

Prompting Large Language Models to Tackle the Full Software Development Lifecycle: A Case Study

Bowen Li^{1*}, Wenhan Wu^{2*†}, Ziwei Tang^{3*†}, Lin Shi^{4*†}, John Yang⁵, Jinyang Li⁶, Shunyu Yao⁵, Chen Qian⁷, Binyuan Hui⁸, Qicheng Zhang¹, Zhiyin Yu¹, He Du¹, Ping Yang⁹, Dahua Lin¹, Chao Peng^{9‡}, Kai Chen^{1‡}

¹Shanghai AI Laboratory, ²Nanjing University, ³BUPT, ⁴Dartmouth College, ⁵Princeton University, ⁶The University of Hong Kong, ⁷Tsinghua University, ⁸Alibaba Group, ⁹ByteDance

Correspondence: chenkai@pjlab.org.cn and pengchao.x@bytedance.com

Abstract

Recent advancements in large language models (LLMs) have significantly enhanced their coding capabilities. However, existing benchmarks predominantly focused on simplified or isolated aspects of coding, such as single-file code generation or repository issue debugging, falling short of measuring the full spectrum of challenges raised by real-world programming activities. In this case study, we explore the performance of LLMs across the entire software development lifecycle with DevEval, encompassing stages including software design, environment setup, implementation, acceptance testing, and unit testing. DevEval features four programming languages, multiple domains, high-quality data collection, and carefully designed and verified metrics for each task. Empirical studies show that current LLMs, including GPT-4, fail to solve the challenges presented within DevEval. Our findings offer actionable insights for the future development of LLMs toward real-world programming applications. ¹

1 Introduction

Given its practical value and reasoning challenges, programming has become an important domain to deploy and evaluate large language models (LLMs), leading to popular products like GitHub Copilot and benchmarks like HumanEval (Chen et al., 2021) and APPS (Hendrycks et al., 2021). While these earlier coding tasks focused on generating a single code file or even a single method from simple instructions, recent works such as SWE-bench (Jimenez et al., 2023) and RepoBench (Liu et al., 2023b) evaluate LLMs on repository-level tasks, which feature longer, more

involved NL2Code problems. Still, these benchmarks concentrate on narrow aspects of software development, leaving a gap in comprehensive studies that encompass the full software development lifecycle across its various phases.

To address these shortcomings and fill this gap, we present DevEval, a comprehensive case study that mirrors real-world software development. DevEval generally evaluates models on the task of constructing a multi-file codebase starting from a product requirement document (PRD) of detailed specifications. Subscribing to the traditional Waterfall software development model (Royce, 1987), DevEval breaks down this process into a diverse set of inter-related development stages, i.e., software design, environment setup, implementation, acceptance and unit testing, as visualized in Figure 1 and Table 1. In contrast to previous works, DevEval is the first to evaluate models' software design and environment setup capabilities. One significant challenge in this study lies in the scarcity of publicly available repositories that include the full range of software development artifacts, particularly design documents and comprehensive testing programs. To overcome this, we curated a collection of 22 repositories across four programming languages (Python, C/C++, Java, JavaScript), spanning various domains such as machine learning, web services, and command-line utilities. By encompassing the multi-faceted, interconnected steps of software development under a single framework, DevEval provides a holistic view of LLMs' capabilities for automated software production, moving beyond the conventional focus on code completion.

Through a comprehensive experimental study, our findings indicate that tested models struggle significantly with the challenges presented. GPT-4-Turbo achieves the highest scores amongst all evaluated models, yet it obtains less than 10% on our repository-level implementation task. Other tasks prove relatively more manageable for mod-

*Equal contribution.

†Work done during internship at Shanghai AI Laboratory.

‡Corresponding authors.

¹Our data and code are available at <https://github.com/open-compass/DevEval>.

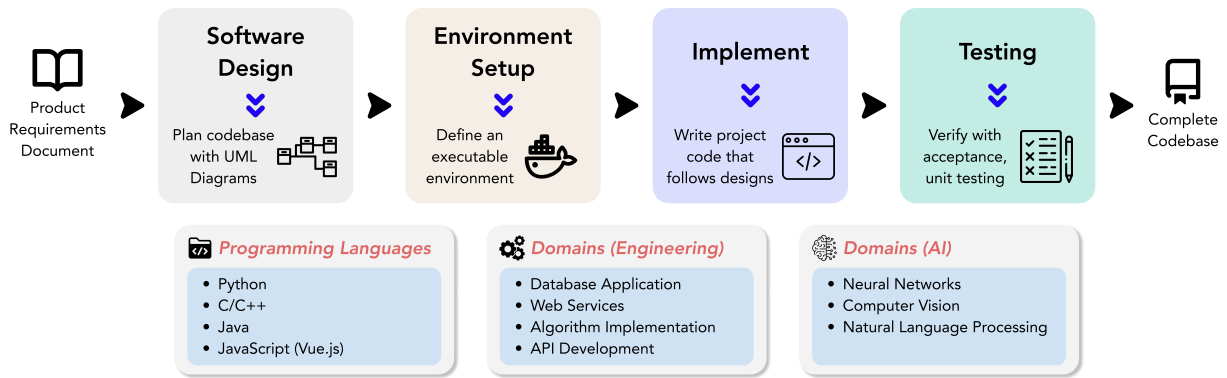


Figure 1: Our DevEval features multiple stages of software development, including software design, environment setup, implementation, and testing (both acceptance and unit testing).

