

Article

Comparative Evaluation of Gemini and DeepSeek for LLM-Generated Code Quality and Architectural Robustness in Backend Software Engineering

Marko Horvat ^{1,*} , Iva Ursić ² and Klara Krmpotić ²¹ Department of Applied Computing, Faculty of Electrical Engineering and Computing, University of Zagreb, Unska 3, HR-10000 Zagreb, Croatia² Independent Researcher, HR-10000 Zagreb, Croatia

* Correspondence: marko.horvat3@fer.unizg.hr

Abstract

The increasing integration of large language models (LLMs) into software engineering workflows under the term *vibe-coding* necessitates systematic empirical evaluation of their code generation capabilities, especially in the context of complex backend development and architectural decision-making. This study compares popular foundational models Google Gemini 3 Pro and DeepSeek-V3.1 for developing a Java/Spring Boot backend application using a structured prompt-chaining protocol following a typical *vibe-coding* process. The generated solutions were evaluated using several quantitative and qualitative criteria, including the number of corrective prompts, the extent of required manual code interventions, functional correctness, architectural robustness, maintainability-related design choices, latency, and test quality. The results show substantial differences between the two models. DeepSeek required twice as many corrective natural language prompts as Gemini, but both models required a similar number of manual interventions in the generated code, with 23 for DeepSeek and 20 for Gemini. The most pronounced difference was in architectural reasoning. Gemini autonomously introduced the Data Transfer Object design pattern, resulting in a decoupled architecture, although at a cost of a minor performance issue. In contrast, DeepSeek was better in development of boilerplate code but exposed raw JPA entities through the application interface leading to tight coupling and other issues. Gemini's solution satisfied 90.25% of evaluated requirements compared to 68.08% for DeepSeek. Additionally, generated tests showed a higher success rate and broader code coverage, achieving 95.7% successful test execution and 55.9% code coverage for Gemini, compared to 74.1% and 45.6% for DeepSeek, respectively. The results indicate that within the paradigm of *vibe-coding*, even the best available foundational LLMs may still require expert human supervision, especially when the generated code is expected to satisfy specific requirements in production-oriented backend systems.



Academic Editor: Weibin Wu

Received: 14 May 2026

Revised: 22 June 2026

Accepted: 23 June 2026

Published: 25 June 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

Keywords: large language models; AI-generated software; backend development; performance evaluation; prompt engineering; *vibe-coding*

1. Introduction

The discipline of software engineering, as an area of computer science, is currently undergoing a fundamental transformation [1]. For many decades, the traditional software development lifecycle was primarily human-driven and relied on highly predictable and

labor-intensive processes [2]. This classical paradigm was characterized by rigorously executed phases of requirements elicitation, system design, implementation, verification, and deployment, which were repeated in cycles [3]. To address the rapidly increasing complexity of developed software, Model-Driven Engineering (MDE) emerged as a widely accepted engineering methodology [4,5]. In this approach, engineers manually translate business logic into domain models, accompanied by extensive architectural documentation [6].

Traditionally, the implementation phase required time-consuming, work-intensive manual code development within a preferred Integrated Development Environment (IDE) using various Computer-Aided Software Engineering (CASE) tools [7]. The developer bore the entire cognitive load of the engineering process, from manual coding to environment configuration and debugging.

Today, this labor-intensive lifecycle is increasingly augmented and even entirely substituted, by a novel approach informally referred to in the developer community as “vibe-coding” [1,8]. The software development process now commonly includes iterative interactions in a natural language with advanced conversational agents (chatbots) based on generative AI (GenAI) and large language models (LLMs) [9]. These models have evolved beyond simple autocomplete engines into intelligent virtual co-pilots that are capable of autonomous context-aware code generation, architectural decisions, system orchestration, and automated test generation [10]. The novel concept of vibe-coding is thoroughly explored in the next section.

Such rapid and uncritical adoption of new GenAI technologies within the developer community introduces critical concerns regarding reliability, architectural soundness, code explainability, long-term maintainability, and, ultimately, the security of generated computer code. The models’ ability to comprehend specific technical systems must be questioned and objectively evaluated before they can be used reliably. While it has been shown that LLMs may be very successful at generating functional boilerplate code [11,12], particularly in domains like web frontend development [13], their capacity to consistently make optimal architectural decisions remains the subject of ongoing academic investigation.

The motivation for conducting this research can be divided into two parts. First, unlike frontend development, which is predominantly concerned with presentation and often stateless user interfaces, backend architectures contain core business logic, manage complex data structures, and enforce transactional integrity. On the backed side, unlike the frontend, architectural or functional errors typically result in a series of catastrophic consequences for performance, data stability and system security. Another inherent challenge with backend assessments is that the server-side behavior requires extensive testing, whereas frontend vibe-coding produces visible and easily assessable results almost immediately. Additionally, evaluation of server-side code can provide a more accurate measure of an LLM’s true capacity for abstract architectural reasoning and enterprise-grade reliability. Second, not all developers use English in their interactions with chatbots but the vast majority of LLM benchmarking is currently conducted using English-language prompts. Moreover, it can be expected that most software developers will use their native languages in interaction with chatbots to reduce cognitive load and facilitate complex and faster communication. Translating highly technical domain concepts from a non-native language to an internal semantic representation may pose a challenge for LLMs which are trained primarily on software development datasets.

Therefore, the motivation for this study, and its distinction with previously published research, may be summarized in these key points:

1. Evaluate the effectiveness and practical applicability of selected foundational LLMs for server-side code generation within the vibe-coding context, using a representative scenario commonly encountered by novice or junior software developers.

2. Compare two popular but distinct LLMs: a proprietary cloud-based model and an open-weight model suitable for local use.
3. Investigate the capability of foundational LLMs to support vibe-coding with low-resource or morphologically complex languages such as Croatian [14–29].

The remainder of the paper is organized as follows: Section 2 explains what the term vibe-coding entails and reviews related work on LLM-assisted software engineering. Section 3 describes the methodology, including the experimental setup, target backend schema, prompting protocol, and evaluation framework for static and dynamic code analysis, as well as the procedure for evaluating the generated SQL schema. Section 4 presents experimental results. It first discusses SQL schema conceptualization and database entity development, followed by separate evaluations of the Google Gemini and DeepSeek models, including their respective evaluation summaries. The section then compares the models in terms of vibe-coding efficiency, human intervention rates, test performance, code coverage, system performance, API call latency, code accuracy, completeness, system architecture, and static code quality. Section 5 discusses the findings and outlines the main research limitations. Finally, the last section concludes the paper and identifies directions for future research.

2. Related Work

2.1. What Is Vibe-Coding?

The vibe-coding paradigm represents an emerging shift in how software is created [17]. Enabled by GenAI and LLMs, this approach moves the developer's primary activity away from writing syntax and implementing repetitive code patterns towards specifying intended system behavior through natural language interaction. In this model, the developer and the AI assistant engage in a continuous dialogue in which requirements, operational goals, code generation tasks, corrections, and refinements are expressed through prompts [8,18]. Rather than manually writing code line by line, the developer acts as a high-level coordinator, guiding the LLM through an iterative process of code generation, evaluation, and modification.

This paradigm shift also changes the skills required of developers. The central challenge is no longer limited to acquiring programming skills or mastering computer language syntax. Through the skill of prompt engineering, the developer describes the desired functionality in natural language and allows the LLM to generate the source code, iteratively refining goals through additional prompts until the expected result is achieved [19]. In its more informal form, vibe-coding may involve generating software entirely without understanding or even inspecting the produced code.

The application of the vibe-coding methodology has significant potential to accelerate prototyping and facilitate development of complex software systems [20]. It may also have implications for computer science education [21], where LLM-based tutors and coding assistants can provide personalized guidance through academic curricula and help students to rapidly prototype and evaluate coursework software solutions [22,23]. However, the vibe-coding paradigm is not without its challenges [10,24,25], particularly in educational settings [26] where challenges such as cognitive debt and cognitive offloading play a major negative role [27,28].

2.2. AI-Assisted Software Development

The intersection of artificial intelligence and software engineering has recently generated substantial academic interest, particularly regarding the automated generation of source code [29–32]. Early research in this domain focused primarily on qualitative studies [33], or syntactic correctness and the ability of models to complete localized, iso-

lated code snippets, using benchmarks like HumanEval, HumanEval-X and MBPP [34–36]. However, as foundational model capabilities have increased to process very large context windows, the research focus has shifted toward system-level architecture, component orchestration, and the generation of complete, interconnected codebases [37]. Recently, a benchmark ViBench specifically developed for evaluation of LLMs in vibe-coding has been published [38]. Unlike previous benchmarks, in ViBench tasks are specified entirely through user-facing requirements without implementation constraints, and evaluation occurs at the abstraction level of an end user interacting with the application.

The recent literature highlights the increasing need of evaluating LLMs not only as advanced syntax generators but also as autonomous architectural reasoning engines [39]. Studies assessing the implementation of standard software design patterns by AI agents indicate significant variance in structural quality [40]. This variance is often highly dependent on the model's training data cutoff date and its exposure to modern framework idioms. Models trained predominantly on legacy code repositories frequently reproduce outdated anti-patterns, while state-of-the-art models increasingly apply modern best practices, such as domain-driven design principles [41].

Additionally, the problem of academic integrity and plagiarism detection in AI-generated code is becoming more pronounced [42]. As generative models become ubiquitous in computer science education, traditional plagiarism detection tools find it difficult to identify AI-authored syntax in text [43] and in computer code [44], creating an urgent need for new heuristic, semantic, and stylometric analysis solutions [42,45].

At the same time, the IT industry is recognizing the problem of the so-called “AI velocity tax” [46], a phenomenon where rapid AI code generation leads to accumulating technical debt, ultimately requiring human developers to spend disproportionate time on maintenance and refactoring instead of feature development [47]. The AI velocity tax can also be identified in educational settings, where students submit AI-generated solutions without engaging in problem-solving, thereby undermining learning outcomes and creating difficulties for educators to confirm code authenticity [48].

The impact of prompt language on code generation quality is an emerging and highly relevant area of software engineering research. While many suggest (for example, [48–50]) that LLMs have the capacity to process internal logic in a wide range of computer languages and code architectural paradigms, empirical research indicates that LLM-based AI co-pilots frequently produce codebases that are difficult to maintain, upgrade, or scale [51,52]. Because the generated source code often results in functionally operational software that appears to meet initial specifications, operators frequently overlook the structural optimality and internal quality of the implementation. However, accepting superficial functionality without rigorously examining the underlying architectural details inevitably leads to systemic deterioration [53]. In the long run, reliance on functional but suboptimal code results in significant structural compromises, as well as incorrect or inefficient execution of core business logic [54].

3. Methodology

To methodically evaluate architectural reasoning, generative code synthesis capabilities, and cross-lingual semantic comprehension of the selected foundational models, we designed a custom empirical methodology. The experimental design simulates the iterative construction of a complete, enterprise-grade RESTful backend application, as might be performed by a typical software engineer giving natural language instructions to two of the best foundational LLMs available at the time of writing: Google Gemini 3 Pro [55] and DeepSeek-V3.1 [56]. These models were selected because both are plausible tools for inexperienced or moderately experienced backend developers seeking assistance in generating

software applications. Comparing them enables an assessment of LLM-generated backend code quality under realistic conditions where developers may rely on easily accessible AI assistants. Furthermore, the models represent different technical choices: Gemini is a proprietary, cloud-hosted managed service [57], while DeepSeek enables open-source and local (on-site) deployment [58]. Because of all these reasons, these models accurately represent the primary configurations that modern vite-coding backend developers are most likely to adopt, ensuring the experimental findings are highly applicable to real-world engineering contexts.

