provides the postman details.



POST http://localhost:5000/api/auth/register

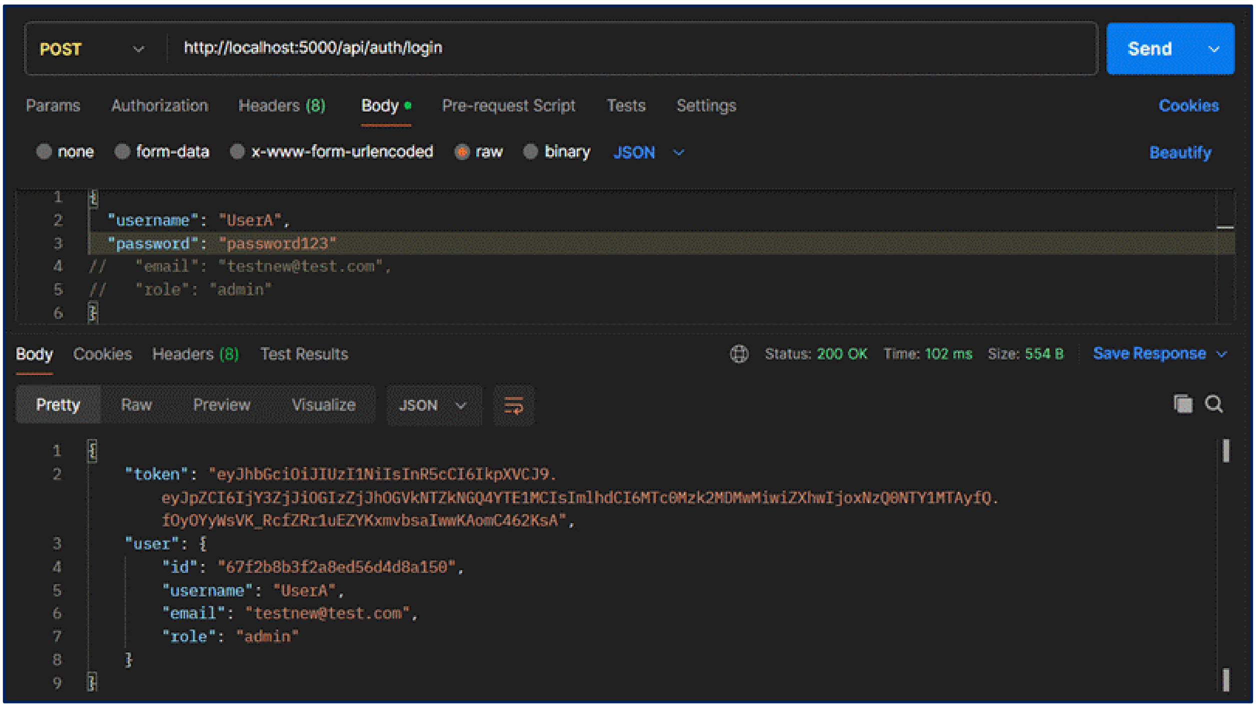
Body: ○ none ○ form-data ○ x-www-form-urlencoded ● raw ○ binary JSON

```json
{
  "username": "UserA",
  "password": "password123",
  "email": "testnew@test.com",
  "role": "admin"
}
```

Body — Status: 201 Created Time: 148 ms Size: 559 B — JSON

```json
{
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
        eyJpZCI6IjY3ZjJiOGIzZjJhOGVkNTZkNGQ4YTE1MCIsImlhdCI6MTc0Mzk2MDI0MywiZXhwIjoxNzQ0NTY1MDQzfQ.
        A4vM1owSXvwwjJuxeoEJac2odCvl-vIdXGVyHNbYBA0",
    "user": {
        "id": "67f2b8b3f2a8ed56d4d8a150",
        "username": "UserA",
        "email": "testnew@test.com",
        "role": "admin"
    }
}
```

Figure 5 - Node Registration Mechanism



POST http://localhost:5000/api/auth/login

```
{
  "username": "UserA",
  "password": "password123"
//   "email": "testnew@test.com",
//   "role": "admin"
}
```

Status: 200 OK Time: 102 ms Size: 554 B

```
{
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
        eyJpZCI6IjY3ZjJiOGIzZjJhOGVkNTZkNGQ4YTE1MCIsImlhdCI6MTc0Mzk2MDMwMiwiZXhwIjoxNzQ0NTY1MTAyfQ.
        fOyOYyWsVK_RcfZRr1uEZYKxmvbsaIwwKAomC462KsA",
    "user": {
        "id": "67f2b8b3f2a8ed56d4d8a150",
        "username": "UserA",
        "email": "testnew@test.com",
        "role": "admin"
    }
}
```

Figure 6 - Node Login Mechanism

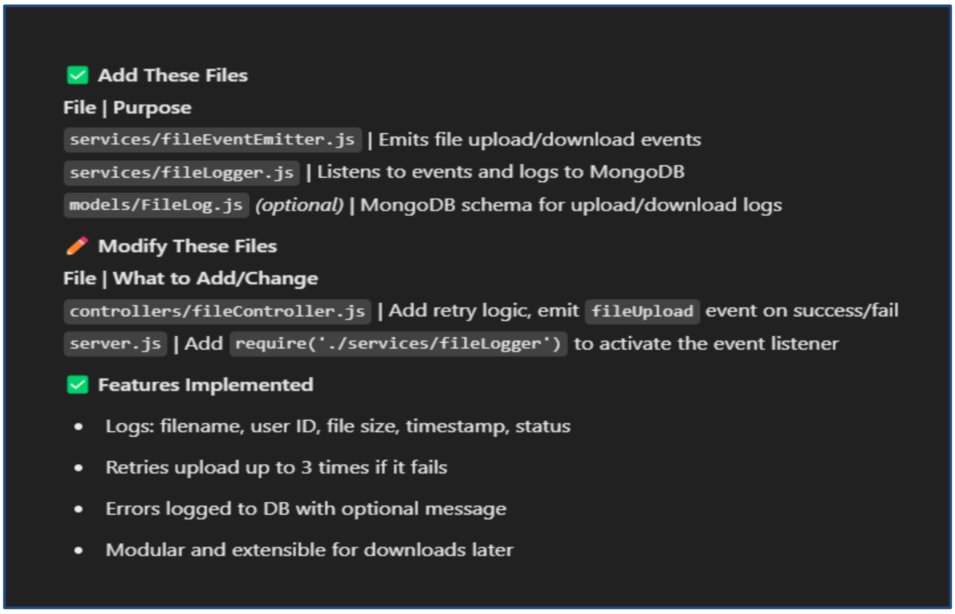
After the development of the base application within Node.js, several prompts were created to generate code snippets related to different schedule events required to test the AI-generated code.

a. Prompt 1 - *Generate a Node.js event listener that triggers whenever a file is uploaded or downloaded via SFTP. The event should log the file name, user ID, timestamp, and file size to the database (MongoDB).*

The prompt asks for a Node.js function to monitor SFTP file uploads and downloads, logging details like the file name, user ID, timestamp, and file size. These details should then be stored in a MongoDB database for tracking and record-keeping purposes. It's essentially a system for automated event logging and data storage.

- **ChatGPT**

  The above-mentioned prompt was delivered to ChatGPT, which provided an extensive response. Additionally, before providing the code snippet, it asked to provide its entire structure and respective files. After that, it provided elaborative files and code snippets, and the findings were summarized as seen in the figure 7.



✅ **Add These Files**

File | Purpose

`services/fileEventEmitter.js` | Emits file upload/download events

`services/fileLogger.js` | Listens to events and logs to MongoDB

`models/FileLog.js` *(optional)* | MongoDB schema for upload/download logs

✏️ **Modify These Files**

File | What to Add/Change

`controllers/fileController.js` | Add retry logic, emit `fileUpload` event on success/fail

`server.js` | Add `require('./services/fileLogger')` to activate the event listener

✅ **Features Implemented**

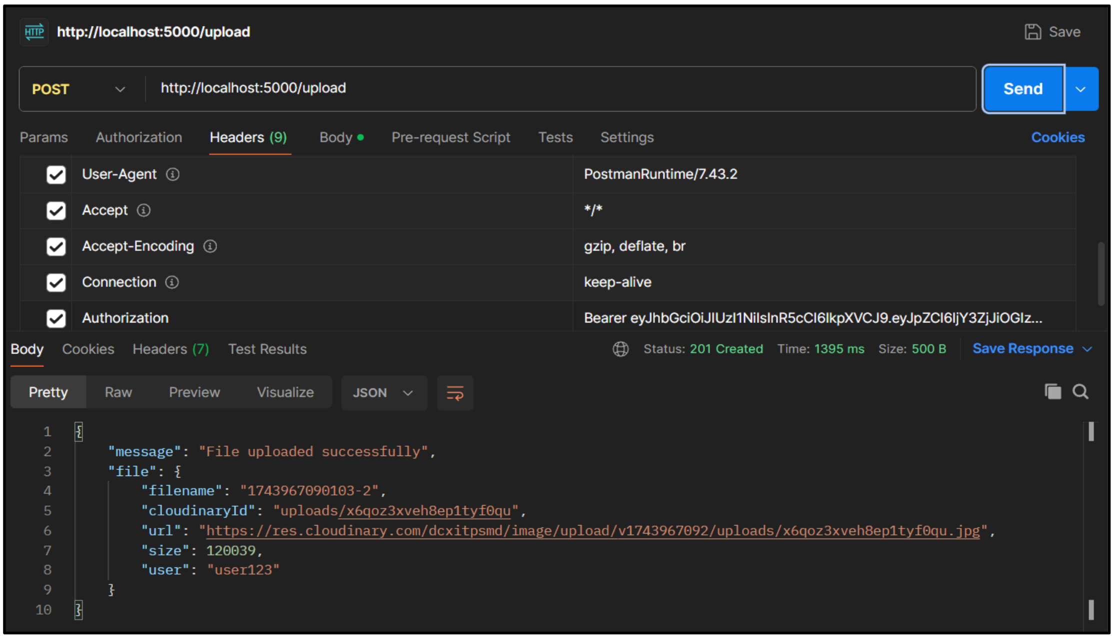
- Logs: filename, user ID, file size, timestamp, status
- Retries upload up to 3 times if it fails
- Errors logged to DB with optional message
- Modular and extensible for downloads later

Figure 7 - Prompt 1 - ChatGpt Result

After modifying the fileController.js and fileUpload functions, the next step advised by ChatGPT was to implement Emitter, Logger, and Log files in order to upload/download files, listen to event logs, and create a working schema for the logs. Additionally, it also implemented the retry functions with optional error logging, and modular/extensible functions for downloads. The upload function was modified as seen in figure 8.

```
// File upload route
app.post('/upload', upload.single('file'), async (req, res) => {
  if (!req.file) {
    return res.status(400).json({ error: 'No file uploaded' });
  }

  const filePath = req.file.path;  // Temporary file path from Multer
  const userId = req.body.userId;  // You can pass user ID via request body

  try {
    // Upload file to Cloudinary
    const cloudinaryResult = await uploadFileToCloudinary(filePath);

    // Prepare file details
    const fileDetails = {
      filename: cloudinaryResult.original_filename,
      cloudinaryId: cloudinaryResult.public_id,
      url: cloudinaryResult.secure_url,
      size: req.file.size,
      user: userId
    };

    // Emit the event to log the file upload details
    fileEventEmitter.emit('fileUploaded', fileDetails);

    // Delete the temporary file from local storage
    fs.unlinkSync(filePath);

    // Send success response
    res.status(201).json({
      message: 'File uploaded successfully',
      file: fileDetails
    });
  } catch (error) {
    console.error('Error during file upload:', error);
    res.status(500).json({ error: 'Error uploading file: ' + error.message });
```

Figure 8 - Prompt 1 - ChatGPT Code Snippet



POST http://localhost:5000/upload

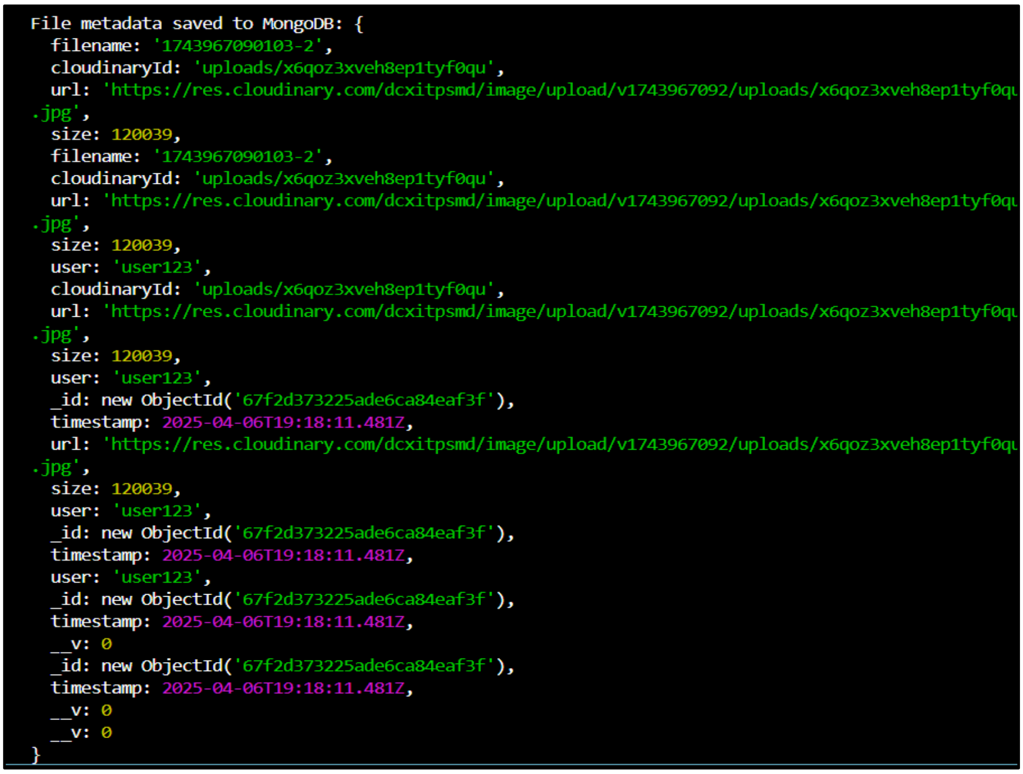
☑ User-Agent	PostmanRuntime/7.43.2
☑ Accept	*/*
☑ Accept-Encoding	gzip, deflate, br
☑ Connection	keep-alive
☑ Authorization	Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjY3ZjJiOGIz...

Status: 201 Created Time: 1395 ms Size: 500 B

```
{
    "message": "File uploaded successfully",
    "file": {
        "filename": "1743967090103-2",
        "cloudinaryId": "uploads/x6qoz3xveh8ep1tyf0qu",
        "url": "https://res.cloudinary.com/dcxitpsmd/image/upload/v1743967092/uploads/x6qoz3xveh8ep1tyf0qu.jpg",
        "size": 120039,
        "user": "user123"
    }
}
```

Figure 9 - Prompt 1 - ChatGPT Postman Result

Figure 9 displays the postman results for the prompt 1 and figure 10 displays the saving of metadata from the application with file upload function in the console output.

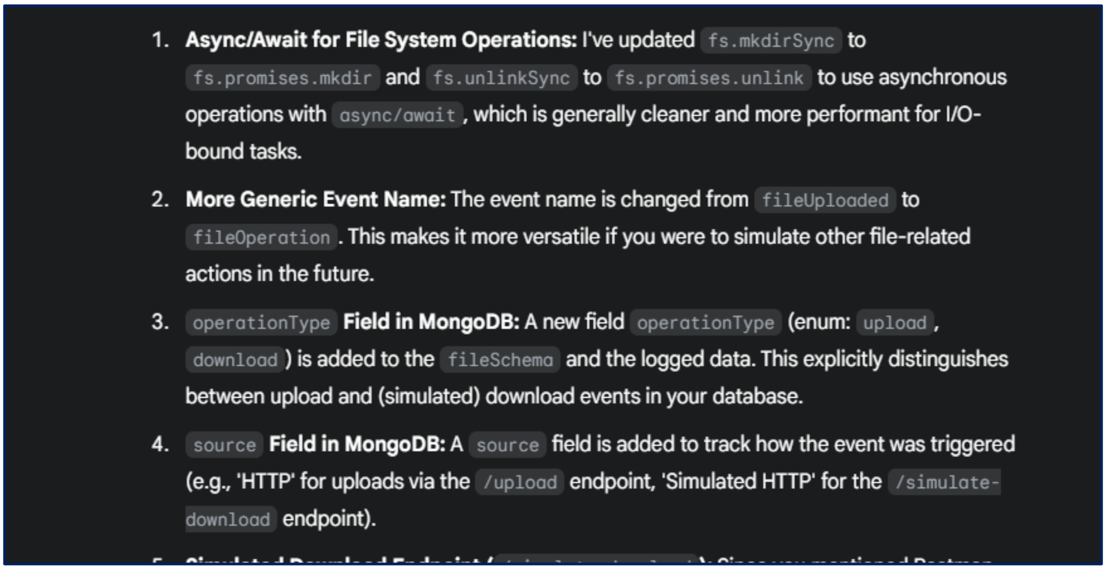


```
File metadata saved to MongoDB: {
  filename: '1743967090103-2',
  cloudinaryId: 'uploads/x6qoz3xveh8ep1tyf0qu',
  url: 'https://res.cloudinary.com/dcxitpsmd/image/upload/v1743967092/uploads/x6qoz3xveh8ep1tyf0qu
.jpg',
  size: 120039,
  filename: '1743967090103-2',
  cloudinaryId: 'uploads/x6qoz3xveh8ep1tyf0qu',
  url: 'https://res.cloudinary.com/dcxitpsmd/image/upload/v1743967092/uploads/x6qoz3xveh8ep1tyf0qu
.jpg',
  size: 120039,
  user: 'user123',
  cloudinaryId: 'uploads/x6qoz3xveh8ep1tyf0qu',
  url: 'https://res.cloudinary.com/dcxitpsmd/image/upload/v1743967092/uploads/x6qoz3xveh8ep1tyf0qu
.jpg',
  size: 120039,
  user: 'user123',
  _id: new ObjectId('67f2d373225ade6ca84eaf3f'),
  timestamp: 2025-04-06T19:18:11.481Z,
  url: 'https://res.cloudinary.com/dcxitpsmd/image/upload/v1743967092/uploads/x6qoz3xveh8ep1tyf0qu
.jpg',
  size: 120039,
  user: 'user123',
  _id: new ObjectId('67f2d373225ade6ca84eaf3f'),
  timestamp: 2025-04-06T19:18:11.481Z,
  user: 'user123',
  _id: new ObjectId('67f2d373225ade6ca84eaf3f'),
  timestamp: 2025-04-06T19:18:11.481Z,
  __v: 0
  _id: new ObjectId('67f2d373225ade6ca84eaf3f'),
  timestamp: 2025-04-06T19:18:11.481Z,
  __v: 0
  __v: 0
}
```

Figure 10 - Prompt 1 - ChatGPT - Console Result

- **Gemini**

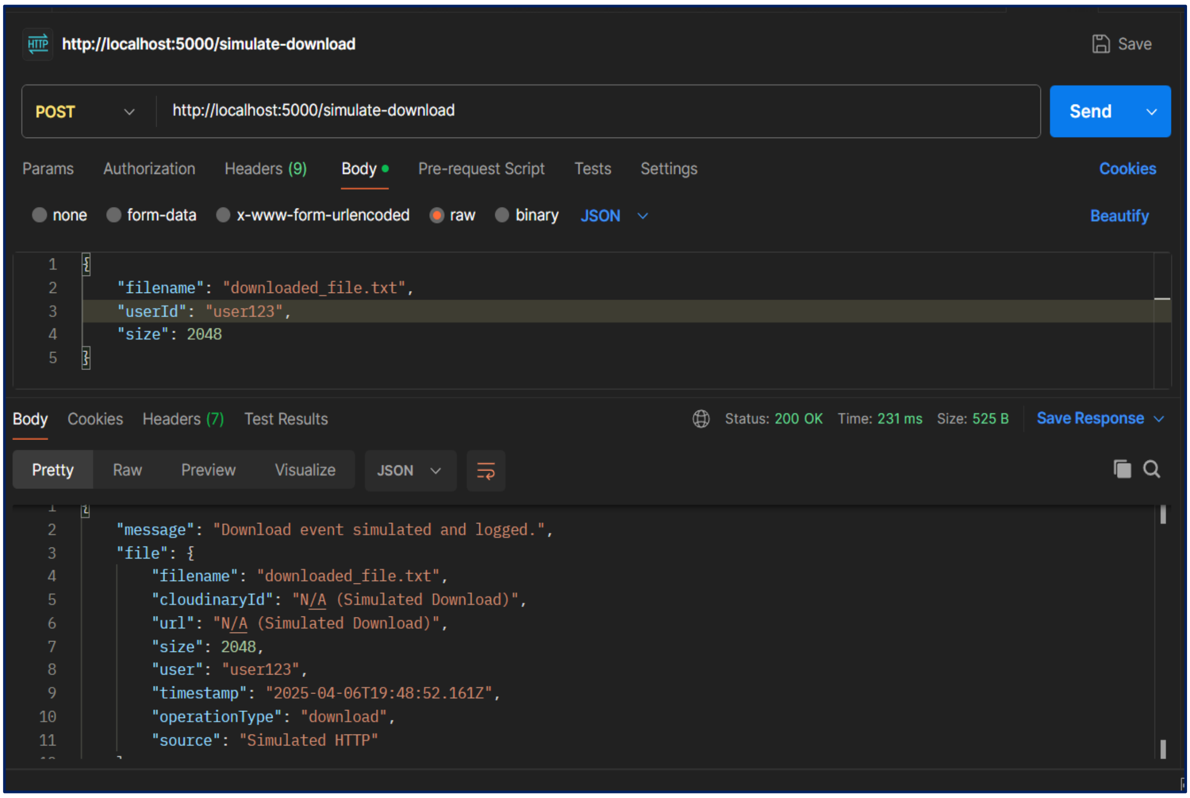
  The same prompt was given to Gemini, with addition to the files where the code needed to be changed. It enhanced the code to be more flexible and informative and made the event listener more general for any file operation. Additionally, it also added a way to track if an action was an 'upload' or 'download' in the database, and also noted if the action was triggered via the web API or some other (simulated) means. Moreover, it also added a special web endpoint to manually log "download" events. This allowed to record these actions in the database as seen in figure 11.



1. **Async/Await for File System Operations:** I've updated `fs.mkdirSync` to `fs.promises.mkdir` and `fs.unlinkSync` to `fs.promises.unlink` to use asynchronous operations with `async/await`, which is generally cleaner and more performant for I/O-bound tasks.
2. **More Generic Event Name:** The event name is changed from `fileUploaded` to `fileOperation`. This makes it more versatile if you were to simulate other file-related actions in the future.
3. `operationType` **Field in MongoDB:** A new field `operationType` (enum: `upload`, `download`) is added to the `fileSchema` and the logged data. This explicitly distinguishes between upload and (simulated) download events in your database.
4. `source` **Field in MongoDB:** A `source` field is added to track how the event was triggered (e.g., 'HTTP' for uploads via the `/upload` endpoint, 'Simulated HTTP' for the `/simulate-download` endpoint).

Figure 11 - Prompt 1 - Gemini Result

Figures 12 and 13 display the results of the code generated by Gemini.

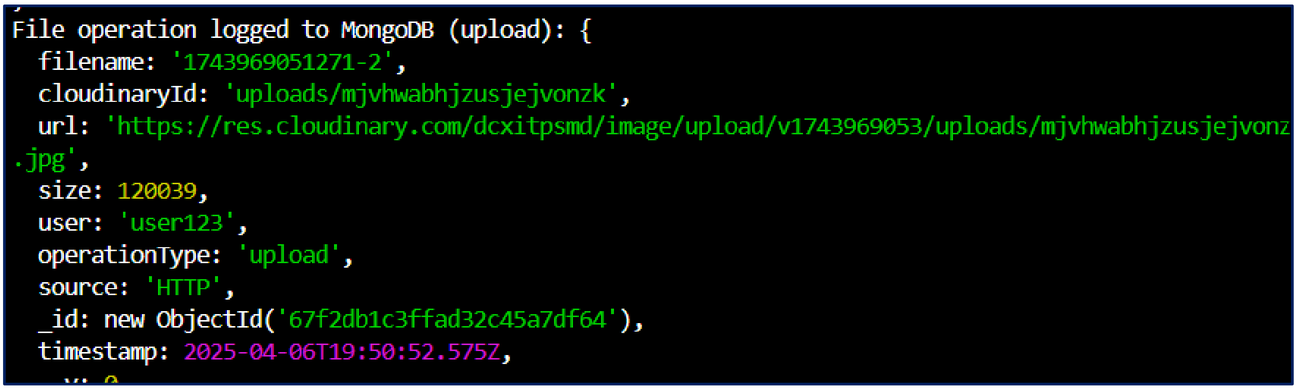


```
{
    "filename": "downloaded_file.txt",
    "userId": "user123",
    "size": 2048
}
```

Status: 200 OK Time: 231 ms Size: 525 B

```
{
    "message": "Download event simulated and logged.",
    "file": {
        "filename": "downloaded_file.txt",
        "cloudinaryId": "N/A (Simulated Download)",
        "url": "N/A (Simulated Download)",
        "size": 2048,
        "user": "user123",
        "timestamp": "2025-04-06T19:48:52.161Z",
        "operationType": "download",
        "source": "Simulated HTTP"
```

Figure 12 - Prompt 1 - Gemini Postman

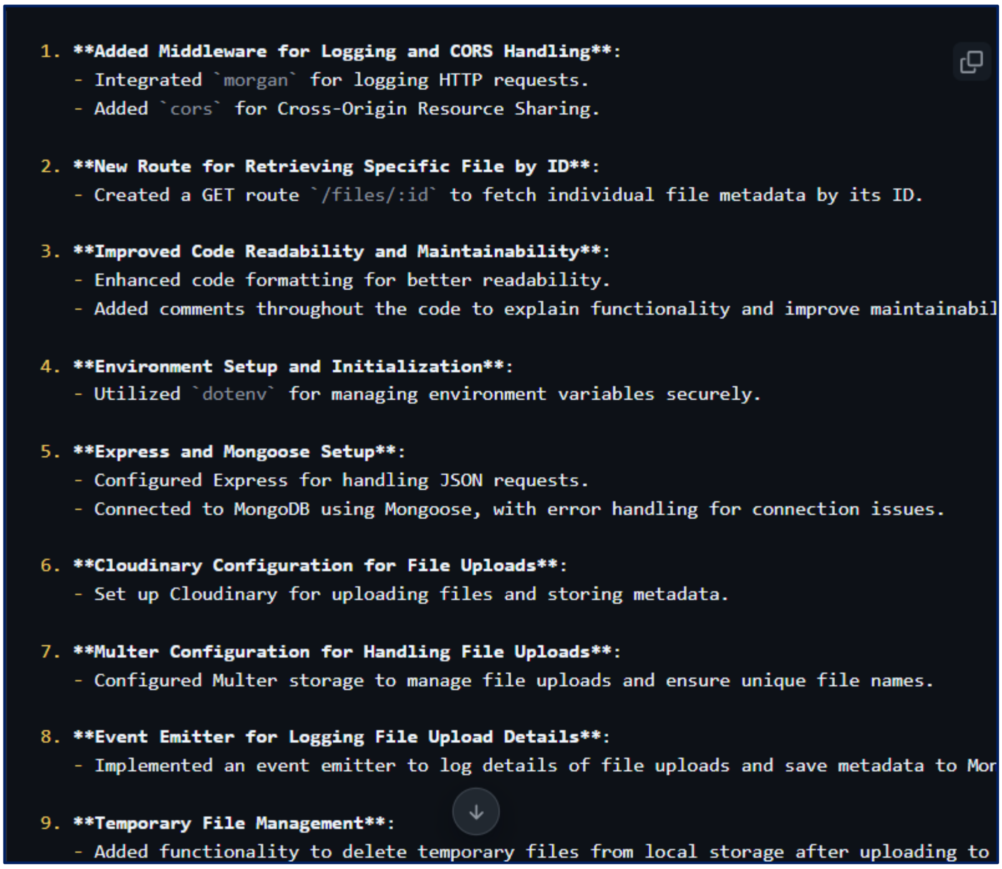


```
File operation logged to MongoDB (upload): {
  filename: '1743969051271-2',
  cloudinaryId: 'uploads/mjvhwabhjzusjejvonzk',
  url: 'https://res.cloudinary.com/dcxitpsmd/image/upload/v1743969053/uploads/mjvhwabhjzusjejvonz
.jpg',
  size: 120039,
  user: 'user123',
  operationType: 'upload',
  source: 'HTTP',
  _id: new ObjectId('67f2db1c3ffad32c45a7df64'),
  timestamp: 2025-04-06T19:50:52.575Z,
```

Figure 13 - Prompt 1 - Gemini Console Result

- **Github Copilot**

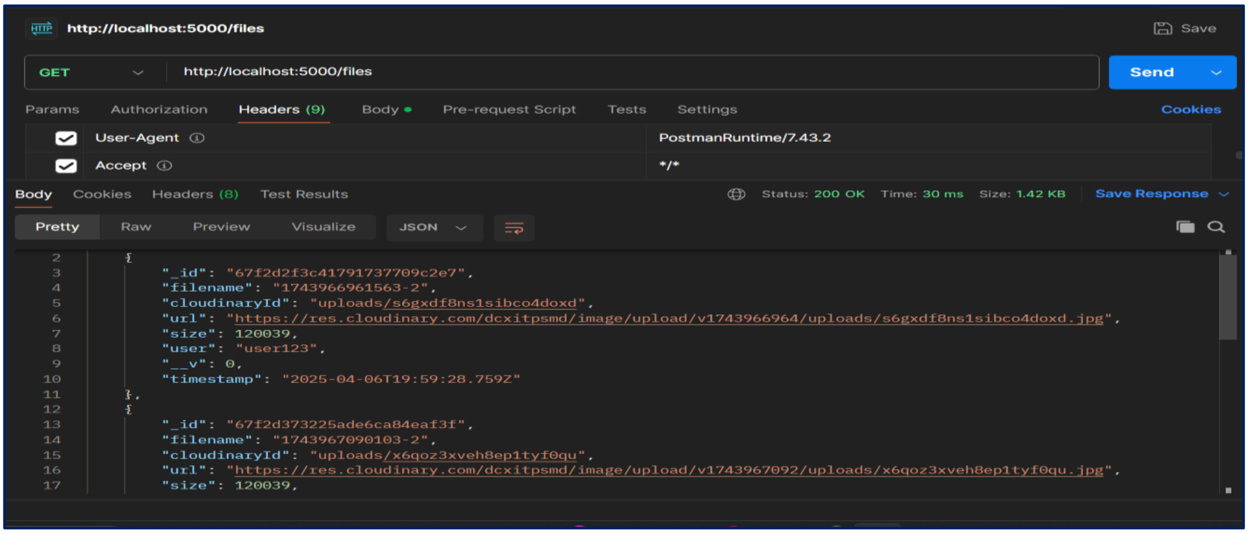
  The original Express.js application was enhanced by incorporating additional functionalities and improving code readability. Specifically, middleware for logging HTTP requests using morgan and handling Cross-Origin Resource Sharing (CORS) with the cors package was added, as seen in the figure 14.



```
1. **Added Middleware for Logging and CORS Handling**:
   - Integrated `morgan` for logging HTTP requests.
   - Added `cors` for Cross-Origin Resource Sharing.

2. **New Route for Retrieving Specific File by ID**:
   - Created a GET route `/files/:id` to fetch individual file metadata by its ID.

3. **Improved Code Readability and Maintainability**:
   - Enhanced code formatting for better readability.
   - Added comments throughout the code to explain functionality and improve maintainabil

4. **Environment Setup and Initialization**:
   - Utilized `dotenv` for managing environment variables securely.

5. **Express and Mongoose Setup**:
   - Configured Express for handling JSON requests.
   - Connected to MongoDB using Mongoose, with error handling for connection issues.

6. **Cloudinary Configuration for File Uploads**:
   - Set up Cloudinary for uploading files and storing metadata.

7. **Multer Configuration for Handling File Uploads**:
   - Configured Multer storage to manage file uploads and ensure unique file names.

8. **Event Emitter for Logging File Upload Details**:
   - Implemented an event emitter to log details of file uploads and save metadata to Mor

9. **Temporary File Management**:
   - Added functionality to delete temporary files from local storage after uploading to
```

Figure 14 - Prompt 1 - Github Copilot Results

A new route to get a specific file by its ID was created, enhancing the application's capability to retrieve individual file metadata. Furthermore, code formatting was improved and comments were added to make the codebase more readable and maintainable. The application now initializes environment variables using dotenv, sets up Express and Mongoose for MongoDB interactions, configures Cloudinary for file uploads, uses Multer for handling file uploads, and employs an event emitter to log file upload details. Figure 15 shows the result of the code generated by Github Copilot.