Task	Input [†]	Output	Environment	Evaluation
Software Design	PRD	UML Diagrams [‡] , Architecture Design	N/A	Subjective Evaluation
Environment Setup	PRD, UML Diagrams, Architecture Design	Dependency Files	Base	Pass Rate on Usage Examples
Implementation	PRD, UML Diagrams, Architecture Design	Implementation Code	Reference	Unit & Acceptance Testing
Acceptance Testing	PRD, UML Diagrams, Architecture Design, Implementation Code*	Acceptance Testing Code	Reference	Oracle Test
Unit Testing	PRD, UML Diagrams, Architecture Design, Implementation Code	Unit Testing Code	Reference	Oracle Test & Coverage

Table 1: Task design in DevEval. †: following our modular evaluation protocol, the input for each task are reference. ‡: includes UML class and sequence diagrams. *: implementation source code is optional for acceptance testing.

els; however, even GPT-4-Turbo struggles to attain scores exceeding 40% on the more complicated ones. For instance, models generally fail to generate executable tests, with oracle test scores falling below 40%. Despite this, the generated testing code demonstrates potential in code coverage, achieving as high as 79.4% when it is executable. Furthermore, our investigation into different prompting methods shows that prompts with external information and execution feedback could yield notable and consistent improvements. Importantly, qualitative analysis shows that models demonstrate difficulties in handling Makefile and Gradle, configuring function arguments and employing advanced object-oriented programming techniques. Overall, DevEval introduces a novel challenge for existing LLMs, and our investigation sheds light on fundamental issues, paving the way for future research.

2 Related Work

Initial code generation datasets focused on self-contained, Python-based completion prob-

lems (Chen et al., 2021; Hendrycks et al., 2021; Austin et al., 2021), while later works expanded complexity by increasing language coverage (Zheng et al., 2023b; Cassano et al., 2023), enhancing execution-based test coverage (Liu et al., 2023a; Wang et al., 2022), incorporating dependencies (Lai et al., 2022; Ding et al., 2023; Liu et al., 2023c), introducing interactive environments (Yin et al., 2022; Yang et al., 2023), and developing short-form task suites (Lu et al., 2021; Muenighoff et al., 2023). DevEval aligns with recent repository-scale coding works (Jimenez et al., 2023; Liu et al., 2023b; Zhang et al., 2023). However, our work distinctly evaluates LLMs’ ability to create entire codebases from extensive natural language descriptions, offering a more comprehensive test of LLM capabilities. In automatic programming, role-play frameworks with communicative agents (Hong et al., 2023; Li et al., 2023; Qian et al., 2023a,b) like MetaGPT (Hong et al., 2023) integrate structured workflows and task modularity, similar to our approach. Compare against

MetaGPT, DevEval stands out by providing a structured evaluation across all phases of the software development lifecycle. Additionally, there is a growing trend in agent-based frameworks specially targeting software engineering tasks (Yang et al., 2024; Zhang et al., 2024; Liu et al., 2024; Wang et al., 2024a).

3 DevEval

In this section, we discuss the design of DevEval, including task specifications and the evaluation criteria (summarized in Table 1). Adhering to the modular approach, our design utilizes reference inputs for each task. This strategy enables and concentrates on evaluating the efficacy of models in executing specific tasks.²

3.1 Task 1: Software Design

During this phase, the model is tasked with interpreting the PRD to create the design of the system before development. The first subtask involves the generation of the UML class diagram using Mermaid syntax, which models the structural aspects of software systems, detailing the classes within the system, their attributes, operations and the relationship among them. Class diagrams help developers understand the system’s foundation before development begins. The second subtask focuses on the creation of the UML sequence diagram using Mermaid syntax. These diagrams clarify the collaboration among components of the system by mapping out the interaction between objects and processes over time, illustrating the sequence of messages exchanged between objects to implement the system’s functionality. The final subtask formulates architectural designs using hierarchical file tree structures, aiming to establish a structured framework for the source code, build scripts and necessary auxiliary files. The architectural design ensures a cohesive structure for coding, testing and maintenance.

Evaluation. Since the Software Design tasks are open-ended, we employ the *LLM-as-a-judge* approach (Zheng et al., 2023a; Wang et al., 2023; Chiang and Lee, 2023) to conduct the automatic evaluation. The evaluation is anchored by two principal metrics: *general principles* and *faithfulness*.

²Notably, our framework can be adeptly configured to facilitate end-to-end evaluations, utilizing the intermediate outputs generated by models across multiple tasks. Moreover, DevEval can also function in a Copilot mode, empowering human users to intervene and refine model outputs, thus enhancing the collaborative synergy between human expertise and automated systems.

The *general principles* metric plays a crucial role, with each task sharing common elements while maintaining specific criteria. For all the subtasks, principles like cohesion and decoupling, and practicability are fundamental. Cohesion and decoupling emphasize the importance of clarity and functionality within individual elements (classes or sequences) and reducing dependencies between different components. In terms of practicability, all tasks require designs to be readable, understandable, and modular, facilitating ease in development, testing, and maintenance. Meanwhile, each task has its unique focus areas: UML Class diagrams are evaluated on complexity; UML Sequence diagrams concentrate on uniformity, integration, and interaction complexity; Architecture Design highlights the distinction between design and coding, and conformance to practical standards. Subsequently, the *faithfulness* metric gauges the extent to which models adhere to specified instructions.

3.2 Task 2: Environment Setup

In the second phase of DevEval, models are provided with the PRD, UML diagrams, and architecture design to generate a dependency file for initializing the development environment. This step is followed by the deployment of a standard installation command utilizing the generated file. For Python, the Conda environment manager is employed; for Java and JavaScript, Gradle and NPM are utilized respectively.³ Setting up an environment often encounters challenges such as missing or outdated dependencies, along with version conflicts, all of which must be resolved to ensure a seamless development environment. Our research aims to investigate the potential of LLMs in automating this cumbersome process, thereby enhancing production efficiency.

Evaluation. The evaluation centers on the execution of dependency files across each programming language within a predetermined base environment delineated in a Docker file. This is followed by the execution of the repository’s example usage code. The principal metric for evaluation in this task is the success rate of the executed example code.

³It is noteworthy that our evaluation does not contain an Environment Setup task for C/C++ due to the absence of a universally acknowledged and user-friendly dependency management system for these languages (Miranda and Pimentel, 2018).

