An empirical comparison of ChatGPT-o3 with DeepSeek-R1

Scope: Code Generation

NIYATI MALIK, NEERAJ SOLANKI, RAUNAK SINGH, AKHIL NAIR, University of Illinois Chicago

1 Introduction

1.1 Original Goals of the Project

The original objective of this project was to systematically evaluate the capabilities and performance of two state-of-the-art large language models—ChatGPT-4o and DeepSeek-R1—in the domain of **Code Generation and Test Case Generation**. Specifically, the goal was to examine how effectively these models could produce maintainable, standards-compliant, and production-grade Java Spring Boot code, along with relevant test cases, when prompted to implement various components of a modular, microservices-based architecture similar to that of a LeetCode-style coding platform.

1.2 Modified Goals

No significant goal modification occurred. However, based on faculty guidance, we narrowed the scope to focus exclusively on **Code Generation** in order to study it in greater depth. Additionally, we updated the model selection from GPT-4o to **GPT-3.5-turbo-instruct (also known as GPT-o3)**, since it is better aligned with DeepSeek-R1 as a reasoning-oriented model. We also expanded our evaluation framework to incorporate human judgment on architectural quality and introduced a use-case-based model recommendation strategy.

1.3 Assumptions

- **Platforms used:** ChatGPT (GPT-o3), DeepSeek-Coder R1. Both models were accessed through their respective web interfaces and prompted in a controlled setting to ensure consistency.
- **Programming language:** Java. We chose Java for its strong ecosystem, robust type system, and relevance in backend microservice development.
- **Frameworks:** Spring Boot (v3.x), Maven (v3.4.4). These were selected to reflect standard industry practices for building scalable RESTful services.
- **Database:** As recommended by the LLM as a part of the Microservice code. (MongoDB, MySQL).
- **Operating Environment:** Windows 11 and MacOS (for local testing and development), GitHub.
- **Code Evaluation Tooling:** SonarQube (Community Edition) was used to quantify code quality and static analysis metrics.
- **Assumptions:**
  - Prompt responses were evaluated without fine-tuning or additional model training.
  - The generated code would reflect a baseline understanding of Spring Boot conventions, REST principles, and microservice separation.

Author's Contact Information: Niyati Malik, Neeraj Solanki, Raunak Singh, Akhil Nair, University of Illinois Chicago.

1.4 Definitions

- **Cyclomatic Complexity:** Measures the number of linearly independent paths in the code. Higher values indicate more complex logic and more effort needed for testing and maintenance.
- **Cognitive Complexity:** Captures how difficult code is to understand for a human, considering nesting, conditionals, and jumps in logic. Focuses on mental effort rather than just structural complexity.
- **Code Smells:** Surface-level indicators of deeper problems in code quality, such as long methods, duplicated code, or poor naming. They signal maintainability and design concerns.
- **Vulnerabilities:** Weaknesses in code that could expose the application to security threats (e.g., injection flaws, insecure endpoints).
- **Microservice Architecture:** An approach where functionality is divided into loosely coupled, independently deployable services with dedicated responsibilities.
- **Separation of Concerns:** A design principle where different parts of the program handle distinct responsibilities (e.g., controller, service, repository layers are kept isolated).
- **Modularity:** A structural principle where the application is broken into smaller, self-contained units that are easier to test, maintain, and scale independently.
- **RESTful APIs:** APIs that follow standard HTTP methods (GET, POST, PUT, DELETE) and resource-based URL structures to support stateless client-server communication.
- **Naming Conventions:** The practice of using meaningful, consistent names for classes, methods, and variables to enhance readability and collaboration.
- **Prompt-Based Evaluation:** A qualitative judgment of how accurately and thoroughly the model implemented the intent of a prompt, including logic, correctness, and semantic coverage.
- **Chain-of-Thought Prompting:** A prompting technique where the model is guided to break down the problem into a sequence of intermediate reasoning steps before generating the final code. This approach encourages logical decomposition and can improve the correctness and structure of generated code, especially for complex tasks.
- **Average Cyclomatic Complexity per Method:** A derived metric calculated by dividing the total cyclomatic complexity of a microservice by its number of methods. This provides a normalized measure of complexity, helping compare services of different sizes by focusing on the average complexity of individual methods.
- **Average Cognitive Complexity per Method:** A normalized metric that divides the total cognitive complexity of a microservice by its number of methods. It reflects the average mental effort required to understand each method, offering a finer-grained view of understandability across services of varying sizes.
- **Test Case Generation (Future Scope):** The ability of LLMs to autonomously create unit tests or integration tests to verify the correctness of generated code.