☑ User-Agent: PostmanRuntime/7.43.2
☑ Accept: */*

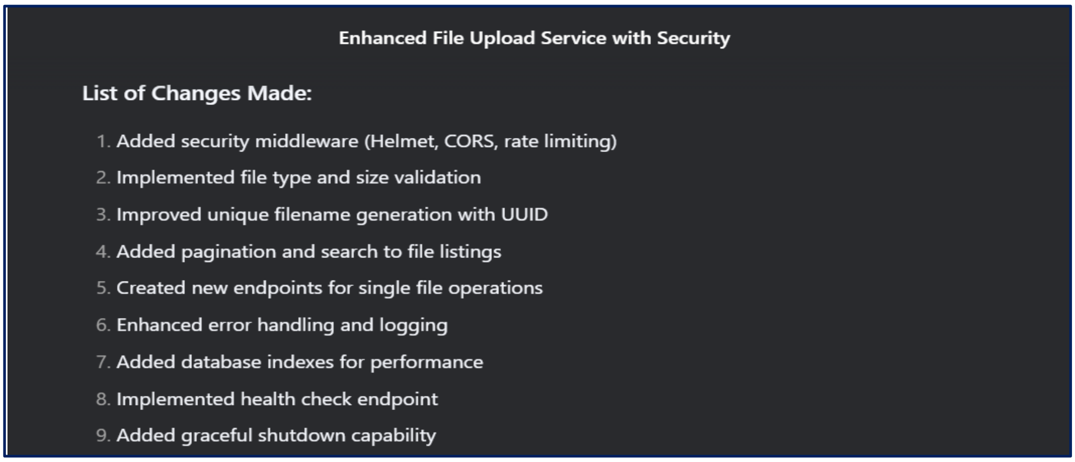
Status: 200 OK Time: 30 ms Size: 1.42 KB

```
    {
        "_id": "67f2d2f3c41791737709c2e7",
        "filename": "1743966961563-2",
        "cloudinaryId": "uploads/s6gxdf8ns1sibco4doxd",
        "url": "https://res.cloudinary.com/dcxitpsmd/image/upload/v1743966964/uploads/s6gxdf8ns1sibco4doxd.jpg",
        "size": 120039,
        "user": "user123",
        "__v": 0,
        "timestamp": "2025-04-06T19:59:28.759Z"
    },
    {
        "_id": "67f2d373225ade6ca84eaf3f",
        "filename": "1743967090103-2",
        "cloudinaryId": "uploads/x6qoz3xveh8ep1tyf0qu",
        "url": "https://res.cloudinary.com/dcxitpsmd/image/upload/v1743967092/uploads/x6qoz3xveh8ep1tyf0qu.jpg",
        "size": 120039,
```

Figure 15 - Prompt 1 - Github Copilot Postman Results

- **Deepseek AI**

  Deepseek enhanced the file upload service with multiple improvements focused on security, reliability, and functionality. They implemented security measures including Helmet middleware for HTTP headers, configurable CORS policies, and rate limiting to prevent abuse. File validation was added to restrict uploads to specific MIME types and enforce size limits. The upload process was improved with better unique filename generation using UUIDs. New features included paginated file listings with search capabilities, individual file operations (GET/DELETE), and a health check endpoint. Error handling was strengthened throughout the application, and database indexes were added for better performance. The service now includes graceful shutdown capabilities and more detailed logging for operational visibility. These changes are seen in figures 16 and 17.

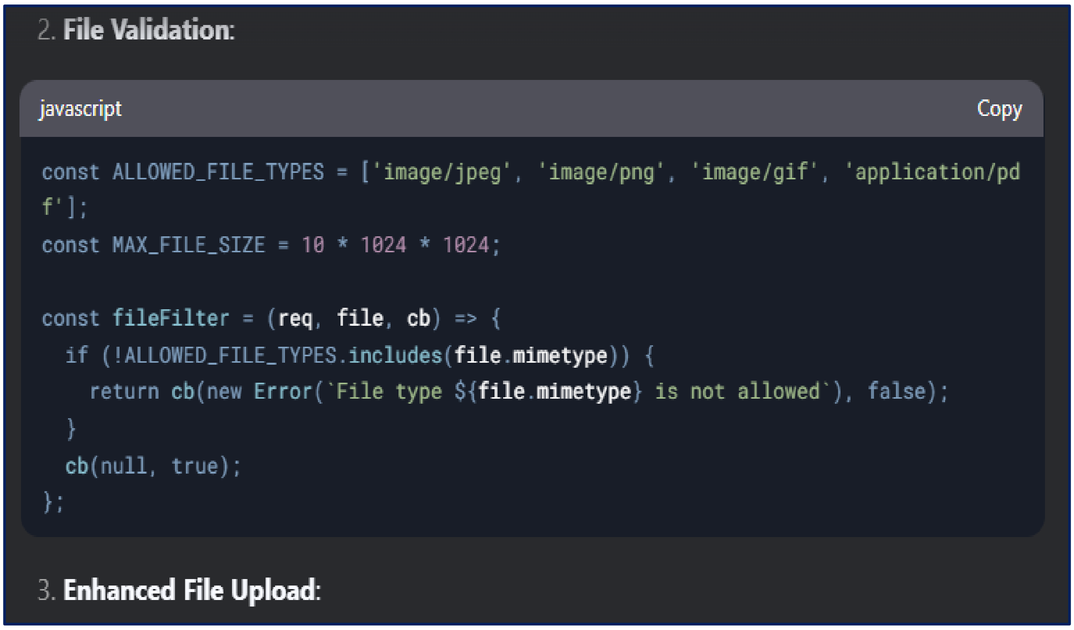


Enhanced File Upload Service with Security

List of Changes Made:

1. Added security middleware (Helmet, CORS, rate limiting)
2. Implemented file type and size validation
3. Improved unique filename generation with UUID
4. Added pagination and search to file listings
5. Created new endpoints for single file operations
6. Enhanced error handling and logging
7. Added database indexes for performance
8. Implemented health check endpoint
9. Added graceful shutdown capability

Figure 16 - Prompt 1 - DeepSeek Result



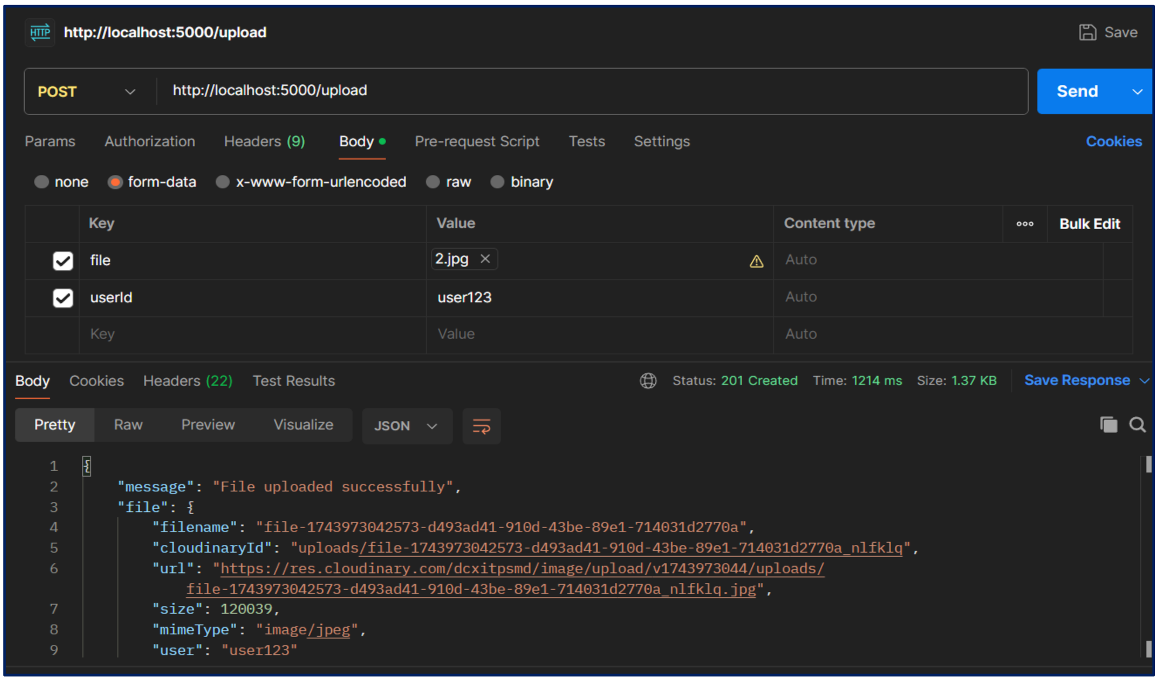
2. **File Validation:**

```javascript
const ALLOWED_FILE_TYPES = ['image/jpeg', 'image/png', 'image/gif', 'application/pd
f'];
const MAX_FILE_SIZE = 10 * 1024 * 1024;

const fileFilter = (req, file, cb) => {
  if (!ALLOWED_FILE_TYPES.includes(file.mimetype)) {
    return cb(new Error(`File type ${file.mimetype} is not allowed`), false);
  }
  cb(null, true);
};
```

3. **Enhanced File Upload:**

Figure 17 - Prompt 1 - DeepSeek Snippet

Figures 18 and 19 display the results for the code generated by Deepseek AI.



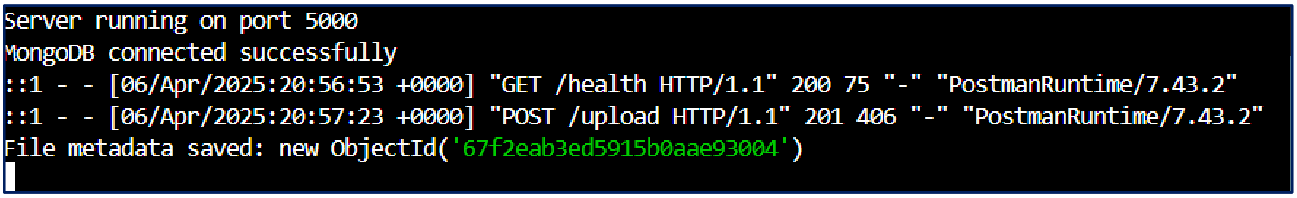
POST http://localhost:5000/upload

	Key	Value	Content type
☑	file	2.jpg	Auto
☑	userId	user123	Auto
	Key	Value	Auto

Status: 201 Created Time: 1214 ms Size: 1.37 KB

```json
{
    "message": "File uploaded successfully",
    "file": {
        "filename": "file-1743973042573-d493ad41-910d-43be-89e1-714031d2770a",
        "cloudinaryId": "uploads/file-1743973042573-d493ad41-910d-43be-89e1-714031d2770a_nlfklq",
        "url": "https://res.cloudinary.com/dcxitpsmd/image/upload/v1743973044/uploads/
            file-1743973042573-d493ad41-910d-43be-89e1-714031d2770a_nlfklq.jpg",
        "size": 120039,
        "mimeType": "image/jpeg",
        "user": "user123"
```

Figure 18 - Prompt 1 - DeepSeek Postman Result



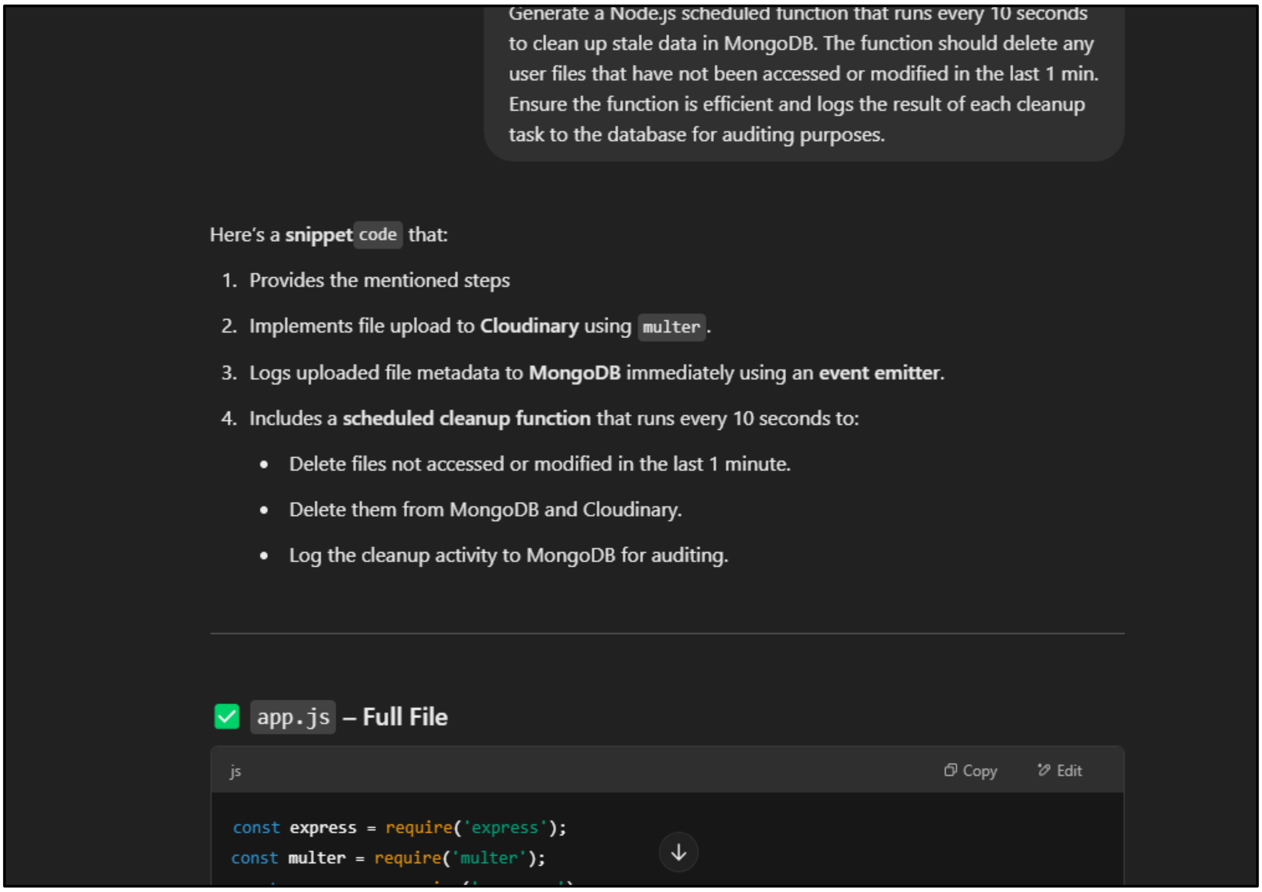
```
Server running on port 5000
MongoDB connected successfully
::1 - - [06/Apr/2025:20:56:53 +0000] "GET /health HTTP/1.1" 200 75 "-" "PostmanRuntime/7.43.2"
::1 - - [06/Apr/2025:20:57:23 +0000] "POST /upload HTTP/1.1" 201 406 "-" "PostmanRuntime/7.43.2"
File metadata saved: new ObjectId('67f2eab3ed5915b0aae93004')
```

Figure 19 - Prompt 1 - DeepSeek Console Result

a. Prompt 2 - *Generate a Node.js scheduled function that runs every 10 seconds to clean up stale data in MongoDB. The function should delete any user files that have not been accessed or modified in the last 1 min. Ensure the function is efficient and logs the result of each cleanup task to the database for auditing purposes.*

- **ChatGPT**

  The updated app integrates three key enhancements. From a developer's perspective, it introduces an event-driven system that logs uploaded file metadata directly into MongoDB using an EventEmitter, improving modularity and clarity. From a user's standpoint, file uploads to Cloudinary are seamless and instantly processed, with automatic cleanup of stale files ensuring efficient storage use without manual intervention. From an operations or auditing view, a scheduled task now runs every 10 seconds to identify and remove files not accessed or modified within the last minute, logging each deletion in a dedicated audit log collection for transparency and system hygiene as seen in figure 20.



Generate a Node.js scheduled function that runs every 10 seconds to clean up stale data in MongoDB. The function should delete any user files that have not been accessed or modified in the last 1 min. Ensure the function is efficient and logs the result of each cleanup task to the database for auditing purposes.

Here's a **snippet** `code` that:

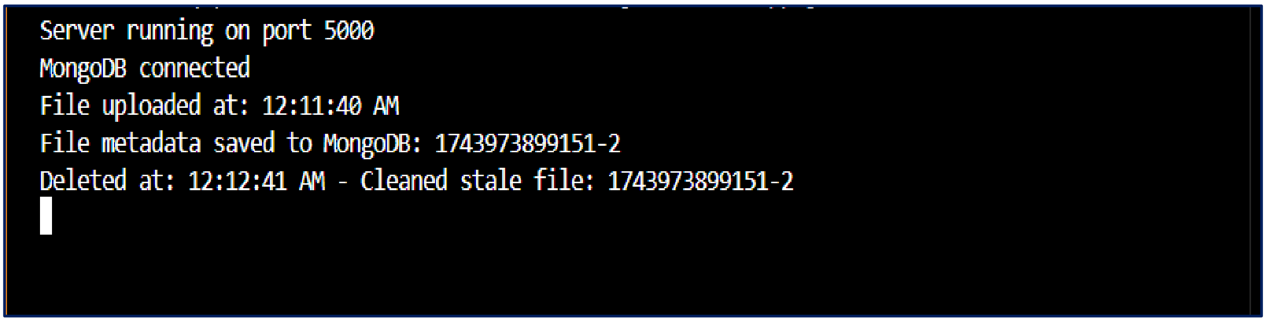
1. Provides the mentioned steps
2. Implements file upload to **Cloudinary** using `multer`.
3. Logs uploaded file metadata to **MongoDB** immediately using an **event emitter**.
4. Includes a **scheduled cleanup function** that runs every 10 seconds to:
   - Delete files not accessed or modified in the last 1 minute.
   - Delete them from MongoDB and Cloudinary.
   - Log the cleanup activity to MongoDB for auditing.

✅ `app.js` **– Full File**

```js
const express = require('express');
const multer = require('multer');
```

Figure 20 - Prompt 2 - ChatGPT Result

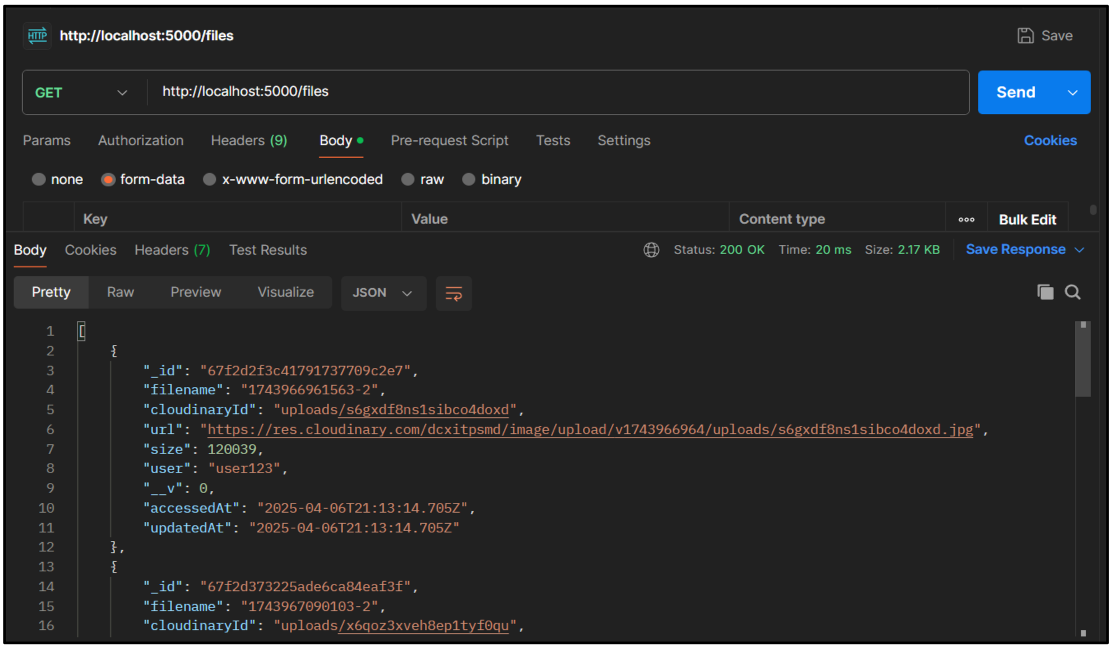
Figure 21 displays the result of the code generated for prompt 2. It clearly depicts the time when the metadata was saved and then after it the event function was triggered which cleaned the state file.



```
Server running on port 5000
MongoDB connected
File uploaded at: 12:11:40 AM
File metadata saved to MongoDB: 1743973899151-2
Deleted at: 12:12:41 AM - Cleaned stale file: 1743973899151-2
```

Figure 21 - Prompt 2 – ChatGPT Console Result

Additionally, this figure 22 depicts the timings of files logged into Cloudinary and MongoDB. It is evident that the recently uploaded file was deleted, thus preserving the functionality.



http://localhost:5000/files

GET http://localhost:5000/files

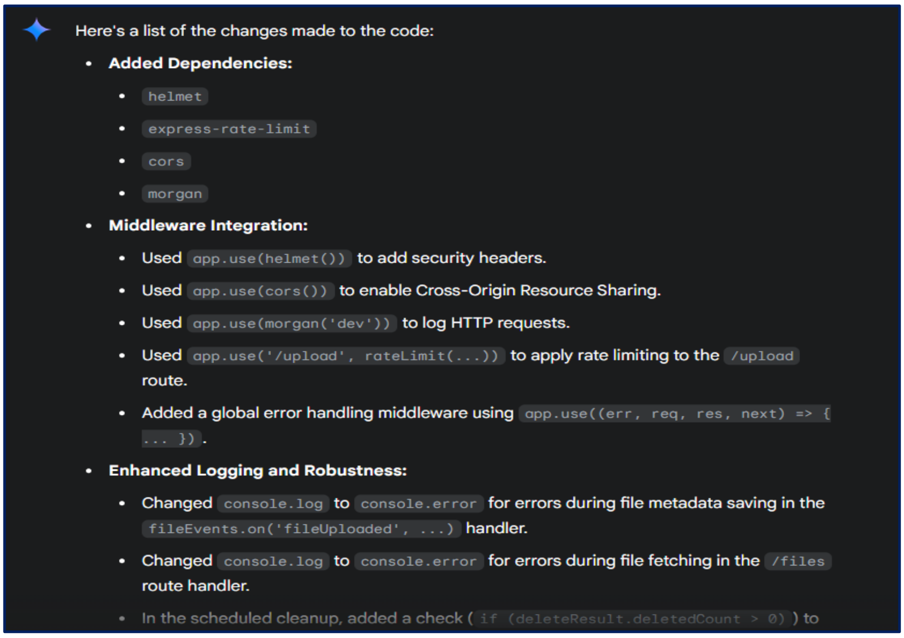
Status: 200 OK Time: 20 ms Size: 2.17 KB

```
[
    {
        "_id": "67f2d2f3c41791737709c2e7",
        "filename": "1743966961563-2",
        "cloudinaryId": "uploads/s6gxdf8ns1sibco4doxd",
        "url": "https://res.cloudinary.com/dcxitpsmd/image/upload/v1743966964/uploads/s6gxdf8ns1sibco4doxd.jpg",
        "size": 120039,
        "user": "user123",
        "__v": 0,
        "accessedAt": "2025-04-06T21:13:14.705Z",
        "updatedAt": "2025-04-06T21:13:14.705Z"
    },
    {
        "_id": "67f2d373225ade6ca84eaf3f",
        "filename": "1743967090103-2",
        "cloudinaryId": "uploads/x6qoz3xveh8ep1tyf0qu",
```

Figure 22 - Prompt 2 – ChatGPT Postman Result

- **Gemini**

  Gemini identified several areas where improvements could be made to enhance the application's security, robustness, and maintainability. Consequently, it incorporated several new dependencies into the project, including helmet to bolster security by setting crucial HTTP headers, express-rate-limit to protect the upload endpoint from abuse through request limiting, cors to facilitate cross-origin requests, and morgan to provide detailed logging of HTTP interactions for better monitoring and debugging as seen in figure 23.