3.3 Task 3: Implementation

For this task, models are provided with the PRD, UML diagrams and architecture design, and are then instructed to develop code for each source code file as specified in the architecture design. Diverging from existing benchmarks on repository-level code generation (Liu et al., 2023b), the implementation task in DevEval is dedicated to assessing LLMs in generating an entire code repository from scratch. A key innovation in our investigations is the detailed level of requirements provided to the LLMs. In contrast to similar systems like MetaGPT (Hong et al., 2023) and ChatDev (Qian et al., 2023a,b), which generate outputs from brief requirement descriptions typically under 100 words, DevEval offers document-level detail to guide the models. This approach ensures the products are more precisely aligned with expectations, and further acceptance testing is employed for verification. As a result, our evaluations better reflect real-world software development scenarios where detailed requirements are essential to capture complex specifications and ensure product quality.

To more accurately simulate real-world development practices and ensure rigorous evaluation, the implementation task in DevEval involves supplying LLMs with comprehensive inputs including the PRD, UML class and sequence diagrams, and architecture design. The models are then prompted to generate code files. Given the constraint of output length, we adopt a sequential generation approach, prompting the models to produce one code file per interaction. Regarding the planning aspect, recent studies have explored prompting LLMs or training a specific planner (Yao et al., 2023; Besta et al., 2023; Wang et al., 2024b). Considering the structured nature of code files and the inherent dependencies among them, we utilize these dependencies as a clue for effective planning. The generation process is guided to adhere to a partial order derived from a predefined directed acyclic graph, thereby ensuring structured and logical code development. We leave the exploration of planning generated by models themselves for future work.

Evaluation. For the evaluation of the implementation task, an automated testing framework has been developed. This framework, tailored to the specific programming language in use, integrates PyTest for Python, GTest for C++, JUnit for Java, and Jest for JavaScript. The evaluation procedure involves executing reference acceptance

and unit tests within a predefined reference environment. Then the evaluation metric is determined by the pass rate of these tests.

	Python	C/C++	Java	JavaScript [‡]
Domain [†]	NLP CV DL ALGO API Tool	DB ALGO Tool	CV DB ALGO Tool	WEB
#Repo	10	5	5	2
Avg. #Code File	2.2	7.0	5.4	6.0
Avg. #Code Line	276	495	524	617
Avg. #Accep. Tests	3.0	5.4	2.4	2
Avg. #Unit Tests	12.4	11.8	8.2	-
Avg. Coverage	91.8	95.0	64.9*	-

Table 2: DevEval Statistics. †: DevEval covers a range of domains including NLP, computer vision, deep learning, algorithm implementation, API applications, Database applications, web service (both frontend and backend), and general tools and utilities. ‡: We do not include unit testing for JavaScript as checking functional correctness is not applicable to pure static web pages. Correctness of page rendering and user interaction handling is checked using acceptance tests. *: Interfaces with thirty-party libraries that are not used to implement the designated functionalities are not supplied with test cases, resulting in relatively lower overall test coverage.

3.4 Task 4: Acceptance Testing

For this task, models are provided with the PRD, UML diagrams, and architecture design, with the option to include the implementation source code to generate acceptance test code. Acceptance testing is critical to verify that the software adheres to requirements and operates effectively. In the context of applications featuring command-line interfaces, acceptance tests interact with the software via shell commands, as specified in the PRD, and subsequently evaluate the accuracy of the output generated. For libraries, acceptance tests are implemented through code that invokes the library’s API, followed by assertions made on the responses of the API. Applying this evaluative approach to LLMs provides valuable insights into their practical effectiveness and dependability within the domain of software development.

Evaluation. The evaluation of this task involves running the generated acceptance tests against the benchmark implementation code in the same testing framework developed for the evaluation for the implementation task.

In this phase, the Oracle Test methodology is

employed to assess the accuracy of acceptance tests generated by the model, which are comprised of both test input and expected output. The testing framework conducts an execution of the reference implementation of the software using the input devised by the model, subsequently comparing the software’s actual output against the model-predicted output. This approach facilitates an understanding of the extent to which LLMs can accurately interpret and predict the correct behavior of the subject software, as delineated in its design documentation.

3.5 Task 5: Unit Testing

In this phase, models are given the PRD, UML diagrams, and architecture design to generate unit test codes that facilitate a comprehensive understanding of the software. Unit testing serves as a fundamental approach to safeguard code integrity and operational accuracy. Distinguished from broader acceptance testing, which focuses on the overall functionality and viability of the software, unit testing examines individual code segments for adherence to specified functionalities. The increasing reliance on LLMs for streamlining software development processes underscores the criticality of their adeptness in writing effective unit tests, a competency integral to the software’s reliability and overall robustness.

Evaluation. The evaluation of LLM-generated unit tests is conducted through their application on the reference source code, employing the previously delineated testing framework. Similar to the acceptance testing evaluation, this phase leverages the Oracle Test, wherein the actual output of code units under specific test inputs is compared to anticipated outputs, as delineated by the oracle.

In addition, code coverage metrics are incorporated, providing a quantitative understanding of test comprehensiveness. Utilizing statement coverage analysis tools integrated within the aforementioned testing frameworks, coverage is mathematically expressed as:

$$\text{Coverage} = \left(\frac{\text{Number of Executed Statements}}{\text{Total Number of Statements}} \right) \times 100\%,$$

where the number of executed statements denotes the count of distinct executable statements within the code that are executed at least once during the testing process, while the total number of statements represents the aggregate count of all executable statements present in the codebase that are subject to potential execution.

3.6 Dataset

The dataset construction process involved three phases: repository preparation, code cleanup, and document preparation. We first selected high-quality repositories from a GitHub dump, applying filters to ensure manageable complexity for evaluation. Postgraduate student annotators then set up the environments, executed the code to verify functionality, and cleaned the repositories by removing unnecessary files and running or creating unit and acceptance tests to ensure oracle test standards. Finally, they prepared standard software design documents, including UML diagrams and architecture designs, following specific guidelines. The curated dataset consists of 22 repositories across Python, C/C++, Java, and JavaScript, with varying complexities and multi-file structures, as detailed in Table 2. Appendix B provides more details.