3.1. Experimental Setup

The experimental setup was based on a modern Java/Spring Boot backend technology stack commonly used in enterprise software development [59]. This choice ensured the evaluation was conducted in a realistic and practically relevant development environment, rather than an artificially simplified programming context.

Java 17 was selected as the programming language because of its stability, long-term support, and broad adoption in existing backend systems [60]. The Java Virtual Machine (JVM) was run locally with standard execution settings. The application was developed using Spring Boot 3.5.0 [61] which automatically configured the HikariCP connection pool with its default settings (10 connections), which remained unchanged for all models. Apache Maven 3.9.9 was used as the project build and dependency management tool. In the experimental setup, Maven provided a standardized mechanism for declaring project dependencies, configuring build plugins, executing automated tests, and packaging the Spring Boot application [61,62]. These steps ensured that both LLM-generated solutions were built and tested under comparable conditions, improving the reproducibility and methodological consistency of the evaluation. The development and testing were carried out in IntelliJ IDEA Community Edition 2024.1, a professional and frequently used IDE supporting advanced code analysis, refactoring, debugging, dependency inspection, and test execution [63].

The persistence layer was implemented using Spring Data JPA and Hibernate [64]. Spring Data JPA provides a high-level abstraction over the Java Persistence API and simplifies the implementation of repository-based data access [65], while Hibernate serves as the underlying object-relational mapping framework responsible for translating interactions with Java entity classes into SQL operations [66]. This part of the technology stack was particularly important for the study because the interaction between generated JPA entities, object-relational mappings, and database access patterns directly influenced several architectural and functional issues observed in the generated solutions.

PostgreSQL 16 was used as the relational database management system [67]. It was selected for its robustness and widespread adoption as an open-source database system [68]. In the experimental application, PostgreSQL was used to store and manage persistent application data, with all database interactions mediated through the Hibernate implementation of JPA. This configuration enabled the study to evaluate not only the syntactic correctness of the generated backend code, but also the models' ability to produce coherent, maintainable, and functionally correct persistence-layer architectures within a realistic Java/Spring Boot environment.

Both models were evaluated on identical hardware: a personal computer with Intel Core i7-12650H processor, 16 GB RAM, 512 GB SSD, and Windows 11 Home Edition, and utilized the same PostgreSQL 16 instance with default configuration settings. Some LLM parameters such as temperature and context-window configuration were not changed for the experiment.

To ensure empirical validity, the methodology simulated a strict vibe-coding paradigm during the primary component generation phases. This means that the developer input was restricted exclusively to natural language instructions, using structured, multi-turn prompts formulated entirely in the Croatian language. This linguistic requirement verified the models' ability to translate non-native technical terms into standard programming terms.

3.2. Target SQL Schema

A specific backend was envisioned to be automatically generated in order to evaluate the both foundational LLMs. This backend was designed to simulate a typical high-transaction e-commerce and inventory management domain. The target backend model contained four core database entities, *Customer*, *Order*, *OrderItem*, and *Product*, as shown in Figure 1. These core entities were mutually connected by bidirectional One-to-Many and Many-to-Many relationships. A *Customer* may have zero or many *Orders*, while a single *Order* always belongs to a specific *Customer*. An *Order* is composed of one or many *OrderItems*, and each *OrderItem* is mapped to the target *Product* catalog.

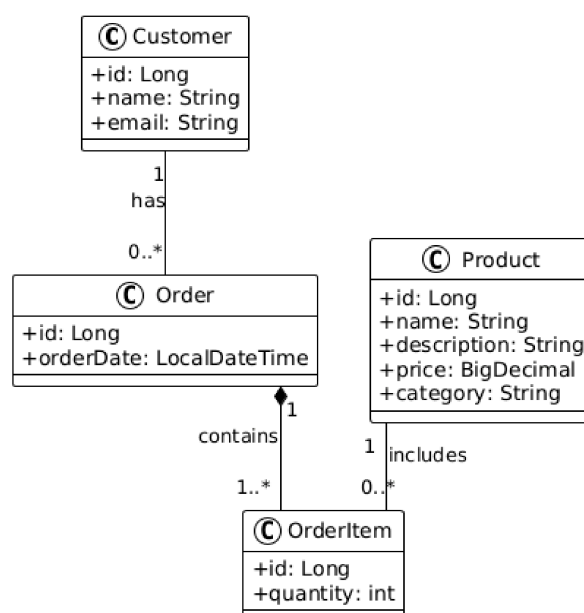


Figure 1. UML class diagram showing the core entities (*Customer*, *Order*, *OrderItem*, and *Product*) and their mutual relationships of the targeted backend that was developed using foundational LLMs.

This structural complexity was intentionally selected to test the models' proficiency in handling advanced enterprise software challenges. By requiring the generation of the specific object graph, the experiment assessed the models' ability for ORM overhead. Specifically, it evaluated whether the models can configure appropriate JPA cascade operations, implement optimal fetch strategies to mitigate memory degradation, enforce strict declarative transaction boundaries, and prevent cyclic dependencies during JSON data serialization. Cyclic dependencies, in particular, are a common hazard when exposing tightly coupled domain graphs to external API presentation layers [69,70].

To enable a realistic backend system evaluation, a SQL script was developed to populate the database schema in Figure 1 with a substantial volume of test data. The created synthetic dataset included 20,000 products and 50,000 orders which enabled performance measurements to be conducted under conditions approximating realistic application load. This setup allowed assessment of both the functional correctness of the generated systems and their behavior when operating on a non-trivial relational dataset. The SQL script used for generating synthetic test data is shown in Appendix A.

3.3. Prompting Protocol

A zero-shot prompt chaining approach was used to instruct the models through the code development process [69,71,72]. Specifically, a predefined set of sequential prompts defining backend architecture and functionalities was given to each model. After each prompt, the generated code was analyzed, and corrective follow-up prompts were issued as needed to fix any errors or address possible misinterpretations. Both models received the same initial development prompts in Croatian, and corrective prompts were based only on objectively observed issues. Both models were given equivalent correction opportunities based on the same stopping and correction criteria. Prompt histories were maintained consistently and separately for each model. This process simulates a realistic, conversational development workflow in the vibe-coding paradigm.

The interaction with the foundational models was conducted exclusively in the Croatian language. This was implemented to assess the models' cross-lingual technical comprehension and their ability to map non-English domain terminology to standard Java syntax. The majority of the global software development workforce, i.e., coders, programmers and computer engineers, will most likely use their native languages when interacting with AI co-pilots, to reduce cognitive load and facilitate much more precise expression of complex business logic. In such a situation evaluating LLMs using only English-centric prompting benchmarks may result in biased assessment of their true usefulness for vibe-coding. As explained before, this part of the experiment aimed to establish a more realistic assessment by requiring the models to interact with morphologically rich language like Croatian [13–16].

The vibe-coding prompt sequence was iterative and divided into four discrete, logical development phases or sequence steps:

- Prompting step 1: Generation of domain entity classes.
- Prompting step 2: Implementation of CRUD (Create–Read–Update–Delete) layers.
- Prompting step 3: Implementation of advanced functionalities.
- Prompting step 4: Configuration and test generation.

Following each of the four main prompts, the generated code was inspected for basic, objective errors that would prevent normal execution, such as compilation failures, missing classes or methods, unresolved dependencies, failed tests, or runtime errors. When such issues were discovered, they were merged into a single comprehensive corrective query and submitted to the model at once, rather than being addressed through a series of separate and shorter corrective prompts. For example: “The code cannot be compiled due to error X in file Y. Also, file Z is missing.” This check-and-correction cycle was repeated throughout each development phase until the generated code had no obvious blocking errors and could be successfully compiled.

The sequence of main prompts is shown in Figures 2–5, in the original language and translated to English.

The first prompt, as shown in Figure 3, was designed to establish the foundation of the application by generating the domain model. The models were instructed to create the required entity classes, define their attributes, specify relationships between entities, and include basic validation constraints. This phase assessed whether the models could correctly interpret the application domain and translate it into a coherent object-relational structure.

The second prompt, as shown in Figure 3, focused on the implementation of standard CRUD architecture. The models were asked to generate the repository, service, and controller layers required for creating, reading, updating, and deleting application data. This phase evaluated the models' ability to follow common Spring Boot architectural conventions, separate responsibilities across layers, and produce functional API endpoints.

Croatian (original)	English (translated)
Ti si backend Java developer s višegodišnjim iskustvom u Spring Bootu i radu s bazama podataka. Tvoj zadatak je izgraditi REST backend sustav za upravljanje narudžbama koristeći sljedeće tehnologije: • Java 17 • Spring Boot 3.5.0 • Spring Data JPA • Maven (kao alat za build) • PostgreSQL (lokalna baza) • JUnit & Mockito (za testove, u kasnijem koraku) Cilj aplikacije: Implementirati sustav za upravljanje narudžbama koji uključuje 4 entiteta: • Product – id (Long) – name (String) – description (String) – price (BigDecimal) – category (String) • Customer – id (Long) – name (String) – email (String) • Order – id (Long) – orderDate (LocalDateTime) – customer (ManyToOne – referenca na Customer) – orderItems (OneToMany – lista stavki) • OrderItem – id (Long) – product (ManyToOne – referenca na Product) – quantity (int) – order (ManyToOne – referenca na Order) Što želim da napraviš u ovom koraku: • Generiraj Java entitetske klase za sve entitete iznad koristeći JPA anotacije. • Postavi ispravne relacije između entiteta (OneToMany, ManyToOne, CascadeType, FetchType). • Ne generiraj servisni ni kontrolerski sloj još. • Nemoj još raditi testove. • Dodaj anotacije za validaciju ako je potrebno (npr. @NotNull, @Email) • Kod neka bude čist, spreman za proširenje, i sa sažetim komentarima	You are a backend Java developer with many years of experience in Spring Boot and working with databases. Your task is to build a REST backend system for order management using the following technologies: • Java 17 • Spring Boot 3.5.0 • Spring Data JPA • Maven (as a build tool) • PostgreSQL (local database) • JUnit & Mockito (for tests, in a later step) Application goal: Implement an order management system that includes 4 entities: • Product – id (Long) – name (String) – description (String) – price (BigDecimal) – category (String) • Customer – id (Long) – name (String) – email (String) • Order – id (Long) – orderDate (LocalDateTime) – customer (ManyToOne – reference to Customer) – orderItems (OneToMany – list of items) • OrderItem – id (Long) – product (ManyToOne – reference to Product) – quantity (int) – order (ManyToOne – reference to Order) What I want you to do in this step: • Generate Java entity classes for all entities above using JPA annotations. • Set up the correct relationships between entities (OneToMany, ManyToOne, CascadeType, FetchType). • Do not generate the service or controller layers yet. • Don't do the tests yet. • Add validation annotations if needed (eg @NotNull, @Email) • Keep the code clean, ready for expansion, and with concise comments.

Figure 2. The first vibe-coding prompt given to the evaluated LLMs. This prompt was intended to establish the foundation of the backend application and generate domain entity classes. The original prompt text in Croatian is on the left, with the English translation on the right.

Croatian (original)	English (translated)
Na temelju ranije generiranih entiteta, generiraj Repository, Service i Controller sloj s REST endpointima za CRUD operacije nad svim entitetima. Koristi REST konvencije i anotacije poput @RestController, @Service, @Repository.	Based on the previously generated entities, generate a Repository, Service and Controller layer with REST endpoints for CRUD operations on all entities. It uses REST conventions and annotations like @RestController, @Service, @Repository.