1.5 Summary of Approach

We designed a modular, backend system resembling the architecture of a LeetCode-like coding practice platform, which was decomposed into four core microservices: *AuthService*, *ProfileService*, *QuestionService*, and *CodeService*. Each microservice was associated with a specific functional responsibility—authentication, user profile management, question management, and code execution respectively. These microservices were implemented independently by each team member to ensure modularity and parallelism. Identical prompts, detailing the requirements and expected functionalities, were provided to both ChatGPT and DeepSeek for each service. The generated code from both models was then systematically collected and subjected to comprehensive evaluation using a combination of automated tools like SonarQube for static analysis and human inspection for architecture quality, maintainability, and adherence to software engineering principles.

1.6 Project Deliverables

- Working microservices for each backend module, generated by both ChatGPT and DeepSeek
- Evaluation framework and scripts
- SonarQube-based code quality reports
- Final comparative analysis document and slides

1.7 Contribution of Individual Team Members

- **Niyati:** QuestionService generation
- **Neeraj:** AuthService generation
- **Raunak:** CodeService generation
- **Akhil:** ProfileService generation
- **All Members:** Evaluation, analysis, documentation, and presentation

2 Background Research

2.1 Primary Research

Our primary research focused on hands-on experimentation with software tools and direct interaction with domain experts:

- We tested multiple versions of SonarQube (Community and Developer Editions) to identify the right setup for automated code quality analysis. We also experimented with SonarLint.
- We tried prompting ChatGPT, as well as DeepSeek on their official platforms with multiple prompt variants to assess behavior consistency.
- Discussions with our course instructor and peer groups helped refine the scope of the project, including the decision to isolate code generation (excluding test generation), and to use SonarQube for uniform evaluation.
- Exploratory use of GitHub Codespaces and local environments helped us standardize the development workflow across operating systems (Windows, macOS).

2.2 Secondary Research

We conducted a thorough literature review and referenced several relevant papers and technical resources:

- **Is Your Code Generated by ChatGPT Really Correct?** [6]: This NeurIPS 2023 paper introduced *EvalPlus*, an automated framework to evaluate LLM-generated code through mutation-based test augmentation, revealing test inadequacies in benchmarks like HumanEval.
- **ChatGPT vs. DeepSeek: A Comparative Study on AI-Based Code Generation** [4]: This work compares ChatGPT and DeepSeek on Python coding tasks using online judges, concluding DeepSeek shows superior correctness while ChatGPT is more concise and efficient.

- **Comprehensive Evaluation of LLMs Across 10 Languages** [5]: This multilingual benchmark analyzed LLM-generated code and highlighted significant performance differences across languages and tasks.
- **Across 10 Programming Languages: Evaluating Code Quality and Maintainability** [1]: This paper focused on static analysis of LLM-generated code using tools like Flake8 and Pylint, emphasizing maintainability and code smells across languages.
- **Evaluating the Correctness of Code Generated by LLMs** [2]: Focused on the limitations of correctness in generated code and emphasized the importance of rigorous, domain-specific test cases for fair model comparison.
- **HumanEval+ and Benchmark Robustness** [3]: Introduced HUMANEVAL+ and mutation-based test case expansion to uncover incorrect outputs undetected by traditional test suites, highlighting flaws in prior evaluations.
- Official documentation of SonarQube and Spring Boot.
- Blogs and tutorials on cyclomatic and cognitive complexity, microservice best practices, and RESTful API design.

2.3 Most Valuable Resources

The most valuable resources were:

- **Empirical studies** such as the ChatGPT vs. DeepSeek paper, which inspired our experiment design.
- **Hands-on testing** with SonarQube, which enabled a quantitative and repeatable evaluation of code quality.
- **Discussions with the instructor**, which led to a clearer, more focused scope and helped align our approach with realistic research expectations.

3 Approach

3.1 Strategy Followed

To systematically compare ChatGPT-o3 and DeepSeek-R1 on backend microservice code generation tasks, we designed a multi-stage evaluation pipeline involving prompt engineering, model invocation, output validation, static and manual analysis, and final synthesis. This process was tailored to closely simulate a professional backend development workflow, ensuring realism.