4 Experiments

4.1 Setup

Models and the Baseline System We evaluate three prominent pre-trained model families with different model sizes, including both proprietary and open-source models: OpenAI GPT (OpenAI, 2023), CodeLlama (Roziere et al., 2023), DeepSeek-Coder (Guo et al., 2024). Specifically, our experiments involve GPT-3.5-Turbo, GPT-4-Turbo from OpenAI GPT⁴, CodeLlama-Instruct 7B/13B/34B, and DeepSeek-Coder-Instruct models 1.3B/6.7B/33B. Regarding the baseline system, we extend ChatDev (Qian et al., 2023a,b) for DevEval, adding support for UML diagrams, architecture design, environment setup, and multi-language execution, with structured PRDs and feedback to reduce hallucinations. Further details are provided in the Appendix C.

Prompting Methods In our baseline system, we explore three prompting methods: *No-Review*, *Normal-Review*, and *Execution-Feedback*. *No-Review* represents a basic zero-shot prompting with built-in task prompts. *Normal-Review* involves a dual-role interaction, where the first role generates a solution and the second role reviews and, where necessary, corrects it. This mode is designed to evaluate the impact of review on model performance, in the absence of external inputs. *Execution-Feedback*, on the other hand, adds more dynamic

⁴We utilize gpt-3.5-turbo-1106, gpt-4-0125-preview, respectively.

Task	Implementation	
	Pass@ Accept. Test [¶]	Pass@ Unit Test [¶]
Evaluation Metric (%)		
GPT-4-Turbo		
No-Review	3.0	0.0
Normal-Review	3.0	0.0
Execution-Feedback	8.9	4.2
CodeLlama-34B-Instruct		
No-Review	0.0	1.4
Normal-Review	0.0	1.4
Execution-Feedback	0.0	1.4
DeepSeek-Coder-33B-Instruct		
No-Review	1.5	4.2
Normal-Review	1.5	4.2
Execution-Feedback	1.5	4.2

Table 3: Results of different prompting methods of the Implementation task on a subset of DevEval. ¶: all results are averaged across all repositories and weighted by the number of code lines, which measures the difficulty of each repository.

interaction to the review process. This feedback includes runtime results, error messages, and performance metrics. Such information could enable the reviewing role to make more informed decisions, potentially leading to more accurate and effective solutions. For computational efficiency, we conduct a single review for all review-involved prompting methods.

Task	Unit Testing	
	Oracle Test [§]	Coverage [§]
Evaluation Metric (%)		
GPT-4-Turbo		
No-Review w/ Src Code	35.1	34.3 (54.8)
Normal-Review w/ Src Code	22.6	25.8 (68.7)
CodeLlama-34B-Instruct		
No-Review w/ Src Code	10.7	18.3 (73.2)
Normal-Review w/ Src Code	12.6	27.0 (72.1)
DeepSeek-Coder-33B-Instruct		
No-Review w/ Src Code	27.2	37.9 (75.8)
Normal-Review w/ Src Code	22.5	35.5 (71.0)

Table 4: Results of different prompting methods of the Unit Testing task on a subset of DevEval. §: the Oracle Test results are averaged across all repositories and weighted uniformly. §: the results on the left side are averaged across all repositories and weighted uniformly, showing the overall scores. The results on the right side in the parenthesis are averaged across all *valid* repositories and weighted uniformly, where models have generated *executable* testing code.

4.2 Main Results

Results across prompting methods We first conduct experiments to examine the effects of different prompting methods on a subset of DevEval us-

Task	Accept. Testing
	Oracle Test [§]
Evaluation Metric (%)	
GPT-4-Turbo	
No-Review	3.6
Normal-Review	7.7
Normal-Review w/ Src Code	14.9
CodeLlama-34B-Instruct	
No-Review	0.0
Normal-Review	0.0
Normal-Review w/ Src Code	0.0
DeepSeek-Coder-33B-Instruct	
No-Review	0.0
Normal-Review	4.2
Normal-Review w/ Src Code	15.6

Table 5: Results of different prompting methods of the Acceptance Testing task on a subset of DevEval. §: all results are averaged across all repositories and weighted uniformly.

ing three representative models: GPT-4-Turbo, CodeLlama-34B-Instruct, and DeepSeek-Coder-33B-Instruct.

Table 3 illustrates the results of the implementation task for our study. In general, Execution-Feedback leads to the optimal performance, where GPT-4-Turbo benefits the most, especially on acceptance tests. In contrast, CodeLlama-34B-Instruct and DeepSeek-Coder-33B exhibited no improvement with any review process. We note that the efficacy of the review process could be understated as our automated testing is too rigorous and sparse to reflect the improvements. We observe that, despite no substantial improvements on reference tests, the code quality notably improved with Execution-Feedback prompt. However, there is no significant improvements using the Normal-Review setting compared with the No-Review setting. Models consistently provide unhelpful suggestions, such as the addition of unnecessary error handling or reorganization. This indicates that the models are unable to comprehend complicated code by merely reading it, lacking external knowledge like execution feedback.