Figure 3. The second prompt was aimed at the implementation of CRUD layers. The original prompt in Croatian on the left and English translation on the right.

Croatian (original)	English (translated)
Dodaj dodatne funkcionalnosti: • Filtriranje proizvoda po cijeni i kategoriji • Sortiranje proizvoda po cijeni (rastuće i padajuće) • Dohvat top 5 najprodavanijih proizvoda (korištenjem agregacije iz OrderItem entiteta) • Dohvat narudžbi po korisniku (preko user ID-a)	Add additional functionality: • Product filtering by price and category • Sorting products by price (ascending and descending) • Retrieving the top 5 best-selling products (using aggregation from the OrderItem entity) • Retrieve orders by user (via user ID)

Figure 4. The third prompt given to the evaluated LLMs. The goal was the implementation of advanced functionalities including product filtering, sorting and retrieval through various data modalities. The original prompt in Croatian on the left and English translation on the right.

The third prompt introduced more complex business requirements beyond basic CRUD operations (Figure 4). It evaluated the models' capacity to implement application-specific logic, handle more demanding functional constraints, and maintain consistency with the previously generated domain model and application architecture. This phase was particularly important for assessing the models' architectural reasoning and their ability to extend the system without introducing functional or structural inconsistencies.

Croatian (original)	English (translated)
Poveži aplikaciju s lokalnim PostgreSQL-om koristeći application.properties konfiguraciju. Pretpostavi sljedeće: • baza: orders • korisnik: postgres • lozinka: mojaLozinka • host: localhost:5432 Generiraj sve potrebne testove za cijelu aplikaciju: • Jedinične testove za servisni sloj koristeći JUnit 5 i Mockito. • Integracijske testove za repozitorije koristeći H2 in-memory bazu i @DataJpaTest. • Integracijske testove za REST kontrolere koristeći @SpringBootTest i MockMvc.	Connect the application to the local PostgreSQL using the application.properties configuration. Assume the following: • database: orders • user: postgres • password: myPassword • host : localhost:5432 Generate all necessary tests for the entire application: • Unit tests for the service layer using JUnit 5 and Mockito . • Integration tests for repositories using H2 in- memory database and @DataJpaTest. • Integration tests for REST controllers using @SpringBootTest and MockMvc.

Figure 5. The fourth and final vibe-coding prompt was for the purpose of system configuration and the creation of unit and integration tests. The original prompt in Croatian on the left and English translation on the right.

Finally, the fourth and last prompt addressed application configuration and testing (Figure 5). The models were instructed to configure database connectivity, adapt the application to the selected PostgreSQL environment, and generate automated tests for the implemented functionality. This phase assessed not only whether the generated application could be executed in the intended technological environment, but also the quality, relevance, and completeness of the test code produced by each model.

3.4. Evaluation Framework for Static and Dynamic Code Analysis

In order to complete the analysis of the backed code quality and architectural robustness, the generated artifacts were also evaluated using quantitative and qualitative static tests. Additionally, the generated SQL schema was evaluated separately from Java codebase using separate metrics, as described in the following section. The generated Java backend code was evaluated using a combination of static and dynamic analysis tools. SonarQube 2025.2 [73] was used for static code analysis, while Postman 11.44.0 [74] was used for the dynamic or runtime validation.

SonarQube tool was used to find so called “code smells” and “technical debt” to determine the readability and internal quality of computer code. “Code smells” is a colloquial term in computer programming describing characteristics of source code that suggest structural design problems [75]. In practice, code smells indicate poor structural practices and design anti-patterns that, although they compile, impede future development [76]. Furthermore, SonarQube aggregates the effort required to fix all “code smells” into a quantitative measure called “technical debt” and assigns the codebase a standardized maintainability grade (denoted A through E, with A being the best grade and E the worst) based on its overall debt density [77]. The appearance of the SonarQube user interface during static analysis of the generated backed code is shown in Figure 6.

Postman was used as the primary tool to validate the generated REST API endpoints and collect basic performance measurements. For each of the five predefined test scenarios, six consecutive HTTP requests were sent to the corresponding endpoint. The first request was treated as a warm-up request and excluded from the analysis in order to reduce the influence of initial execution overhead. Response latency was then calculated as the arithmetic mean of the remaining five requests. The Postman interface is shown in Figure 7.

SonarQube and Postman were used with their default configurations. Consequently, the obtained results should be interpreted within the context of these default settings, since changes to the SonarQube rule profile, Postman execution settings, or related analysis parameters could affect the reported code quality and runtime evaluation outcomes.

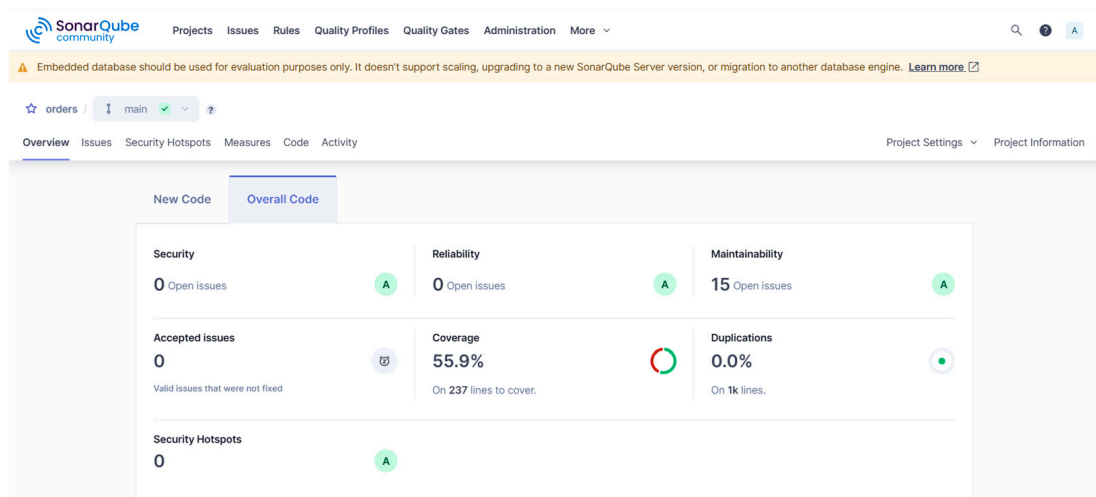


Figure 6. SonarQube user interface with the results of maintainability and code coverage tests.

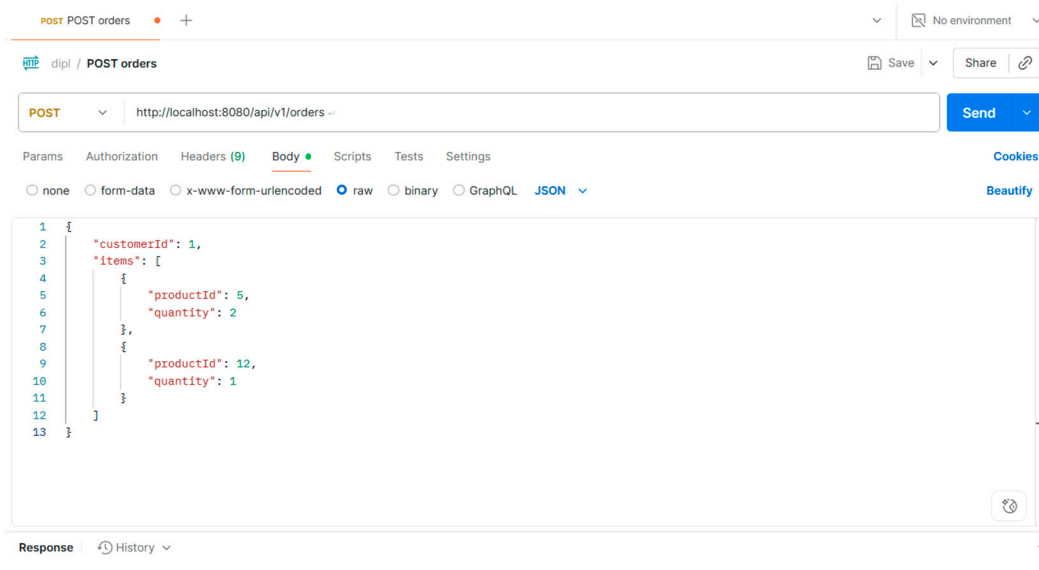


Figure 7. Postman tool was used to validate REST API behavior at runtime and to measure response latency.

3.5. Evaluation of Generated SQL Schema

In addition to assessing the generated Java backend code, and in order to enhance the comprehensiveness of the experiment, the methodology also included an evaluation of the LLM-assisted generation of the underlying data layer. This separate part of the analysis concentrated only on the vibe-coding of SQL schema of the relational database based on the supplied target UML diagram, as a picture (shown in Figure 1).

4. Results

The results of the experiment demonstrate significant differences in software generation capabilities, internal knowledge representations, and architectural awareness between the two foundational models. The data from the evaluated Java/Spring Boot scenario reveal a clear distinction between mechanical syntax generation and advanced semantic reasoning.

4.1. Google Gemini Model Evaluation

In the first prompting step, Gemini correctly interpreted the initial requirements and generated all four required entity classes with appropriate JPA annotations, including @Entity, @Table, @Id, @GeneratedValue, @ManyToOne, and @OneToMany. The relationships between the entities were logically defined and generally consistent with the intended relational model. However, code inspection revealed two errors that prevented successful compilation. Both errors were addressed in a single corrective prompt. After the model generated the corrected version, five additional manual interventions were required to make the code fully functional. This prompting phase indicates that Gemini was able to understand the domain structure and correctly model JPA relationships, but it did not generate completely executable code without further human intervention.

In the second prompting phase or step, Gemini generated the required repository, service, and controller layers for the *Customer* and *Order* entities. However, for the *Product* and *OrderItem* entities, it provided only generic implementation instructions in comments instead of complete source code. This represented a significant omission because the prompt requirements were not fully implemented. In addition, key Maven dependencies for JPA and web functionality were missing. These issues were addressed through one corrective prompt. After the first correction, four additional manual interventions were still necessary to add the missing implementations and correct the dependency configuration. The results of the second prompting step show that Gemini was able to follow the intended layered Spring Boot architecture, but only partially.

Gemini performed particularly well in the third prompting step in regard to the implementation of advanced functionalities. No errors were identified in the generated code. The model correctly implemented all requested advanced functionalities, including a complex aggregation query in *OrderItemRepository* and a flexible endpoint in *ProductController* capable of accepting multiple parameters. This prompting step demonstrates Gemini's strong ability to understand and implement more complex business requirements. Unlike the previous phases, the model preserved consistency with the existing codebase and generated functionally correct code without requiring corrective prompts or manual modifications.

The final and fourth prompting step proved to be the most problematic in the interaction with Gemini. Although the model correctly generated the *application.properties* file required for database connectivity, it showed substantial weaknesses in generating complete and correct test code. These weaknesses appeared in two main forms: for some of the requested test classes, the model generated no code at all, while for others it produced only general textual instructions instead of concrete and executable test implementations. The initial response contained seven structural errors, primarily related to completely missing test classes. After the first corrective prompt, Gemini generated the requested files. However, manual inspection of the newly generated code revealed a second layer of problems: although the files were now present, their content contained logical and syntactic errors that prevented successful compilation and correct test execution. The remaining errors were specified in a second corrective prompt. After this correction, two manual changes were sufficient to bring the code to a functional state. Nevertheless, the complete correction process in this phase required two corrective prompts and eleven manual interventions before all tests became fully operational.