Our strategy unfolded in the following major phases:

(1) **Microservice Decomposition:** We conceptualized a backend system inspired by platforms like LeetCode, and divided it into four logical microservices—*AuthService*, *ProfileService*, *QuestionService*, and *CodeService*. Each was scoped to encapsulate distinct business logic such as authentication, user profile CRUD, question filtering and fetching, and code evaluation. This modular design enabled isolated analysis and streamlined testing.

(2) **Prompt Design and Engineering:** For each service, detailed prompts were constructed, specifying the requirements, expected endpoints, and technologies to use (e.g., Java, Spring Boot). We made deliberate use of prompting best practices including step-by-step task breakdown, role definition ("You are a Java backend developer..."), and iterative refinement. Prompts were reused verbatim across models to maintain a consistent testing baseline.

(3) **Model Invocation and Code Collection:** We used the official interfaces of ChatGPT and DeepSeek, manually submitting prompts and collecting code. Every major response was timestamped, logged, and stored in our GitHub repository. We ensured clean project builds in IntelliJ IDEA to validate code correctness and structure. Additional helper code or dependencies suggested by the LLMs were noted and incorporated.

(4) **Iterative Prompting and Functional Verification:** Code was not always correct on the first attempt. Hence, we adopted a controlled loop of prompting → evaluation → re-prompting. We continued this loop until the generated code compiled without errors and fulfilled the intended functional behavior. A Google Sheet was used to log prompt counts and reasons for retries (e.g., missing controller annotations, database connection issues, bad DTO usage).

(5) **Tracking Prompt Efficiency:** We treated the number of prompts required to reach a working solution as a proxy for model efficiency. Lower prompt counts indicated better contextual understanding and fewer hallucinations. This data was later aggregated and visualized across services.

(6) **Static Code Analysis with SonarQube:** Each working microservice was analyzed using SonarQube Community Edition running on localhost. We used the Maven Sonar plugin to integrate analysis with our build process. SonarQube reports were used to extract metrics like Cyclomatic Complexity, Cognitive Complexity, Code Smells, and Vulnerabilities.

(7) **Manual Architectural Evaluation:** Automated metrics do not capture system-level design nuances. We performed human-led code walkthroughs to check separation of controller, service, and repository layers, proper use of REST conventions, DTO usage, and externalized configurations.

(8) **Derived Metrics (Novel Contribution):** We proposed and computed normalized complexity scores for all Microservices for each LLM:
   - *Average Cyclomatic Complexity per Method* = Total Cyclomatic Complexity / Number of Methods
   - *Average Cognitive Complexity per Method* = Total Cognitive Complexity / Number of Methods

   These helped in fairer comparisons between microservices of different sizes.

(9) **Result Compilation and Model Comparison:** Each service's output from both models was compiled into a comparative matrix. We used bar graphs to highlight trade-offs. Recommendations were then derived based on scenario.

(10) **Use-Case-Based Model Suggestion:** Our final step involved mapping use cases to model strengths.

3.2 Motivation for Our Strategy

The choice of strategy was influenced by the need to balance objectivity, realism, and practicality.

- **Prompt Engineering Realism:** We treated each LLM like a junior developer—providing one prompt at a time and reviewing results iteratively. This mirrors real-world usage where developers give iterative prompts over time to achieve desired results.
- **Code-Level Evaluation:** By using SonarQube and metrics like Cyclomatic and Cognitive Complexity, we could evaluate not just whether the code worked, but how clean, maintainable, and scalable it was. These metrics are used by teams in production pipelines.
- **Microservice Architecture as Testbed:** Choosing a modular microservice system ensured isolated testing of capabilities. It also gave us insights into how well LLMs could reason across related components—such as whether the DTO matched what the controller and service expected.
- **Human Insight:** Automated tools have limitations in assessing software design quality and architectural decisions. Hence, we supplemented static analysis with structured human evaluation, inspired by qualitative rubrics used in software design reviews.

3.3 Team Collaboration and Task Division

The project was executed collaboratively with consistent weekly meetups, real-time discussion, and progress tracking.