For the testing tasks, which are relatively easier than the implementation, there are various observations. In Table 4, we find no clear evidence that the review process brings stable benefits to unit testing. However, on the acceptance testing in Table 5, Normal-Review brings enhancement to the performance. The degradation in unit testing performance with review mainly stems from extended input length challenging models’ long-context com-

Task	Environment Setup	Implementation		Acceptance Testing	Unit Testing	
Evaluation Metric (%)	Pass@ Example Usage [§]	Pass@ Accept. Test [¶]	Pass@ Unit Test [¶]	Oracle Test [§]	Oracle Test [§]	Coverage [§]
GPT-3.5-Turbo	<i>33.3</i>	4.2	4.3	11.7	28.7	24.6 (61.4)
GPT-4-Turbo	<i>41.7</i>	7.1	8.0	29.2	36.5	33.2 (66.3)
CodeLlama-7B-Instruct	8.3	0.0	0.0	0.0	3.0	3.6 (71.0)
CodeLlama-13B-Instruct	<i>25.0</i>	0.6	0.0	0.0	5.1	8.6 (57.6)
CodeLlama-34B-Instruct	<i>16.7</i>	0.6	0.5	4.5	21.1	25.4 (72.6)
DeepSeek-Coder-1.3B-Instruct	8.3	0.0	0.1	0.0	5.6	2.7 (27.0)
DeepSeek-Coder-6.7B-Instruct	<i>25.0</i>	2.9	3.9	20.5 [♡]	23.5	28.2 (70.6)
DeepSeek-Coder-33B-Instruct	<i>16.7</i>	4.4	5.5	13.6	32.8	35.7 (79.4)

Table 6: Tasks 2 to 5 results on DevEval. *Italic figures*: test cases for the Environment Setup task are quite scarce compared to other tasks, therefore the results are more influenced by the randomness⁶. §: all results are averaged across all repositories and weighted uniformly. ¶: all results are averaged across all repositories and weighted by the number of code lines. §: the results on the left side are averaged across all repositories and weighted uniformly, showing the overall scores. The results on the right side in the parenthesis are averaged across all *valid* repositories and weighted uniformly, where models have generated *executable* testing code. ♡: the model has generated meaningless but executable testing code.

prehension. Imprecise reviewer suggestions may also reduce output quality. Regarding the visibility of implementation source code, it is common practice not to expose source code and execution feedback for acceptance testing, while it’s allowed to employ the source code as input for unit testing. As shown in Table 5, models can barely generate executable acceptance testing code and incorporating source code as additional input dramatically increases the performance.

Results across models Table 6 illustrates the main results on DevEval with optimal prompting methods applied for each task.⁵ We find that GPT-4-Turbo demonstrates superior performance compared to other models, while all models are far from satisfactory. DevEval can effectively distinguish between models of varying capabilities. Smaller models, such as CodeLlama-7B/13B-Instruct and DeepSeek-Coder-1.3B-Instruct, demonstrate inherent limitations, frequently unable to generate syntactically accurate code or follow the instructions. These models tend to generate mere code skeletons or fill the function body with only comments. Larger open-sourced models and GPT models, while generating more reasonable code, still struggle with the subtleties of complex code structure and logic, such as variable type conversion, function arguments and object-oriented classes.

⁵Execution-Feedback is used for Environment Setup and Implementation; Normal-Review w/ Src Code for Acceptance Testing; No-Review w/ Src Code for Unit Testing as Normal-Review w/ Src Code shows no clear advantage.

Results across tasks Generally, the models’ performances on the implementation task are all below the 10% pass rate. The highest-performing model, GPT-4-Turbo, registers only a 7.1% pass rate on reference acceptance tests and 8.0% on unit tests, while some other models score zero, failing all reference tests. Despite prior research such as HumanEval (Chen et al., 2021) indicating that models can manage simple code-writing tasks, substantial challenges remain in more complex coding scenarios within DevEval. We break down the implementation results into different languages in Figure 8 and find that models particularly struggle in handling Java and C/C++, whose stringent syntax requirements tend to magnify the models’ deficiencies in managing intricate details. This points to the necessity for more enriched and diverse training data across programming languages to bridge this competency gap.

Compared with the implementation task, other tasks are relatively simple but still challenging. Regarding the environment setup task, we note that the test cases for the environment setup task are quite scarce compared to other tasks⁶, therefore the results are more influenced by the randomness. Roughly speaking, GPT-4-Turbo reaches a 41.7% pass rate in building environments, while open-

⁶Except for C/C++ and easy Python repositories that are absent, only 12 repositories are involved in the environment setup task. We will resolve this issue in future work. However, DevEval contains rich test cases for other tasks. Table 2 evidences that DevEval features fruitful tests for the implementation task. For the testing generation tasks, our prompts depict fine-grained requirements and ensure the quantity of generated testing cases.

sourced models largely fall behind it. For the testing tasks, GPT-4-Turbo still obtains the highest scores and open-sourced models perform worse. We identify an outlier that DeepSeek-Coder-6.7B-Instruct achieves the highest score among open-sourced models and approach GPT-4-Turbo in acceptance testing. The model somehow *cheats* on this task by generating meaningless but executable testing code (detail in Section E.4). With respect to the unit testing, the overall Oracle Test and Coverage scores are quite low, which are averaged across all repositories. However, for those generated testing codes that can be successfully executed (obtain non-zero Oracle Test score), the coverage scores are relatively high, suggesting the models’ promising capability on this problem.

Results on software design Table 7 shows the results of software design using GPT-4-Turbo as the Judge and GPT-3.5-Turbo as the baseline for comparison. More details on how we evaluate the software design are found in Appendix D. GPT-4-Turbo dominantly outperforms GPT-3.5-Turbo with extraordinarily high win rates in all cases. Regarding the open-sourced models, as the size increases, models consistently produce higher quality design documents on both metrics, while they are relatively inferior on *faithfulness*.

We compare the LLM-as-a-Judge results with human majority annotations. Low agreements are observed with tie considered, which aligns with previous studies (Zheng et al., 2023a). It is reasonable as a tie is hard to define and judge, especially for highly complicated and structured software design documents. Without tie, GPT-4-Turbo reaches 79.2% and 83.2% agreements on the *general principles* and *faithfulness* metrics, respectively. This means GPT-4-Turbo’s judgments align with the majority of humans and could serve as a good alternative for automated software design evaluation.