The last prompting step reveals a significant limitation of Gemini in test-code generation. While the model was able to configure the application environment correctly, it was considerably less reliable when generating tests that required an understanding of interactions between different application layers, dependencies, repositories, services, controllers, and the test execution context. In the end, the entire vibe-coding process with the Gemini model required a total of four corrective prompts and twenty manual code interventions.

The main strength of the Gemini-generated solution is its modern and robust architecture. Without explicit instruction, the model independently implemented the Data Transfer Object (DTO) design pattern [78,79]. This approach is widely used in the software development industry and employs separate classes for REST API communication, while JPA entities are reserved for internal domain and persistence logic. Additionally, the service layer also includes mapping logic for converting data between entities and DTOs. An example of such Java class is *ProductDTO* in Figure 8.

```
public record ProductDTO(
    Long id,
    @NotBlank String name,
    String description,
    @NotNull @DecimalMin("0.01") BigDecimal price,
    String category
){}
```

Figure 8. An example of the industry standard Data Transfer Object successfully implemented by Gemini LLM without being explicitly prompted to do so.

The self-directed architectural decision to implement DTO pattern provides two important benefits: (1) layer separation and (2) enhanced security and stability. First, it clearly separates the domain model from the external API representation, making the system easier to maintain and extend. Second, it improves security and stability by preventing problems such as infinite recursion during serialization and by allowing precise control over which data are exposed through the API.

Another indicator of architectural quality is the implementation of advanced functionality. The generated *ProductController* class contains a single but highly flexible endpoint for searching and sorting products. This represents an elegant and efficient solution, as the endpoint supports multiple optional parameters while maintaining a compact and coherent API design.

At the same time, Gemini's main weakness was observed in the generation of repetitive boilerplate code, particularly in the CRUD and testing layers, where it occasionally omitted entire classes. Another noteworthy issue is the specific implementation of DTO mapping in the service layer, where the order items are retrieved inside a loop for each order. Although this code is functionally correct, it may introduce performance inefficiencies. The impact of this implementation choice on runtime performance is more closely examined in Sections 4.7 and 4.8.

4.2. DeepSeek Model Evaluation

The development process with DeepSeek-V3.1 exposed a different set of strengths and weaknesses compared with the system generated by Gemini 3 Pro.

In the first prompting step the model's response initially appeared correct. It generated complete Java classes, including getters and setters, and used the modern *jakarta.** namespace required by recent versions of Spring Boot and Jakarta Persistence. However, closer inspection showed that the generated code could not be compiled because of a critical configuration error which initiated a problematic multi-step manual correction process following the vibe-coding interaction paradigm. The original code contained three distinct errors: two configuration-related errors, which manifested as import-related compilation problems, and one syntactic error. After these problems were reported to the model in the first corrective prompt, DeepSeek made an incorrect inference. Instead of adding the missing dependency required for the existing *jakarta.** imports, it replaced the correct *jakarta.** imports with outdated *javax.** imports which was not acceptable since the

experimental setup was based on a modern Spring Boot environment that relies on the Jakarta namespace. After the problem was prompted again, the model continued to use the obsolete *javax.** packages in its third response. Resolving this initially minor configuration issue therefore required three corrective prompts, including an explicit instruction to use the *jakarta.** namespace. In total, seven manual code interventions were needed to bring the entity-layer code into a correct configuration that can be compiled.

In the second prompting step and in contrast to Gemini, DeepSeek immediately generated all requested CRUD classes for all entities. This included the repository, service, and controller layers, resulting in a more complete initial implementation of the repetitive CRUD structure. However, the generated code contained one configuration-related error that prevented the entire controller layer from compiling. The problem was manifested through unresolved imports related to *org.springframework.http* and *org.springframework.web*. It was resolved with one corrective prompt. After the correction, one additional manual intervention was required to add the appropriate dependency to the *pom.xml* file.

In the third prompting step, the generated code contained two distinct errors that prevented successful compilation and correct execution. Both errors were addressed with a single corrective prompt. After this, two manual code interventions were still required. The first involved adding a missing method definition to the *ProductService* interface. The second involved correctly injecting a missing dependency that was used in the generated implementation but not properly declared.

The fourth prompting step proved to be the most problematic stage in the interaction with DeepSeek. The model's initial response was incomplete. It contained eleven errors, three of which were related to missing Maven dependencies required for testing, while the remaining eight concerned the complete omission of requested test classes. This meant that the generated test layer was not only incomplete, but also impossible to execute in the intended project environment. After the first corrective prompt, the DeepSeek model generated the missing files and added the required dependencies. However, this introduced a new fundamental issue that completely prevented the tests from running. Even after this issue was explicitly pointed out, DeepSeek's attempted correction was unsuccessful, and the problem persisted. A third corrective prompt was therefore required, containing a direct instruction to set the *junit-jupiter-api* dependency to the stable version 5.12.2. Only after receiving this explicit instruction did the model successfully correct the test configuration.

After completing all four development phases, bringing the system generated by DeepSeek to a functional state required a total of eight corrective prompts and twenty-three manual code interventions.

The main demonstrated weakness of the DeepSeek-generated solution is in architectural design. The model used an approach that is widely regarded as an anti-pattern in modern backend development: returning raw JPA entities directly from REST controllers [70]. The issue is particularly evident in the bidirectional relationship between the Customer and Order entities, as shown in Figure 9, which is exposed directly through the controller layer.

This architectural decision introduces a known risk in Java/Spring Boot applications. When entities with bidirectional relationships are serialized directly into JSON, serialization problems may occur, including infinite recursion, excessive data exposure, unstable API behavior, and tight coupling between persistence concerns and client-facing representations. The practical impact of this design choice on system functionality is analyzed in more detail in the following section.

Compared to the Gemini general-purpose conversational model, DeepSeek was somewhat less reliable in generating a complete and modern backend system. Its main advantage was in producing more complete code for repetitive tasks, particularly in the CRUD layers,

which reduced the need for manual completion in those parts of the system. However, these benefits were outweighed by several fundamental shortcomings. First, the model demonstrated outdated domain knowledge, repeatedly generating code and configuration fragments that were not fully aligned with the modern Java/Spring Boot environment. Second, the model made inferior architectural decisions by exposing JPA entities directly through REST controllers, thereby undermining robustness, maintainability, and API stability.

```
@GetMapping("/customer/{customerId}")
public ResponseEntity<List<Order>> getOrdersByCustomerId(
    @PathVariable Long customerId
){
    List<Order> orders = orderService.getOrdersByCustomerId(customerId);
    return ResponseEntity.ok(orders); // Incorrect entity return
}
```

Figure 9. An example of DeepSeek’s generated code where the REST controller returns a list of raw JPA *Order* entities directly to the client. Such code may result in stack overflows, excessive response payloads, failed API responses, and unstable runtime behavior.

4.3. Vibe-Coding Efficiency and Human Intervention Rates

The efficiency of the development process was evaluated by the number of required interactions (corrective queries) and the amount of manual work (number of changes) required to achieve a functional solution. The cumulative results after all four prompting phases are presented below in Table 1.

Table 1. Comparison of development process efficiency, the number of corrective prompts, and manual software code corrections.

Vibe-Coding Success Metric	Gemini	DeepSeek
Total number of errors	16	22
Total number of corrective LLM prompts	4	8
Total number of manual code corrections	20	23

As can be seen in Table 1, in terms of raw process efficiency and workload during vibe-coding, Gemini 3 Pro demonstrated a significantly more efficient and predictable development lifecycle. DeepSeek-V3.1 required twice as many corrective prompts to achieve a baseline functional state (eight corrective instances compared to Gemini’s four). A more thorough analysis of the prompt history logs showed that DeepSeek’s failures were primarily caused by an outdated internal representation of the Java technological ecosystem. It frequently hallucinated deprecated Spring Boot annotations, attempted to utilize obsolete dependency configurations, and struggled to apply Java 25 native features, requiring the human operator to continuously force alignment with contemporary standards.

4.4. Test Performance and Code Coverage

Code coverage is a software metric measuring the percentage of source code executed during automated testing, identifying untested areas to improve quality. It highlights which lines, branches, or functions are exercised, helping developers find hidden bugs and ensure comprehensive testing, often aiming for 70–80% [80–82].

The accuracy of the implementation, the successful execution of the generated tests, and the tests code coverage as key quantitative indicators of the quality and reliability of the produced code are provided in Table 2.

Table 2. Comparison of test performance and code coverage.

TEST Metric	Gemini	DeepSeek
Total number of generated tests	23	27
Successful test rate (%)	95.7	74.1
Code test coverage (%)	55.9	45.6

At first glance, DeepSeek-V3.1 produced more unit tests, with 27 compared to Gemini's 23. However, the test success rate reveals a more significant qualitative difference. Gemini achieved a test success rate of 95.7%, with one failed test, indicating that its generated tests were almost entirely executable and functionally valid. In contrast, 74.1% of DeepSeek's tests were successfully executed. It can be concluded that a higher number of generated tests does not necessarily correspond to better test quality.

The same pattern can be noticed in code coverage. Gemini's higher test success rate resulted in better code coverage, reaching 55.9% compared to DeepSeek's 45.6%. This suggests that Gemini's tests are more reliable and more significant by employing a larger portion of the implemented backend business logic. Therefore, in the testing segment of the experiment, Gemini demonstrated greater practical usefulness, while DeepSeek showed a tendency to generate a larger volume of test code with lower functional reliability.

Despite this difference in the necessity for corrective prompting, both models ultimately required a comparable amount of manual code interventions before the application could be considered as ready for production.

4.5. System Performance and API Call Latency

System performance was measured by response time (latency) in milliseconds (ms) for five key API calls, i.e., test scenarios. The results are given in Table 3.

Table 3. Comparison of API call latency. The results of the first four scenarios are valid, while the result for the fifth scenario is invalid and should be excluded from analysis.

N	API Call Test Scenarios	Gemini Latency [ms]	DeepSeek Latency [ms]
1	Retrieval of a specific product by ID	10	11
2	Frequent category filter	35	27
3	Complex search (filter + sort)	23	30
4	Retrieval of the top 5 best-selling products	175	170
5	Retrieval of all orders for a specific user	620	15 ¹

¹ This value is invalid as the API call returned unusable results caused by infinite recursion. Therefore, the findings from the fifth API call test scenario should be discarded.

The results show that for standard CRUD operations and search functionalities (API call test scenarios 1–4), the performance of both generated systems was almost identical and generally very efficient. In these cases, both models produced backend implementations with similar response times, suggesting that neither solution introduced significant runtime overhead in simple request–response operations.

The reported comparison for scenario 5, which evaluated the retrieval of complex nested objects, is invalid. In this scenario the Gemini-generated system showed significantly higher latency than in previous scenarios (620 ms). This performance degradation was a direct result of the $N + 1$ select problem caused by the DTO mapping logic, where related data were repeatedly retrieved instead by using a more optimized query strategy. On

the other hand, the DeepSeek-generated system showed a much lower latency of 15 ms. However, this result should be disregarded because the corresponding API response was functionally incorrect due to a critical serialization bug.