- **Niyati Malik:** Developed *QuestionService*, designed prompt templates for filtering logic, and analyzed cognitive complexity results.
- **Neeraj Solanki:** Implemented *AuthService*, and contributed to code quality script automation and architectural diagrams.
- **Raunak Singh:** Built *CodeService*, set up SonarQube locally and maintained project structure across services.
- **Akhil Nair:** Created *ProfileService*, and helped standardize evaluation results.
- **Shared Responsibilities:** Beyond individual microservices, all team members collaborated closely on every phase of the project. Prompt design and refinement for both ChatGPT and DeepSeek were discussed collectively to ensure fairness and clarity. Evaluation rubrics were co-developed and revised through multiple review cycles to align with course objectives. Weekly meetings—mostly held in the library—helped maintain steady progress and allowed for real-time troubleshooting.
  We also jointly carried out manual inspection of the generated code to assess architecture quality, modularity, and RESTful principles. Responsibilities such as result analysis, SonarQube dashboard interpretation, documentation, and presentation development were distributed organically based on task load. Collaboration and version control were maintained using shared Google Docs for reports and slides, GitHub for code organization, and a shared spreadsheet to track the number of prompts, model responses, and evaluation scores.

4 Implementation

4.1 Design of the Software

The hypothetical software system on the basis of which we were evaluating the models was based on a modular microservice architecture, similar to real-world scalable platforms such as LeetCode or HackerRank. It was decomposed into four core services:

- **AuthService:** Responsible for user authentication, registration, and JWT token generation/validation.
- **ProfileService:** Manages user profile information, including CRUD operations for personal details and user settings.
- **QuestionService:** Handles storage, filtering, and retrieval of coding problems categorized by difficulty and topic.
- **CodeService:** Accepts user-submitted code, interacts with the language execution environment, and returns results.

Each service was developed independently using Java Spring Boot, adhering to the MVC design pattern. RESTful APIs ensured modularity and clear separation of concerns. DTOs (Data Transfer Objects) were used to decouple internal models from API payloads, and `application.properties` externalized environment-specific configurations.

4.2 External Components Used

- **Java 17:** The programming language used for implementing all backend microservices. Java was selected due to its widespread adoption in enterprise environments, robust type system, and seamless compatibility with the Spring ecosystem.

- **Spring Boot 3.x:** This modern framework was chosen for rapid development of production-grade microservices. It offers built-in dependency injection, REST controller abstractions, and auto-configuration features, reducing boilerplate code and improving modularity. Each microservice was structured using the Model-View-Controller (MVC) pattern facilitated by Spring Boot annotations.
- **Maven 3.4.4:** Used as the build automation tool and dependency manager. It helped maintain version control over libraries (e.g., Spring Boot Starter, MongoDB Connector) and allowed uniform builds across development machines using the ‘pom.xml‘ configuration file.
- **MongoDB / MySQL:** NoSQL and SQL database technologies were incorporated based on the output generated by the models. MongoDB was commonly selected by both ChatGPT and DeepSeek for schema-flexible services (e.g., QuestionService), whereas MySQL was suggested where relational constraints (ProfileService) were needed. We validated and integrated the suggested database into each service accordingly.
- **SonarQube (Community Edition):** An open-source static analysis platform used to evaluate code quality and collect empirical metrics such as Cyclomatic Complexity, Cognitive Complexity, Code Smells, and Security Vulnerabilities. SonarQube also provided a web-based dashboard to visualize and compare model outputs.
- **GitHub and IntelliJ IDEA:** IntelliJ IDEA (on both Windows and macOS) served as our primary integrated development environment (IDE) for writing, building, and testing the Java Spring Boot microservices. After local development and verification, all code was pushed to GitHub for version control, collaboration, and centralized storage. This setup ensured smooth coordination among team members and reproducibility of builds across machines.

4.3 Design of Empirical Studies

Our empirical evaluation pipeline included the following stages:

- **Prompt-Based Code Generation:** Identical microservice prompts were given to both models. Iterative prompting continued until the code compiled and ran as intended.
- **Prompt Tracking:** We recorded the number of prompts needed to reach a working state for each model and microservice, serving as a proxy for prompt efficiency.
- **Static Analysis Metrics:** SonarQube provided Cyclomatic Complexity, Cognitive Complexity, Code Smells, and Vulnerabilities.
- **Derived Metrics:**
  - **Cyclomatic Complexity per Method (CCPM):** $\frac{\text{Total Cyclomatic Complexity per Microservice}}{\text{Number of Methods in Microservice}}$
  - **Cognitive Complexity per Method (CogCPM):** $\frac{\text{Total Cognitive Complexity per Microservice}}{\text{Number of Methods in Microservice}}$
- **Qualitative Evaluation:** Architecture quality, naming practices, and modular design were judged manually using structured rubrics.
- **Model Recommendation Strategy:** Based on results, we proposed when to prefer one model over another based on project needs such as readability or simplicity.