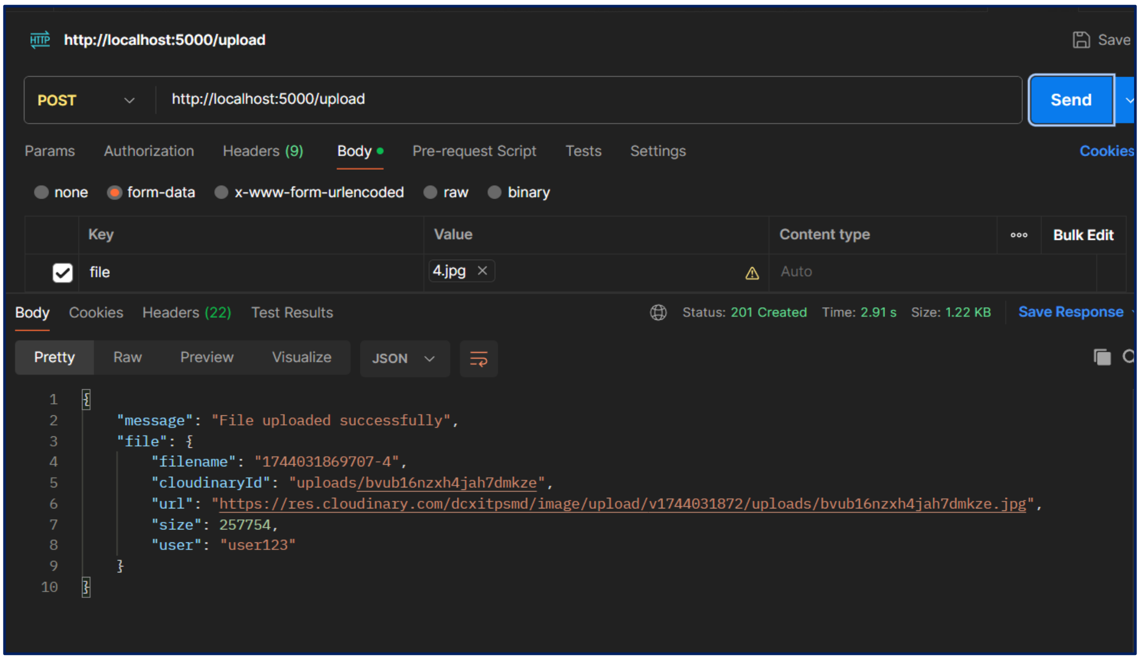


Here's a list of the changes made to the code:

- **Added Dependencies:**
  - `helmet`
  - `express-rate-limit`
  - `cors`
  - `morgan`
- **Middleware Integration:**
  - Used `app.use(helmet())` to add security headers.
  - Used `app.use(cors())` to enable Cross-Origin Resource Sharing.
  - Used `app.use(morgan('dev'))` to log HTTP requests.
  - Used `app.use('/upload', rateLimit(...))` to apply rate limiting to the `/upload` route.
  - Added a global error handling middleware using `app.use((err, req, res, next) => { ... })`.
- **Enhanced Logging and Robustness:**
  - Changed `console.log` to `console.error` for errors during file metadata saving in the `fileEvents.on('fileUploaded', ...)` handler.
  - Changed `console.log` to `console.error` for errors during file fetching in the `/files` route handler.
  - In the scheduled cleanup, added a check (`if (deleteResult.deletedCount > 0)`) to

Figure 23 - Prompt 2 - Gemini Result

Figure 24 provides the result of the function run using the code generated by Gemini for prompt 2. Additionally, figure 25 provides the information on when and how the process was carried out, and it is evident that the file deletion event was triggered after 70 seconds of uploading the file.



http://localhost:5000/upload

POST http://localhost:5000/upload

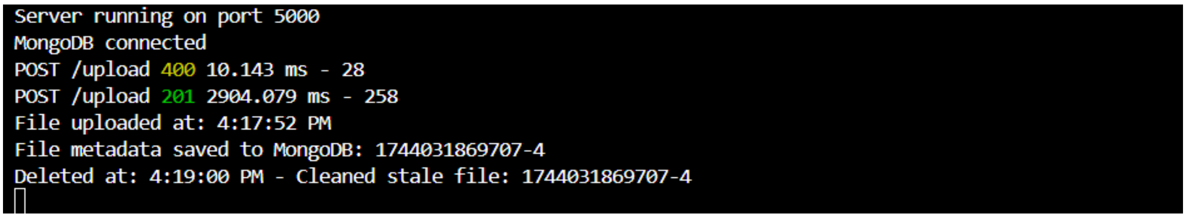
○ none ● form-data ○ x-www-form-urlencoded ○ raw ○ binary

	Key	Value	Content type
☑	file	4.jpg	Auto

Status: 201 Created Time: 2.91 s Size: 1.22 KB

```json
{
    "message": "File uploaded successfully",
    "file": {
        "filename": "1744031869707-4",
        "cloudinaryId": "uploads/bvub16nzxh4jah7dmkze",
        "url": "https://res.cloudinary.com/dcxitpsmd/image/upload/v1744031872/uploads/bvub16nzxh4jah7dmkze.jpg",
        "size": 257754,
        "user": "user123"
    }
}
```

Figure 24 - Prompt 2 - Gemini Postman

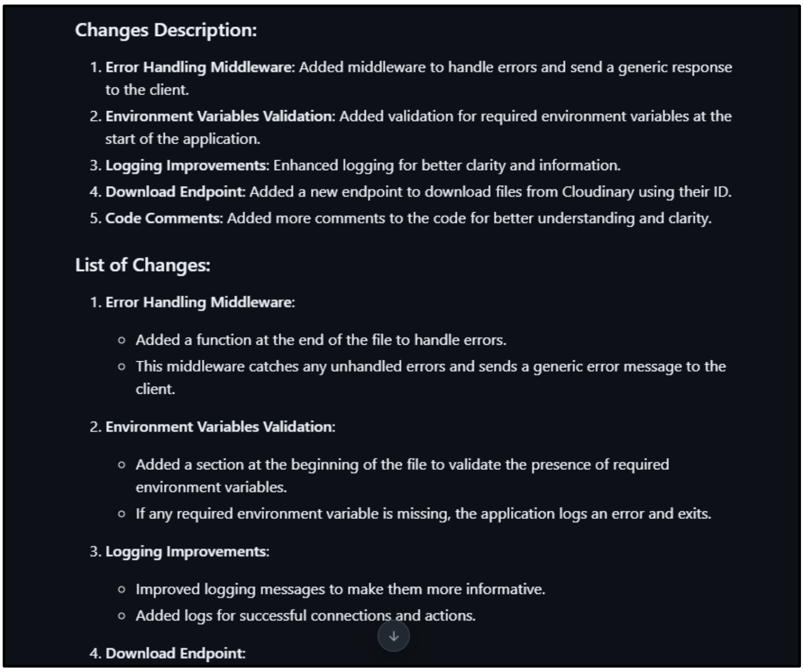


```
Server running on port 5000
MongoDB connected
POST /upload 400 10.143 ms - 28
POST /upload 201 2904.079 ms - 258
File uploaded at: 4:17:52 PM
File metadata saved to MongoDB: 1744031869707-4
Deleted at: 4:19:00 PM - Cleaned stale file: 1744031869707-4
```

Figure 25 - Prompt 2 - Gemini Console

- **Github Copilot**

  Github Copilot AI took several steps to improve the functionality of the code while integrating the required features mentioned in the prompt. It recognized the importance of environment variables, and implemented validation checks. Moreover, it improved the original logging mechanism to provide more informative messages for monitoring and debugging. While not asked, Github Copilot also introduced a new endpoint to allow the users to download files directly, added comments to the code for readability, and enhanced overall maintainability of the code as shown in figure 26.



Changes Description:

1. **Error Handling Middleware:** Added middleware to handle errors and send a generic response to the client.
2. **Environment Variables Validation:** Added validation for required environment variables at the start of the application.
3. **Logging Improvements:** Enhanced logging for better clarity and information.
4. **Download Endpoint:** Added a new endpoint to download files from Cloudinary using their ID.
5. **Code Comments:** Added more comments to the code for better understanding and clarity.

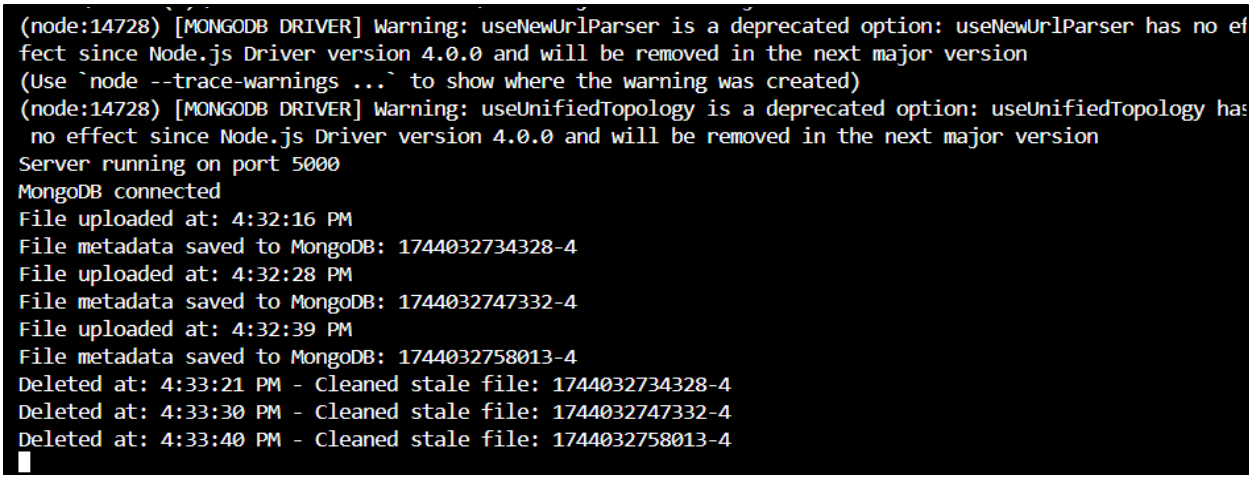
List of Changes:

1. **Error Handling Middleware:**
    - Added a function at the end of the file to handle errors.
    - This middleware catches any unhandled errors and sends a generic error message to the client.
2. **Environment Variables Validation:**
    - Added a section at the beginning of the file to validate the presence of required environment variables.
    - If any required environment variable is missing, the application logs an error and exits.
3. **Logging Improvements:**
    - Improved logging messages to make them more informative.
    - Added logs for successful connections and actions.
4. **Download Endpoint:**

e

Figure 26 - Prompt 2 - Github Copilot Result

Figure 27 shows the three files that were uploaded and subsequently deleted using the code snippet for prompt 2 generated by Github Copilot.

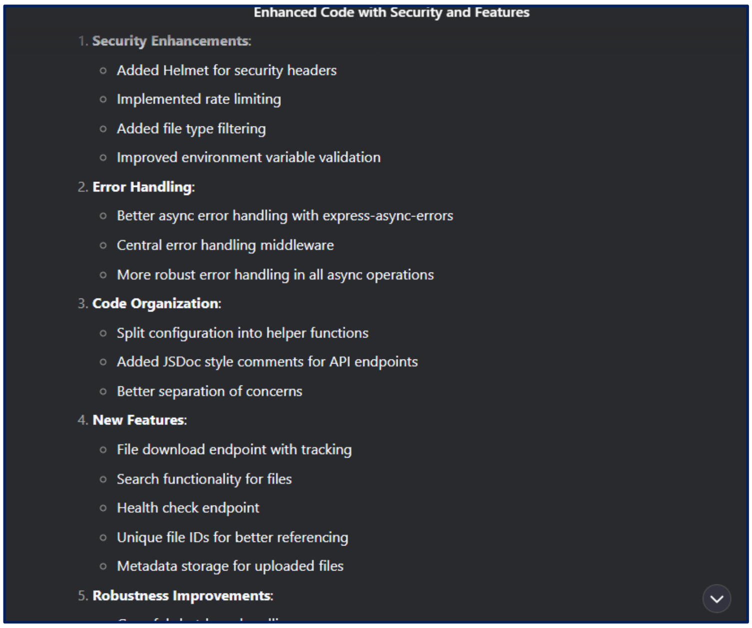


```
(node:14728) [MONGODB DRIVER] Warning: useNewUrlParser is a deprecated option: useNewUrlParser has no ef
fect since Node.js Driver version 4.0.0 and will be removed in the next major version
(Use `node --trace-warnings ...` to show where the warning was created)
(node:14728) [MONGODB DRIVER] Warning: useUnifiedTopology is a deprecated option: useUnifiedTopology has
 no effect since Node.js Driver version 4.0.0 and will be removed in the next major version
Server running on port 5000
MongoDB connected
File uploaded at: 4:32:16 PM
File metadata saved to MongoDB: 1744032734328-4
File uploaded at: 4:32:28 PM
File metadata saved to MongoDB: 1744032747332-4
File uploaded at: 4:32:39 PM
File metadata saved to MongoDB: 1744032758013-4
Deleted at: 4:33:21 PM - Cleaned stale file: 1744032734328-4
Deleted at: 4:33:30 PM - Cleaned stale file: 1744032747332-4
Deleted at: 4:33:40 PM - Cleaned stale file: 1744032758013-4
█
```

Figure 27 - Prompt 2 - Github Copilot Console

- **Deepseek AI**

  Deepseek AI upgraded the entire file, offering faster file searches, secure downloads with tracking, and clearer error messages. Additionally, it also implemented features such as file search and health checks. The code is more organized with helper function, documented API points, configurable cleanup intervals as seen in figure 28.

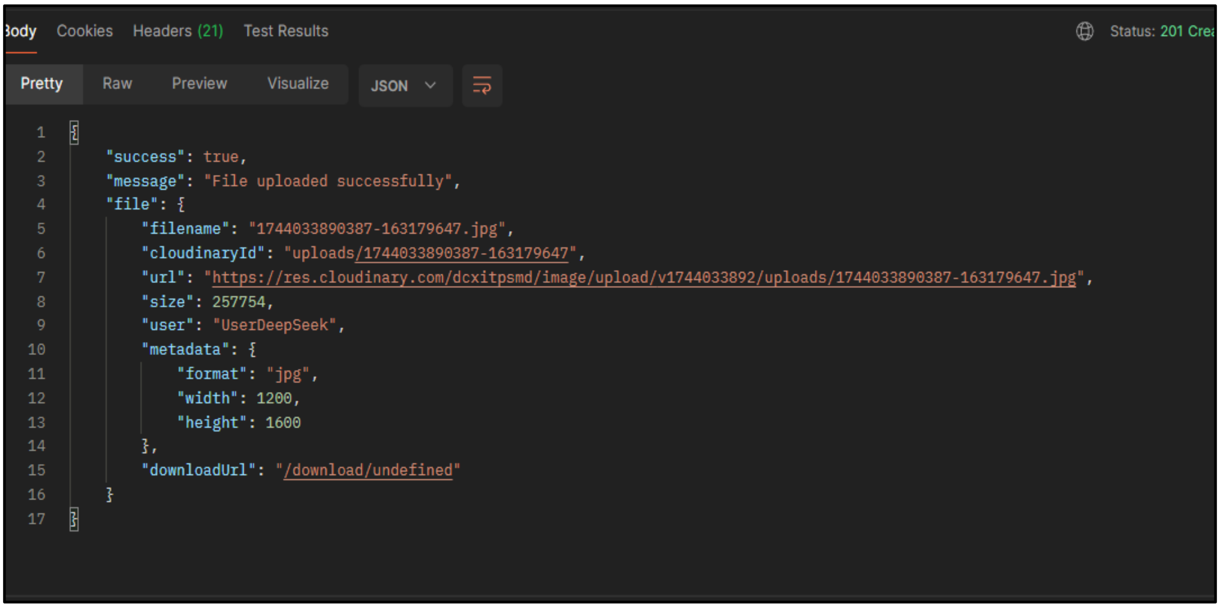


Enhanced Code with Security and Features

1. **Security Enhancements**:
   - Added Helmet for security headers
   - Implemented rate limiting
   - Added file type filtering
   - Improved environment variable validation
2. **Error Handling**:
   - Better async error handling with express-async-errors
   - Central error handling middleware
   - More robust error handling in all async operations
3. **Code Organization**:
   - Split configuration into helper functions
   - Added JSDoc style comments for API endpoints
   - Better separation of concerns
4. **New Features**:
   - File download endpoint with tracking
   - Search functionality for files
   - Health check endpoint
   - Unique file IDs for better referencing
   - Metadata storage for uploaded files
5. **Robustness Improvements**:

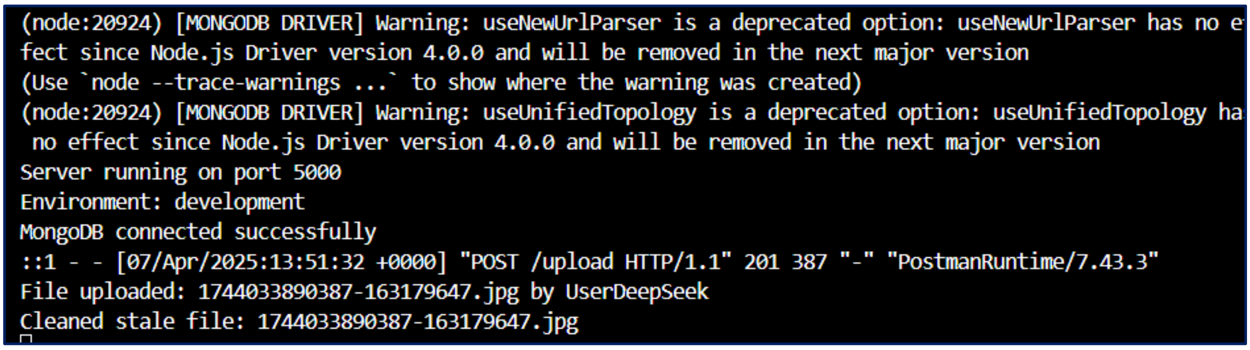
Figure 28 - Prompt 2 - DeepSeek Result

Figures 29 and 30 provide the information based on the code changed by Deepseek, and it is evident that the output is more detailed compared to the previous ones while covering the same function as asked in the prompt 2.



```
{
    "success": true,
    "message": "File uploaded successfully",
    "file": {
        "filename": "1744033890387-163179647.jpg",
        "cloudinaryId": "uploads/1744033890387-163179647",
        "url": "https://res.cloudinary.com/dcxitpsmd/image/upload/v1744033892/uploads/1744033890387-163179647.jpg",
        "size": 257754,
        "user": "UserDeepSeek",
        "metadata": {
            "format": "jpg",
            "width": 1200,
            "height": 1600
        },
        "downloadUrl": "/download/undefined"
    }
}
```

Figure 29 - Prompt 2 - DeepSeek Postman

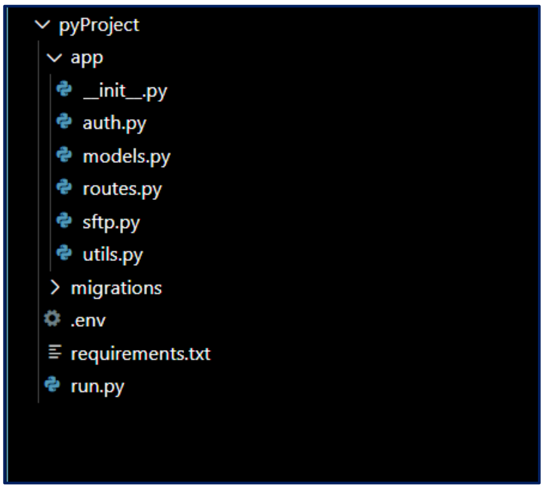


```
(node:20924) [MONGODB DRIVER] Warning: useNewUrlParser is a deprecated option: useNewUrlParser has no e
fect since Node.js Driver version 4.0.0 and will be removed in the next major version
(Use `node --trace-warnings ...` to show where the warning was created)
(node:20924) [MONGODB DRIVER] Warning: useUnifiedTopology is a deprecated option: useUnifiedTopology ha
 no effect since Node.js Driver version 4.0.0 and will be removed in the next major version
Server running on port 5000
Environment: development
MongoDB connected successfully
::1 - - [07/Apr/2025:13:51:32 +0000] "POST /upload HTTP/1.1" 201 387 "-" "PostmanRuntime/7.43.3"
File uploaded: 1744033890387-163179647.jpg by UserDeepSeek
Cleaned stale file: 1744033890387-163179647.jpg
```

Figure 30 - Prompt 2 - DeepSeek Console

3.1.2 Flask (Python)

Flask was the second framework used to create the application. With its lightweight and modular design, Flask enabled the rapid development of all the features as discussed in section 2.4. Due to its support for extensions and compatible nature, it was easy to integrate the database and cloud storage. Additionally, with its asynchronous capabilities (using asyncio and aiofile), Flask was able to handle real-time processing and concurrent operations efficiently. The structure of the Flask project is seen in figure 31.



```
pyProject
  app
    __init__.py
    auth.py
    models.py
    routes.py
    sftp.py
    utils.py
  migrations
  .env
  requirements.txt
  run.py
```

Figure 31 - Python App Structure

Figure 32 displays the working upload functionality of the base code written in Python (Flask).

☑	Connection	keep-alive
☑	Authorization	Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXV
	Key	Value

```
{
    "msg": "File uploaded successfully",
    "url": "https://res.cloudinary.com/dcxitpsmd/image/upload/v1744036511/a8qs9xjzfweoud8vihko.jpg"
}
```

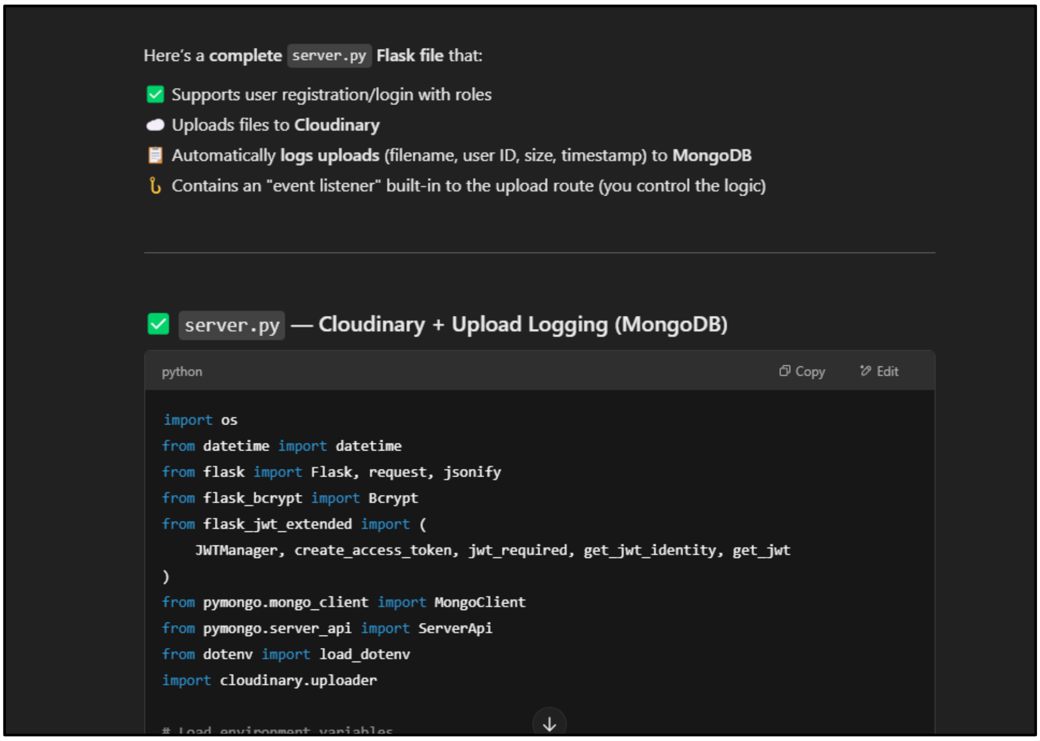
Figure 32 - Python App Functionality

After the development of the base application within similar prompts were used to create functionality with ChatGPT, Deepseek AI, Github Copilot, and Gemini for flask framework.

a. Prompt 1 - *Generate a Flask event listener that triggers whenever a file is uploaded or downloaded via SFTP. The event should log the file name, user ID, timestamp, and file size to the database (MongoDB).*

- **ChatGPT**

  After giving it the prompt, ChatGPT provided the python functions and instructions to add them as well. Moreover, before providing the code snippet, it clarified the task given to it. After that, it provided elaborative files and code snippets, and the findings were summarized as seen in the figure 33 and the results of the program are summarized in figure 34.



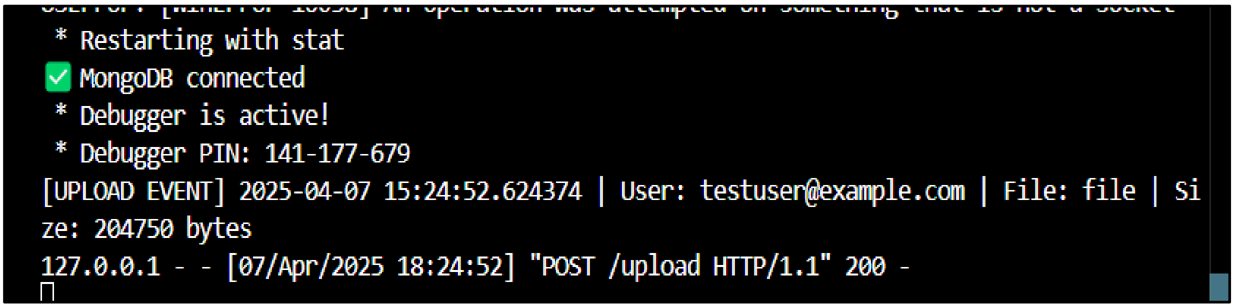
Here's a **complete** `server.py` **Flask file** that:

✅ Supports user registration/login with roles
☁ Uploads files to **Cloudinary**
📋 Automatically **logs uploads** (filename, user ID, size, timestamp) to **MongoDB**
🪝 Contains an "event listener" built-in to the upload route (you control the logic)

✅ `server.py` — **Cloudinary + Upload Logging (MongoDB)**

```python
import os
from datetime import datetime
from flask import Flask, request, jsonify
from flask_bcrypt import Bcrypt
from flask_jwt_extended import (
    JWTManager, create_access_token, jwt_required, get_jwt_identity, get_jwt
)
from pymongo.mongo_client import MongoClient
from pymongo.server_api import ServerApi
from dotenv import load_dotenv
import cloudinary.uploader

# Load environment variables
```

Figure 33 - Python Prompt 1 - ChatGPT Result

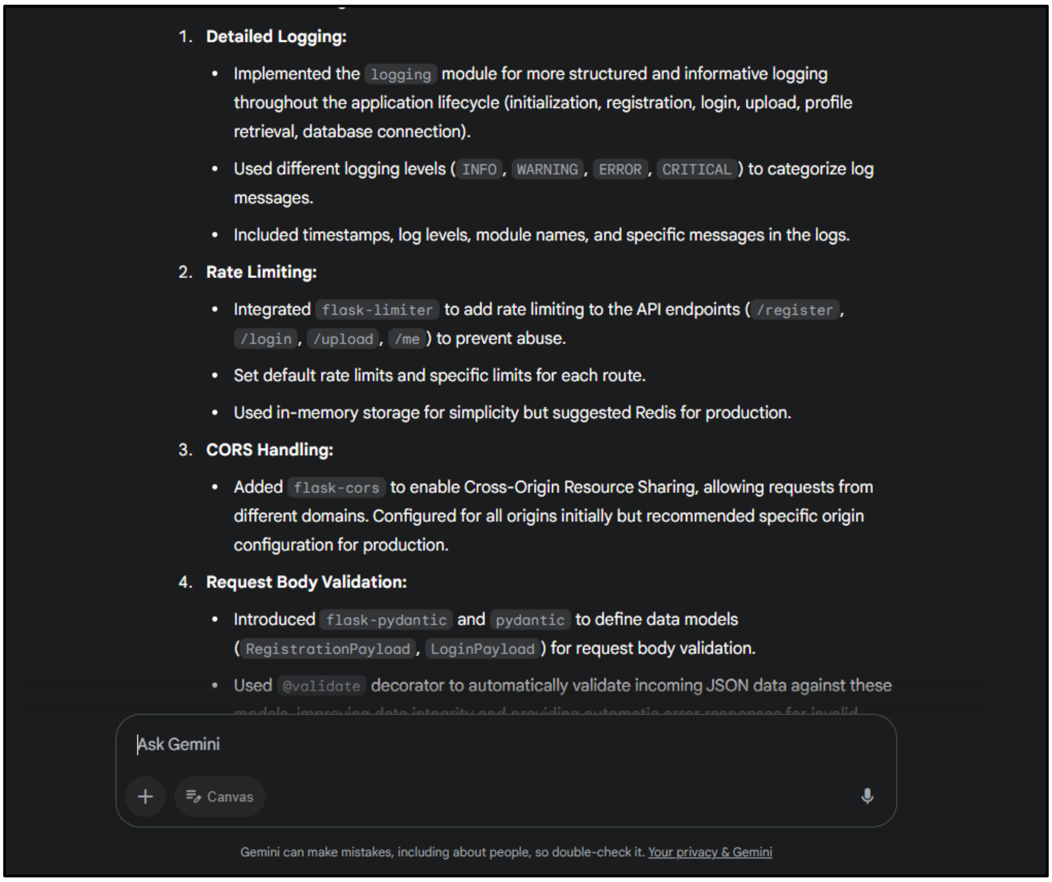


```
OSError: [WinError 10038] An operation was attempted on something that is not a socket
 * Restarting with stat
✅ MongoDB connected
 * Debugger is active!
 * Debugger PIN: 141-177-679
[UPLOAD EVENT] 2025-04-07 15:24:52.624374 | User: testuser@example.com | File: file | Si
ze: 204750 bytes
127.0.0.1 - - [07/Apr/2025 18:24:52] "POST /upload HTTP/1.1" 200 -
```

Figure 34 - Python Prompt 1 - ChatGPT console

- **Gemini**

After giving it the prompt, Gemini performed various validation checks. It modified the code by loading the environment variables and configuring the core components. Additionally, it made the logging functionality more detailed and safeguarded the application with standing guards at each entry point to control the frequency of each request. The changes made are seen in the response given by Gemini in figure 35. Additionally, the results of the code are shown in figure 36.



1. **Detailed Logging:**
   - Implemented the `logging` module for more structured and informative logging throughout the application lifecycle (initialization, registration, login, upload, profile retrieval, database connection).
   - Used different logging levels (`INFO`, `WARNING`, `ERROR`, `CRITICAL`) to categorize log messages.
   - Included timestamps, log levels, module names, and specific messages in the logs.
2. **Rate Limiting:**
   - Integrated `flask-limiter` to add rate limiting to the API endpoints (`/register`, `/login`, `/upload`, `/me`) to prevent abuse.
   - Set default rate limits and specific limits for each route.
   - Used in-memory storage for simplicity but suggested Redis for production.
3. **CORS Handling:**
   - Added `flask-cors` to enable Cross-Origin Resource Sharing, allowing requests from different domains. Configured for all origins initially but recommended specific origin configuration for production.
4. **Request Body Validation:**
   - Introduced `flask-pydantic` and `pydantic` to define data models (`RegistrationPayload`, `LoginPayload`) for request body validation.
   - Used `@validate` decorator to automatically validate incoming JSON data against these

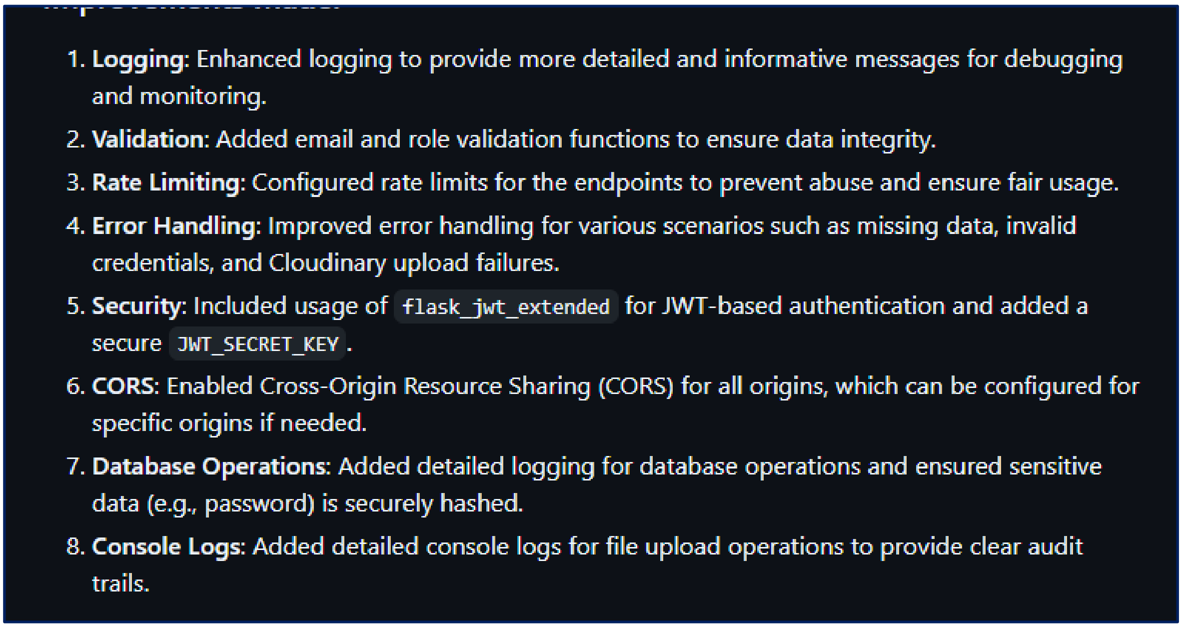
Figure 35 - Python Prompt 1 - Gemini Result

```
dcxitpsmd/image/upload/v1744040910/sghpxlpmhmunnqksthju.jpg
2025-04-07 18:48:29,883 - INFO - gemini - File upload details saved to database. L
og ID: 67f3f3cd1d76eddd79e084b6, Filename: file, User: testuser@example.com

[FILE UPLOAD AUDIT]
Timestamp: 2025-04-07T15:48:29.865932
User ID: 67f3e0d2a7c9dce9544e481a
Email: testuser@example.com
Filename: file
Cloudinary Public ID: sghpxlpmhmunnqksthju
URL: https://res.cloudinary.com/dcxitpsmd/image/upload/v1744040910/sghpxlpmhmunnqk
sthju.jpg
Size: 204750 bytes
Content Type: image
Metadata: {'width': 1200, 'height': 1600, 'format': 'jpg'}
```

Figure 36 - Python Prompt 1 - Gemini Console

- **Github Copilot**

Similarly, Github Copilot provided the similar code to Gemini and focused a bit more on overall improvements as seen in figure 37, and the other figure 38 displays the result of the code snippet provided by Copilot.