It is important to emphasize that the $N + 1$ select problem observed in the Gemini-generated implementation should not be interpreted as evidence of a fundamental architectural deficiency but rather as an implementation-level performance defect in the generated repository and mapping logic. In a Spring Boot and Hibernate framework, such a problem could typically be mitigated by optimizing data-loading strategy, for example by using an explicit JOIN FETCH query, named entity graphs (@NamedEntityGraph and @EntityGraph annotations) or batch fetching.

4.6. Code Accuracy and Completeness

The implementation accuracy and completeness were manually assessed to determine how well each model satisfied the requirements specified during the four prompt-chaining phases. This metric assesses not only whether the generated code was syntactically correct, but also whether it met the intended functional, architectural, and configuration requirements of the backend application.

The development task was divided into twelve key functional units covering the full scope of the experiment. Each functional unit belonged to one of the functional categories: (1) domain model, (2) CRUD architecture, (3) advanced functionality, (4) configuration or testing. Each functional unit was evaluated using a three-level scoring scheme: a score of 1.0 was assigned when the requirement was fully implemented, 0.5 when it was only partially implemented or implemented with significant deficiencies, and 0.0 when the requirement was not implemented. The evaluation was conducted by three independent evaluators or raters. Scoring criteria were discussed prior to evaluation and disagreements were resolved through discussion. The evaluation results are presented in Table 4.

Table 4. Detailed evaluation of the implementation accuracy and completeness by functional categories and units. The evaluation was conducted by three independent evaluators.

Functional Category	Functional Unit	Evaluator 1	
		Gemini	DeepSeek
Domain model	Implementing entities and JPA relations	1	1
Domain model	Applying validation annotations	1	1
CRUD architecture	Generating complete layers (repo, service, controller)	1	1
CRUD architecture	Implementing CRUD endpoint for <i>Customer</i> and <i>Order</i> entities	1	1
CRUD architecture	Implementing CRUD endpoint for <i>Product</i> and <i>OrderItem</i> entities	1	1
Advanced functionality	Implementing filtering and product sorting	1	0.5
Advanced functionality	Implementing aggregation (top 5 products' retrieval)	1	0.83
Advanced functionality	Implementing order retrieval by user	0.5	0
Configuration	Correct configuration for PostgreSQL database	1	0.83
Testing	Generating unit tests (service)	1	0.67
Testing	Generating integration tests (repo)	0.83	0.17
Testing	Generating integration tests (controller)	0.5	0.17
Total points		10.83/12	8.17/12
Average		0.9	0.68
Std. dev.		0.19	0.36
Success ratio (%)		90.25	68.08

Inter-rater reliability, measured using Fleiss' Kappa, was 0.51 for the Gemini evaluation and 0.53 for the DeepSeek. The unanimous agreement rate, defined as the percentage of features received a 1.0 from all three raters, was 83.33% and 58.33% for Gemini and DeepSeek, respectively. The unanimous agreement rate metric alone strongly reinforces the necessity for strict manual code review when deploying LLM generated architectures.

Although both Gemini and DeepSeek eventually produced backend systems that contained the required structural components, their paths to functional completion differed significantly. Also, both models showed different strengths and weaknesses in their solutions.

Regarding the generation of the basic application structure, both models generated the necessary CRUD layers. However, Gemini required additional corrective prompts and several manual interventions because parts of its initial CRUD implementation were incomplete. DeepSeek was more comprehensive in this task, creating the repetitive CRUD structure more completely in its initial prompt response. This suggests that DeepSeek was more effective at producing boilerplate backend code, at least for structurally predictable implementation tasks such as this one.

More substantial differences appeared in the implementation of complex business logic, which was the most demanding part of the experiment. Two examples were particularly indicative: product filtering and sorting, and the retrieval of related data. In the case of product filtering and sorting, Gemini demonstrated a stronger understanding of REST API design conventions. It generated a single flexible endpoint capable of handling multiple optional parameters, resulting in a compact implementation as in Figure 10, and received the maximum score for this functional requirement. In contrast, DeepSeek implemented the same functionality only partially, using several separate endpoints. This solution did not fully satisfy the requirement for combined search and sorting functionality and was therefore evaluated as not completely correct.

```
// Class ProductController.java
@GetMapping
public ResponseEntity<List<ProductDTO>> getAllProducts(
    @RequestParam(required = false) String category,
    @RequestParam(required = false) BigDecimal minPrice,
    @RequestParam(required = false) BigDecimal maxPrice,
    @RequestParam(required = false, defaultValue = "asc") String sortDir
){
    return ResponseEntity.ok(
        productService.getAllProducts(category, minPrice, maxPrice, sortDir)
    );
}
```

Figure 10. An example of flexible endpoint in *ProductController* entity generated by Gemini 3 Pro.

The task of retrieving orders by customer revealed weaknesses in both generated backends, but with different severity. Gemini's solution (Figure 11) was functionally correct and as such partially satisfied the requirement. However, it introduced the aforementioned performance issue by retrieving related data through repeated database queries inside a loop. For this reason, it received only a partial score.

However, DeepSeek's solution for retrieving orders by customer was more problematic than Gemini's: by returning raw JPA entities with bidirectional relationships directly from the REST controller, it caused infinite recursive serialization. As a result, the endpoint would be functionally unusable as the generated code was not manually changed and received a score of zero for this requirement.

```
// Class OrderService.java
public List<OrderDTO> getOrdersByCustomerId(Long customerId) {
    return orderRepository.findById(customerId).stream()
        .map(this::toDto) // <-- Loop causing N+1 queries
        .collect(Collectors.toList());
}
```

Figure 11. Code generated by Gemini for *OrderService* entity contains the $N + 1$ select anti-pattern, where a new query to the database is triggered within the loop for each order.

Both models correctly implemented the retrieval of the top five products, indicating they could automatically generate simpler advanced queries when the expected logic was clearly defined and less dependent on architectural design decisions.

The differences were also present in test generation. Gemini produced a more reliable test suite, with only one recorded failure in the controller layer. DeepSeek generated slightly more problematic tests: five tests could not be executed because of errors, while two additional tests failed during execution. DeepSeek was less reliable in generating correct and executable test code, especially when the tests depended on proper configuration, dependency management, and interaction between multiple application layers.

Overall, the final implementation accuracy score clearly favors Gemini, which achieved 90.25% compared to 68.08% for DeepSeek. DeepSeek showed greater efficiency in generating simple and repetitive backend code, especially CRUD structures. However, its weaknesses in complex functionality, dependency configuration, test generation, and architectural decision-making significantly reduced the quality of the final system. Gemini, despite occasional incompleteness and a performance issue caused by the $N + 1$ select problem, produced a architecturally better and functionally more reliable backend solution.

4.7. System Architecture Analysis

The most significant and revealing difference between the models' behavior was in their autonomous architectural decision-making, especially concerning the boundary between the persistence layer (database) and the presentation layer (REST API).

The Gemini model demonstrated superior architectural reasoning by autonomously identifying the need for the DTO design pattern. Without explicit instruction in the Croatian prompts, Gemini recognized the inherent security and coupling risks of exposing internal database entities directly to the client. It proactively generated immutable Java Records to map internal JPA entities to external API JSON payloads, representing best practice and ensuring strict data encapsulation. However, the model's implementation was not without architectural issues; its persistence fetching strategy introduced $N + 1$ select performance degradation issue by failing to use JPA Entity Graphs or explicit JOIN FETCH directives for nested collection retrieval.

On the other hand, DeepSeek generated an architecturally inferior solution. It bypassed the DTO abstraction pattern by returning raw JPA @Entity instances directly from the @RestController endpoints. Given the bidirectional relationships defined in the provided UML domain model, this decision resulted in failure (Figure 12). When the Jackson JSON parsing library attempted to serialize the deeply nested entities, it immediately triggered an infinite recursion loop (for example, the *Order* entity referencing the *Customer* entity, which in turn references the *Order*, and so on) until the entire call stack memory space was exhausted.

Returning an *Order* object from the API causes the JSON serializer to enter an infinite loop, resulting in a deeply nested and unusable response, as shown in the abbreviated example (*id* = 7639) in Figure 13.

```
// Class Customer.java
@OneToMany(mappedBy = "customer")
private List<Order> orders;
...

// Class Order.java
@ManyToOne
@JoinColumn(name = "customer_id")
private Customer customer;
...
```

Figure 12. An example of the bidirectional relationship anti-pattern generated by DeepSeek-V3.1.

```
[
  {
    "id": 7639,
    "customer": {
      "id": 2137,
      "orders": [
        {
          "id": 7639,
          "customer": {
            "id": 2137,
            "orders": [{ ... }]
          }
        }
      ]
    },
    ...
  }
]
```

Figure 13. An example of an infinite recursion in the JSON response generated by DeepSeek-V3.1.

This critical functional bug caused immediate *StackOverflowError* exceptions at runtime, making the data retrieval API entirely unusable without significant manual refactoring by an expert developer understanding the cause of the problem. Furthermore, on the system's client side, tools such as Postman do not semantically detect recursive object structures such as this. Instead, they react to the consequences of the server-side serialization problem, such as an excessively large response, interrupted data transfer, connection failure, or response-size limit being reached. If the server manages to start sending the response before failing, Postman may terminate the display of the response or report an error related to maximum response size. Such behavior is therefore not the root cause of the problem, but only its visible manifestation on the client side.

Regarding the security implications of LLM-generated backend code, it should be noted that directly exposing raw JPA entities through REST controllers, as seen in the DeepSeek-generated solution, is not only a maintainability and serialization concern but may also increase the risk of unintended data exposure and weaken control over the public API contract. In addition, dependency misconfiguration and uncontrolled serialization behavior represent security risks that require expert review before LLM-generated backend code can be considered suitable for production use. However, these complex issues require specific and separate investigation.

4.8. Static Code Quality Analysis

The SonarQube tool provided details on the maintainability and reliability of the generated codebases. The results of the static backed code quality analysis are shown in Table 5 below.

Table 5. A comparison of code quality based on SonarQube analysis.

Quality Metric	Gemini	DeepSeek
Code smells	15	10
Technical debt (estimated time)	2 h 20 min	1 h 32 min

The results of the analysis may seem counterintuitive at first, but a more detailed contextual analysis strongly supports the results of the previous architectural assessment [83].

DeepSeek had fewer code smells (10 compared to Gemini's 15) and a lower estimated technical debt (1 h and 32 min versus 2 h and 20 min). This is expected, given the structural simplicity of the source code it generates. By removing the DTO layer and its associated object mapping logic, DeepSeek reduced the chance for potential syntactic and structural irregularities that SonarQube could detect. In contrast, Gemini had more code smells (mostly minor issues; string literal duplication and static analyzer suggestions to use newer Java API methods such as *Stream.toList()*) which increased its technical debt.

Most importantly, this highlights a known limitation of static analysis tools: SonarQube does not have the deeper semantic context to identify fundamental architectural anti-patterns in the code generated by DeepSeek. An infinite recursion flaw during JSON serialization was not flagged by the static analyzer as a bug or vulnerability, since the underlying code was still syntactically valid and compiled successfully.

4.9. SQL Schema Conceptualization and Database Entities Development

The structural integrity of the generated database schemas was evaluated based on their adherence to the provided UML class model diagram and the execution of SQL queries. Initial findings indicate that, while both tested models successfully identified core entities such as *Customer*, *Product*, and *Orders*, differences emerged regarding the normalization of the relationship between orders and products. Gemini demonstrated better initial reasoning by implementing a four-table structure, including *OrderItems*, to manage the Many-to-Many relationship. On the other hand, DeepSeek initially failed to generate the connecting table, requiring corrective prompting to achieve the desired SQL schema capable of tracking specific product quantities per order.