5 Results

In this section, we present the empirical findings of our comparative study between ChatGPT-o3 and DeepSeek-R1 on backend code generation tasks. The generated microservices—AuthService, ProfileService, QuestionService, and CodeService—were evaluated using multiple dimensions of code quality and architecture.

5.1 Empirical Results

5.1.1 Cyclomatic Complexity. Cyclomatic Complexity measures the number of independent paths in the code. Lower values are generally preferred as they indicate simpler control flow and easier testability.

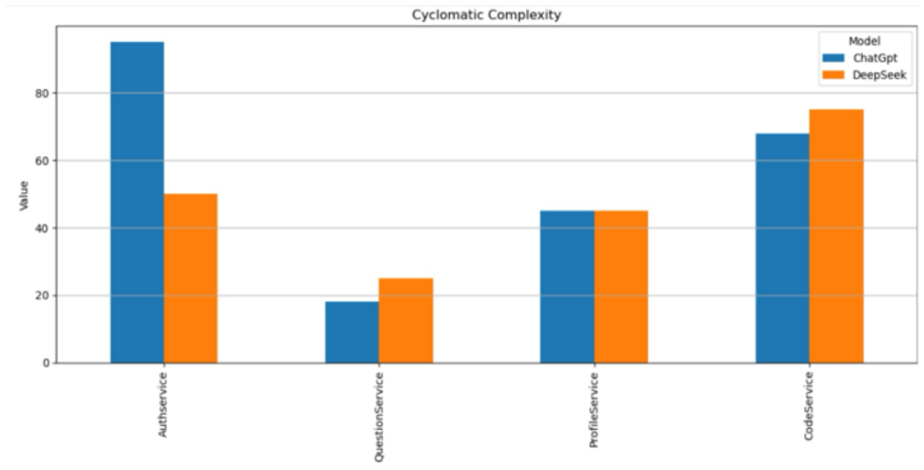

Fig. 1. Cyclomatic Complexity per Microservice for ChatGPT and DeepSeek

As shown in Figure 1, DeepSeek produced significantly lower complexity for the `AuthService` (50 vs 95), whereas ChatGPT performed better on `QuestionService` and `CodeService`. This suggests that **ChatGPT** generally produced **simpler control structures**, though **DeepSeek** showed strength in specific contexts such as **authentication workflows**.

5.1.2 Cognitive Complexity. This metric evaluates how difficult code is to understand. It accounts for mental overhead due to nesting and flow interruptions.

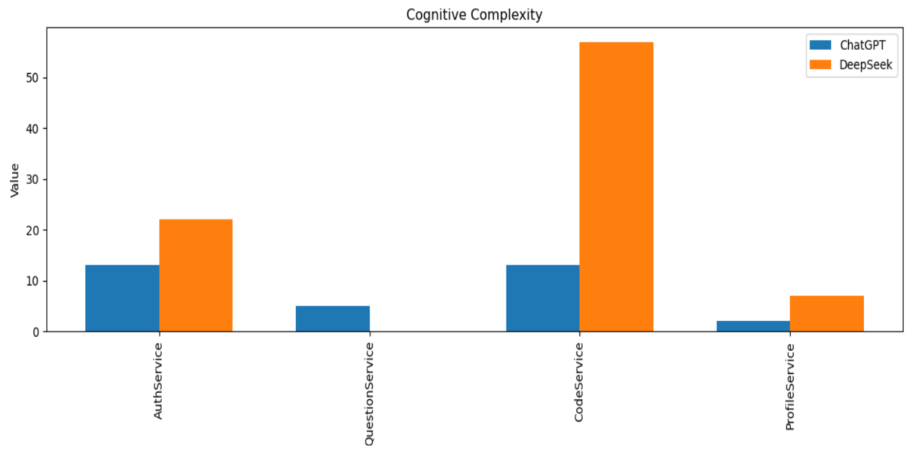

Fig. 2. Cognitive Complexity per Microservice for ChatGPT and DeepSeek

In Figure 2, **ChatGPT** consistently produced **more readable code** in `AuthService`, `CodeService` and `QuestionService`. **DeepSeek's** `CodeService` had the **highest cognitive complexity** (57), suggesting a steeper learning curve for developers.