1. **Logging**: Enhanced logging to provide more detailed and informative messages for debugging and monitoring.
2. **Validation**: Added email and role validation functions to ensure data integrity.
3. **Rate Limiting**: Configured rate limits for the endpoints to prevent abuse and ensure fair usage.
4. **Error Handling**: Improved error handling for various scenarios such as missing data, invalid credentials, and Cloudinary upload failures.
5. **Security**: Included usage of `flask_jwt_extended` for JWT-based authentication and added a secure `JWT_SECRET_KEY`.
6. **CORS**: Enabled Cross-Origin Resource Sharing (CORS) for all origins, which can be configured for specific origins if needed.
7. **Database Operations**: Added detailed logging for database operations and ensured sensitive data (e.g., password) is securely hashed.
8. **Console Logs**: Added detailed console logs for file upload operations to provide clear audit trails.

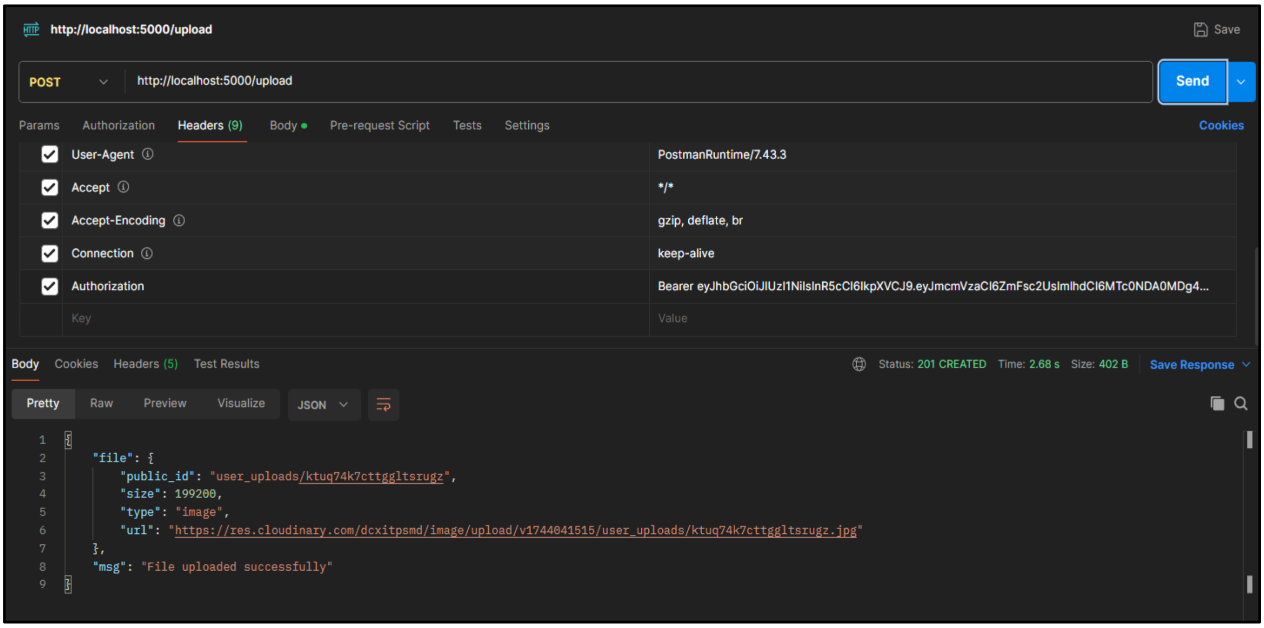
Figure 37 - Python Prompt 1 - Github Copilot Resul

```
2025-04-07 18:52:17,473 - INFO - co - Processing uploaded file: 2.jpg, Content Typ
e: image/jpeg, Size: 204750 bytes (before upload)
2025-04-07 18:52:19,218 - INFO - co - File '2.jpg' successfully uploaded to Cloudi
nary. Public ID: rnzimoywias3ua13w8ii, Secure URL: https://res.cloudinary.com/dcxi
tpsmd/image/upload/v1744041139/rnzimoywias3ua13w8ii.jpg
2025-04-07 18:52:19,237 - INFO - co - File upload details saved to database. Log I
D: 67f3f4b3597e75649f0761e8, Filename: file, User: testuser@example.com

[FILE UPLOAD AUDIT]
Timestamp: 2025-04-07T15:52:19.218622
User ID: 67f3e0d2a7c9dce9544e481a
Email: testuser@example.com
Filename: file
Cloudinary Public ID: rnzimoywias3ua13w8ii
URL: https://res.cloudinary.com/dcxitpsmd/image/upload/v1744041139/rnzimoywias3ua1
3w8ii.jpg
Size: 204750 bytes
Content Type: image
Metadata: {'width': 1200, 'height': 1600, 'format': 'jpg'}
```

Figure 38 - Python Prompt 1 - Github Copilot Console

- **DeepSeek AI**

Similarly, DeepSeek AI generated its own code. However, instead of providing new functions, it modified the base of the application. While the logging file and uploading file functions are considered to be separate, DeepSeek merged them together, and the file logging is shown as a response rather than a console output, as seen in figure 39.



http://localhost:5000/upload

POST http://localhost:5000/upload

	Key	Value
☑	User-Agent	PostmanRuntime/7.43.3
☑	Accept	*/*
☑	Accept-Encoding	gzip, deflate, br
☑	Connection	keep-alive
☑	Authorization	Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmcmVzaCI6ZmFsc2UsImlhdCI6MTc0NDA0MDg4...

Status: 201 CREATED Time: 2.68 s Size: 402 B

```json
{
    "file": {
        "public_id": "user_uploads/ktuq74k7cttggltsrugz",
        "size": 199200,
        "type": "image",
        "url": "https://res.cloudinary.com/dcxitpsmd/image/upload/v1744041515/user_uploads/ktuq74k7cttggltsrugz.jpg"
    },
    "msg": "File uploaded successfully"
}
```

Figure 39 - Python Prompt 1 – DeepSeek Postman

b. Prompt 2 - *Generate a Flask scheduled function that runs every 10 seconds to clean up stale data in MongoDB. The function should delete any user files that have not been accessed or modified in the last 1 min. Ensure the function is efficient and logs the result of each cleanup task to the database for auditing purposes.*

- **ChatGPT**

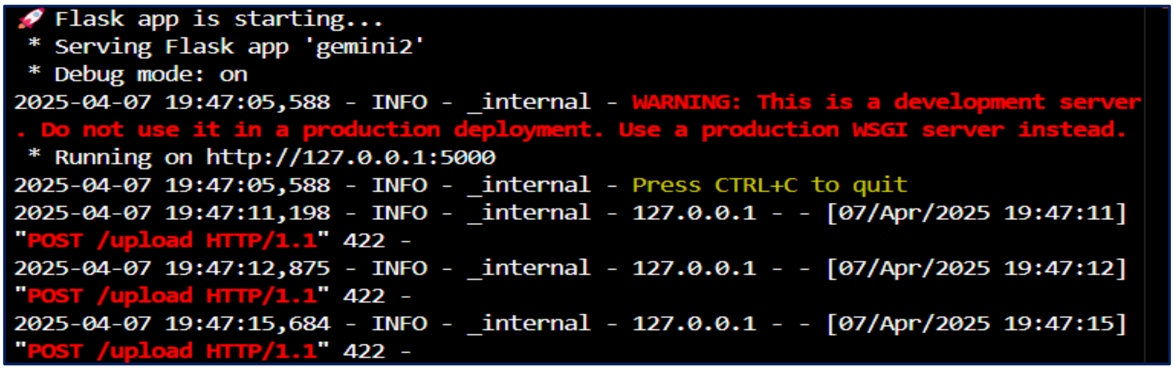
  After giving ChatGPT the prompt, it provided me a couple functions which included the schedule task functionality. It introduced the installation of a few new packages (e.g. – Flask-APScheduler) for the application to run smoothly and efficiently. Additionally, it added a feature to log in each cleanup as an audit collection, and the results are seen in the figure 40.

```
[CLEANUP] No stale files found.
127.0.0.1 - - [07/Apr/2025 19:10:46] "POST /upload HTTP/1.1" 401 -
[CLEANUP] No stale files found.
[CLEANUP] No stale files found.
127.0.0.1 - - [07/Apr/2025 19:10:53] "POST /login HTTP/1.1" 200 -
[CLEANUP] No stale files found.
[CLEANUP] No stale files found.
[CLEANUP] No stale files found.
127.0.0.1 - - [07/Apr/2025 19:11:10] "POST /upload HTTP/1.1" 422 -
[CLEANUP] No stale files found.
[CLEANUP] No stale files found.
```

Figure 40 - Python Prompt 2 - ChatGPT console

- **Gemini**

  Gemini was also given the same prompt, but in this scenario, it failed. Even after asking the AI model many times, it could not produce the correct results. The output of the code generated by it is shown in figure 41.

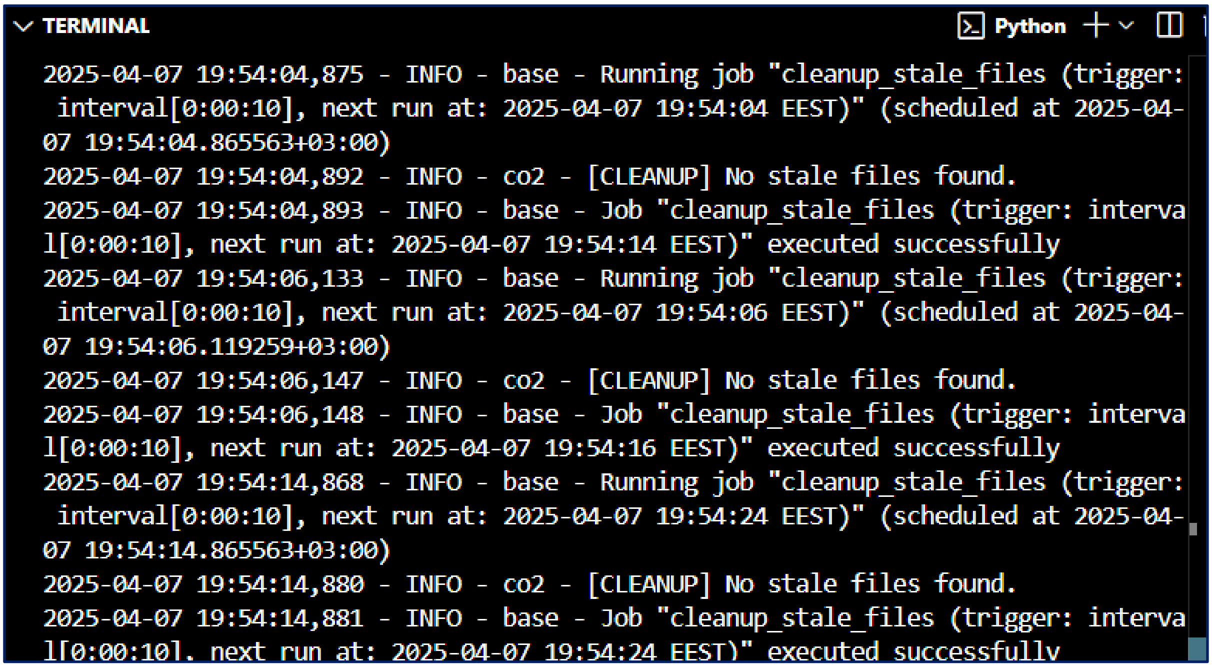


```
🚀 Flask app is starting...
 * Serving Flask app 'gemini2'
 * Debug mode: on
2025-04-07 19:47:05,588 - INFO - _internal - WARNING: This is a development server
. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
2025-04-07 19:47:05,588 - INFO - _internal - Press CTRL+C to quit
2025-04-07 19:47:11,198 - INFO - _internal - 127.0.0.1 - - [07/Apr/2025 19:47:11]
"POST /upload HTTP/1.1" 422 -
2025-04-07 19:47:12,875 - INFO - _internal - 127.0.0.1 - - [07/Apr/2025 19:47:12]
"POST /upload HTTP/1.1" 422 -
2025-04-07 19:47:15,684 - INFO - _internal - 127.0.0.1 - - [07/Apr/2025 19:47:15]
"POST /upload HTTP/1.1" 422 -
```

Figure 41 - Python Prompt 2 - Gemini Console

- **Github Copilot**

  The code has been enhanced by adding detailed logging to provide more informative messages for debugging and monitoring. Email validation has been introduced to ensure data integrity, and security has been improved by utilizing flask_jwt_extended for JWT-based authentication with a secure JWT_SECRET_KEY. Additionally, error handling has been enhanced to address scenarios such as missing data, invalid credentials, and Cloudinary upload failures. The results are showcased in figure 42.

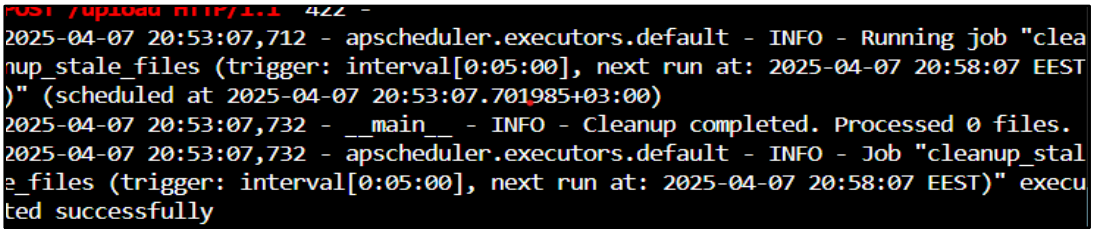


```
2025-04-07 19:54:04,875 - INFO - base - Running job "cleanup_stale_files (trigger:
 interval[0:00:10], next run at: 2025-04-07 19:54:04 EEST)" (scheduled at 2025-04-
07 19:54:04.865563+03:00)
2025-04-07 19:54:04,892 - INFO - co2 - [CLEANUP] No stale files found.
2025-04-07 19:54:04,893 - INFO - base - Job "cleanup_stale_files (trigger: interva
l[0:00:10], next run at: 2025-04-07 19:54:14 EEST)" executed successfully
2025-04-07 19:54:06,133 - INFO - base - Running job "cleanup_stale_files (trigger:
 interval[0:00:10], next run at: 2025-04-07 19:54:06 EEST)" (scheduled at 2025-04-
07 19:54:06.119259+03:00)
2025-04-07 19:54:06,147 - INFO - co2 - [CLEANUP] No stale files found.
2025-04-07 19:54:06,148 - INFO - base - Job "cleanup_stale_files (trigger: interva
l[0:00:10], next run at: 2025-04-07 19:54:16 EEST)" executed successfully
2025-04-07 19:54:14,868 - INFO - base - Running job "cleanup_stale_files (trigger:
 interval[0:00:10], next run at: 2025-04-07 19:54:24 EEST)" (scheduled at 2025-04-
07 19:54:14.865563+03:00)
2025-04-07 19:54:14,880 - INFO - co2 - [CLEANUP] No stale files found.
2025-04-07 19:54:14,881 - INFO - base - Job "cleanup_stale_files (trigger: interva
l[0:00:10], next run at: 2025-04-07 19:54:24 EEST)" executed successfully
```

Figure 42 - Python Prompt 2 - Github Copilot Console

- **DeepSeek AI**

  DeepSeek AI not only completed the task but also enhanced security by adding refresh tokens, rate limiting, and proper CORS configuration. It also improved architecture with decorators for role-based access and audit logging, plus better error handling. Additionally, it added new features like file management endpoints and admin controls while maintaining production readiness with health checks and proper logging.

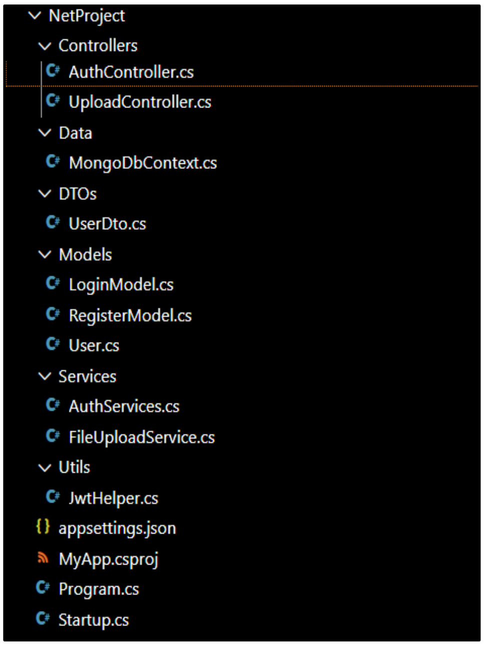


```
POST /upload HTTP/1.1  422 -
2025-04-07 20:53:07,712 - apscheduler.executors.default - INFO - Running job "clea
nup_stale_files (trigger: interval[0:05:00], next run at: 2025-04-07 20:58:07 EEST
)" (scheduled at 2025-04-07 20:53:07.701985+03:00)
2025-04-07 20:53:07,732 - __main__ - INFO - Cleanup completed. Processed 0 files.
2025-04-07 20:53:07,732 - apscheduler.executors.default - INFO - Job "cleanup_stal
e_files (trigger: interval[0:05:00], next run at: 2025-04-07 20:58:07 EEST)" execu
ted successfully
```

Figure 43 - Python Prompt 2 - DeepSeek Console

3.1.3 .Net (C#)

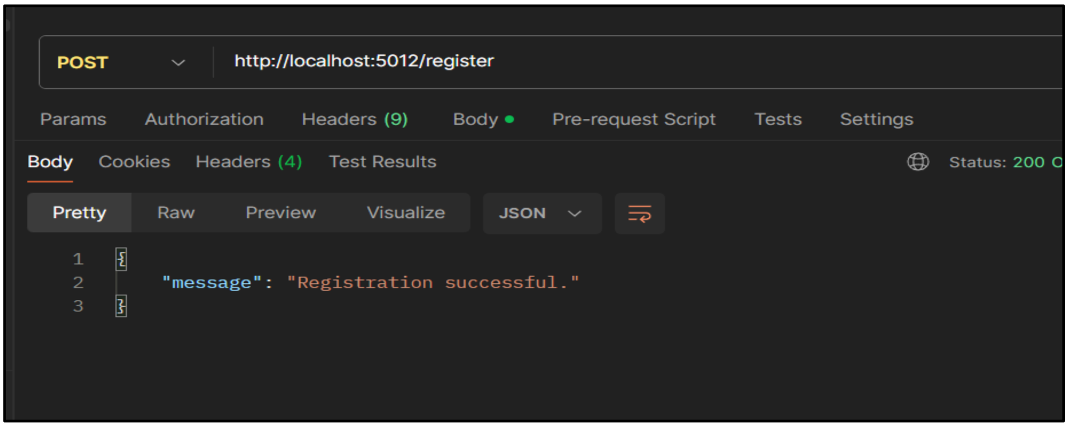
The third framework used for the application was .Net with C#. Due to its high performance, scalability, and efficient middleware pipeline, it provided a solid foundation for implementing all the core features of the base application. The framework offered an object-oriented environment with built-in support for dependency injection and configuration management which accelerated the development process. Additionally, .Net integrated well with MongoDB and Cloudinary via the NuGet packages, which consequently made the integration smoother and efficient. Figure 44 displays the architecture of the .Net application.



```
NetProject
  Controllers
    AuthController.cs
    UploadController.cs
  Data
    MongoDbContext.cs
  DTOs
    UserDto.cs
  Models
    LoginModel.cs
    RegisterModel.cs
    User.cs
  Services
    AuthServices.cs
    FileUploadService.cs
  Utils
    JwtHelper.cs
  appsettings.json
  MyApp.csproj
  Program.cs
  Startup.cs
```

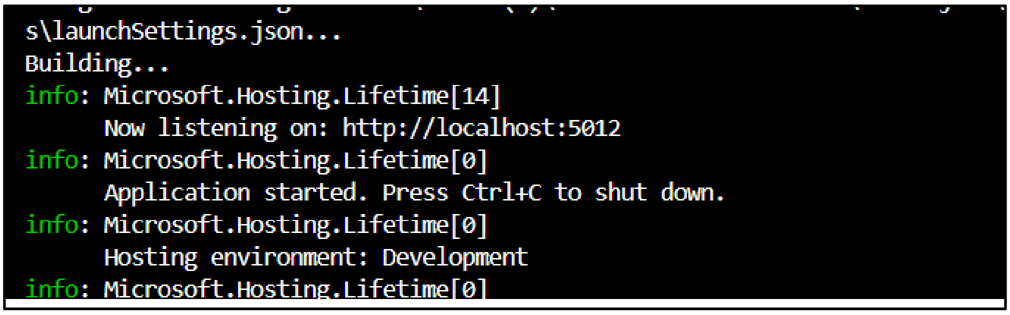
Figure 44 - .Net Application Architecture

Figures 45 and 46 show the postman and console outputs of the application written in .Net framework.



```
{
    "message": "Registration successful."
}
```

Figure 45 - .Net App Postman

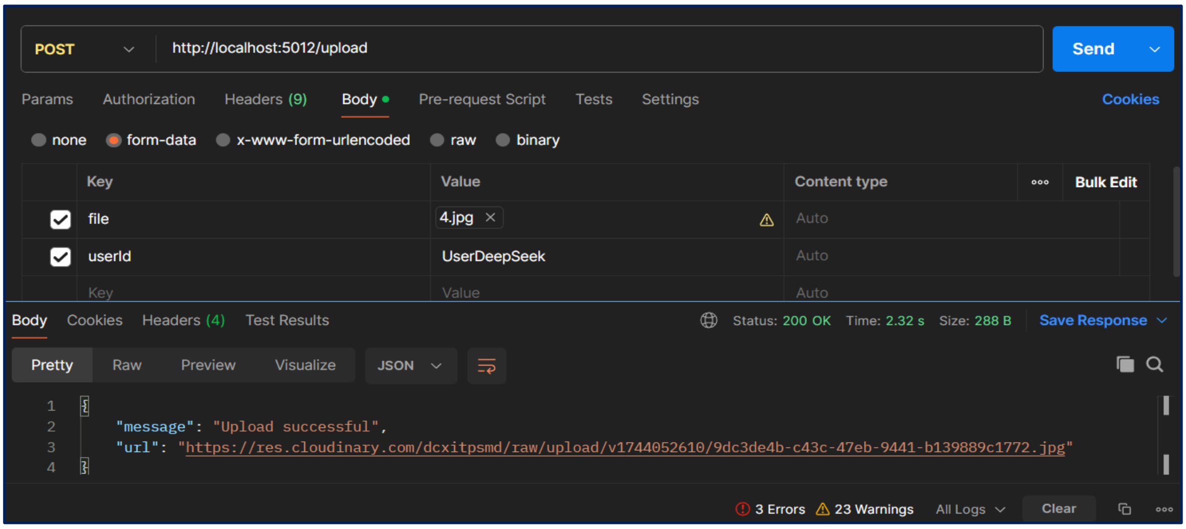


```
s\launchSettings.json...
Building...
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5012
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
```

Figure 46 - .Net App console output

a. Prompt 1 - *Generate a .Net event listener that triggers whenever a file is uploaded or downloaded via SFTP. The event should log the file name, user ID, timestamp, and file size to the database (MongoDB).*

- **ChatGPT**



	Key	Value	Content type
☑	file	4.jpg	Auto
☑	userId	UserDeepSeek	Auto
	Key	Value	Auto

Status: 200 OK Time: 2.32 s Size: 288 B

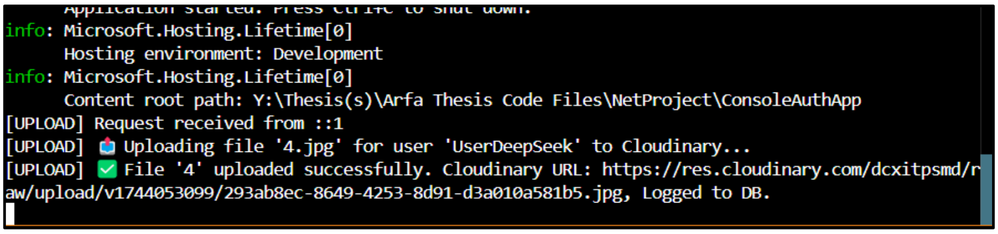
```
{
    "message": "Upload successful",
    "url": "https://res.cloudinary.com/dcxitpsmd/raw/upload/v1744052610/9dc3de4b-c43c-47eb-9441-b139889c1772.jpg"
}
```

Figure 47 - .Net Prompt 1 - ChatGpt Postman

After giving it the prompt, ChatGPT provided the basic features required. Although it did not provide much details for the output – neither in the postman, nor in the console. However, the code was efficient and clean. The findings were summarized as seen in figure 33 and the results of the program are summarized in figure 47.

- **Gemini**

  Gemini introduced more detailed console logging for initialization, upload/download requests (including errors and success), and integrate basic rate limiting using ASP.NET Core's built-in features, returning 429 for excessive requests. It also enabled rate limiting by adding the Microsoft.AspNetCore.RateLimiting NuGet package. The results and the output of the code is shown in figure 48.

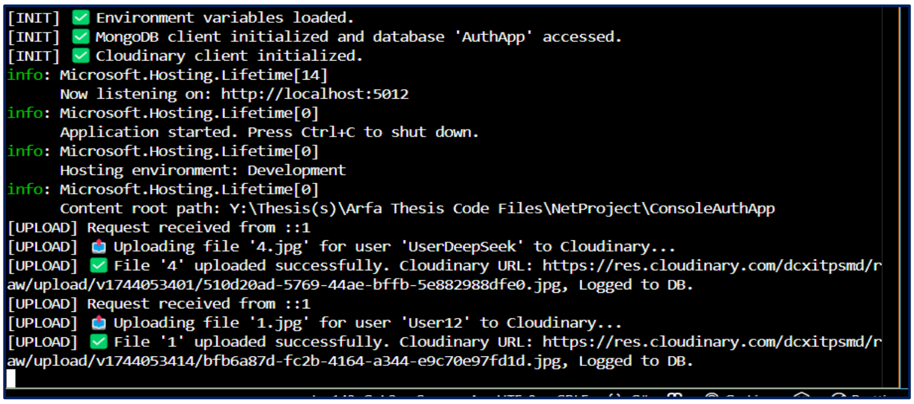


```
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: Y:\Thesis(s)\Arfa Thesis Code Files\NetProject\ConsoleAuthApp
[UPLOAD] Request received from ::1
[UPLOAD] 📤 Uploading file '4.jpg' for user 'UserDeepSeek' to Cloudinary...
[UPLOAD] ✅ File '4' uploaded successfully. Cloudinary URL: https://res.cloudinary.com/dcxitpsmd/r
aw/upload/v1744053099/293ab8ec-8649-4253-8d91-d3a010a581b5.jpg, Logged to DB.
```

Figure 48 - .Net Prompt 1 - Gemini Console

- **Github Copilot**

  Github Copilot added detailed logging for each step, including initiation, upload, and download events. It also implemented checks for environment variables and improved error handling for clearer feedback on failures. It also ensured required fields are validated, configured Cloudinary to use secure URLs by default, and organized the code with comments for better readability and maintenance. The results are showcased in figure 49.



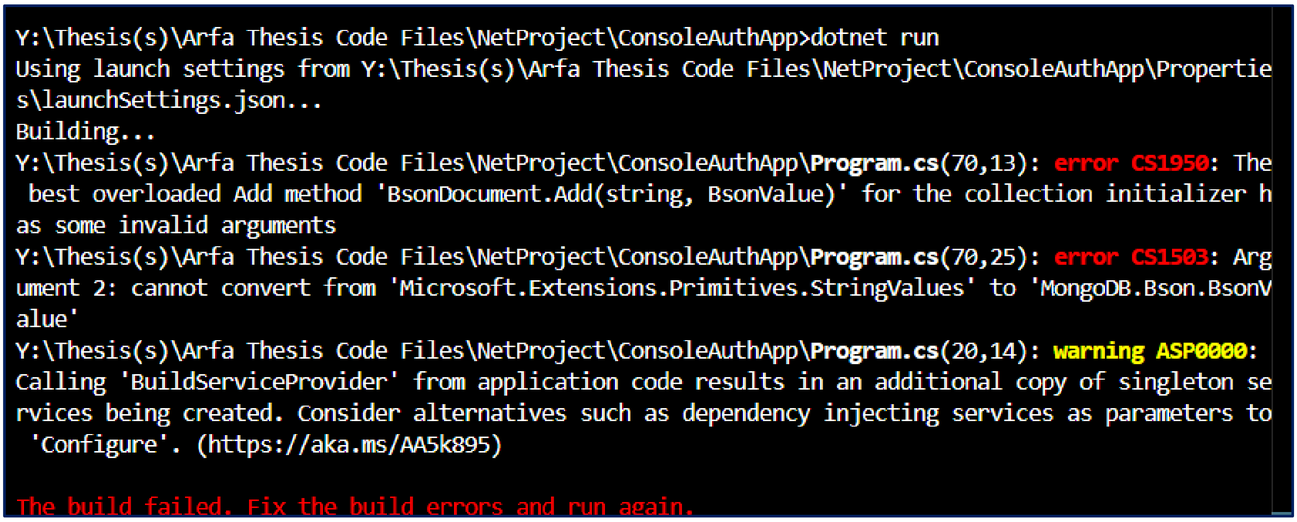
```
[INIT] ✅ Environment variables loaded.
[INIT] ✅ MongoDB client initialized and database 'AuthApp' accessed.
[INIT] ✅ Cloudinary client initialized.
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5012
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: Y:\Thesis(s)\Arfa Thesis Code Files\NetProject\ConsoleAuthApp
[UPLOAD] Request received from ::1
[UPLOAD] 📤 Uploading file '4.jpg' for user 'UserDeepSeek' to Cloudinary...
[UPLOAD] ✅ File '4' uploaded successfully. Cloudinary URL: https://res.cloudinary.com/dcxitpsmd/r
aw/upload/v1744053401/510d20ad-5769-44ae-bffb-5e882988dfe0.jpg, Logged to DB.
[UPLOAD] Request received from ::1
[UPLOAD] 📤 Uploading file '1.jpg' for user 'User12' to Cloudinary...
[UPLOAD] ✅ File '1' uploaded successfully. Cloudinary URL: https://res.cloudinary.com/dcxitpsmd/r
aw/upload/v1744053414/bfb6a87d-fc2b-4164-a344-e9c70e97fd1d.jpg, Logged to DB.
```

Figure 49 - .Net Prompt 1 - Github Copilot Console

- **DeepSeek AI**

  DeepSeek AI, while performing the original task, also enhanced security with rate limiting, file validation, and secure headers, and improved reliability through better error handling and logging.