Regarding attribute generation consistency, the models exhibited similar levels of precision in complying with natural language prompts. Firstly, Gemini's initial output mostly mirrored the target UML diagram given in the prompt, successfully including all necessary fields in the *Product* table, such as *description* and *category*. However, Gemini incorrectly created the attributes *first_name* and *last_name* in *Customer* instead of *name* as required, failed to create *description* and *category* in *Product*, and failed to define the primary key in *OrderItem*. The adherence to the camel case naming format was also an issue. DeepSeek initially introduced redundant attributes such as *created_at* and *phone* fields, while omitting the required *category* attribute. Corrective prompts were necessary to change naming format, removing additional fields and to create *OrderItems* table. Despite these initial structural discrepancies, after corrective prompts both models eventually converged on a functional PostgreSQL schema and correctly implemented SERIAL primary keys and appropriate foreign key constraints to maintain relational integrity. Both Gemini and DeepSeek required three additional corrective prompts each. Interestingly, both models struggled with the required camel case (e.g., *customerOrder*, *orderItem*) and named tables as

well as attributes using the snake case (e.g., *customer_order*, *order_item*). The results for the generation of the SQL schema are listed in Table 6.

Table 6. Comparison of the Gemini 3 Pro and DeepSeek-V3.1 for generating SQL schema.

	Gemini	DeepSeek
Number of tables created	4	3
Primary key	BigSerial (Long for PostgreSQL)	BigSerial (Long for PostgreSQL)
Attributes	<i>Customer</i> (email with additional <i>first_name</i> and <i>last_name</i>) <i>Product</i> (name, price, but omitted <i>description</i> and <i>category</i>) <i>Orders</i> (correct; <i>customerId</i> and <i>orderId</i>) <i>OrderItems</i> (correct; <i>orderId</i> , <i>productId</i> , <i>quantity</i>)	<i>Customer</i> (email with additional <i>first_name</i> , <i>last_name</i> , <i>phone number</i> and <i>createdAt</i>) <i>Product</i> (name, price, <i>description</i> , but omitted <i>category</i> , <i>stock quantity</i> and <i>createdAt</i>) <i>Orders</i> (<i>customerId</i> and <i>orderId</i> , with additional <i>total amount</i> and <i>status</i>) <i>OrderItems</i> (entirely omitted)
Naming consistency	Used snake_case naming format, but did correct it to camel case format when prompted In SQL queries used snake_case naming format	Used snake_case naming format, but did correct it to camel case format when prompted Correctly used camel case naming format in SQL queries
Number of additional prompts	3	3

The efficacy of the generated schemas was further validated through the execution of complex data retrieval tasks involving multiple JOIN operations. Both models successfully generated syntactically correct queries for identifying high-value orders and calculating total customer expenditure. Notably, DeepSeek demonstrated greater robustness in its query logic by incorporating the COALESCE function to handle NULL values for customers without existing orders, ensuring data integrity in aggregate reports. Compared to the generation of the Java/Spring Boot stack, the development of the SQL schema was much more streamlined, and any issues that arose could be quickly resolved, even by novice developers.

5. Discussion

The overall analysis shows that under the examined experimental conditions, Gemini 3 Pro and DeepSeek-V3.1 exhibited distinctly different behavioral patterns during development. Both models contributed to a functional backend system, but they differed significantly in architectural quality, completeness, error severity, and the extent of human correction required.

In this experiment Gemini 3 Pro demonstrated stronger architectural reasoning and better handling of complex backend requirements. Its key strength was the autonomous implementation of the DTO pattern, which separated the persistence model from the REST API layer and helped avoid critical serialization problems. It also produced a more coherent implementation of advanced functionalities, particularly in product filtering and sorting,

where it generated a flexible endpoint aligned with REST API design practices. These decisions resulted in a more accurate final solution and more reliable tests. However, Gemini was not consistently complete. Its main weakness was the omission of repetitive boilerplate code, especially in the CRUD and testing layers, where some classes were missing or replaced with generic comments. Additionally, although its architecture was generally robust, the generated DTO mapping issues demonstrated that architectural maturity does not necessarily guarantee optimal runtime performance.

DeepSeek-V3.1 showed the opposite tendency. It was more effective at mechanically generating commonplace programming patterns and produced more complete CRUD structures in its initial prompt responses, making it useful for rapidly generating predictable application scaffolding. In many cases junior or novice developers will require only this level of vibe-coding output. However, its weaknesses were different than with Gemini. The model showed inconsistent knowledge of the specific Java/Spring Boot ecosystem mixing different versions. More importantly, it exposed raw JPA entities directly through REST controllers, causing an infinite recursion problem during JSON serialization when bidirectional relationships were present.

Taken together the results show that LLM-generated backend code should be evaluated not only for initial completeness, but also for additional elements such as architectural quality, maintainability, framework compatibility, runtime behavior, and the severity of generated errors.

However, the obtained findings should be interpreted with caution and strictly within the scope of this experiment. The results should not be understood as evidence of universal superiority of one model over another, nor as a general statement about all LLMs in backend software engineering. Rather, they indicate how the compared models behaved within the evaluated Java/Spring Boot scenario.

Research Limitations

Several limitations should be considered when interpreting the results of this study. The first and most obvious research limitation is the choice of LLMs that limit the experiment's generalizability. However, the goal of the study was not to compare all available LLMs, as it would not be possible to be completely exhaustive, but rather to compare two practically relevant model categories that would be used by inexperienced or moderately experienced backend developers in a realistic vibe-coding scenario. Gemini 3 Pro was chosen as an example of a proprietary, cloud-hosted, managed LLM service that can be accessed directly without requiring local installation or infrastructure configuration. DeepSeek-V3.1 was chosen as a conceptually opposite example of an openly available model family that allows for local or self-hosted deployment, making it useful for users who want more control over execution, data handling, and deployment conditions. Additionally, the experiment was conducted based on a single structured natural language interaction sequence per model. Since LLM outputs may vary, the results should not be interpreted as statistically generalizable evidence about the full distribution of possible model outputs.

Second, the experiment was conducted on a relatively simple backend system. Although the application included several typical components of a Java/Spring Boot backend, it did not have the complexity of large production systems. This design was intentional, as the study's goal was to create a baseline backend system that could subsequently be expanded into more complex systems, which could then be uniformly compared to this initial architecture. Future research should expand the experiment to include larger and more complex backend architectures.

Third, the experiment did not simulate the complete software development Continuous Integration and Continuous Delivery (CI/CD) pipeline [84] with repeated change

requests, evolving requirements, bug reports, regression testing, code reviews, and long-term maintenance tasks. As a result, the results primarily reflect the models' ability to generate and correct an initial backend implementation. It should be expected that in an actual CI/CD production environment, because of the increased complexity, the reported problems should be even more pronounced or amplified.

Fourth, the study did not include a cost analysis of the entire vibe-coding development process. The use of LLMs incurs token-based computational costs [20–24] which are dynamic pricing and LLM provider-dependent. Future research could include cost-efficiency metrics like cost per functional requirement, cost per successful test, or cost per manual intervention.

Finally, the rapid evolution of different LLMs represents an inherent limitation of any empirical comparison in this field [85–87]. New models and updated versions are continuously released, and their capabilities may improve significantly in a short period of time. Also, LLM behavior may change across versions, deployment modes, and configuration settings. As a result, the absolute performance values presented in this study should be interpreted as time-dependent. Nonetheless, the issues identified in this experiment are likely to remain present, even if their frequency decreases in newer models. It should also be noted that model progress is not always linear: in some cases, newer versions may perform worse than older versions on specific tasks [87]. Consequently, future research should periodically replicate the experiment with newer models.

6. Conclusions

This study provides a comprehensive, empirically driven comparative evaluation of two commonly used LLMs, Gemini and DeepSeek, in the context of enterprise backend software development, utilizing non-English prompting and enforcing modern technological standards for code analysis.

The presented results showed that the evaluated models can be successfully used in modern software development for vibe-coding, as they accelerate the development of functional backend architectures. However, our findings, even within a relatively limited scenario, highlight the mandatory need for caution. The most important insight from the study is that even the best foundational models are not capable of fully autonomous deployment, even in the most basic enterprise backend development scenarios. As a result, the generative process requires continuous guidance and expert validation from an experienced software engineer with a comprehensive understanding of the generated codebase. The vibe-coding paradigm should be understood as a joint human–LLM development workflow rather than as standalone software engineering.

The authors plan to address several important directions in future research. First, it would be valuable to systematically compare general-purpose conversational models and vibe-coding tools, on one side, and traditional code generation models that do not rely on GenAI on the other. This research could determine whether narrow specialization in code generation leads to better software engineering solutions than the broader contextual NLP capabilities of general-purpose LLMs. Second, the experimental results indicate that many generated errors were not immediately visible in individual code fragments but emerged later during compilation or in testing and runtime. Therefore, future research, within the area of software engineering, could empirically investigate the usefulness of integration LLM-assisted backend development in CASE tools. Finally, an ablative study of prompt engineering techniques for backend vibe-coding, which was beyond the scope of this manuscript, should be conducted. Comparing identical tasks when prompted with different methods and in various natural languages would provide a better insight into the reasoning capabilities of contemporary LLMs in the software engineering context. Such

research could reveal whether certain prompts introduce reduced architectural precision and if LLM performance differs from results obtained through interaction conducted exclusively in English.

All parts of the dataset used in the experiment that are required to replicate the reported results, including the Croatian prompts, English translations of the prompts, corrective prompts, generated code, SQL schema scripts, and supplementary implementation materials, are freely available for research and non-commercial use upon contacting the corresponding author.

Author Contributions: Conceptualization, M.H. and I.U.; methodology, M.H. and I.U.; software, I.U.; validation, M.H., I.U. and K.K.; formal analysis, I.U.; investigation, I.U. and K.K.; resources, I.U. and K.K.; data curation, I.U.; writing—original draft preparation, M.H., I.U. and K.K.; writing—review and editing, M.H., I.U. and K.K.; visualization, M.H., I.U. and K.K.; supervision, M.H. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The data supporting the findings of this study are available from the corresponding author upon reasonable request.

Acknowledgments: During the preparation of this manuscript, some of the authors used Google Gemini Pro 3.1, QuillBot Premium and InstaText Premium tools to improve the language and check spelling. The authors have reviewed and edited the output and take full responsibility for the content of this publication.

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A

As described in Section 3.2, the following SQL script was employed to populate the database with a sufficient volume of synthetic data (5000 customers, 20,000 products, 50,000 orders) and to establish realistic conditions for evaluating the system's performance:

-- 1. GENERATE Customers

INSERT INTO "customers" (name, email)

SELECT

'Kupac ' || n, -- Name in the format "Kupac 1", "Kupac 2", ..., "Kupac 5000", in English "Customer 1", "Customer 2", etc.

'kupac.' || n || '@example.com'

FROM generate_series(1, 5000) AS n;

-- 2. GENERATE Products

INSERT INTO "products" (name, description, price, category)

SELECT

'Proizvod ' || n, -- Name in the format "Proizvod 1", "Proizvod 2", ..., "Proizvod 20,000", in English "Product 1", "Product 2", etc.

'Detaljan opis za proizvod ' || n, -- Description in the format "Detaljan opis za proizvod 1", "Detaljan opis za proizvod 2", ..., in English "Detailed description for product 1", "Detailed description for product 2", etc.