5.1.3 Code Smells. Code smells represent maintainability concerns like long methods, large classes, or poor separation.

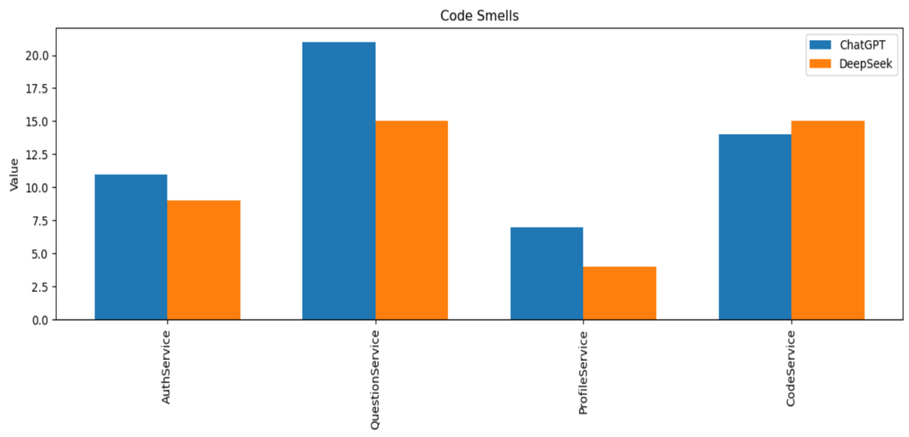

Fig. 3. Code Smells Detected by SonarQube per Microservice for ChatGpt and DeepSeek

From Figure 3, **DeepSeek** had **fewer code smells** in most microservices, indicating cleaner code structure overall.

5.1.4 Vulnerabilities. Security vulnerabilities were assessed through SonarQube's static analysis.

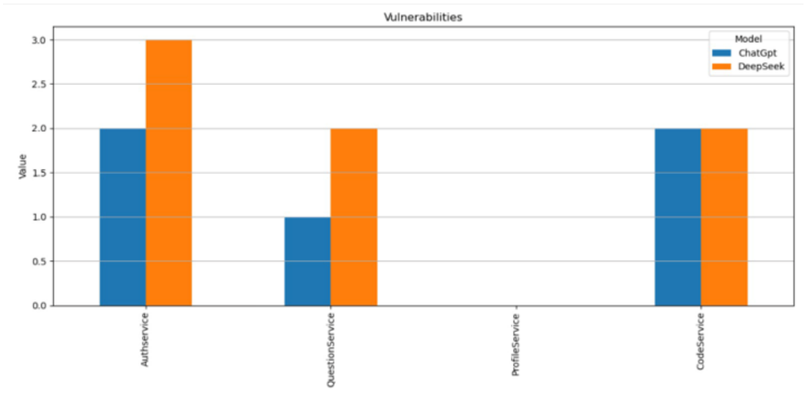

Fig. 4. Vulnerability Count per Microservice for ChatGpt and DeepSeek

Figure 4 indicates that both models performed **comparably with very low vulnerability counts** across all services. While **DeepSeek** had **slightly more** reported vulnerabilities in two services, the other two showed parity—including one where **both models had zero vulnerabilities**—suggesting no critical security concerns in either model's output.

5.1.5 Derived Metrics. We introduced two novel derived metrics: average complexity per method.

- **Average Cyclomatic Complexity per Method:** Indicates control complexity normalized by method count.
- **Average Cognitive Complexity per Method:** Indicates understandability per function.