4.3 Analysis

Our experiments reveal several challenges faced by LLMs, such as difficulties in generating accurate Makefiles and Gradle files, handling function redefinitions, and managing file references in multi-file repositories. Models also struggle with correct function parameter usage, naming conventions, type handling, and managing variable scope. Furthermore, they frequently fabricate variables or misinterpret data files, leading to hallucination issues. For more detailed experimental findings and

	w/ Tie		w/o Tie	
	G [†]	F [‡]	G	F
GPT-4-Turbo	97.9	97.9	100.0	100.0
CodeLlama-7B-Instruct	4.2	8.3	4.2	4.5
CodeLlama-13B-Instruct	18.8	14.6	10.5	5.3
CodeLlama-34B-Instruct	39.6	33.3	33.3	21.4
DeepSeek-Coder-1.3B-Instruct	16.7	16.7	5.5	5.6
DeepSeek-Coder-6.7B-Instruct	35.4	35.4	31.6	29.4
DeepSeek-Coder-33B-Instruct	52.1	50.0	53.8	50.0
Agree w/ Human Majority	60.4	51.6	79.2	83.2

Table 7: Win rate of pairwise comparison against GPT-3.5-Turbo on Software Design on a subset of DevEval. Results are averaged across different repositories and sub-tasks uniformly. †: *general principles*. ‡: *faithfulness*. w/ Tie: inconsistent results are considered as a tie. We also report agreement with Human Majority.

analysis, please refer to the Appendix E.

5 Conclusion

The DevEval framework presents a leap forward in studying LLMs within the domain of automated software development. By employing a multi-stage evaluation process, DevEval comprehensively assesses LLMs across a spectrum of tasks including design, environment setup, implementation, and testing. Empirical findings reveal that pre-trained models like GPT-4-Turbo are still confronted with substantial challenges within DevEval. Through analysis, we identify models’ limitations in understanding the complex repository structures and handling the nuanced demands of comprehensive software development. These insights elucidate critical pathways for future model development.

6 Limitations

One limitation of our work is the limited number of repositories used in the study, with only 22 curated examples across four programming languages. This relatively small dataset may not fully capture the wide variety of challenges and complexities present in real-world software development. Although we selected diverse repositories to represent different domains and programming paradigms, a broader and more extensive collection of repositories would provide a more comprehensive evaluation of LLM performance. Future work could address this by incorporating additional repositories to better generalize the findings.

Acknowledgement

This project is supported by Shanghai Artificial Intelligence Laboratory. We would like to thank the reviewers for their valuable feedback and constructive suggestions.

References

- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. [Program synthesis with large language models](#). *Preprint*, arXiv:2108.07732.
- Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Michal Podstawski, Hubert Niewiadomski, Piotr Nyczyk, et al. 2023. Graph of thoughts: Solving elaborate problems with large language models. *arXiv preprint arXiv:2308.09687*.
- Federico Cassano, John Gouwar, Daniel Nguyen, Sy Duy Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q. Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2023. [Multipl-e: A scalable and polyglot approach to benchmarking neural code generation](#). *IEEE Transactions on Software Engineering*, 49:3675–3691.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Cheng-Han Chiang and Hung-yi Lee. 2023. Can large language models be an alternative to human evaluations? *arXiv preprint arXiv:2305.01937*.
- Yanguibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2023. [Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion](#). *ArXiv*, abs/2310.11248.
- Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. [Deepseek-coder: When the large language model meets programming – the rise of code intelligence](#). *Preprint*, arXiv:2401.14196.
- Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *NeurIPS*.
- Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2023. [Metagpt: Meta programming for a multi-agent collaborative framework](#). *Preprint*, arXiv:2308.00352.
- Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.
- Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Yih, Daniel Fried, Si yi Wang, and Tao Yu. 2022. [Ds-1000: A natural and reliable benchmark for data science code generation](#). *ArXiv*, abs/2211.11501.
- Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. Camel: Communicative agents for "mind" exploration of large scale language model society. *arXiv preprint arXiv:2303.17760*.
- Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023a. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210*.
- Tianyang Liu, Canwen Xu, and Julian McAuley. 2023b. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*.
- Yizhou Liu, Pengfei Gao, Xinchun Wang, Jie Liu, Yexuan Shi, Zhao Zhang, and Chao Peng. 2024. [Marscode agent: Ai-native automated bug fixing](#). *Preprint*, arXiv:2409.00899.
- Yuliang Liu, Xiangru Tang, Zefan Cai, Junjie Lu, Yichi Zhang, Yan Shao, Zexuan Deng, Helan Hu, Zengxian Yang, Kaikai An, Ruijun Huang, Shuzheng Si, Sheng Chen, Haozhe Zhao, Zheng Li, Liang Chen, Yiming Zong, Yan Wang, Tianyu Liu, Zhiwei Jiang, Baobao Chang, Yujia Qin, Wangchunshu Zhou, Yilun Zhao, Arman Cohan, and Mark B. Gerstein. 2023c. [Ml-bench: Large language models leverage open-source libraries for machine learning tasks](#). *ArXiv*, abs/2311.09835.
- Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.
- André Miranda and João Pimentel. 2018. On the use of package managers by the c++ open-source community. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 1483–1491.
- Niklas Muennighoff, Qian Liu, Qi Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra,