-- Price between 10.00 and 1010.00

(random() * 1000 + 10)::numeric(10, 2),

-- Randomly select one of the 5 categories (Elektronika, Odjeća, Knjige, Kućanstvo, Sport) with equal probability. In English, Electronics, Clothing, Books, Home, Sports.

(ARRAY['Elektronika', 'Odjeća', 'Knjige', 'Kućanstvo', 'Sport'])[floor(random() * 5 + 1)]

FROM generate_series(1, 20,000) AS n;

```

-- 3. GENERATE Orders
INSERT INTO "orders" ("order_date", "customer_id")
SELECT
    -- Random date and time in the last 5 years
    NOW() - (random() * 365 * 5) * '1 day'::interval,
    -- Random customer ID between 1 and 5000
    floor(random() * 5000 + 1)::bigint
FROM generate_series(1, 50000) AS n;

-- 4. GENERATE OrderItems
DO $$
DECLARE
    order_id bigint;
    num_items int;
    i int;
BEGIN
    -- Loop through each order that was just created
    FOR order_id IN SELECT id FROM "orders" LOOP
        -- Each order will have between 1 and 7 items (randomly determined)
        num_items := floor(random() * 7 + 1);
        FOR i IN 1..num_items LOOP
            INSERT INTO "order_items" ("order_id", "product_id", "quantity")
            VALUES (
                order_id,
                -- Random product ID between 1 and 20,000
                floor(random() * 20,000 + 1)::bigint,
                -- Quantity between 1 and 5
                floor(random() * 5 + 1)
            );
        END LOOP;
    END LOOP;
END $$;

```

References

1. Meske, C.; Hermanns, T.; Von der Weiden, E.; Loser, K.U.; Berger, T. Vibe coding as a reconfiguration of intent mediation in software development: Definition, implications, and research agenda. *IEEE Access* **2025**, *13*, 213242–213259. [\[CrossRef\]](#)
2. Leau, Y.B.; Loo, W.K.; Tham, W.Y.; Tan, S.F. Software development life cycle AGILE vs traditional approaches. In *Proceedings of the International Conference on Information and Network Technology*; IACSIT Press: Singapore, 2012; Volume 37, pp. 162–167.
3. Stober, T.; Hansmann, U. Traditional software development. In *Agile Software Development: Best Practices for Large Software Development Projects*; Springer: Berlin/Heidelberg, Germany, 2009; pp. 15–33.
4. Michael, J.; Cleophas, L.; Zschaler, S.; Clark, T.; Combemale, B.; Godfrey, T.; Khelladi, D.E.; Kulkarni, V.; Lehner, D.; Rumpe, B.; et al. Model-driven engineering for digital twins: Opportunities and challenges. *Syst. Eng.* **2025**, *28*, 659–670.
5. Schmidt, D.C. Model-driven engineering. *Computer* **2006**, *39*, 25. [\[CrossRef\]](#)
6. Verbruggen, C.; Snoeck, M. Practitioners' experiences with model-driven engineering: A meta-review. *Softw. Syst. Model.* **2023**, *22*, 111–129.
7. Shafiee, S.; Wautelet, Y.; Friis, S.C.; Lis, L.; Harlou, U.; Hvam, L. Evaluating the benefits of a computer-aided software engineering tool to develop and document product configuration systems. *Comput. Ind.* **2021**, *128*, 103432. [\[CrossRef\]](#)
8. Sarkar, A.; Drosos, I. Vibe coding: Programming through conversation with artificial intelligence. *arXiv* **2025**, arXiv:2506.23253.
9. Fischer, M.; Lanquillon, C. Evaluation of generative AI-assisted software design and engineering: A user-centered approach. In *Proceedings of the International Conference on Human-Computer Interaction*; Springer Nature: Cham, Switzerland, 2024; pp. 31–47.
10. Elgendy, I.A.; Dwivedi, Y.K.; Al-Sharafi, M.A.; Hosny, M.; Helal, M.Y.; Crick, T.; Hughes, L.; Alwahaishi, S.; Mahmud, M.; Dutot, V.; et al. Responsible Vibe Coding: Architecture, Opportunities, and Research Agenda. *J. Comput. Inf. Syst.* **2026**, 1–19. [\[CrossRef\]](#)

11. Sapkota, R.; Roumeliotis, K.I.; Karkee, M. Vibe coding vs. agentic coding: Fundamentals and practical implications of agentic ai. *arXiv* **2025**, arXiv:2505.19443.
12. Horvat, M.; Kralj, B.; Gledec, G. A Comparative Study of Vibe Coding with ChatGPT and Gemini in Front-end Web Development. In *Proceedings of the 36th International Scientific Conference: Central European Conference on Information and Intelligent Systems (CECIIS 2025)*; University of Zagreb, Faculty of Organization and Informatics: Varaždin, Croatia, 2025; pp. 787–796.
13. Ljubi, I.; Grgić, Z.; Vuković, M.; Gledec, G. Detecting Disinformation in Croatian Social Media Comments. *Future Internet* **2025**, *17*, 178. [\[CrossRef\]](#)
14. Divjak, D.; Sharoff, S.; Erjavec, T. Slavic corpus and computational linguistics. *J. Slav. Linguist.* **2017**, *25*, 171–199. [\[CrossRef\]](#)
15. Gledec, G.; Sokele, M.; Horvat, M.; Mikuc, M. Error pattern discovery in spellchecking using multi-class confusion matrix analysis for the Croatian language. *Computers* **2024**, *13*, 39. [\[CrossRef\]](#)
16. Gledec, G.; Horvat, M.; Mikuc, M.; Blašković, B. A Comprehensive Dataset of Spelling Errors and Users' Corrections in Croatian Language. *Data* **2023**, *8*, 89. [\[CrossRef\]](#)
17. Gadde, A. Democratizing software engineering through generative ai and vibe coding: The evolution of no-code development. *J. Comput. Sci. Technol. Stud.* **2025**, *7*, 556–572. [\[CrossRef\]](#)
18. Ge, Y.; Mei, L.; Duan, Z.; Li, T.; Zheng, Y.; Wang, Y.; Wang, L.; Yao, J.; Liu, T.; Cai, Y.; et al. A survey of vibe coding with large language models. *arXiv* **2025**, arXiv:2510.12399.
19. Osmani, A. *Beyond Vibe Coding: From Coder to AI-Era Developer*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2025.
20. Ray, P.P. A review on vibe coding: Fundamentals, state-of-the-art, challenges and future directions. *TechRxiv* **2025**. [\[CrossRef\]](#) [\[PubMed\]](#)
21. Kusper, G.; Szabó, C. Vibe Coding in Education. In *Proceedings of the 2025 International Conference on Emerging eLearning Technologies and Applications (ICETA)*; IEEE: New York, NY, USA, 2025; pp. 506–511.
22. Geng, F.; Shah, A.; Li, H.; Mulla, N.; Swanson, S.; Soosai Raj, G.; Zingaro, D.; Porter, L. Exploring student-AI interactions in vibe coding. In *Proceedings of the 28th Australasian Computing Education Conference*; Association for Computing Machinery: New York, NY, USA, 2026; pp. 45–54.
23. Šarčević, A.; Tomićić, I.; Merlin, A.; Horvat, M. Enhancing programming education with open-source generative AI chatbots. In *Proceedings of the 2024 47th MIPRO ICT and Electronics Convention (MIPRO)*; IEEE: New York, NY, USA, 2024; pp. 2051–2056.
24. Horvat, M. What is Vibe coding and when should you use it (or not)? *TechRxiv* **2025**. [\[CrossRef\]](#) [\[PubMed\]](#)
25. Fan, Y.; Tang, L.; Le, H.; Shen, K.; Tan, S.; Zhao, Y.; Shen, Y.; Li, X.; Gašević, D. Beware of metacognitive laziness: Effects of generative artificial intelligence on learning motivation, processes, and performance. *Br. J. Educ. Technol.* **2025**, *56*, 489–530.
26. Kosmyrna, N.; Hauptmann, E.; Yuan, Y.T.; Situ, J.; Liao, X.H.; Beresnitzky, A.V.; Braunstein, I.; Maes, P. Your brain on ChatGPT: Accumulation of cognitive debt when using an AI assistant for essay writing task. *arXiv* **2025**, arXiv:2506.08872.
27. Gerlich, M. AI tools in society: Impacts on cognitive offloading and the future of critical thinking. *Societies* **2025**, *15*, 6. [\[CrossRef\]](#)
28. Cabot, J. Vibe modeling: Challenges and opportunities. In *Proceedings of the International Conference on Conceptual Modeling*; Springer Nature: Cham, Switzerland, 2025; pp. 105–118.
29. Revuri, J.; Sakthivel, R.K.; Nagasubramanian, G. Artificial intelligence (AI) technologies and tools for accelerated software development. In *Advances in Computers*; Elsevier: Amsterdam, The Netherlands, 2026; Volume 141, pp. 115–159.
30. Alharbi, M.; Alshayeb, M. Automatic Code Generation Techniques: A Systematic Literature Review. *Autom. Softw. Eng.* **2026**, *33*, 4.
31. Odeh, A. Exploring AI innovations in automated software source code generation: Progress, hurdles, and future paths. *Informatica* **2024**, *48*, 125–136. [\[CrossRef\]](#)
32. Chou, Y.H.; Jiang, B.; Chen, Y.W.; Weng, M.; Jackson, V.; Zimmermann, T.; Jones, J.A. Building Software by Rolling the Dice: A Qualitative Study of Vibe Coding. *arXiv* **2025**, arXiv:2512.22418.
33. Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H.P.d.O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. Evaluating large language models trained on code. *arXiv* **2021**, arXiv:2107.03374.
34. Zheng, Q.; Xia, X.; Zou, X.; Dong, Y.; Wang, S.; Xue, Y.; Shen, L.; Wang, Z.; Wang, A.; Li, Y.; et al. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*; Association for Computing Machinery: New York, NY, USA, 2023; pp. 5673–5684.
35. Yu, Z.; Zhao, Y.; Cohan, A.; Zhang, X.P. Humaneval pro and mbpp pro: Evaluating large language models on self-invoking code generation. *arXiv* **2024**, arXiv:2412.21199.
36. Fawzy, A.; Tahir, A.; Blincoe, K. Vibe Coding in Practice: Motivations, Challenges, and a Future Outlook—A Grey Literature Review. *arXiv* **2025**, arXiv:2510.00328.
37. Jahić, J.; Sami, A. State of practice: Llms in software engineering and software architecture. In *Proceedings of the 2024 IEEE 21st International Conference on Software Architecture Companion (ICSA-C)*; IEEE: New York, NY, USA, 2024; pp. 311–318.