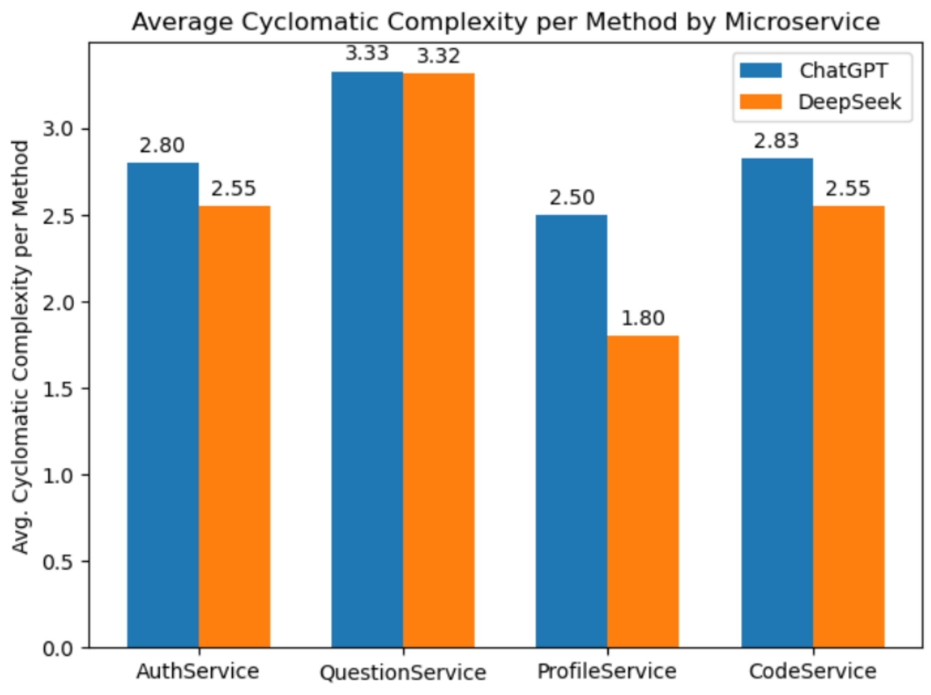

Fig. 5. Average Cyclomatic Complexity per Method

Figure 5 shows that **DeepSeek consistently generated code with lower or comparable cyclomatic complexity per method** across all services. In particular, **ProfileService** exhibited the most significant difference, with DeepSeek averaging nearly one point lower complexity than ChatGPT. For **CodeService** and **AuthService**, the difference was marginal but still favored DeepSeek, suggesting slightly simpler control logic. In **QuestionService**, both models performed nearly identically, indicating no substantial difference in structural complexity. Overall, these results imply that **DeepSeek tends to produce leaner, more focused method-level logic**, which may improve maintainability in certain modules.

Figure 6 illustrates that **ChatGPT consistently produced code with higher cognitive complexity per method** across all services. The difference was most pronounced in **ProfileService**, where ChatGPT's average exceeded DeepSeek's by over 1.7 points. While **ChatGPT's code may be more expressive or feature-rich**, the increased complexity may lead to **greater effort in understanding, maintaining, or modifying the code**. On the other hand, **DeepSeek demonstrated a stronger tendency toward simplicity and readability**, which can be beneficial in production environments requiring frequent updates or developer handovers.

5.1.6 Architecture and Best Practices Evaluation. We conducted a human-evaluated checklist to assess architectural quality:

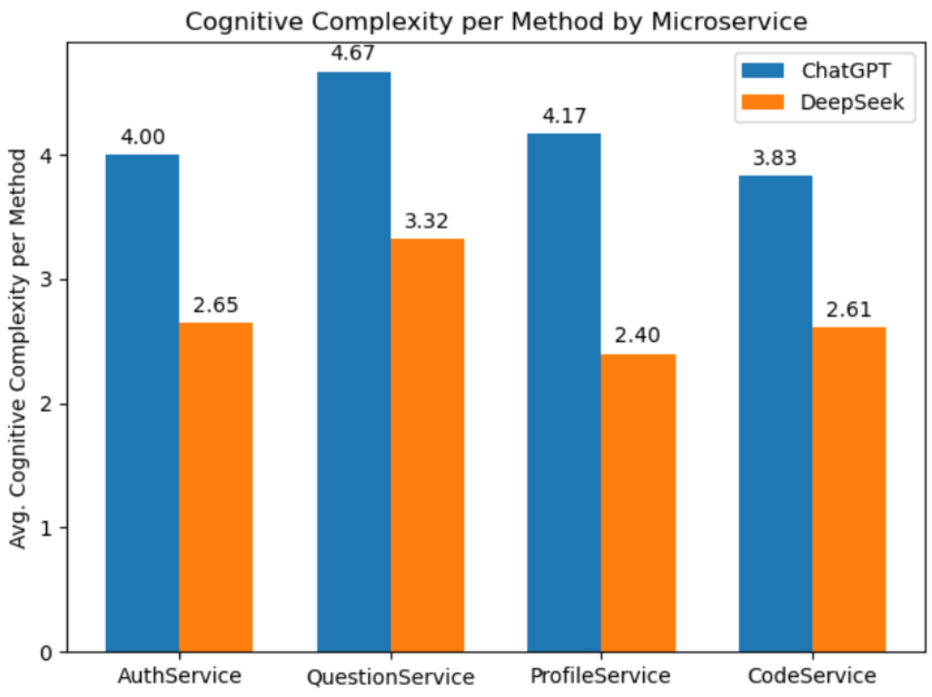

Fig. 6. Average Cognitive Complexity per Method

- **Separation of Concerns**
- **Modularity**
- **RESTful APIs**
- **Naming Conventions**