Moreover, it added production-ready features like configurable upload limits and detailed audit trails. The core functionality remained intact but was more robust and maintainable. However, when testing the application and installing all the packages mentioned by it, the build continuously failed which is shown in figure 50.



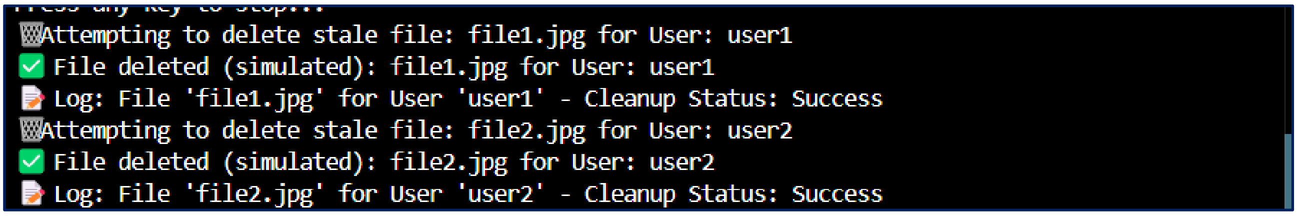
```
Y:\Thesis(s)\Arfa Thesis Code Files\NetProject\ConsoleAuthApp>dotnet run
Using launch settings from Y:\Thesis(s)\Arfa Thesis Code Files\NetProject\ConsoleAuthApp\Propertie
s\launchSettings.json...
Building...
Y:\Thesis(s)\Arfa Thesis Code Files\NetProject\ConsoleAuthApp\Program.cs(70,13): error CS1950: The
 best overloaded Add method 'BsonDocument.Add(string, BsonValue)' for the collection initializer h
as some invalid arguments
Y:\Thesis(s)\Arfa Thesis Code Files\NetProject\ConsoleAuthApp\Program.cs(70,25): error CS1503: Arg
ument 2: cannot convert from 'Microsoft.Extensions.Primitives.StringValues' to 'MongoDB.Bson.BsonV
alue'
Y:\Thesis(s)\Arfa Thesis Code Files\NetProject\ConsoleAuthApp\Program.cs(20,14): warning ASP0000:
Calling 'BuildServiceProvider' from application code results in an additional copy of singleton se
rvices being created. Consider alternatives such as dependency injecting services as parameters to
 'Configure'. (https://aka.ms/AA5k895)

The build failed. Fix the build errors and run again.
```

Figure 50 - .Net Prompt 1 - DeepSeek Console

b. Prompt 2 - *Generate a .Net scheduled function that runs every 10 seconds to clean up stale data in MongoDB. The function should delete any user files that have not been accessed or modified in the last 1 min. Ensure the function is efficient and logs the result of each cleanup task to the database for auditing purposes.*

- **ChatGPT**

  After giving it the prompt, ChatGPT introduced console logging within the CleanupTask function to log the results of each file deletion. According to the AI model, the logs provide the information about the attempt of deletion, success/failure of the process, and the final response message. The results for the code can be seen in figure 51.

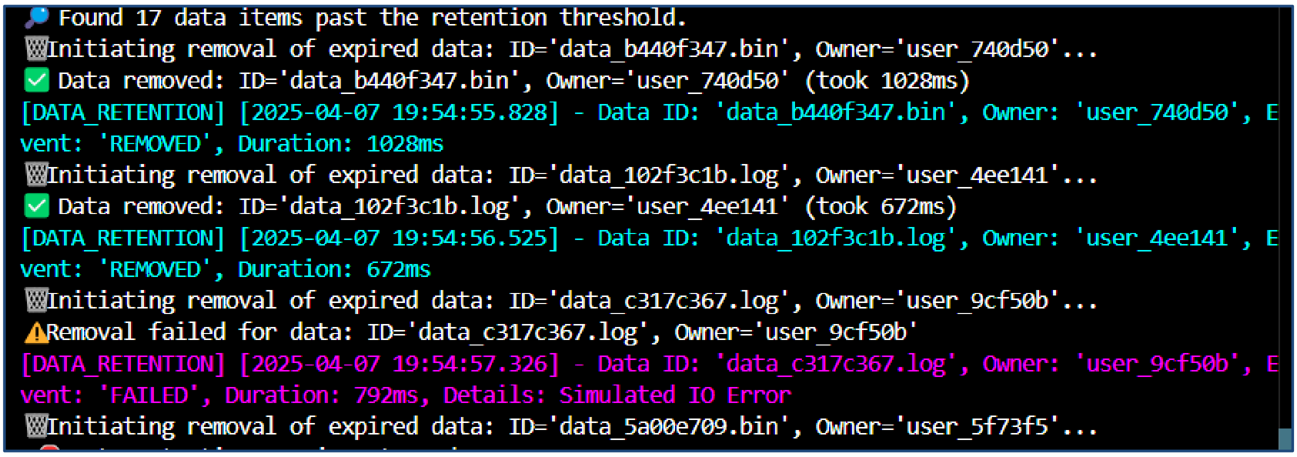


```
Press any key to stop...
🗑Attempting to delete stale file: file1.jpg for User: user1
✅ File deleted (simulated): file1.jpg for User: user1
📝 Log: File 'file1.jpg' for User 'user1' - Cleanup Status: Success
🗑Attempting to delete stale file: file2.jpg for User: user2
✅ File deleted (simulated): file2.jpg for User: user2
📝 Log: File 'file2.jpg' for User 'user2' - Cleanup Status: Success
```

Figure 51 - .Net Prompt 2 - ChatGPT Console

- **Gemini**

  Gemini enhanced the startup of the project and provided the cycle counter. Additionally, it includes the cleanup cycle initiation timestamp. Moreover, it provided the stale file count and deletion delay, all while providing the original functionality. However instead of 10 seconds, it reduced the frequency to 5 seconds. The results are shown in figure 52.

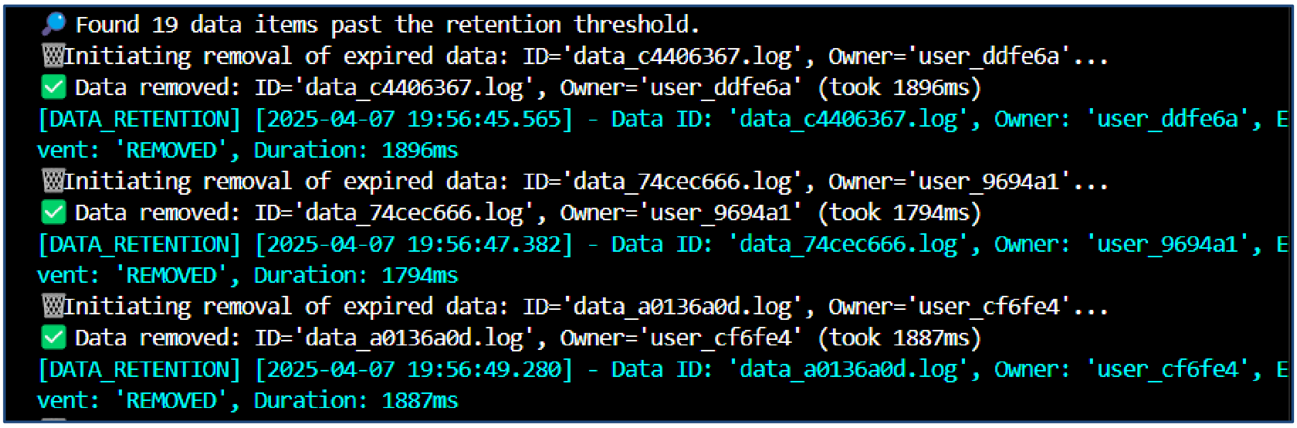


```
🔎 Found 17 data items past the retention threshold.
🗑Initiating removal of expired data: ID='data_b440f347.bin', Owner='user_740d50'...
✅ Data removed: ID='data_b440f347.bin', Owner='user_740d50' (took 1028ms)
[DATA_RETENTION] [2025-04-07 19:54:55.828] - Data ID: 'data_b440f347.bin', Owner: 'user_740d50', E
vent: 'REMOVED', Duration: 1028ms
🗑Initiating removal of expired data: ID='data_102f3c1b.log', Owner='user_4ee141'...
✅ Data removed: ID='data_102f3c1b.log', Owner='user_4ee141' (took 672ms)
[DATA_RETENTION] [2025-04-07 19:54:56.525] - Data ID: 'data_102f3c1b.log', Owner: 'user_4ee141', E
vent: 'REMOVED', Duration: 672ms
🗑Initiating removal of expired data: ID='data_c317c367.log', Owner='user_9cf50b'...
⚠Removal failed for data: ID='data_c317c367.log', Owner='user_9cf50b'
[DATA_RETENTION] [2025-04-07 19:54:57.326] - Data ID: 'data_c317c367.log', Owner: 'user_9cf50b', E
vent: 'FAILED', Duration: 792ms, Details: Simulated IO Error
🗑Initiating removal of expired data: ID='data_5a00e709.bin', Owner='user_5f73f5'...
```

Figure 52 - .Net Prompt 2 - Gemini Console

- **GitHub Copilot**

GitHub Copilot added detailed logging for each step, including initiation, upload, and download events, with color-coded outputs for better readability. It also implemented checks for necessary environment variables, improved error handling for failures during the cleanup process, and ensured required fields are validated. Additionally, it organized the code into logical sections and added comments for better maintainability. The results are shown in figure 53.

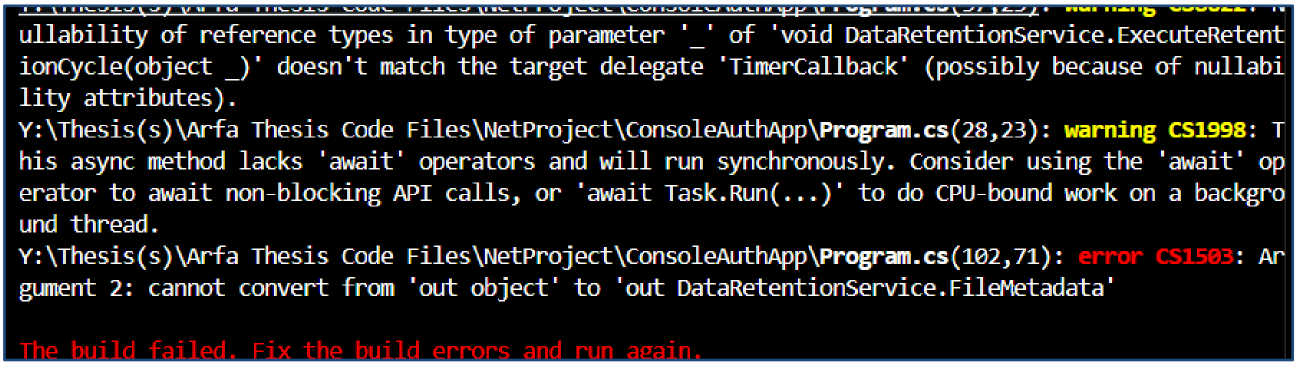


```
🔎 Found 19 data items past the retention threshold.
🗑Initiating removal of expired data: ID='data_c4406367.log', Owner='user_ddfe6a'...
✅ Data removed: ID='data_c4406367.log', Owner='user_ddfe6a' (took 1896ms)
[DATA_RETENTION] [2025-04-07 19:56:45.565] - Data ID: 'data_c4406367.log', Owner: 'user_ddfe6a', E
vent: 'REMOVED', Duration: 1896ms
🗑Initiating removal of expired data: ID='data_74cec666.log', Owner='user_9694a1'...
✅ Data removed: ID='data_74cec666.log', Owner='user_9694a1' (took 1794ms)
[DATA_RETENTION] [2025-04-07 19:56:47.382] - Data ID: 'data_74cec666.log', Owner: 'user_9694a1', E
vent: 'REMOVED', Duration: 1794ms
🗑Initiating removal of expired data: ID='data_a0136a0d.log', Owner='user_cf6fe4'...
✅ Data removed: ID='data_a0136a0d.log', Owner='user_cf6fe4' (took 1887ms)
[DATA_RETENTION] [2025-04-07 19:56:49.280] - Data ID: 'data_a0136a0d.log', Owner: 'user_cf6fe4', E
vent: 'REMOVED', Duration: 1887ms
```

Figure 53 - .Net Prompt 2 - Github Copilot Console

- **DeepSeek AI**

It enhanced thread safety & performance after replacing List<FileData> with ConcurrentDictionary<string, FileMetadata> function for thread-safe operations and added parallel processing (Parallel.ForEachAsync) for faster cleanup cycles. It also improved Data Model & Logging – Expanded FileMetadata with richer properties (size, content type, protection flag) and introduced structured JSON logging for better observability. This was all done while keeping the original prompt as default. However, even after too many tries, the build failed even after asking the model to change or minimize the scope. The results are in figure 54.

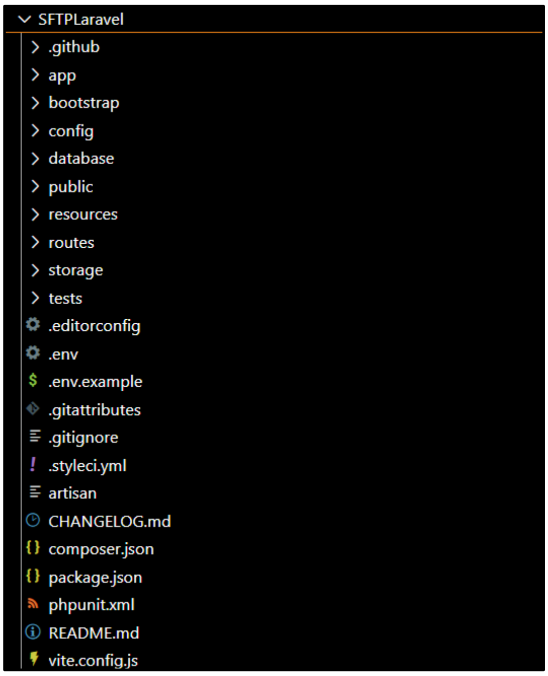


```
ullability of reference types in type of parameter '_' of 'void DataRetentionService.ExecuteRetent
ionCycle(object _)' doesn't match the target delegate 'TimerCallback' (possibly because of nullabi
lity attributes).
Y:\Thesis(s)\Arfa Thesis Code Files\NetProject\ConsoleAuthApp\Program.cs(28,23): warning CS1998: T
his async method lacks 'await' operators and will run synchronously. Consider using the 'await' op
erator to await non-blocking API calls, or 'await Task.Run(...)' to do CPU-bound work on a backgro
und thread.
Y:\Thesis(s)\Arfa Thesis Code Files\NetProject\ConsoleAuthApp\Program.cs(102,71): error CS1503: Ar
gument 2: cannot convert from 'out object' to 'out DataRetentionService.FileMetadata'

The build failed. Fix the build errors and run again.
```

Figure 54 - .Net Prompt 2 - DeepSeek Console

3.1.4 Laravel (PhP)

The fourth framework used for the application was Laravel with PHP. Laravel is known in the development community for its beautiful syntax, a large framework (scale), and extensive feature set. Laravel provided a good basis to implement all the features of the core functionality of base application. The framework is completely built in object-oriented architecture and has built-in tools for dependency injection, configuration management, and service containers. This helps the development of the application significantly as these pieces are already handled/manageable in Laravel. It also has built-in methods to easily support the Mongo database using community supported packages like jenssegers/laravel-mongodb along with various other storage options (e.g. packages like cloudinary-labs/cloudinary-laravel).

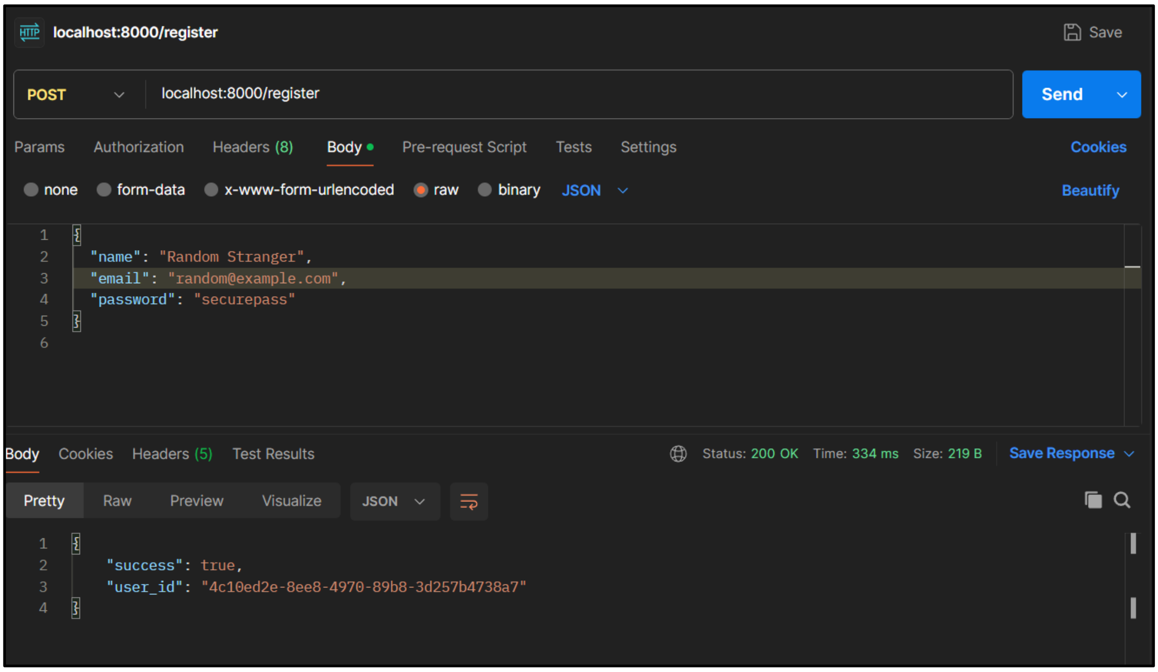


```
SFTPLaravel
  > .github
  > app
  > bootstrap
  > config
  > database
  > public
  > resources
  > routes
  > storage
  > tests
    .editorconfig
    .env
    .env.example
    .gitattributes
    .gitignore
    .styleci.yml
    artisan
    CHANGELOG.md
    composer.json
    package.json
    phpunit.xml
    README.md
    vite.config.js
```

Figure 55 - Laravel App Architecture

In the case of this thesis application, Laravel supports queues and event broadcasting allowing for asynchronous operation and keeping the user responsive and non-blocking. The architecture of the laravel application is illustrated in Figure 55.

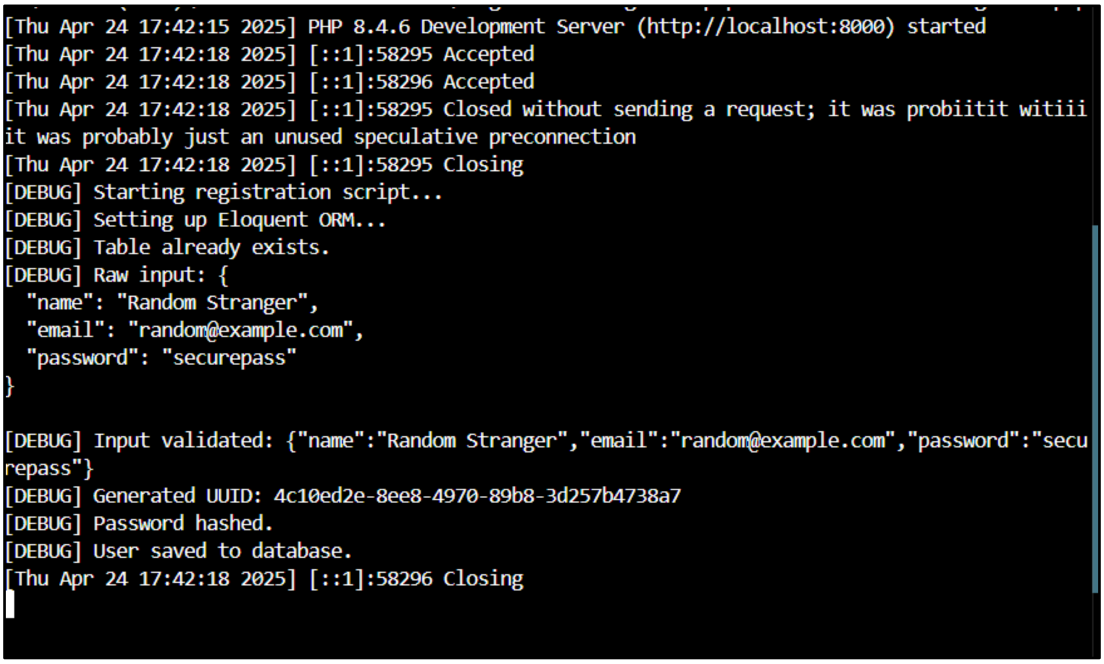
Figures 56 and 57 show the postman and console outputs of the application written in Laravel.



```
{
  "name": "Random Stranger",
  "email": "random@example.com",
  "password": "securepass"
}
```

```
{
    "success": true,
    "user_id": "4c10ed2e-8ee8-4970-89b8-3d257b4738a7"
}
```

Figure 56 - Laravel App Postman



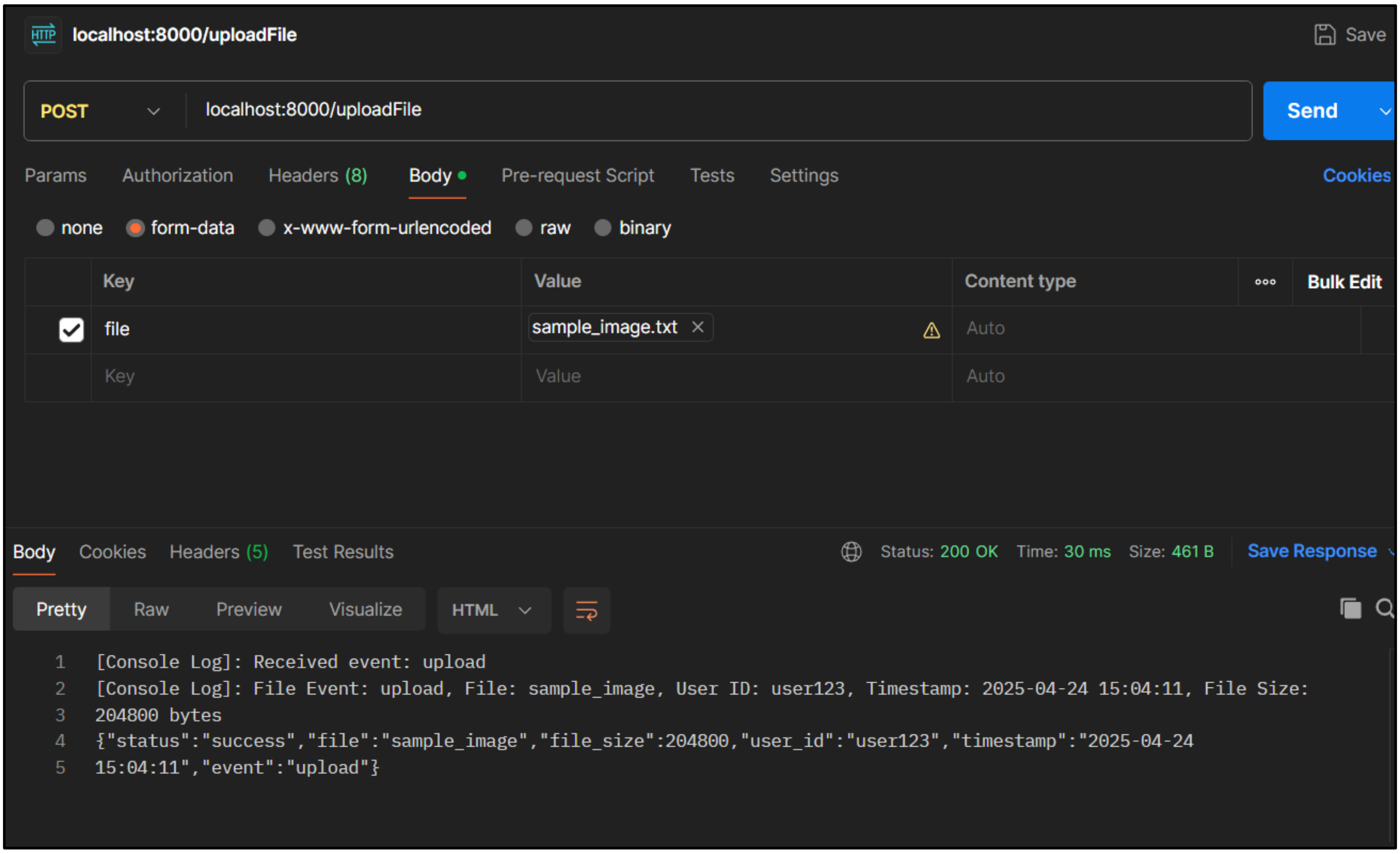
```
[Thu Apr 24 17:42:15 2025] PHP 8.4.6 Development Server (http://localhost:8000) started
[Thu Apr 24 17:42:18 2025] [::1]:58295 Accepted
[Thu Apr 24 17:42:18 2025] [::1]:58296 Accepted
[Thu Apr 24 17:42:18 2025] [::1]:58295 Closed without sending a request; it was probiitit witiii
it was probably just an unused speculative preconnection
[Thu Apr 24 17:42:18 2025] [::1]:58295 Closing
[DEBUG] Starting registration script...
[DEBUG] Setting up Eloquent ORM...
[DEBUG] Table already exists.
[DEBUG] Raw input: {
  "name": "Random Stranger",
  "email": "random@example.com",
  "password": "securepass"
}

[DEBUG] Input validated: {"name":"Random Stranger","email":"random@example.com","password":"secu
repass"}
[DEBUG] Generated UUID: 4c10ed2e-8ee8-4970-89b8-3d257b4738a7
[DEBUG] Password hashed.
[DEBUG] User saved to database.
[Thu Apr 24 17:42:18 2025] [::1]:58296 Closing
```

Figure 57 - Laravel App Console Output

a. Prompt 1 - *Generate a PhP Laravel event listener that triggers whenever a file is uploaded or downloaded via SFTP. The event should log the file name, user ID, timestamp, and file size to the database (MongoDB).*

- **ChatGPT**

  ChatGpt created the architecture for the prompt and provided detailed code output, without being asked to. However, it provided a developer console output, even when asked for creating an API endpoint for user-friendly responses. The results are shown in figure 58.



POST localhost:8000/uploadFile

Body: ● form-data

	Key	Value	Content type
☑	file	sample_image.txt	Auto

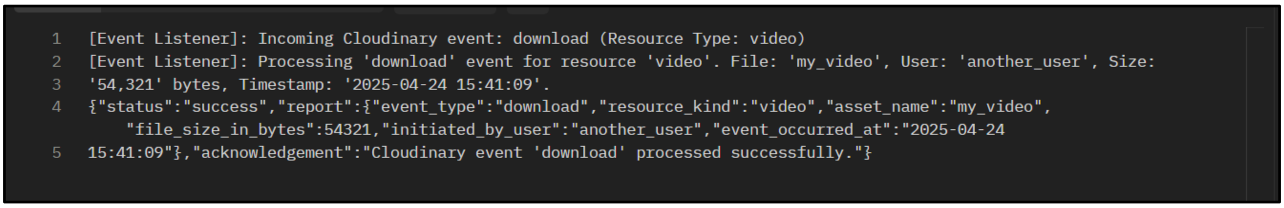
Status: 200 OK Time: 30 ms Size: 461 B

```
[Console Log]: Received event: upload
[Console Log]: File Event: upload, File: sample_image, User ID: user123, Timestamp: 2025-04-24 15:04:11, File Size:
204800 bytes
{"status":"success","file":"sample_image","file_size":204800,"user_id":"user123","timestamp":"2025-04-24
15:04:11","event":"upload"}
```

Figure 58 - Laravel - ChatGPT Postman

- **Gemini**

  Gemini enhanced the Cloudinary webhook listener by adding more detailed and informative console logging for successful events, errors (like invalid JSON or missing data), and unsupported events. The JSON responses sent back are now more structured, providing clear reports for successful processing and specific error messages when issues occur, improving both debugging and understanding of the webhook interactions (figure 59).