38. Zhong, P.; Vaezipoor, P.; Cui, F.; Kumar, V.; Asgarian, A.; Austin, J.; Ho, T.; Inder, P.; Kedir, I.; Catasta, M.; et al. ViBench: A Benchmark on Vibe Coding. In *Proceedings of the 1st ACM Conference on Agentic and AI Systems (CAIS'26)*; ACM: New York, NY, USA, 2026.
39. Liu, J.; Wang, K.; Chen, Y.; Peng, X.; Chen, Z.; Zhang, L.; Lou, Y. Large language model-based agents for software engineering: A survey. In *ACM Transactions on Software Engineering and Methodology*; Association for Computing Machinery: New York, NY, USA, 2024.
40. Panichella, S. Vulnerabilities introduced by llms through code suggestions. In *Large Language Models in Cybersecurity: Threats, Exposure and Mitigation*; Springer Nature: Cham, Switzerland, 2024; pp. 87–97.
41. Xie, Y.; Wu, S.; Chakravarty, S. AI meets AI: Artificial intelligence and academic integrity-A survey on mitigating AI-assisted cheating in computing education. In *Proceedings of the 24th Annual Conference on Information Technology Education*; Association for Computing Machinery: New York, NY, USA, 2023; pp. 79–83.
42. Chen, B.; Lewis, C.M.; West, M.; Zilles, C. Plagiarism in the age of generative ai: Cheating method change and learning loss in an intro to CS course. In *Proceedings of the Eleventh ACM Conference on Learning @ Scale*; Association for Computing Machinery: New York, NY, USA, 2024; pp. 75–85.
43. Pan, W.H.; Chok, M.J.; Wong, J.L.S.; Shin, Y.X.; Poon, Y.S.; Yang, Z.; Yang, Z.; Chong, C.Y.; Lo, D.; Lim, M.K. Assessing ai detectors in identifying ai-generated code: Implications for education. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*; Association for Computing Machinery: New York, NY, USA, 2024; pp. 1–11.
44. Baek, J.; Yamazaki, T.; Morihata, A.; Mori, J.; Yamakata, Y.; Taura, K.; Chiba, S. LLM-Based Explainable Detection of LLM-Generated Code in Python Programming Courses. In *Proceedings of the 57th ACM Technical Symposium on Computer Science Education V. 1*; Association for Computing Machinery: New York, NY, USA, 2026; pp. 80–86.
45. Song, F.; Agarwal, A.; Wen, W. The impact of generative AI on collaborative open-source software development: Evidence from GitHub Copilot. *arXiv* **2024**, arXiv:2410.02091.
46. Becker, J.; Rush, N.; Barnes, E.; Rein, D. Measuring the impact of early-2025 AI on experienced open-source developer productivity. *arXiv* **2025**, arXiv:2507.09089.
47. Esteban Cuellar Argotty, J.; Manrique, R. AI-Generated Code Detection: An Examination of Current Tools in Education. In *Proceedings of the International Conference on Intelligent Tutoring Systems*; Springer Nature: Cham, Switzerland, 2025; pp. 192–201.
48. Mekterović, I.; Brkić, L.; Horvat, M. Scaling automated programming assessment systems. *Electronics* **2023**, *12*, 942. [[CrossRef](#)]
49. Jiang, J.; Wang, F.; Shen, J.; Kim, S.; Kim, S. A survey on large language models for code generation. *ACM Trans. Softw. Eng. Methodol.* **2026**, *35*, 58. [[CrossRef](#)]
50. Zheng, D.; Wang, Y.; Shi, E.; Zhang, H.; Zheng, Z. How well do llms generate code for different application domains? benchmark and evaluation. *arXiv* **2024**, arXiv:2412.18573.
51. Huynh, N.; Lin, B. Large language models for code generation: A comprehensive survey of challenges, techniques, evaluation, and applications. *arXiv* **2025**, arXiv:2503.01245.
52. Stoyanova, M. Integrating Logic Programming with Large Language Models: Opportunities and Challenges. In *Strategic Responses to Global Uncertainty: Rethinking Markets, Governance and Innovation*; University of Economics–Varna: Varna, Bulgaria, 2025; pp. 512–524.
53. Zhong, L.; Wang, Z. Can llm replace stack overflow? A study on robustness and reliability of large language model code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*; Association for the Advancement of Artificial Intelligence: Washington, DC, USA, 2024; Volume 38, pp. 21841–21849.
54. Paul, D.G.; Zhu, H.; Bayley, I. Does LLM Generated Code Smell? In *Proceedings of the 2025 9th International Conference on Cloud and Big Data Computing (ICCBDC)*; Association for Computing Machinery: New York, NY, USA, 2025; pp. 68–73.
55. Comanici, G.; Bieber, E.; Schaekermann, M.; Pasupat, I.; Sachdeva, N.; Dhillon, I.; Blistein, M.; Ram, O.; Zhang, D.; Rosen, E.; et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv* **2025**, arXiv:2507.06261.
56. Liu, A.; Feng, B.; Xue, B.; Wang, B.; Wu, B.; Lu, C.; Zhao, C.; Deng, C.; Zhang, C.; Ruan, C.; et al. Deepseek-v3 technical report. *arXiv* **2024**, arXiv:2412.19437.
57. Saeidnia, H.R. Welcome to the Gemini era: Google DeepMind and the information industry. *Libr. Hi Tech News* **2023**, *43*, 18–20. [[CrossRef](#)]
58. Deng, Z.; Ma, W.; Han, Q.L.; Zhou, W.; Zhu, X.; Wen, S.; Xiang, Y. Exploring DeepSeek: A survey on advances, applications, challenges and future directions. *IEEE/CAA J. Autom. Sin.* **2025**, *12*, 872–893. [[CrossRef](#)]
59. Chaturvedi, V. Modern software development with Java, Spring Boot, and Python: A survey of frameworks and best practices. *ESP J. Eng. Technol. Adv.* **2023**, *3*, 188–197.
60. Abdulkareem Hamaamin, R.; Mohammed Amin Ali, O.; Wahhab Kareem, S. Java programming language: Time permanence comparison with other languages: A review. *ITM Web Conf.* **2024**, *64*, 01012. [[CrossRef](#)]

61. de Oliveira, C.E.; Turnquist, G.L.; Antonov, A. *Developing Java Applications with Spring and Spring Boot: Practical Spring and Spring Boot Solutions for Building Effective Applications*; Packt Publishing Ltd.: Birmingham, UK, 2018.
62. Bharathan, R. *Apache Maven Cookbook*; Packt Publishing Ltd.: Birmingham, UK, 2015.
63. Krochmalski, J. *IntelliJ IDEA Essentials*; Packt Publishing Ltd.: Birmingham, UK, 2014.
64. Bonteanu, A.M.; Tudose, C. Performance analysis and improvement for CRUD operations in relational databases from java programs using JPA, hibernate, spring data JPA. *Appl. Sci.* **2024**, *14*, 2743. [[CrossRef](#)]
65. Tudose, C.; Odubăşteanu, C. Object-relational mapping using JPA, hibernate and spring data JPA. In *Proceedings of the 2021 23rd International Conference on Control Systems and Computer Science (CSCS)*; IEEE: New York, NY, USA, 2021; pp. 424–431.
66. Tudose, C.; Bauer, C.; King, G. *Java Persistence with Spring Data and Hibernate*; Simon and Schuster: New York, NY, USA, 2023.
67. Drake, J.D.; Worsley, J.C. *Practical PostgreSQL*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2002.
68. Douglas, K.; Douglas, S. *PostgreSQL: A Comprehensive Guide to Building, Programming, and Administering PostgreSQL Databases*; SAMS Publishing: Indianapolis, IN, USA, 2003.
69. Wang, Z.; Liu, R.; Fu, J. Prompting. In *Interactive Natural Language Processing: Language Model as Agent*; Springer Nature: Cham, Switzerland, 2026; pp. 87–101.
70. Parker, G.; Kim, S.; Al Maruf, A.; Cerny, T.; Frajtak, K.; Tisnovsky, P.; Taibi, D. Visualizing anti-patterns in micro-services at runtime: A systematic mapping study. *IEEE Access* **2023**, *11*, 4434–4442.
71. Sharma, A.; Chaturvedi, A.; Tripathi, A.K. From problem descriptions to user stories: Utilizing large language models through prompt chaining. In *Proceedings of the 2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT)*; IEEE: New York, NY, USA, 2024; pp. 1–6.
72. Horvat, M. Which Prompting Technique is Better? Improving Vibe Coding Code Quality with Efficient Prompt Design for Web Front-End Development. *Interaction* **2026**, *9*, 10.
73. Marcilio, D.; Bonifácio, R.; Monteiro, E.; Canedo, E.; Luz, W.; Pinto, G. Are static analysis violations really fixed? a closer look at realistic usage of sonarqube. In *Proceedings of the 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*; IEEE: New York, NY, USA, 2019; pp. 209–219.
74. Gupta, S.; Bhatia, M.; Memoria, M.; Manani, P. Prevalence of GitOps, DevOps in fast CI/CD cycles. In *Proceedings of the 2022 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COM-IT-CON)*; IEEE: New York, NY, USA, 2022; Volume 1, pp. 589–596.
75. Paiva, T.; Damasceno, A.; Figueiredo, E.; Sant'Anna, C. On the evaluation of code smells and detection tools. *J. Softw. Eng. Res. Dev.* **2017**, *5*, 7. [[CrossRef](#)]
76. Fontana, F.A.; Dietrich, J.; Walter, B.; Yamashita, A.; Zaroni, M. Antipattern and code smell false positives: Preliminary conceptualization and classification. In *Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*; IEEE: New York, NY, USA, 2016; Volume 1, pp. 609–613.
77. Das, D.; Maruf, A.A.; Islam, R.; Lambaria, N.; Kim, S.; Abdelfattah, A.S.; Cerny, T.; Frajtak, K.; Bures, M.; Tisnovsky, P. Technical debt resulting from architectural degradation and code smells: A systematic mapping study. *ACM SIGAPP Appl. Comput. Rev.* **2022**, *21*, 20–36.
78. Pantaleev, A.; Rountev, A. Identifying data transfer objects in EJB applications. In *Proceedings of the Fifth International Workshop on Dynamic Analysis (WODA'07)*; IEEE: New York, NY, USA, 2007; p. 5.
79. Pardede, C.; Sihombing, W.; Nainggolan, W. Comparative Study of Manual and Generated Data Transfer Object Implementation Performance. *J. Appl. Inform. Comput.* **2025**, *9*, 2912–2919. [[CrossRef](#)]
80. Hemmati, H. How effective are code coverage criteria? In *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security*; IEEE: New York, NY, USA, 2015; pp. 151–156.
81. Al-Ahmad, B. Using Code Coverage Metrics for Improving Software Defect Prediction. *J. Softw.* **2018**, *13*, 654–674. [[CrossRef](#)]
82. Elbaum, S.; Gable, D.; Rothermel, G. The impact of software evolution on code coverage information. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2001)*; IEEE: New York, NY, USA, 2001; pp. 170–179.
83. Wikantayasa, I.M.A.; Kurniawan, A.P.; Rochimah, S. CK metric and architecture smells relations: Towards software quality assurance. In *Proceedings of the 2023 14th International Conference on Information & Communication Technology and System (ICTS)*; IEEE: New York, NY, USA, 2023; pp. 13–17.
84. Arachchi, S.A.I.B.S.; Perera, I. Continuous integration and continuous delivery pipeline automation for agile software project management. In *Proceedings of the 2018 Moratuwa Engineering Research Conference (MERCon)*; IEEE: New York, NY, USA, 2018; pp. 156–161.
85. Rahmani, A.M.; Hemmati, A.; Abbasi, S. The Rise of Large Language Models: Evolution, Applications, and Future Directions. *Eng. Rep.* **2025**, *7*, e70368. [[CrossRef](#)]

86. Harsha, K.; Tarun Kumar, K.; Sumathi, D.; Ajith Jubilson, E. A survey on LLMs: Evolution, applications, and future frontiers. In *Generative AI: Current Trends and Applications*; Springer Nature: Singapore, 2024; pp. 289–327.
87. Patil, R.; Gudivada, V. A review of current trends, techniques, and challenges in large language models (LLMs). *Appl. Sci.* **2024**, *14*, 2074. [[CrossRef](#)]

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.