- and S. Longpre. 2023. [Octopack: Instruction tuning code large language models](#). *ArXiv*, abs/2308.07124.
- OpenAI. 2023. [Openai gpt](#).
- Chen Qian, Xin Cong, Wei Liu, Cheng Yang, Weize Chen, Yusheng Su, Yufan Dang, Jiahao Li, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023a. [Communicative agents for software development](#). *Preprint*, arXiv:2307.07924.
- Chen Qian, Yufan Dang, Jiahao Li, Wei Liu, Weize Chen, Cheng Yang, Zhiyuan Liu, and Maosong Sun. 2023b. [Experiential co-learning of software-developing agents](#). *arXiv preprint arXiv:2312.17025*.
- Winston W Royce. 1987. [Managing the development of large software systems: concepts and techniques](#). In *Proceedings of the 9th international conference on Software Engineering*, pages 328–338.
- Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. [Code llama: Open foundation models for code](#). *arXiv preprint arXiv:2308.12950*.
- Lin Shi, Chiyu Ma, Wenhua Liang, Weicheng Ma, and Soroush Vosoughi. 2024. [Judging the judges: A systematic investigation of position bias in pairwise comparative assessments by llms](#). *Preprint*, arXiv:2406.07791.
- Peiyi Wang, Lei Li, Liang Chen, Dawei Zhu, Binghui Lin, Yunbo Cao, Qi Liu, Tianyu Liu, and Zhifang Sui. 2023. [Large language models are not fair evaluators](#). *arXiv preprint arXiv:2305.17926*.
- Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2024a. [Open-devin: An open platform for ai software developers as generalist agents](#). *Preprint*, arXiv:2407.16741.
- Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2022. [Execution-based evaluation for open-domain code generation](#). In *Conference on Empirical Methods in Natural Language Processing*.
- Zihao Wang, Shaofei Cai, Guanzhou Chen, Anji Liu, Xiaojian Ma, and Yitao Liang. 2024b. [Describe, explain, plan and select: Interactive planning with large language models enables open-world multi-task agents](#). *Preprint*, arXiv:2302.01560.
- John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. [Swe-agent: Agent-computer interfaces enable automated software engineering](#). *Preprint*, arXiv:2405.15793.
- John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. 2023. [Intercode: Standardizing and benchmarking interactive coding with execution feedback](#). *ArXiv*, abs/2306.14898.
- Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. [Tree of thoughts: Deliberate problem solving with large language models](#). *arXiv preprint arXiv:2305.10601*.
- Pengcheng Yin, Wen-Ding Li, Kefan Xiao, A. Eashaan Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, Oleksandr Polozov, and Charles Sutton. 2022. [Natural language to code generation in interactive data science notebooks](#). *ArXiv*, abs/2212.09248.
- Fengji Zhang, B. Chen, Yue Zhang, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. [Repocoder: Repository-level code completion through iterative retrieval and generation](#). In *Conference on Empirical Methods in Natural Language Processing*.
- Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. [Autocoderover: Autonomous program improvement](#). *Preprint*, arXiv:2404.05427.
- Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2023a. [Judging llm-as-a-judge with mt-bench and chatbot arena](#). *arXiv preprint arXiv:2306.05685*.
- Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. 2023b. [Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x](#). *arXiv preprint arXiv:2303.17568*.

Appendix Table of Contents

A Software Engineering Tasks with a Running Example	11
A.1 Software Design	11
A.1.1 Class Diagrams	11
A.1.2 Sequence Diagrams	13
A.1.3 Architecture Design	13
A.2 Software Development	13
A.3 Quality Assurance	14
B Dataset	15
B.1 Dataset Construction	15
B.2 Dataset Statistics	15
C The Baseline System	15
D Software Design Evaluation	16
E Experimental Discussions	16
E.1 Model Capacity	16
E.2 Instruction Following	18
E.3 Hallucination	19
E.4 Limitations in Testing	19
F Repositories statistics in DevEval	20

Appendix

A Software Engineering Tasks with a Running Example

We describe concepts of software engineering tasks using one of the subjects of DevEval, named Actor Relationship Game. The Actor Relationship Game is a Java-based application that allows users to explore connections between popular actors through their movie collaborations, using data from The Movie Database (TMDB) API. It constructs an actor graph and identifies the shortest path of relationships between any two actors.

A.1 Software Design

Software design is the process by which an agent creates a specification of a software artifact, intended to accomplish goals, using a set of primitive components and subject to constraints. It is a phase in the software development lifecycle that bridges the gap between software requirements analysis and the actual implementation of the software system.

During the software design phase, software engineers or designers define the way a software application will work to meet the specified requirements in the form of a Product Requirement Document (PRD). They create diagrams that determine the data structures, software architecture, interface designs, and module specifications with Unified Markup Language (UML) diagrams.

A good software design is crucial as it impacts the quality, maintainability, performance, scalability, and robustness of the software product. It facilitates a smoother implementation phase, allows for better understanding and communication among team members, and helps in identifying potential issues early in the development process.

A.1.1 Class Diagrams

Class diagrams are a cornerstone of object-oriented design, offering a static snapshot of the system structure. These diagrams illustrate the classes within the system, their attributes, methods, and the relationships among the classes, such as inheritance and associations. Class diagrams are instrumental in providing an abstract representation of the system's components and their interactions, facilitating a deeper understanding of the software's overall architecture and design patterns.

Figure 2 shows the class diagram of the Actor Relationship Game repository. This UML