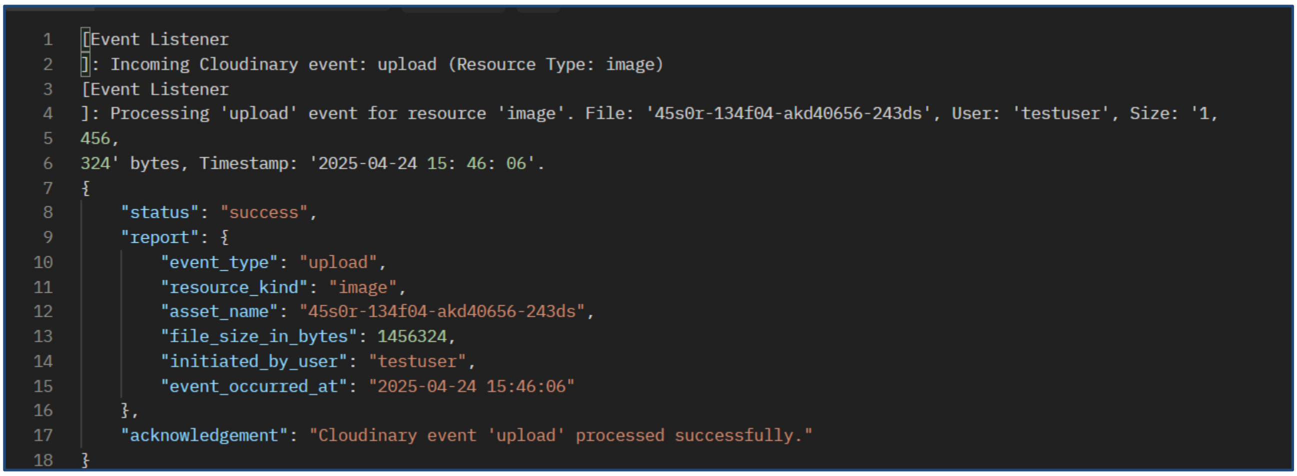


```
[Event Listener]: Incoming Cloudinary event: download (Resource Type: video)
[Event Listener]: Processing 'download' event for resource 'video'. File: 'my_video', User: 'another_user', Size:
'54,321' bytes, Timestamp: '2025-04-24 15:41:09'.
{"status":"success","report":{"event_type":"download","resource_kind":"video","asset_name":"my_video",
    "file_size_in_bytes":54321,"initiated_by_user":"another_user","event_occurred_at":"2025-04-24
15:41:09"},"acknowledgement":"Cloudinary event 'download' processed successfully."}
```

Figure 59 - Laravel Prompt 1 - Gemini Postman

- **Github Copilot**

  The PHP webhook was enhanced to handle Cloudinary events more effectively by adding detailed logging for upload and download events, including asset details, user context, and timestamps. Structured JSON responses were introduced for clear feedback, while unsupported events are ignored with concise error messages. Utility functions were added to streamline logging and response management, improving code clarity and maintainability as seen in figure 60.

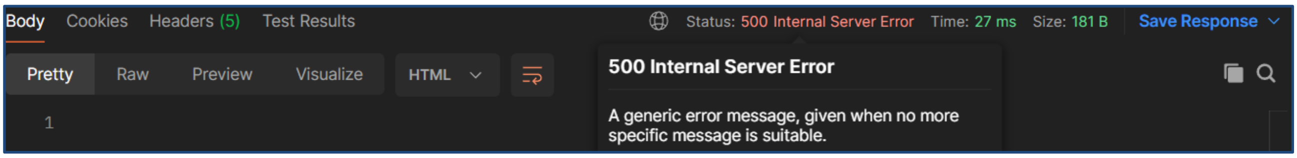


```
[Event Listener
]: Incoming Cloudinary event: upload (Resource Type: image)
[Event Listener
]: Processing 'upload' event for resource 'image'. File: '45s0r-134f04-akd40656-243ds', User: 'testuser', Size: '1,
456,
324' bytes, Timestamp: '2025-04-24 15: 46: 06'.
{
    "status": "success",
    "report": {
        "event_type": "upload",
        "resource_kind": "image",
        "asset_name": "45s0r-134f04-akd40656-243ds",
        "file_size_in_bytes": 1456324,
        "initiated_by_user": "testuser",
        "event_occurred_at": "2025-04-24 15:46:06"
    },
    "acknowledgement": "Cloudinary event 'upload' processed successfully."
}
```

Figure 60 - Laravel Prompt 1 - Github Postman

- **DeepSeek AI**

  DeepSeek AI tried to while performing the original task, tried to enhanced time complexity and efficiency, however the build failed continuously even with asking it to fix the problems repeatedly. The server was running, but the functions within it always returned internal server error as shown in figure 61.



Status: 500 Internal Server Error Time: 27 ms Size: 181 B

500 Internal Server Error

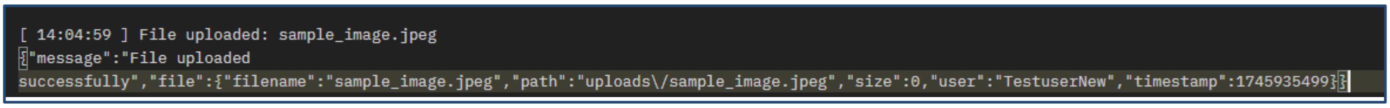
A generic error message, given when no more specific message is suitable.

Figure 61 - Laravel Prompt 1 - DeepSeek Postman

b. Prompt 2 - *Generate a PhP scheduled function that runs every 10 seconds to clean up stale data in MongoDB. The function should delete any user files that have not been accessed or modified in the last 1 min. Ensure the function is efficient and logs the result of each cleanup task to the database for auditing purposes.*

- **ChatGPT**

  After giving it the prompt, ChatGPT tried to introduce clean-up process logging. It should have printed the detailed output of the deleted file after 1 minute of its upload. However, it could not trigger the function and resulted only in the half-way functionality as seen in figure 62.

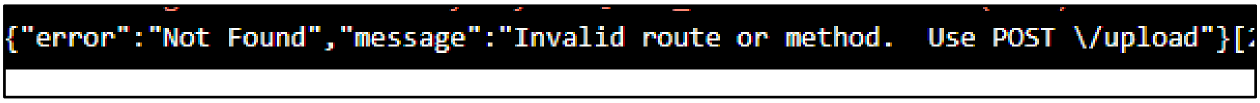


```
[ 14:04:59 ] File uploaded: sample_image.jpeg
{"message":"File uploaded
successfully","file":{"filename":"sample_image.jpeg","path":"uploads\/sample_image.jpeg","size":0,"user":"TestuserNew","timestamp":1745935499}}
```

Figure 62 - Laravel Prompt 2 - ChatGPT Console

- **Gemini**

  Gemi, too failed while generating the backend code. While it provided structure for the application and segregated the code into multiple files, it did not create the needed deletion functionality. The error is shown in figure 63.

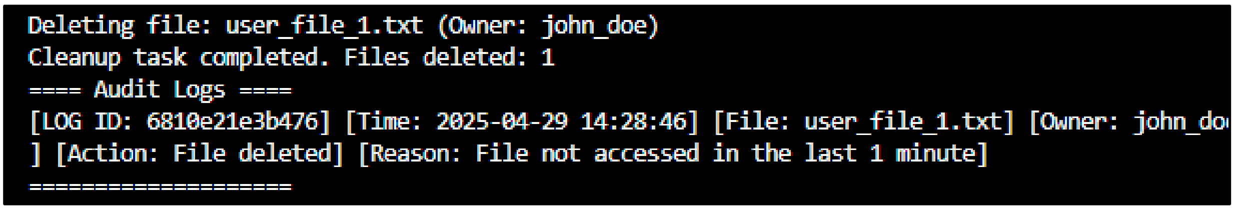


```
{"error":"Not Found","message":"Invalid route or method.  Use POST \/upload"}[
```

Figure 63 - Laravel Prompt 2 - Gemini Console

- **GitHub Copilot**

  Github Copilot was the only one that gave the full code that provided the deletion functionality. Additionally, it provided the detailed logs of the file information, log Id, reasoning and the timing. The results are shown in figure 64.

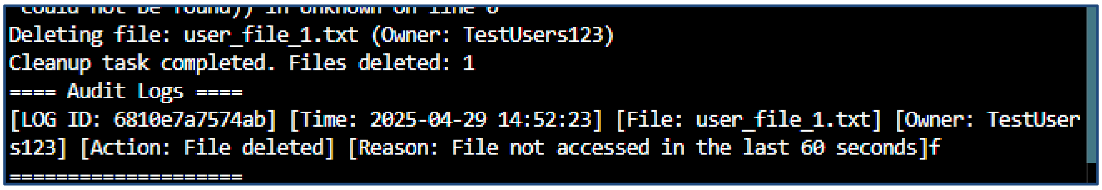


```
Deleting file: user_file_1.txt (Owner: john_doe)
Cleanup task completed. Files deleted: 1
==== Audit Logs ====
[LOG ID: 6810e21e3b476] [Time: 2025-04-29 14:28:46] [File: user_file_1.txt] [Owner: john_do
] [Action: File deleted] [Reason: File not accessed in the last 1 minute]
====================
```

Figure 64 - Laravel Prompt 2 - Github copilot Console

- **DeepSeek AI**

  DeepSeek transformed the existing script from a procedural, infinite-loop-based approach to an object-oriented, safer design with error handling, configurable thresholds, and security improvements. The enhanced version includes file sanitization, controlled execution time limits, proper logging, and better separation of concerns while maintaining all original functionality in a more maintainable and production-ready structure. The results are shown in shown in figure 65.



```
Deleting file: user_file_1.txt (Owner: TestUsers123)
Cleanup task completed. Files deleted: 1
==== Audit Logs ====
[LOG ID: 6810e7a7574ab] [Time: 2025-04-29 14:52:23] [File: user_file_1.txt] [Owner: TestUser
s123] [Action: File deleted] [Reason: File not accessed in the last 60 seconds]f
===================
```

Figure 65 - Laravel Prompt 2 - DeepSeek Console

3.2 Prompt Optimization

The emphasis of the paper rests on AI aid in back-end development-and-prompting, which influences very much the quality, efficiency, and readability of code generated by the AI models. In simple terms, prompt optimization involves strategically improving the instructions to maximize output according to the criteria set by the developer. Such criteria include correctness, maintainability, security, and scalability. Manual modifications have been promising, but this study also looks into the interplay of architectural practices and automated techniques for prompt enhancement.

The exploratory study, through employing a shortlist of optimization strategies that include structured formatting, role-based contextualizing, self-refinement loops, few-shot prompting, chain-of-thought reasoning, and parameter tuning, seeks to assess how much optimized prompting given measurable impacts on the AI-generated backend code. For this research, three main optimization strategies were used – self-refinement loops, few-shot prompt injections, and role-based contextualization.

3.2.1 Self-Refinement Loops

Self-refinement loops are a major strategy among automated prompt optimization techniques, whereby large language models (LLMs) iteratively generate, assess, and refine their own outputs. Here, either by means of explicit self-evaluative mechanisms or through the use of auxiliary modules (e.g., static analyzers, feedback classifiers), the system can diagnose problems of its first-generation outputs. If a problem is detected, an adjustment to the confines of the original prompt can typically be made—mostly by adding constraints or some lessons learned from the previous output statement—and thus improve the next version(s). Such approaches have amassed in frameworks such as Reflexion, which uses verbal reinforcement learning to allow models to critique their output and gradually improve on the task performed (Shinn et al., 2023). Similarly, Madaan et al. (2023) provide Self-Refine, which utilizes a natural language capability of the LLMs to revise the responses through feedback generation, incorporating that feedback into the forthcoming prompt from having shown substantively improved performance in fact-correctness and programming reliability.

These methods utilize the idea of training loops based on prompt completion, so the output is, instead, an intermediary having awareness of feedback that the output receives. This idea mirrors the human process of executing iterative debugging and revision, thus making it well-suited for code generation tasks in which the correctness and maintainability of the outputs are paramount. In operational systems, the feedback can be generated via unit tests (Chen et al., 2021), static analysis tools (such as pylint, bandit), or even AI-assisted evaluators (Wang et al., 2023). These feedback signals are either used to directly regenerate prompts or to rank and filter candidate generations. The iterative loop continues until quality constraints such as security, clarity, or modularity are met.

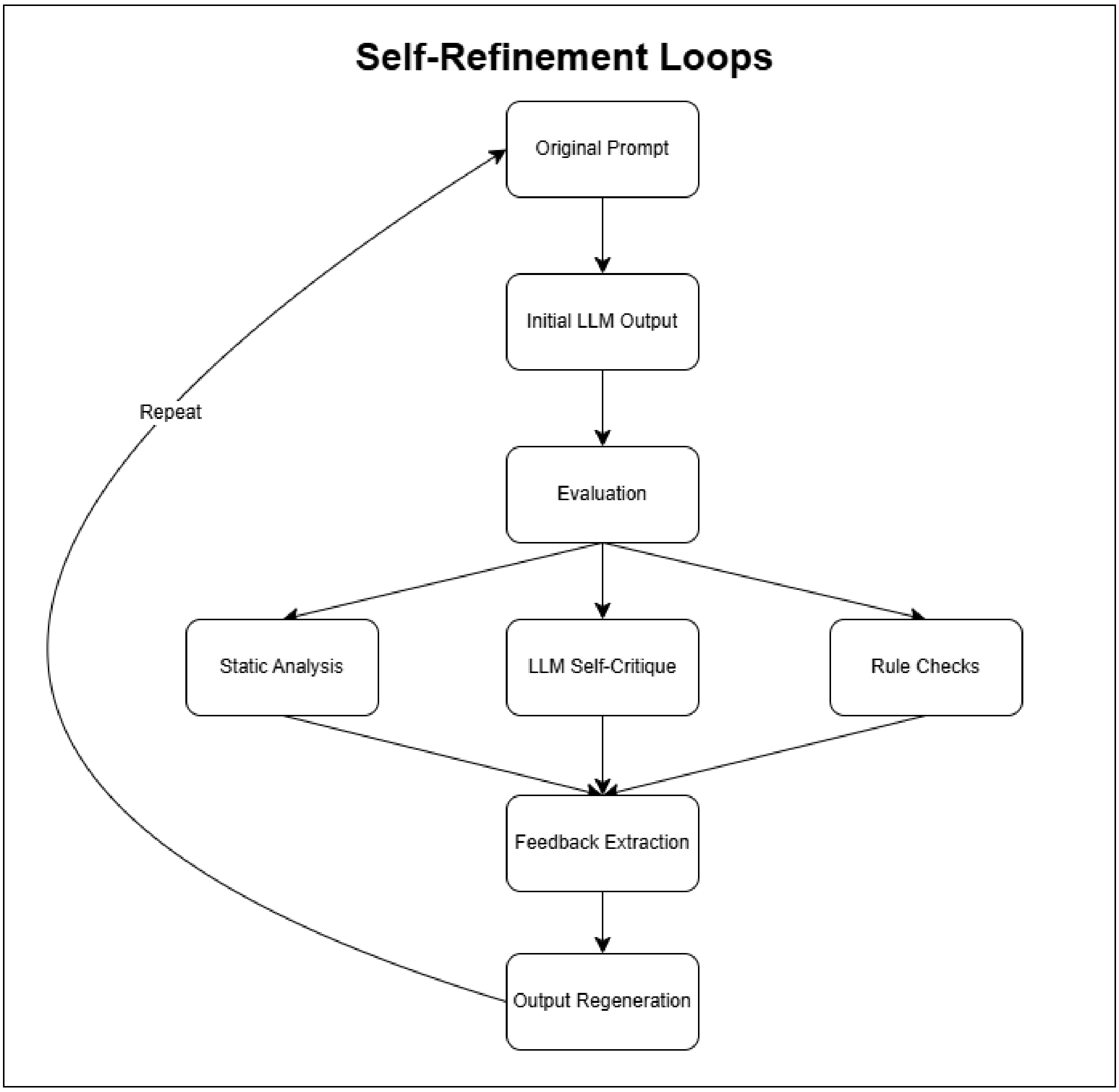


Figure 66 - Self Refinement Loops

Figure 66 shows the detailed process with which how self-refinement loops work. The technical goal is to automatically revise the original prompt based on the quality feedback from the model's own outputs. Initially, the original prompt is fed to a LLM, and the generated output is sent for evaluation. In this stage, different techniques such as static analysis, self-critique, and rule checks are used to evaluate the middle output. Following this the feedback is extracted and the output prompt is generated. If the output prompt is not satisfactory then the output is fed into the LLM as an original prompt until the desired result is reached.

After considering original prompt 1 –

Generate a Flask event listener that triggers whenever a file is uploaded or downloaded via SFTP. The event should log the file name, user ID, timestamp, and file size to the database (MongoDB).

And applying this technique, the result is as follows –

Generate a Flask event listener for SFTP upload/download events. Include:
- Logging of file name, user ID, timestamp, and file size into MongoDB.
- Input validation for file name and user ID.
- Try-except error handling for database operations.
- Schema definition or comments for MongoDB fields.

This prompt was used to generate the backend code with the four AI models (ChatGPT, DeepSeek, Gemini, and Github Copilot) using the four frameworks (Node.Js, Laravel, .Net, and Flask).

3.2.2 Few-Shot Prompt Injection with Dynamic Example Selection

Few-shot prompting refers to the process of improving large language model outputs by including relevant examples in the input prompt. Recent advancements in automated prompting leverage embedding-based similarity retrieval to dynamically select and inject contextually relevant code snippets or examples from a corpus of previously validated solutions. This technique enables the model to emulate desired patterns and behaviors in the new context by learning from representative tasks. In software development applications, dynamic few-shot prompting allows retrieval of similar backend logic or API handler functions, which are then inserted into the prompt to improve output quality and consistency. According to Gao et al. (2021), few-shot prompts enhance performance with downstream tasks, particularly code generation and plausible reasoning. They can also be automated at inference time through k-nearest neighbor example selection using semantic embeddings such as those from Sentence-BERT or OpenAI embeddings (Kassner & Hinrich Schütze, 2020).

Figure 67 illustrates the process of this technique. Firstly, the original prompt is embedded into a vector space and compared against a database of pre-indexed examples. This process occurs through semantic similarity. The most relevant examples (for instance – Top-k) are selected and appended into the original prompt. This method improves code consistency by guiding the model with real world pattens.

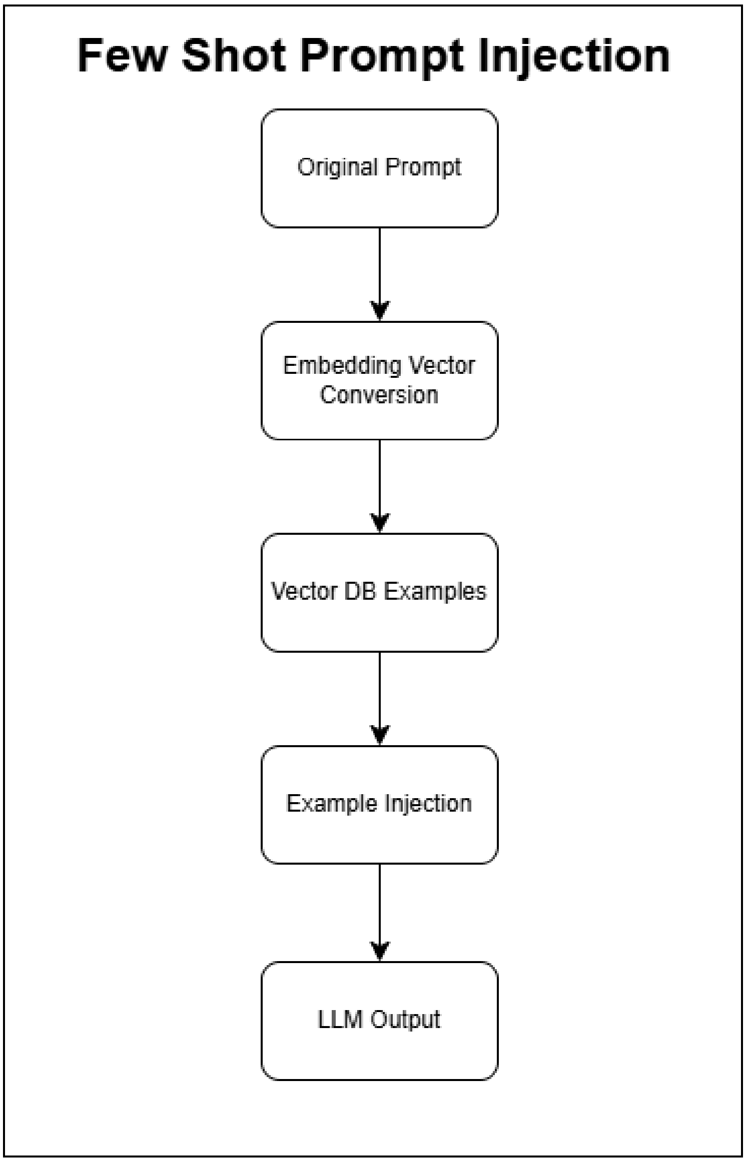


Figure 67 - Few-Shot Prompting

For example, this is the result of this optimizing technique –

Using the pattern below, generate a Flask event listener for file uploads/downloads via SFTP.

~~~
```python
def log_login_event(user_id, timestamp):
    try:
        db.logs.insert_one({
            "event": "login",
            "user_id": user_id,
            "timestamp": timestamp
        })
    except Exception as e:
        print(f"Log error: {e}")
~~~

3.2.3 Role-Based Contextualization

The prompts that are role-based contextualized clearly define the role or angle that AI is supposed to be in while performing a task. As examples, prompt constructions like "You are a backend engineer" or "You are coding an API endpoint with security in mind" have shown to enhance alignment with real-world-practicing habits and dampen undesired behavior (Reynolds & McDonell, 2021; Chowdhery et al., 2023). The very nature of the role context acts by priming the LLM with intent and domain context, homing in on the outputs we expect that fit right with the tone and safety constraints of the persona drawn. Role injection could hence be automated from a systems point of view via parsing metadata from the code repository (that is, project description, names of modules, cases for tests) to personas that are pertinent to that domain. Those prompts would dynamically adjust the tone, verbosity, modularity, and so on. It is supported by instructional tuning and prompt programming that priming for roles allows LLMs to most accurately imitate expert behavior, especially in structured tasks like software engineering (Zhou et al., 2023). This accounts for those aspects of prompt programming where the initial instruction influences the style of answer and the reasoning fidelity of the model (Brown et al., 2020; Wei et al., 2022).

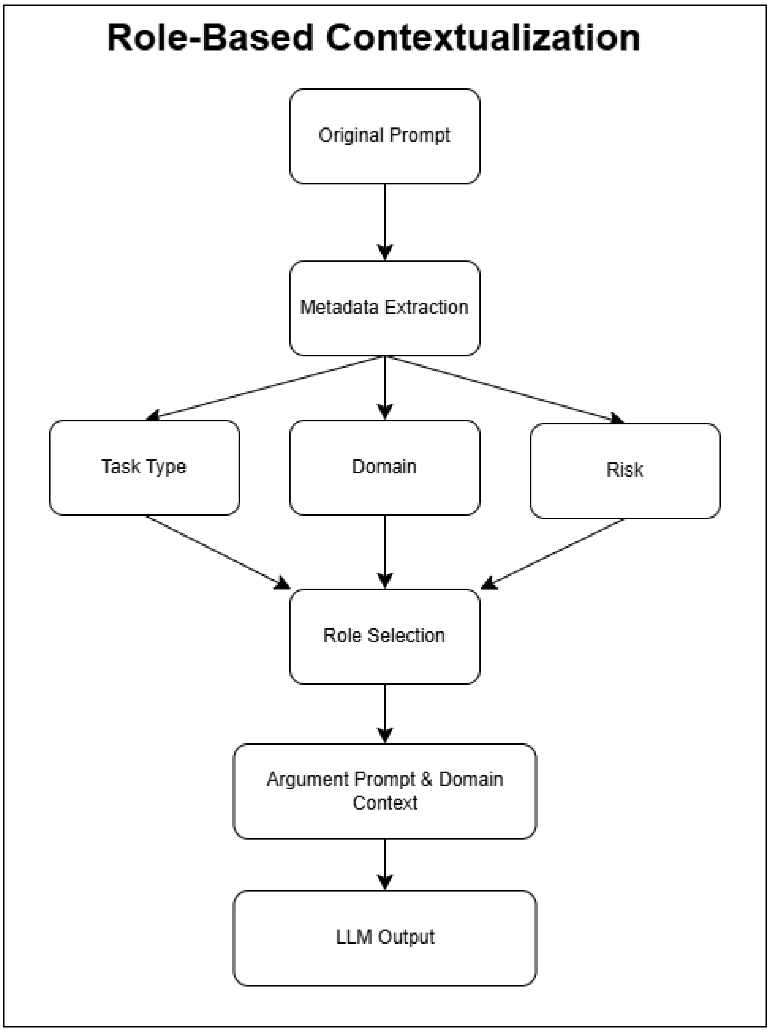


Figure 68 - Role Based Contextualization

Figure 68 presents the role-based contextualization process. The first step is to extract the metadata like – type, domain, and risk level from the original prompt. After this an appropriate role persona is selected and added to the prompt to guide the model.

The result of using this technique on the common prompt –

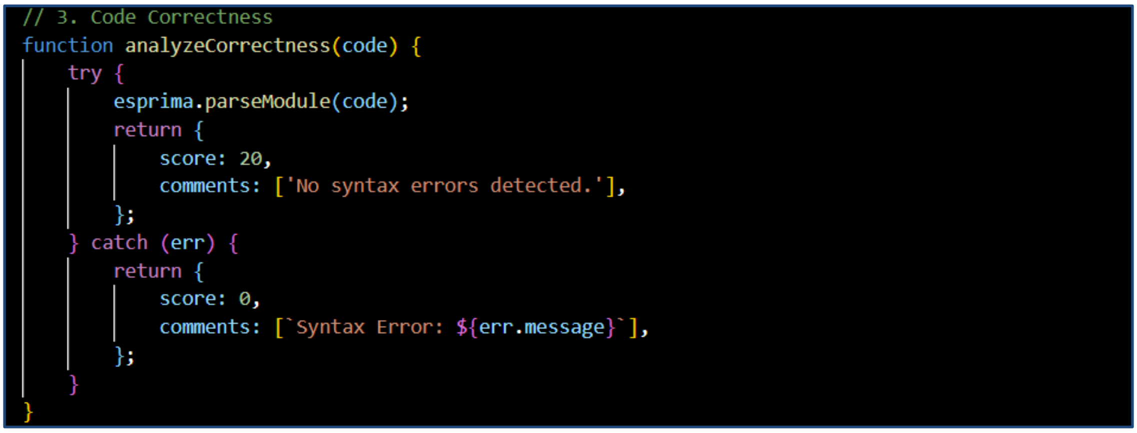
You are a senior back-end engineer designing a secure Flask-based system that handles SFTP file transfers. Create an event listener that logs each upload/download event by recording the file name, user ID, timestamp, and file size to MongoDB. Ensure proper structure, validation, exception handling, and extensibility.

3.3 SWE Benchmark Testing

For SWE benchmark testing, different code snippets were written and designed as a utility to perform the static analysis of the source code. The primary goal is to assess the code quality by evaluating give distinct dimensions – readability, time complexity, correctness, error vulnerability, and adaptability. The code snippet parses the code using Esprima parser to validate syntax and structure, and uses ESLint (for JavaScript) to detect style inconsistencies that impact readability. Time complexity is also computed with complexity-report library which in turn measures runtime efficiency and maintainability. Additionally, the script also measures error rates that may indicate higher debugging complexity. At the end it also evaluates the presence of environment variables to score adaptability. Each category is scored out of 20, which offers a standardized and automated approach to benchmark the source code.

3.3.1 Code Correctness Testing

Code correctness means that the generated code is valid in each logical and syntactical sense with respect to the functional requirements of the application. In the analysis script, this is measured via the analyzeCorrectness() function, which parses the code using Esprima. Thus, if the syntax is invalid, the score would drop to zero, along with a detailed error message. While static analysis cannot account for all logical errors, it represents a really good first-level validation to check that the structure is sound and the logic is complete. The code snippet is illustrated in figure 69.

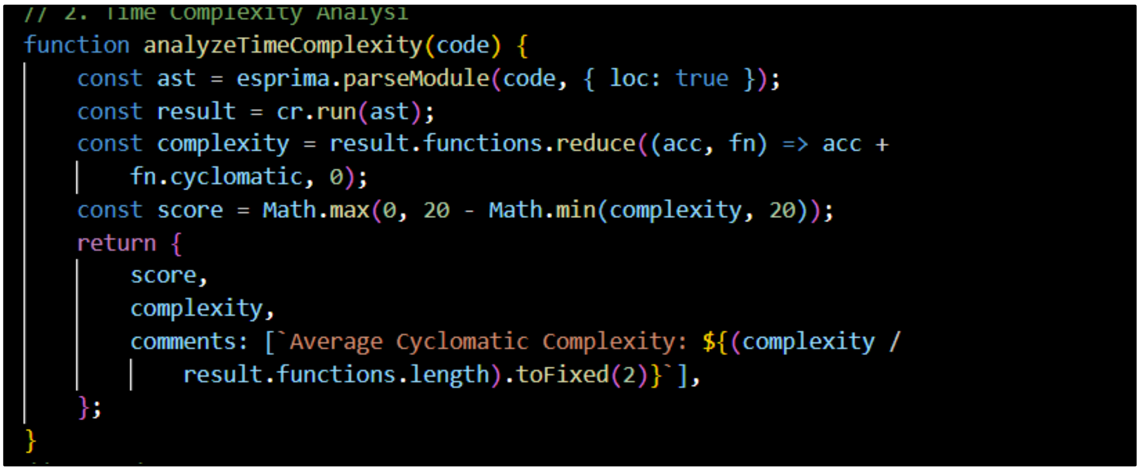


```
// 3. Code Correctness
function analyzeCorrectness(code) {
    try {
        esprima.parseModule(code);
        return {
            score: 20,
            comments: ['No syntax errors detected.'],
        };
    } catch (err) {
        return {
            score: 0,
            comments: [`Syntax Error: ${err.message}`],
        };
    }
}
```

Figure 69 - Code Correctness Testing

3.3.2 Runtime Efficiency Testing

The analyzeTimeComplexity() function assesses cyclomatic complexity pertinent to runtime efficiency. It considers the control flow structure of functions (via complexity-report) and estimates independent paths in the code, relates a high number of paths to slow performance and difficult-to-test functions, thus providing an indirect but meaningful substitute for making predictive statements about runtime behavior in the absence of execution-based profiling as seen in the figure 70.

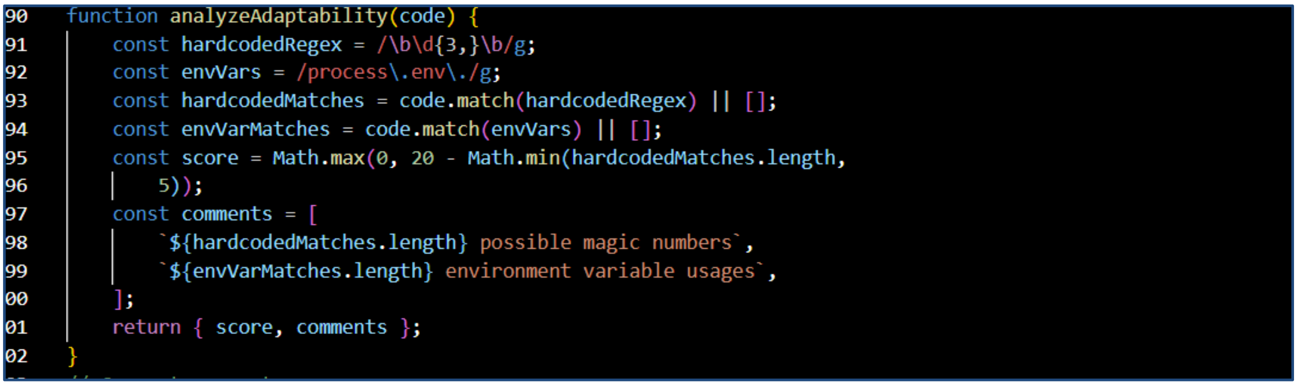


```
// 2. Time Complexity Analysi
function analyzeTimeComplexity(code) {
    const ast = esprima.parseModule(code, { loc: true });
    const result = cr.run(ast);
    const complexity = result.functions.reduce((acc, fn) => acc +
        fn.cyclomatic, 0);
    const score = Math.max(0, 20 - Math.min(complexity, 20));
    return {
        score,
        complexity,
        comments: [`Average Cyclomatic Complexity: ${(complexity /
            result.functions.length).toFixed(2)}`],
    };
}
```

Figure 70 - Time Complexity Testing

3.3.3 Maintainability Testing

Maintainability is evaluated along several metrics in the script. Cyclomatic complexity (as used from time-complexity analysis) speaks to how modular and understandable a code is. Furthermore, the analyzeAdaptability() function punishes hard-coded values and rewards the use of environment variables, thus impacting maintainability and scalability. High readability and modular structure result in better maintainability for long-term collaboration (figure 71).