Both models adhered well to software engineering best practices. No violations were found in layering or RESTful principles. Hence, they both scored equally in this category.

5.2 Software Deliverables

The following software artifacts and documentation were produced as part of this project:

- **Microservice Codebases:** Fully functional Java Spring Boot implementations of four core backend microservices—`AuthService`, `ProfileService`, `QuestionService`, and `CodeService`—each generated independently by both ChatGPT-o3 and DeepSeek-R1. These codebases include RESTful endpoints, controller and service logic, DTOs, and basic integration with database.
- **Evaluation Artifacts:** Comprehensive SonarQube reports covering code quality metrics such as cyclomatic complexity, cognitive complexity, code smells, security vulnerabilities, and maintainability indices for each generated microservice.

- **Presentation and Summary Material:** Final presentation slide deck used for project demonstration and evaluation, alongside a written comparative summary outlining key findings, model strengths and weaknesses, and recommendations for future use of LLMs in backend development.

6 Future Scope

While the project successfully achieved its primary objectives of generating and evaluating microservice code using ChatGPT-o3 and DeepSeek-R1, several important extensions and exploratory goals had to be deferred due to time constraints, and can be done in the future:

- **End-to-End Integration Testing:** We aimed to deploy all generated microservices in a unified runtime environment and evaluate their interoperability through full-stack API testing. Due to limited time for service wiring, Docker configuration, and test sequencing, this component was postponed.
- **Frontend UI Generation and Binding:** We considered generating a simple frontend (e.g., Angular or React) using LLMs that connects to the generated APIs. This would have enabled us to evaluate not only API fidelity but also usability from a client-facing perspective.
- **Test Case Generation and Comparison:** A significant area we were unable to pursue was evaluating the models' ability to generate meaningful unit and integration test cases. We wanted to compare the completeness, correctness, and coverage of test suites generated by each model, particularly for REST endpoints and business logic layers.
- **Expanded Code Quality Metrics:** We planned to incorporate additional analysis such as maintainability index, documentation coverage, and static analysis metrics beyond what SonarQube provided, but processing and aligning these across all services exceeded our timeline.
- **Model Failure Case Analysis:** While we observed certain failure patterns (e.g., missing null checks, improper auth logic), we did not develop a structured error taxonomy. A formal classification of failure types per service could guide future model fine-tuning or evaluation benchmarks.

These remain promising avenues for future work and could significantly enrich both the empirical evaluation and real-world applicability of LLM-based code generation systems.

References

[1] Anonymous. 2023. Across 10 Programming Languages: Evaluating Code Quality and Maintainability. *Unpublished* (2023). Accessed as part of internal review material.

[2] Anonymous. 2023. Evaluating the Correctness of Code Generated by LLMs. *Internal Notes* (2023). Used to identify common pitfalls in correctness evaluation.

[3] Xiang Chen, Jie Liu, et al. 2023. HumanEval+: Uncovering Hidden Flaws in LLM Code Generation Benchmarks. *arXiv preprint arXiv:2307.13648* (2023). https://arxiv.org/abs/2307.13648

[4] Md Motaleb Hossen Manik. 2024. ChatGPT vs. DeepSeek: A Comparative Study on AI-Based Code Generation. *arXiv preprint arXiv:2502.18467* (2024). https://arxiv.org/abs/2502.18467

[5] Yuan Tian, Xipeng Qiu, et al. 2023. Comprehensive Evaluation of Large Language Models across 10 Programming Languages. *arXiv preprint arXiv:2308.12950* (2023). https://arxiv.org/abs/2308.12950

[6] Yilun Wang, Vighnesh Nair, Xinyi Wang, and Graham Neubig. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of LLMs for Code Generation. In *NeurIPS*. https://arxiv.org/abs/2305.18463