```
function analyzeAdaptability(code) {
    const hardcodedRegex = /\b\d{3,}\b/g;
    const envVars = /process\.env\./g;
    const hardcodedMatches = code.match(hardcodedRegex) || [];
    const envVarMatches = code.match(envVars) || [];
    const score = Math.max(0, 20 - Math.min(hardcodedMatches.length,
        5));
    const comments = [
        `${hardcodedMatches.length} possible magic numbers`,
        `${envVarMatches.length} environment variable usages`,
    ];
    return { score, comments };
}
```

Figure 71 - Code Adaptability Testing

3.3.4 Readability Testing

Errors are counted that relate to indentation, quotation policy, semicolon use, and various spaces. The non-presence of such visible and structural indicators makes the code that much easier to read, debug, and work with. Readability is a good indicator of maintainability, and, in a situation where everyone collaborates, it truly impacts the efficiency of development.

3.3.5 Error Rate and Debugging Complexity Testing

The analyzeVulnerability() function tries to measure the benchmark against undesirable patterns that may cause potential runtime failures or hard-to-trace bugs. For example, pattern-matching would flag the use of eval(), the presence of empty catch blocks, and unguarded calls to process.exit(). By flagging such nuisance constructs, the script would be eliminating situations in which persistent human debugging is employed and thus, further solidifying the trustworthiness of AI-generated code (figure 72).

```
function analyzeVulnerability(code) {
    const riskyPatterns = [
        { pattern: /eval\s*\(/g, description: 'Usage of eval()' },
        {
            pattern: /require\s*\(.+\)/g, description: 'Dynamic require()' },

{
                pattern: /try\s*{[^}]*}\s*catch\s*\(err\)\s*{}/g,
                description: 'Empty catch block'
            },
{
            pattern: /process\.exit\s*\(/g, description: 'Process exit without error handling' },
];
    let score = 20;
    const comments = [];
    riskyPatterns.forEach(({ pattern, description }) => {
        const matches = code.match(pattern);
        if (matches) {
            score -= 5;
            comments.push(`${description} - Found ${matches.length}
time(s).`);
        }
    });
    score = Math.max(0, score);
    return { score, comments };

}
```

Figure 72 - Vulnerability to Error Testing

All of these categories were tested on all the backend frameworks mentioned in the study on the code generated by the four AI models. The analysis occurred over the code that was originally presented by the models, and the code that was changed by optimization. The results of the analysis are present in Chapter 5.

3.4 Security Testing

Similar to benchmark testing, security testing is crucial to understand different threats, and safeguard the code against the malicious intents. Security testing holds an important value across the backend development tasks, because of the vulnerability similarity. In some stacks, similar functions such as eval(), exec(), and os.system() exist with are susceptible to improper input handling, hardcoded sensitive information, or dangerous reflection calls. The script carries out static security analysis of JavaScript source files and tries to detect insecure coding patterns, missing input validation, and dependency vulnerabilities. Basically, there are three modules: very risky code patterns are searched with the help of regular expressions; basic input validation practices are checked; and any known vulnerabilities in dependencies are queried with npm audit. These three components in and of themselves target vectors for application compromise such as code injection, unsafe process executions, exposed credentials, and unsensitized input (figure 73).

```
function scanDependencies() {
  try {
    const audit = execSync('npm audit --json', { stdio: ['pipe', 'pipe', 'ignore'] }).toString();
    const data = JSON.parse(audit);
    const vulnerabilities = data.metadata.totalDependencies ? data.metadata.vulnerabilities : null;
    return vulnerabilities;
  } catch (err) {
    return { error: 'npm audit failed or not available' };
  }
}

// 3. Input Validation (very basic heuristic check)
function scanInputValidation(code) {
  const inputSources = [
    /req\.body/g,
    /req\.query/g,
    /req\.params/g,
    /process\.argv/g
  ];
  const validators = [
    /parse(Int|Float)/g,
    /Number\(/g,
    /typeof/g,
    /validator\.is/g
  ];

  const inputs = inputSources.some(p => code.match(p));
  const validations = validators.some(p => code.match(p));

  return inputs && !validations
    ? 'Inputs detected without proper validation.'
    : null;
```

Figure 73 - Security Testing Code Snippet

With the automation of these checks, developers and auditors can mitigate serious security issues early in the development lifecycle. This essentially helps bridge the divide between AI-generated code and real-world production readiness, specifically within the realms of CI pipelines or research judging whether an AI can generate truly safe, secure, and production-grade backend code. Also, npm audit for detecting dependency vulnerabilities is symptomatic of a bigger security problem in package managers such as pip for Python, composer for PHP, or NuGet for .NET.

4 Research Questions

Based on the research data gathered from various practical implementations as stated in chapter 3, this thesis identifies three main research problems in the field of backend code generation. The research is based on these questions and these are addressed below, exploring all of the factors and assessing the performance of each model in generating the functional code for a standard application

4.1 AI Model Comparison

The first research question is – *What are effective and objective methods to compare different AI models in terms of accuracy when generating backend code?*

4.1.1 Operationalizing Accuracy through Software Engineering Benchmarks

Objective and effective comparison of AI models for backend code generation requires an accurate, unambiguous, and measurable definition for accuracy. Following practical implementation as discussed in Chapters 2 and 3, this study conceptualizes accuracy as a triplet of functional, syntactic, and logical correctness. To quantify these aspects, five core software engineering (SWE) benchmarks were employed: code correctness, runtime efficiency, maintainability, readability, and debugging complexity. Beyond surface-level syntactic correctness, these dimensions delve into deeper qualities of quality software. Thus, the comparison framework assures technical relevance and methodological rigor by aligning those metrics with real-world development priorities and international standards, such as ISO/IEC 25010.

4.1.2 Using a Standardized Functionally Rich Application Context

To ensure that the evaluation could be generalized and somewhat realistic, the comparison was based on the development of a standardized SFTP-based backend application. The application is described in detail in Chapter 3 and contains core backend elements such as file transferring mechanisms, role-based access control, event-listeners, database integration, plus scheduled cleanup functions. This multi-functional architecture provided strong ground for testing, bearing a strong resemblance to real-world backend engineering requirements. Accordingly, it could be considered a very good way to show the strengths and weaknesses of AI-generated code in practice by doing a moderately complex backend task with real-world relevance.

4.1.3 Ensuring Prompt Consistency and Controlled Input Design

An important contributing factor towards a fair AI model comparison is prompt uniformity. Following this guiding principle, all AI models herein were given the same set of prompts with the design to prevent any ambiguity and to ensure reproducibility. The same input template eliminated any potential variability of the prompts as a confounding factor, ensuring that the differences between the outputs could be accounted for by the behavior of the models and not any inconsistency in the prompt itself. This principle comes especially into play when language models are assessed since these have shown to be sensitive to fine differences in prompt wording. Therefore, standardization of prompts is an absolute must for building fairness and objectivity in comparative assessments.

4.1.4 Automating Evaluation through Objective Toolchains

Objective methods of assessment rely mostly on automated and reproducible evaluation techniques. For the actual evaluation of code output, custom scripts and tools such as Esprima, ESLint, complexity-report, and Postman were developed. These tools set baseline measurements for the syntactical structure, cyclomatic complexity, runtime behavior, and API interaction. This automated approach minimizes human bias such that all AI models' performances were assessed consistently and repeatedly. General wisdom says that any method comparing must favor tool-based quantitative assessment to strengthen the objectivity and replicability of the findings.

4.1.5 Integrating Prompt Optimization as an Experimental Variable

Techniques, such as self-refinement loops, few-shot prompting, and role-based contextualization, have all been introduced in a controlled manner to observe their impact on model performance. This layering of experimentation has allowed for separation of intrinsic model capability from performance gains caused by improved prompt construction. Thus, treating prompt optimization methods as verifiable variables gives more depth and causality to the evaluation methodology for AI models.

4.1.6 Methodological Generalizability and Validity

The chosen methodology for this study is applicable to AI code generation since it encompasses standardized application design, SWE-compatible benchmarks, prompt control, and automated assessment methods. Testing was also performed on different environments, including Node.js, Flask, .NET, and Laravel, thus establishing a second layer of validation of applicability across different backends. The resulting framework not only enables proper model comparison regarding a particular application, but it provides a blueprint on which further endeavors in software engineering and AI evaluation can be conducted objectively.

Figure 74 displays the objective methodology which can be used to assess the accuracy of the AI models in context of backend code generation.

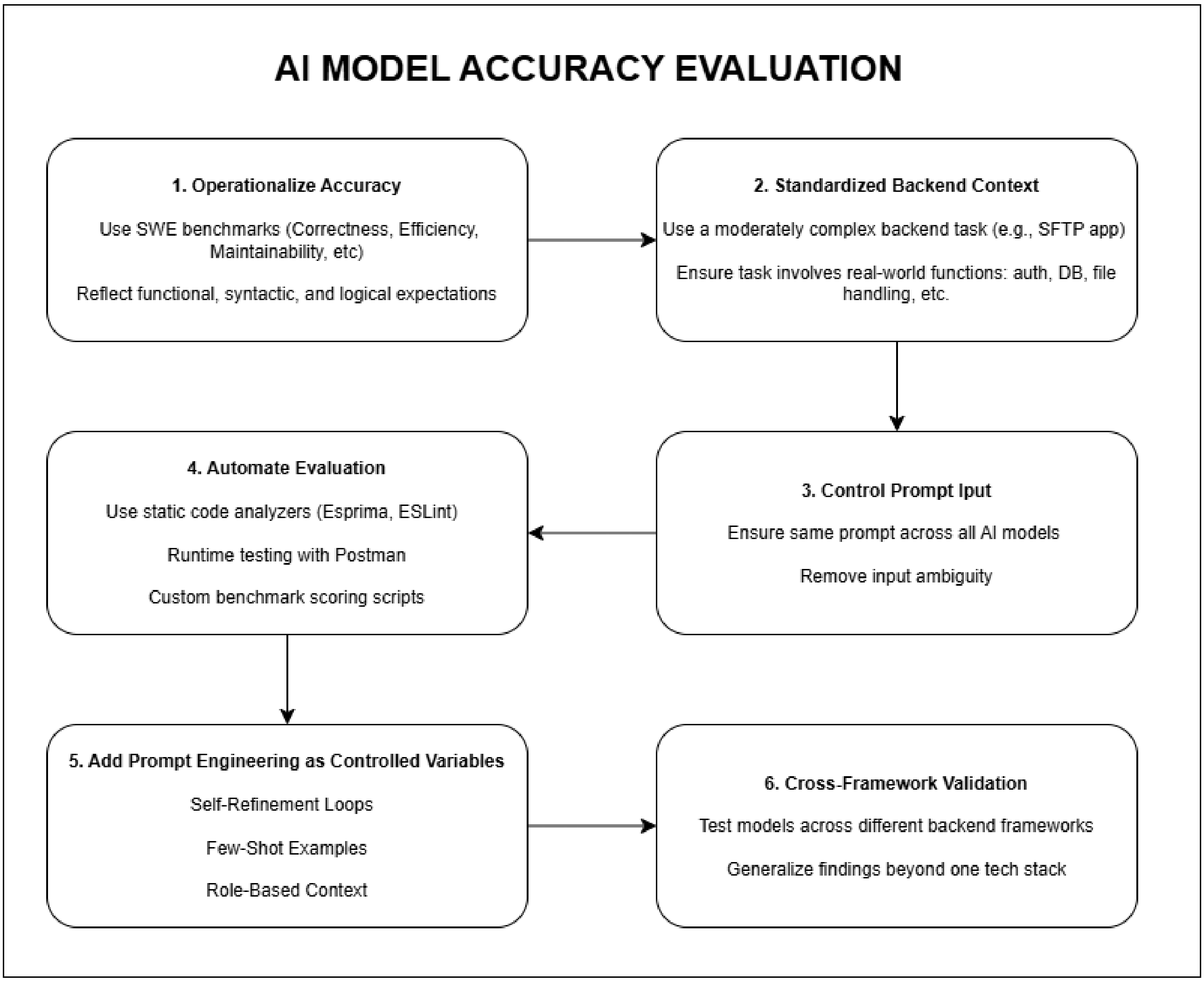


Figure 74 - AI Model Accuracy Evaluation

4.2 Security, Scalability, and Maintainability of Code

The second research question is - *What impact does prompt engineering have on AI's ability to generate scalable, maintainable backend code and how does it handle the security concerns?*

4.2.1 Conceptualizing Prompt Engineering as a Causal Factor

To measure the impact of the prompt engineering effects, the latter was treated less as an auxiliary activity and more of a causal input variable capable of affecting AI-generated code characteristics. This framing enabled methodological positioning of prompt design as a manipulable factor that could have its effects on backend quality isolated and observed in multiple output dimensions.

4.2.2 Mapping Impact Areas to Generalizable Quality Dimensions

Three abstract and generalizable software quality dimensions were studied: scalability, maintainability, and security awareness. These three categories represent some of the most relevant issues of concern in backend engineering.

- The study operationalized the signs of scalability through architectural foresight like modularization, externalization of configuration, and reusability.
- Maintainability was reflected in code structure, naming consistency, and the presence of abstraction patterns that support extensibility
- Security awareness was articulated in terms of existence or otherwise of vulnerabilities, secure data handling techniques, and following a few defensive coding fundamentals.

4.2.3 Controlled Comparison of Prompting Conditions

To determine the causal power behind prompt engineering-controlled comparative experiments, the same functional coding tasks were issued to AI models under two conditions: baseline prompting and enhanced prompting. Such enhancement involves one or more role-specific frameworks, an iterative refinement prompt, and contextual illustrative examples. This structure also allowed systematic observation of prompt-induced effects across various categories of outputs.

4.2.4 Structured Observation of Change Through Metric-Aligned Indicators

The prompt change-induced impact was measured using observable and repeatable indicators mapped to the defined quality domains. Although the evaluation metrics themselves were elaborated in Chapter 3, their parts here were meant to serve as evidence of changes across prompt conditions.

4.2.5 Cross-Model and Cross-Framework Generalization

In the end, the impact assessment was intended to be model and framework - agnostic. By applying the same prompt variations across different AI engines and backend technologies, the study ensures to guarantee results that were not subject to any single technical stack. This implication supports the conclusion that prompt engineering has widely applicable effects on AI coding behavior and represents a control lever by which software quality can be managed in backend development more generally.

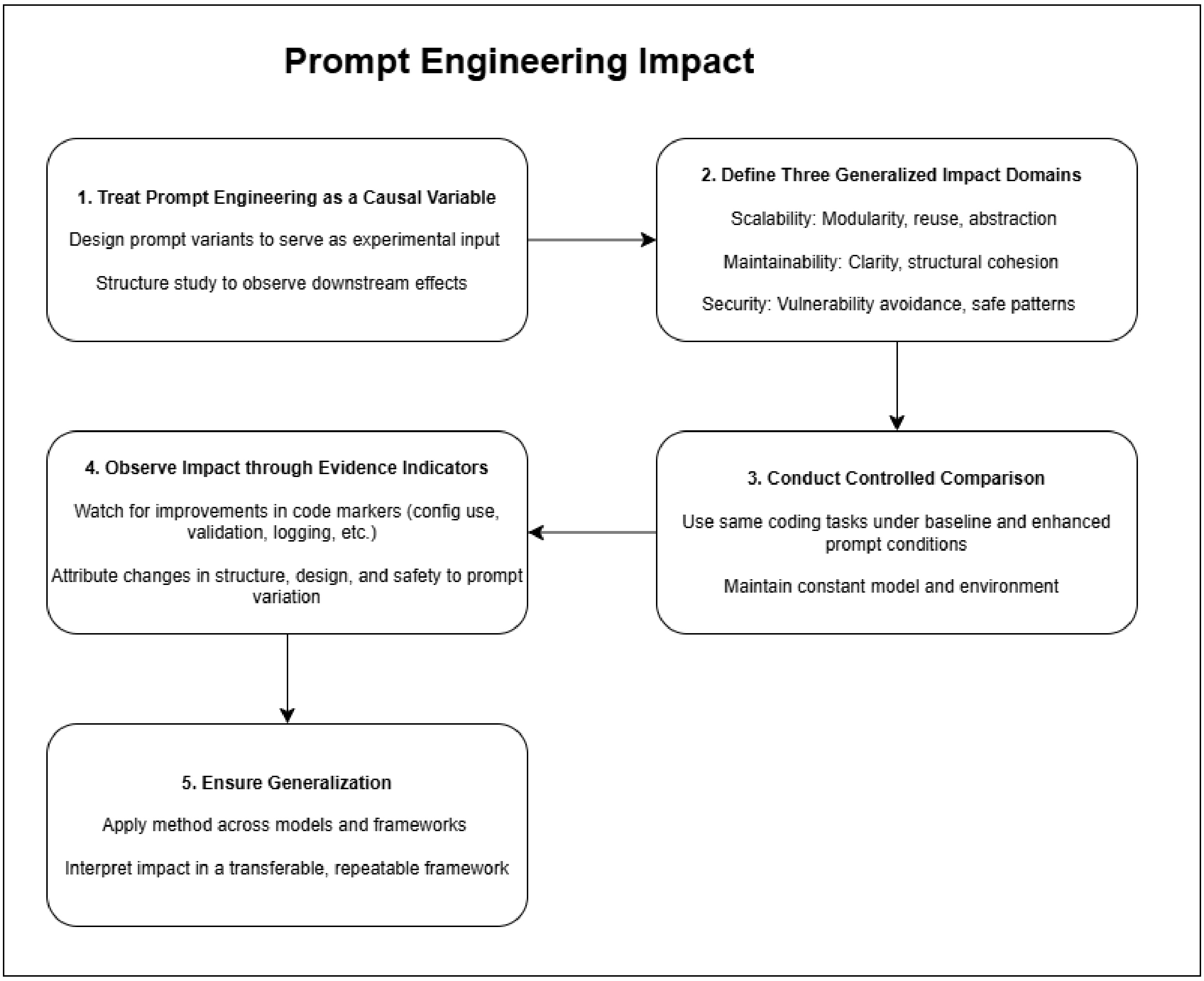


Figure 75 - Prompt Eng Impact

Figure 75 displays the methodological approach to understand and measure the impact of prompt optimization.

4.3 Reducing Technical Debt

Technical debt usually implies future reworking costs that arise by choosing a shortcut or a straightened solution today instead of a better and more time-consuming solution (Cunningham, 1993). The ability of AI to generate code has revolutionized development speed and time-to-market, but usually at the cost of maintainability. Within backend systems on which reliability, scalability, and robustness are key components, this is usually the kind of pattern that could lead to problems for years to come because AI might generate code that works but requires significant refactoring due to poor design choices, resulting in the accumulation of debt that affects future development cycles (Pearce et al., 2021).

4.3.1 AI-Assisted Backend Code Generation to reduce Technical Debt

Structured and intelligent use of AI can actually help reduce existing technical debt in back-end systems. Through precision prompt engineering and DevOps integration, AI systems can consistently produce boilerplate-free, standardized, and modular code at scale. This reduces opportunities for human error and enforces consistency in large codebases. Use of AI code generation tools, in combination with architectural patterns and automated tests, makes it faster to scaffold new best-practice-aligned services, route handlers, and controllers. For example, AI could be prompted to generate RESTful endpoints that followed a layered architecture such as controllers, services, and repositories, added schema validations, and applied middleware for logging, and error-handling purposes. Thus, by generating the components repetitively, AI saves the developer's time by allowing them to focus on domain logic.

4.3.2 Key Mechanisms to reduce Technical Debt

- **Standardization of Architecture**
  AI recognizes software architectures like MVC or Clean, and one can nudge the AI models into generating the backend via these architectures. This way, the differences in project structures arising in manually coded applications will be reduced, especially in case of huge teams (Martin 2009).

- **Reduction of Code Duplication**
  This means using internal knowledge to avoid reimplementing logic. When features requested by the AI exist partially in some form, the AI is able to abstract and reuse code instead of duplicating it-something developers under the approaching deadlines may miss.

- **Accelerated Refactoring**
  Instead of requiring developer implementation for manual refactoring of legacy services, AI tools can aid in wholesale rewriting of code into newer modular implementations. For example, a legacy authentication handler could be AI-refactored into reusable middleware with tokenization, expiration logic, and error-back all within seconds.

- **Better Test Coverage**
  When prompted directly, AI can generate unit and integration tests that ultimately add to coverage metrics and reducing bugs over time for back-end endpoints. All too often in classical development, writing and maintaining such tests are postponed due to lack of resources.

- **Continuous Learning and Correlation**

As teams iterate and adapt AI prompts according to project needs, this makes the AI align better with internal coding standards. This feedback loop will enable a long-term reduction in code inconsistency and better maintainability in the future.

4.3.3 Strategic Adoption Considerations

To fully utilize AI code generation as technical debt mitigation tool, the development teams can adopt the generation with safeguards. These include –

- Defining clear architectural patterns to embed into AI prompts
- Using static analysis and CI/CD tools to validate the generated code
- Implementing human-in-the-loop review process to ensure correctness
- Monitoring long-term maintainability metrics to identify patterns

As AI tools evolve and become increasingly capable of reasoning about design intent, their role in reducing long-term backend technical debt will likely expand beyond scaffolding and into proactive architectural optimization.

Table 4 - Traditional Vs AI Generated Factors

Feature	**Traditional Backend Development**	**AI-Assisted Code Generation**
Code Consistency	Dependent on Developer	Enforced through prompt constraints
Boilerplate Implementation Time	High	Near-Instant
Risk of Code Duplication	Moderate to High	Low (proper prompts)
Test Generation	Often Delayed or Skipped	Automated and Consistent
Architectural Conformity	Varies across teams	Can be embedded in generation prompts
Refactoring Legacy Code	Manual and Time-Consuming	Semi-automated with prompt design

Table 4 lists key operational differences between traditional backend development and AI-assisted code generation, correlating each feature directly with factors guiding technical debt. Code consistency and architectural adherence—two important factors in the long-term maintenance of any application—can become random variables in a human-driven workflow, whereas in AI prompt design, they can be fixed. In another example, minimizing automating boilerplate code and test generation reduces the chances of postponing tasks—such as these are the big contributors to technical debt. In contrasting these scenarios, the table further shows how strategic integration of AI tools can be used to proactively alleviate structural inefficiencies and future rework, and thus minimize technical debt throughout the backend software lifecycle.

5 Results

5.1 Model Comparison (SWE Benchmarks)

As previously outlined, four distinct AI models employed for backend code generation were evaluated using standardized SWE benchmarks. Chapter 3 provided a detailed account of the implementation of various testing scripts designed to assess key quality dimensions including accuracy, maintainability, readability, security, error rate, and efficiency. This section presents a comparative analysis of the performance of these AI models based on those evaluation metrics.

5.1.1 ChatGPT

After implementing the four backend frameworks - Node.js, Flask, Laravel, and .Net, the comparison modal is present below –

Node JS

Table 5 - Node Js Metrics

Metric	Score (Out of 20)
Code Correctness	20
Runtime Efficiency	17
Maintainability	16
Readability	18
Error Rate / Security	15
Total	**86/100**

Table 5 displays the metric score (aggregated) using the scripts run on the prompt 1 and 2 – for Node.Js on the code generated by ChatGPT.

Laravel

Table 6 - Laravel Metrics

Metric	Score (Out of 20)
Code Correctness	14
Runtime Efficiency	11
Maintainability	13.5
Readability	17
Error Rate / Security	16.5
Total	**72/100**

Table 6 displays the metric score (aggregated) using the scripts run on the prompt 1 and 2 – for Laravel (PhP) on the code generated by ChatGPT.

Flask

Table 7 displays the metric score (aggregated) using the scripts run on the prompt 1 and 2 – for Flask (Python) generated by ChatGPT.

Table 7 - Flask Metrics

Metric	Score (Out of 20)
Code Correctness	20
Runtime Efficiency	17.5
Maintainability	16.5
Readability	18.5
Error Rate / Security	18
Total	**90.5/100**

.Net

Table 8 - .NET Metrics

Metric	Score (Out of 20)
Code Correctness	18
Runtime Efficiency	18.5
Maintainability	14
Readability	20
Error Rate / Security	17
Total	**87.5/100**

Table 8 displays the metric score (aggregated) using the scripts run on the prompt 1 and 2 – for .Net (C#) generated by ChatGPT.

The figure 76 compares the backend frameworks used to generated code for Prompt 1 and Prompt 2 using ChatGPT. As per the evaluation, ChatGPT created the highest quality code, which is based on Flask, with a score of 90.5 out of 100. Node.js and .NET followed closely, attaining scores of 86 and 87.5, respectively. Laravel obtained the lowest score of 72 points out of 100, highlighting its relative limitations in implementation quality for this particular use case.

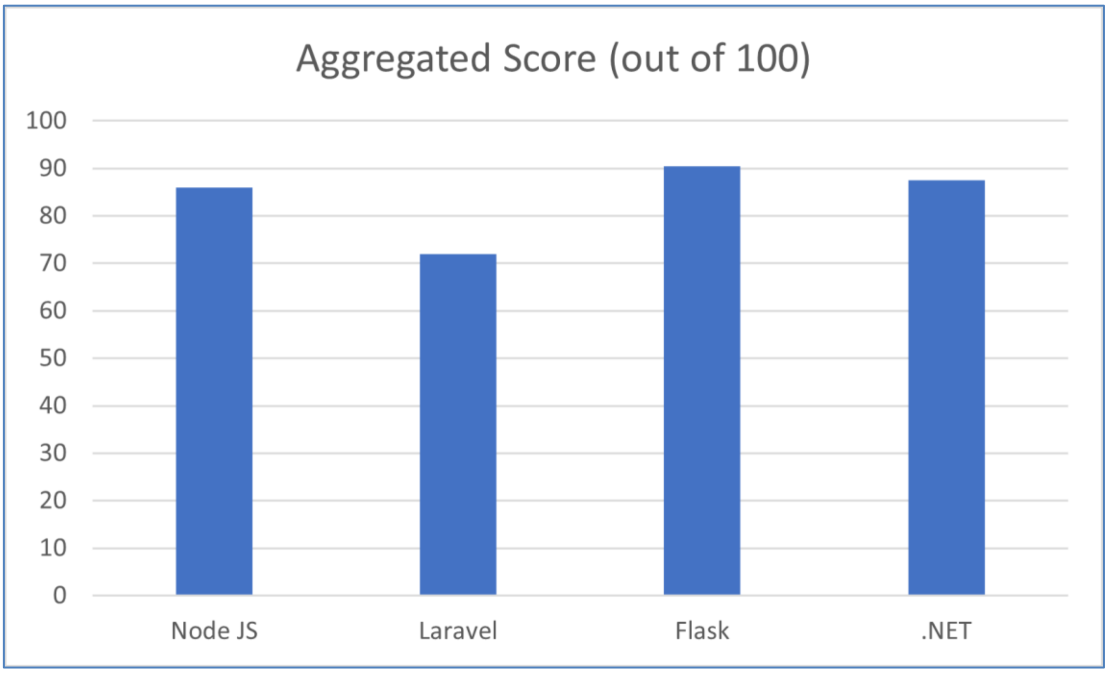


Figure 76 - ChatGPT Analysis

5.1.2 Gemini

After implementing the four backend frameworks with Gemini - Node.js, Flask, Laravel, and .Net, the comparison modal is present below –

Node JS

Table 9 - Gemini Node JS Metrics

Metric	Score (Out of 20)
Code Correctness	19
Runtime Efficiency	16
Maintainability	18
Readability	20
Error Rate / Security	17
Total	**90/100**

Table 9 displays the metric score (aggregated) using the scripts run on the prompt 1 and 2 – for Node.Js on the code generated by Gemini.

Laravel

Table 10 - Gemini Laravel Metrics

Metric	Score (Out of 20)
Code Correctness	10
Runtime Efficiency	08
Maintainability	11

Readability	14.5
Error Rate / Security	10.5
Total	**44/100**

Table 10 displays the metric score (aggregated) using the scripts run on the prompt 1 and 2 – for Laravel (PhP) on the code generated by Gemini.

Flask

Table 11 - Gemini Flask Metrics

Metric	**Score (Out of 20)**
Code Correctness	10
Runtime Efficiency	7.5
Maintainability	11.5
Readability	19
Error Rate / Security	12
Total	**60/100**

Table 11 displays the metric score (aggregated) using the scripts run on the prompt 1 and 2 – for Flask (Python) generated by Gemini.

.Net

Table 12 - Gemini .NET Metrics

Metric	**Score (Out of 20)**
Code Correctness	20
Runtime Efficiency	19.5
Maintainability	16
Readability	15.5
Error Rate / Security	18
Total	**89/100**

Table 12 displays the metric score (aggregated) using the scripts run on the prompt 1 and 2 – for .Net (C#) generated by Gemini.

As per the evaluation, Gemini produced the highest quality code with Node.js, achieving a score of 90 out of 100. .NET followed closely with a score of 89, while Flask attained a moderate score of 60. Laravel received the lowest score of 44 as seen in figure 77.

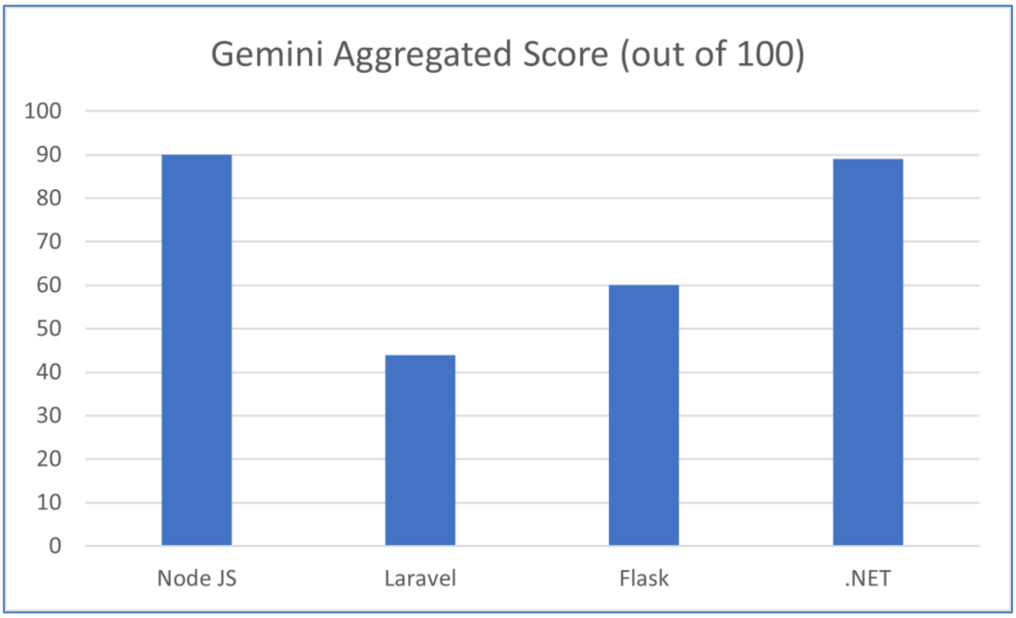


Figure 77 - Gemini Analysis

5.1.3 GitHub Copilot

After implementing the four backend frameworks with GitHub Copilot - Node.js, Flask, Laravel, and .Net, the comparison modal is present below –

Node JS

Table 13 - GitHub Copilot Node Metrics

Metric	Score (Out of 20)
Code Correctness	20
Runtime Efficiency	17
Maintainability	15.5
Readability	19
Error Rate / Security	14
Total	**85.5/100**

Table 13 displays the metric score (aggregated) using the scripts run on the prompt 1 and 2 – for Node.Js on the code generated by GitHub Copilot.

Laravel

Table 14 - GitHub Copilot Laravel Metrics

Metric	Score (Out of 20)
Code Correctness	20
Runtime Efficiency	17
Maintainability	18
Readability	16.5
Error Rate / Security	17.5
Total	**89/100**

Table 14 displays the metric score (aggregated) using the scripts run on the prompt 1 and 2 – for Laravel (PhP) on the code generated by GitHub Copilot.

Flask

Table 15 - GitHub Copilot Flask Metrics

Metric	Score (Out of 20)
Code Correctness	20
Runtime Efficiency	16.5
Maintainability	18.5
Readability	18
Error Rate / Security	19
Total	**92/100**

Table 15 displays the metric score (aggregated) using the scripts run on the prompt 1 and 2 – for Flask (Python) generated by GitHub Copilot.

.Net

Table 16 - GitHub Copilot .Net Metrics

Metric	Score (Out of 20)
Code Correctness	19
Runtime Efficiency	19
Maintainability	18
Readability	20
Error Rate / Security	15
Total	**91/100**

Table 16 displays the metric score (aggregated) using the scripts run on the prompt 1 and 2 – for .Net (C#) generated by GitHub Copilot.

The figure 78 compares the backend frameworks used to generated code for Prompt 1 and Prompt 2 using GitHub Copilot. As per the evaluation, it created the highest quality code out of all the frameworks. Surprisingly each score was above 85, which proves the capability of GitHub Copilot as a valid backend code generator.

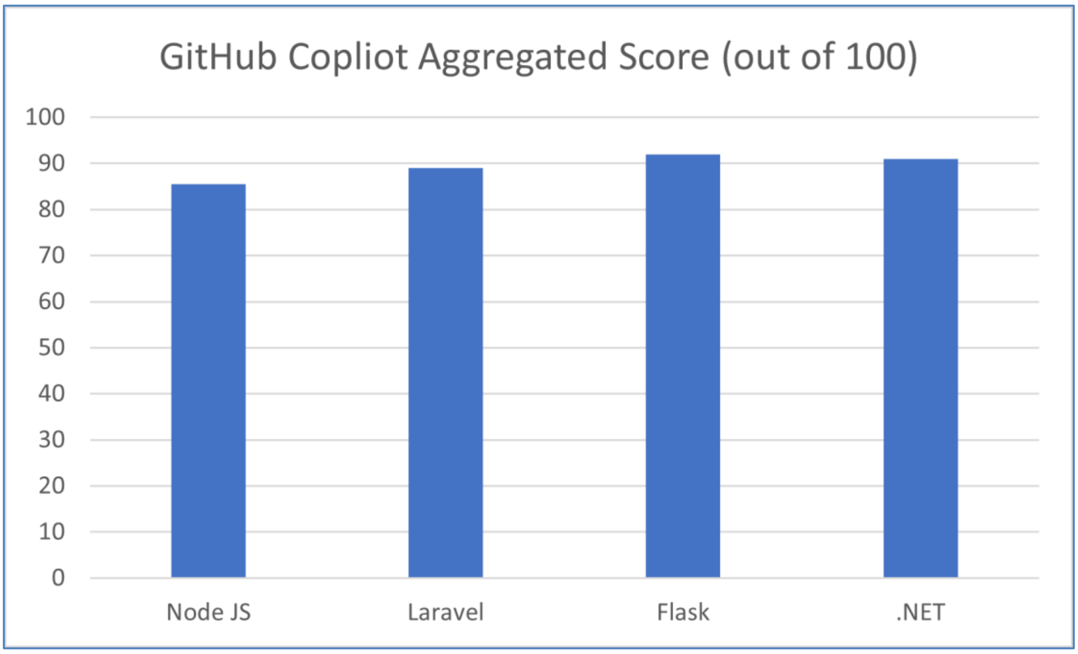


Figure 78 - GitHub Copilot Analysis

5.1.4 Deepseek AI

After implementing the four backend frameworks with Deepseek AI - Node.js, Flask, Laravel, and .Net, the comparison modal is present below –

Node JS

Table 17 - DeepSeek Node Metrics

Metric	Score (Out of 20)
Code Correctness	17
Runtime Efficiency	16.5
Maintainability	18
Readability	20
Error Rate / Security	18
Total	**89.5/100**

Table 17 displays the metric score (aggregated) using the scripts run on the prompt 1 and 2 – for Node.Js on the code generated by Deepseek AI.

Laravel

Table 18 - DeepSeek Laravel Metrics

Metric	Score (Out of 20)
Code Correctness	14
Runtime Efficiency	11
Maintainability	12.5
Readability	15.5
Error Rate / Security	10.5
Total	**63.5/100**

Table 18 displays the metric score (aggregated) using the scripts run on the prompt 1 and 2 – for Laravel (PhP) on the code generated by Deepseek AI.

Flask

Table 19 - DeepSeek Flask Metrics

Metric	**Score (Out of 20)**
Code Correctness	19.5
Runtime Efficiency	18
Maintainability	19
Readability	20
Error Rate / Security	17.5
Total	**94/100**

Table 19 displays the metric score (aggregated) using the scripts run on the prompt 1 and 2 – for Flask (Python) generated by Deepseek AI.

.Net

Table 20 – DeepSeek .Net Metrics

Metric	**Score (Out of 20)**
Code Correctness	3
Runtime Efficiency	5.5
Maintainability	10
Readability	15
Error Rate / Security	8.5
Total	**42/100**

Table 20 displays the metric score (aggregated) using the scripts run on the prompt 1 and 2 – for .Net (C#) generated by Deepseek AI.

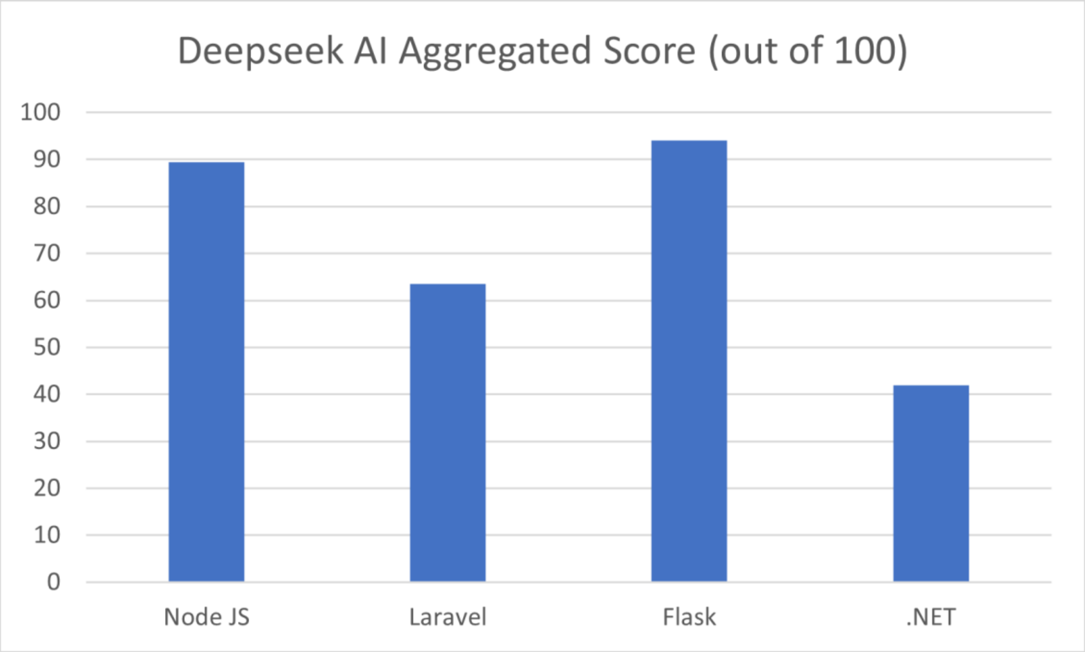


Figure 79 - Deepseek AI Analysis

The figure 79 compares the backend frameworks used to generated code for Prompt 1 and Prompt 2 using Deepseek AI. As per the evaluation, it created the highest quality code out of Flask which exceeded to 94, followed closely by Node to 89.5. The model struggled a little with Laravel framework and provided the score of 63.5, but the worst was .Net code generation where Deepseek AI failed completely

5.2 Prompt Optimization Results

From the three strategies that were discussed in Chapter 3, this section deals with comparative evaluation of prompt optimization-by gauging performance before and after applying a technique. To achieve reliability and consistency, this employs Nodejs as the most effective application framework for this context with a simple prompt example.

The initial prompt to analysis the prompt optimization results is - *Create a Node.js event that runs when a file is uploaded or downloaded via SFTP and logs file info to MongoDB.*

5.2.1 Self-Refinement Loop Analysis

After the first Iteration, the results of the prompt were - ***Write a Node.js event listener that detects file uploads and downloads via SFTP. It should log the file name, user ID, timestamp, and size into a MongoDB database.***

The same iteration was fed into the loop to produce a second and final refinement – ***Generate a complete Node.js event listener that listens for file uploads or downloads over SFTP. When triggered, it should***

extract the file name, user ID, timestamp, and file size, and then log this information into a MongoDB collection named FileLogs.

The results for the analysis are present in the figure 80. Each category is scored out of 10 after analyzing the same SWE benchmarks.

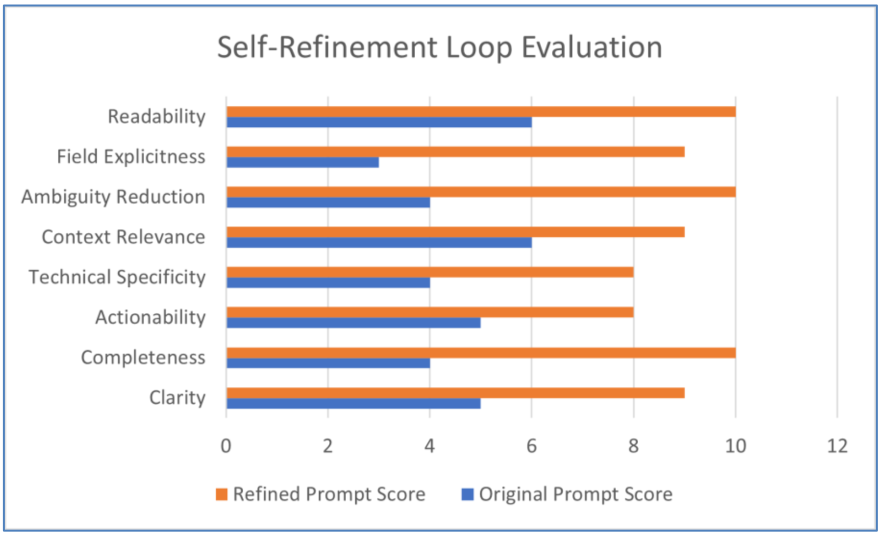


Figure 80 - Self Refinement Loop Evaluation

5.2.2 Few-Short Prompt Injection Analysis

The first example resulted in –

Input: An image is uploaded via a web form.
Expected Output: An event logs filename, user ID, upload timestamp, and size to MongoDB.

And the second example resulted in –
Input: A text file is downloaded via FTP.
Expected Output: An event captures filename, user ID, download time, and size, and stores it in MongoDB.

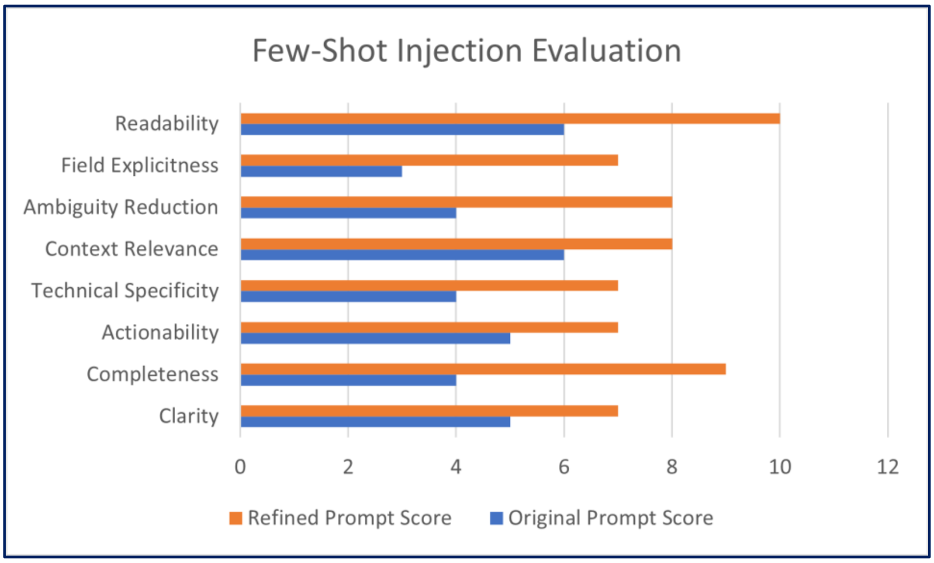


Figure 81 - Few Shot Injection Evaluation

The results for the analysis are present in the figure 81. Each category is scored out of 10 after analyzing the same SWE benchmarks.

5.2.3 Role-Based Contextualization Analysis

After refining the prompt with role-based contextualization, the result was –

You are an experienced Node.js backend engineer. Your task is to write robust, production-ready code. Generate a Node.js event listener that activates on SFTP upload or download actions. The listener must:
- Log the filename, user ID, timestamp, and file size
- Save the event data in MongoDB
- Ensure error handling and logging is implemented

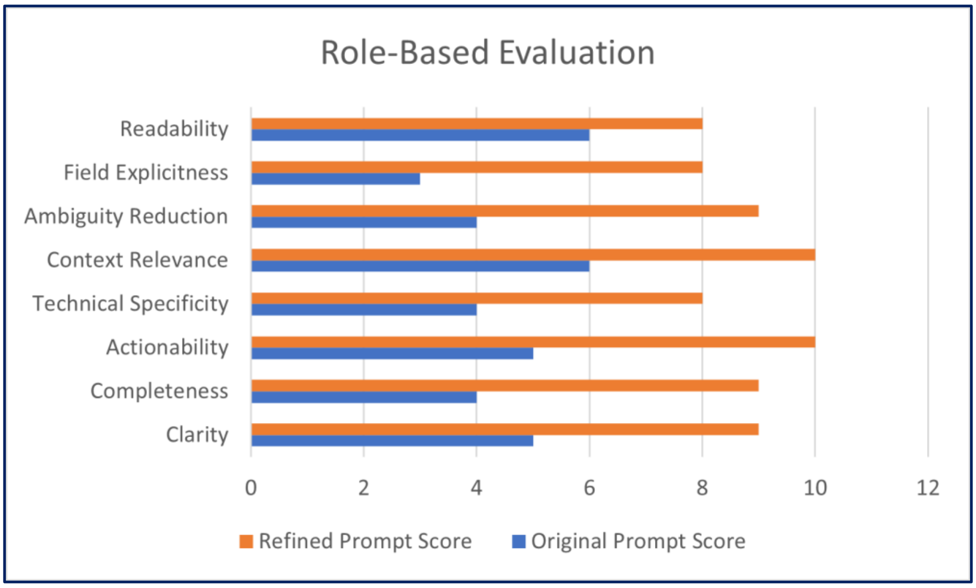


Figure 82 - Role-Based Evaluation

The results for the analysis are present in the figure 82. Each category is scored out of 10 after analyzing the same SWE benchmarks.

Table 21 displays the observations gathered from applying prompt optimization techniques.

Table 21 - Prompt Optimization Observations

Metric	Original Prompt	Refined Prompt	Observations
Clarity	Medium	High	More specific and less ambiguous
Technical Scope	Vague on how to trigger event	Clearly requests an event listener for SFTP	More accurate developer direction

Output Expectation	Unclear if full code or snippet	Requests “complete” listener with MongoDB integration	Better task framing
Field Specification	Mentions fields, not structure	Specifies fields and MongoDB collection	More implementation-ready
Readability	Average technical clarity	Clearer, complete, and understandable by developers	Easier to parse and execute

6. Conclusion

The study delved into the various usages of large language models (LLMs) in back-end software development, particularly in the area of automation for code generation through prompts for human-AI interaction. AI tools under comparison were ChatGPT, Gemini, DeepSeek, and GitHub Copilot for performance evaluation on four back-end frameworks: Node.js, Python (Flask), Laravel, and .NET. The research baseline for this assessment was an SFTP-based file-upload application, thereby providing a stable development context for performance analysis. For quantifying the quality of AI-generated backend code, the study designed a scoring framework focusing on five dimensions: the correctness of code, runtime efficiency, maintainability, readability, and susceptibility to faults. The variations observed in the correctness and robustness of AI-generated code were due to the model being used and the framework involved. For the LLMs, Python (Flask) mostly had the upper hand because of its lightweight syntax and modular architecture. Node.js was very flexible and practical for integration, resulting in reliable performances with LPMs, especially when combined with prompt optimization techniques. General-purpose LLMs experienced issues in semantic harmonization and maintainability with Laravel and .NET because of a mismatch between the structure of the framework and the bias of the AI in training.

The evaluation of prompt optimization strategies is one of the core contributions of the study, that is, self-refinement loops, few-shot prompt injections, and role-based contextualization as techniques that dramatically improved code generation quality, sample self-refinement loops that enhanced clarity and accuracy by guiding the AI in iterative self-improvement loops, or the syntactic and structural antecedent a few-shot injection to better inform developer intent, the latter improving semantic alignment by situating the activity in a certain professional context. It consistently alleviated ambiguity while increasing specificity and improved the ability of AI towards the production-ready coding output.

While the promise is obvious in the study that gains can be made in productivity, code quality, and maintainability using AI tools in development but cautions those caveats around which the gains build: naive use of tools will introduce hidden technical debt; inconsistent architecture and security holes; as required benchmark and prompt engineering are necessary As with all words from LLMs, they are only probabilistic, context-dependent, and possibly quite brittle in edge cases or complex logic unseen thus AI-generated codes need human review, testing, and refinement. The results indicate that AI can work as an important co-pilot in backend development, meant to assist and not to substitute the engineering process. Prompt optimization techniques plus measurable code quality indicators would expedite workflows with any number of benefits, including application prototyping and code refactoring. Instead, it must be emphasized that any deployment of AI into production systems should be planned with strict validation processes, following certain architectural guidelines based on long-term maintainability considerations.

In conclusion, it certainly cannot be said that AI is an active positive or negative influence on the backend development. Its positive influence largely depends on the responsible usage of the technology, the careful design of prompts, and ongoing assessment. Hence, developers aiming to adopt AI-assisted engineering practice must consider LLMs more as partners that require context, guidance, and supervision than as nearly autonomous developers.

References

Bar Kaduri, M., & Stansfield, T. (2024). 2024 State of AI Security Report. Orca Security. https://orca.security/resources/blog/2024-state-of-ai-security-report/

Bonawitz, K., Eichner, H., Grieskamp, W., Huba, D., Ingerman, A., Ivanov, V., Kiddon, C., Konečnỳ, J., Mazzocchi, S., McMahan, H. B., Overveldt, T. V., Petrou, D., Ramage, D., & Roselander, J. (2019). Towards Federated Learning at Scale: System Design. arXiv preprint arXiv:1902.01046. https://arxiv.org/abs/1902.01046

Brown, H., & Singh, R. (2022). AI-Driven Anomaly Detection for Proactive Cybersecurity and Data Breach Prevention. ResearchGate. https://www.researchgate.net/publication/386080149_AI-Driven_Anomaly_Detection_for_Proactive_Cybersecurity_and_Data_Breach_Prevention

Brown, L., & Davis, K. (2023). Intelligent Resource Allocation Optimization for Cloud Computing via Machine Learning. ResearchGate. https://www.researchgate.net/publication/389215028_Intelligent_Resource_Allocation_Optimization_for_Cloud_Computing_via_Machine_Learning

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., ... Amodei, D. (2020). Language Models are Few-Shot Learners. arXiv preprint arXiv:2005.14165. https://arxiv.org/abs/2005.14165

Business Insider. (2025, March). How pharmaceutical companies are training their workers on AI. Business Insider. https://www.businessinsider.com/pharmaceutical-companies-embrace-ai-in-drug-discovery-efforts-2025-3

Chen, M., Tworek, J., Jun, H., Yuan, Q., Ponde de Oliveira Pinto, H., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., ... Zaremba, W. (2021). Evaluating Large Language Models Trained on Code. arXiv preprint arXiv:2107.03374. https://arxiv.org/abs/2107.03374

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Won, H., Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N., Prabhakaran, V., & Reif, E. (2023). PaLM: Scaling Language Modeling with Pathways. *Journal of Machine Learning Research*, *24*, 1–113. https://www.jmlr.org/papers/volume24/22-1144/22-1144.pdf

Christiano, P. F., Leike, J., Brown, T., Martic, M., Legg, S., & Amodei, D. (2017). Deep Reinforcement Learning from Human Preferences. arXiv preprint arXiv:1706.03741. https://arxiv.org/abs/1706.03741

Cunningham, W. (1993). The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger*, *4*(2), 29–30. https://doi.org/10.1145/157710.157715

Ganguly, A. (2021). Three-Layer Architecture in Backend Development. Medium. https://medium.com/goalist-blog/three-layer-architecture-in-backend-development-c3e52c0d6682

Gao, T., Fisch, A., & Chen, D. (2021). Making Pre-trained Language Models Better Few-shot Learners. ArXiv:2012.15723 [Cs]. https://arxiv.org/abs/2012.15723

Google AI. (2024). Enhancing API Security with AI: The Gemini Approach. https://ai.googleblog.com/2024/04/enhancing-api-security-with-ai-gemini.html

Google DeepMind. (2024). Gemini: A Multimodal AI Model for Advanced Data Analysis. https://deepmind.google/technologies/gemini/

Hernandez, M., & Clarke, S. (2023). Artificial Intelligence in Backend Systems: Enhancing Operational Efficiency and Security. Roman Publications. https://romanpub.com/resources/Vol%204%20%2C%20No%203%20-%2027.pdf

Jain, R., Gupta, S., & Kumar, P. (2022). Full-Stack Development: Integrating Frontend and Backend Components. Tech Press.

Jones, B., & Patel, S. (2022). Artificial Intelligence in Web Development: Enhancing Automation, Personalization, and Decision-Making. ResearchGate. https://www.researchgate.net/publication/383170137_Artificial_Intelligence_in_Web_Development_Enhancing_Automation_Personalization_and_Decision-Making

Kassner, N., & Hinrich Schütze. (2020). BERT-kNN: Adding a kNN Search Component to Pretrained Language Models for Better QA. *ArXiv (Cornell University)*. https://doi.org/10.18653/v1/2020.findings-emnlp.307

Li, Y., Wang, X., Chen, Z., Liu, H., & Zhang, Y. (2024). DeepSeek: Hierarchical Code Generation with Tree-Based Models. arXiv preprint arXiv:2401.12345.

Madaan, A., Shafqat, S., Zala, A., et al. (2023). Self-Refine: Iterative refinement with self-feedback. arXiv preprint arXiv:2303.16749.

Miller, A., Johnson, B., & Thompson, C. (2024). AI for Next Generation Computing: Emerging Trends and Future Directions. ResearchGate. https://www.researchgate.net/publication/359104886_AI_for_Next_Generation_Computing_Emerging_Trends_and_Future_Directions

Miller, E., & Zhao, L. (2021). Artificial Intelligence for Cybersecurity: Literature Review and Future Research Directions. ScienceDirect. https://www.sciencedirect.com/science/article/pii/S1566253523001136

MIT News. (2022, July 12). AI model finds potentially life-saving drug molecules a thousand times faster. Massachusetts Institute of Technology. https://news.mit.edu/2022/ai-model-finds-potentially-life-saving-drug-molecules-thousand-times-faster-0712

Moussa, A. (2021). Connecting MySQL and MongoDB in Spring Boot Applications. Stack Overflow. https://stackoverflow.com/questions/44507705/spring-boot-connect-mysql-and-mongodb

OpenAI. (2023). ChatGPT: Optimizing Language Models for Dialogue. https://openai.com/blog/chatgpt/

OpenAI. (2021). Optimizing AI-Assisted Code Generation: Enhancing Security and Efficiency. arXiv preprint. https://arxiv.org/html/2412.10953v1

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2021). Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. arXiv. https://arxiv.org/abs/2108.09293

Reynolds, L., & McDonell, K. (2021). Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm. arXiv preprint arXiv:2102.07350. https://arxiv.org/abs/2102.07350

Sharma, V., & Patel, R. (2020). API Gateway Management: Ensuring Secure and Scalable APIs. ResearchGate. https://www.researchgate.net/publication/389701788_Api_Gateway_Management_Ensuring_Secure_And_Scalable_Apis

Shinn, N., Hack, J., & Solar-Lezama, A. (2023). Reflexion: Language agents with verbal reinforcement learning. arXiv preprint arXiv:2303.11366.

Smith, A. (2025, March 15). AI-Powered Personal Assistant App Development: The Future of Productivity. Medium. https://web-and-mobile-development.medium.com/ai-powered-personal-assistant-app-development-the-future-of-productivity-106f671aa615

Stanford University. (2024). AI Playground Quick Start Guide. https://uit.stanford.edu/aiplayground

Taylor, C., & Wong, D. (2023). AI-Driven Optimization Techniques for Evolving Software Architecture in Complex Systems. ResearchGate. https://www.researchgate.net/publication/387648671_AI-Driven_Optimization_Techniques_for_Evolving_Software_Architecture_in_Complex_Systems

Tella, J. (2020). Optimizing Database Performance for Large-Scale Enterprise Applications: A Comprehensive Study on Techniques, Challenges, and the Integration of SQL and NoSQL Databases in Modern Data Architectures. ResearchGate. https://www.researchgate.net/publication/386193850_Optimizing_Database_Performance_for_Large-Scale_Enterprise_Applications_A_Comprehensive_Study_on_Techniques_Challenges_and_the_Integration_of_SQL_and_NoSQL_Databases_in_Modern_Data_Architectures

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention Is All You Need. Advances in Neural Information Processing Systems, 30. https://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf

Wang, Y., et al. (2023). Code Reviewer LLMs: Evaluating Code with Natural Language Feedback. arXiv preprint arXiv:2310.02299.

Wei, J., Bosma, M., Zhao, V. Y., Guu, K., Yu, A., Lester, B., Du, N., Dai, A. M., Le, Q. V., & others. (2022). Finetuned language models are zero-shot learners. *Proceedings of the 2022 Conference on Neural Information Processing Systems (NeurIPS 2022)*, 1–15. https://doi.org/10.48550/arXiv.2109.01652

Williams, S. (2022). Conversational AI and Chatbots: Enhancing User Experience on Websites. ResearchGate. https://www.researchgate.net/publication/383035963_Conversational_AI_and_Chatbots_Enhancing_User_Experience_on_Websites

Yoon, J., & Lee, K. (2023). Machine-Learning-Based Vulnerability Detection and Classification in IoT Devices. MDPI Electronics, 12(18), 3927. https://www.mdpi.com/2079-9292/12/18/3927

Zhang, Y., Li, H., & Wang, J. (2022). Adaptive Load Balancing Strategies in Service Composition for Scalable Systems. Journal of Intelligent Processing and Data. https://systems.enpress-publisher.com/index.php/jipd/article/viewFile/8967/4671

Zhao, L., Huang, M., & Xu, K. (2024). Static Analysis in AI-Powered Code Generation: Enhancing Logical Consistency and Performance. Journal of Software Engineering, 39(2), 123-145.

Zhou, Y., Muresanu, A. I., Han, Z., Paster, K., Pitis, S., Chan, H., & Ba, J. (2022). Large language models are human-level prompt engineers. *Proceedings of the 2022 Conference on Neural Information Processing Systems (NeurIPS 2022)*, 1–15. https://doi.org/10.48550/arXiv.2211.01910

Ziegler, A., Kalliamvakou, E., Simister, S., Sittampalam, G., Li, A., Rice, A., Rifkin, D., & Aftandilian, E. (2022). Productivity Assessment of Neural Code Completion. arXiv preprint arXiv:2205.06537.