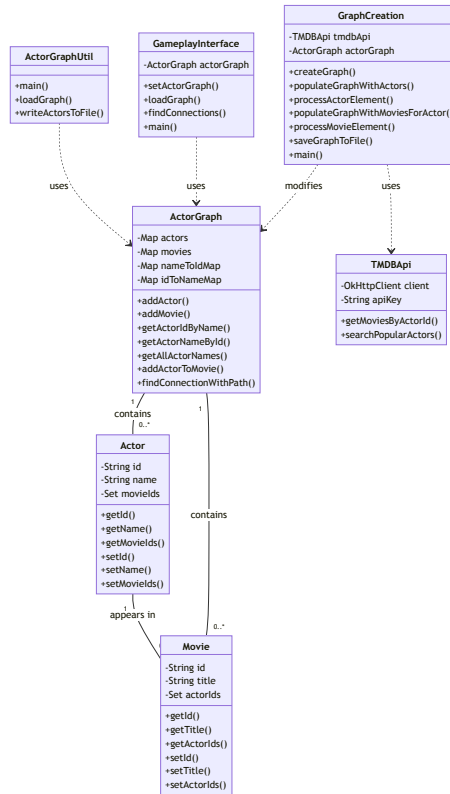


Figure 2: UML Class Diagram for the Example Repository

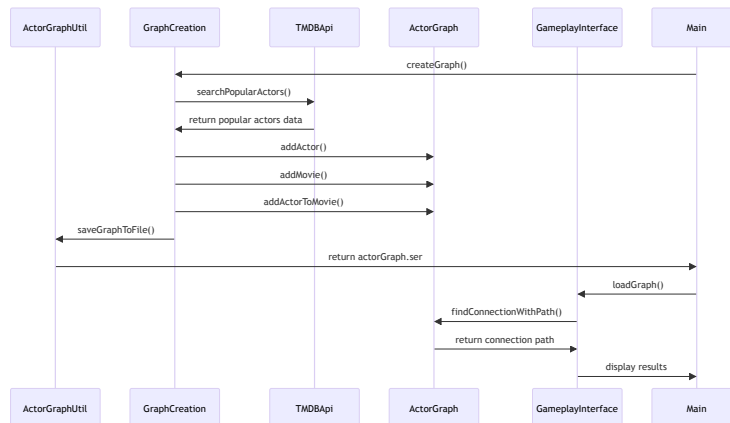


Figure 3: UML Sequence Diagram for the Example Repository

class diagram delineates the architecture of a system designed to model and analyze the network of relationships between actors and movies through an Actor Graph. It encompasses classes such as ‘Actor’ and ‘Movie’ to represent individual entities, alongside an ActorGraph

class that serves as a repository and management layer for these entities and their associations. Utility and operational classes like ActorGraphUtil, GameplayInterface, and GraphCreation provide mechanisms for manipulating the graph—ranging from data inges-

tion using the `TMDBApi` to utility functions and gameplay interfaces that leverage the graph for various applications. The relationships between classes, including associations, aggregations, and dependencies, are meticulously outlined to depict interactions such as actors appearing in movies and the construction and utilization of the actor-movie graph for finding connections and supporting gameplay or analysis tasks.

A.1.2 Sequence Diagrams

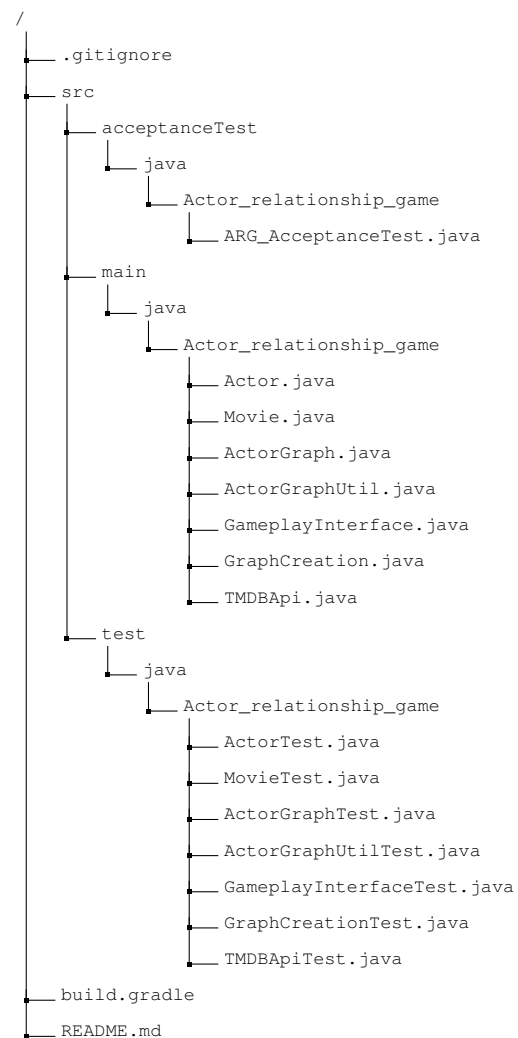
Sequence diagrams, different than class diagrams, focus on the dynamic aspects of the system. They depict how objects interact with each other across time, outlining the sequence of messages exchanged between objects to accomplish a specific functionality or process within the system. Sequence diagrams are invaluable for visualizing and analyzing the flow of operations, timing constraints, and the interaction patterns among system components, making them essential for detailed behavioral analysis.

The sequence diagram of Actor Relationship Game repository is illustrated in Figure 3. This diagram illustrates the flow of operations for creating, populating, and utilizing an actor-movie graph. Initially, the `Main` function triggers the graph creation process by calling `createGraph()` on the `GraphCreation` module, which then interacts with the `TMDBApi` to fetch popular actors' data. Upon receiving this data, `GraphCreation` populates the `ActorGraph` with actors, movies, and their associations. After constructing the graph, `GraphCreation` delegates the responsibility of saving this graph to a file to `ActorGraphUtil`, which then returns a serialized file (`actorGraph.ser`) back to `Main`. Subsequently, `Main` instructs the `GameplayInterface` to load this graph and use it to find connections between actors via `findConnectionWithPath()`, a method in `ActorGraph`. The path found is then returned to `GameplayInterface`, which finally displays the results back in the `Main` function. This sequence encapsulates a complete lifecycle from graph creation, through data population and serialization, to utilization for finding actor connections, showcasing a systematic approach to managing and analyzing actor-movie relationships.

A.1.3 Architecture Design

Architecture design using file tree representation refers to a method of visualizing and organizing the structural layout of a software system's components in a hierarchical format. This approach delineates the organization of software modules, packages, libraries, and other assets in a tree-like structure, where each node represents a file or a directory containing more files or directories. Such a representation is crucial in conveying the architectural blueprint of a software project, illustrating how its various parts are interrelated.

The text-based representation of the file tree for the Actor Relationship Game repository, including test classes for each Java class, is shown as below.



A.2 Software Development

Software development is the comprehensive process of programming, documenting, optimization, and fixing involved in creating and maintaining applications, frameworks, or other software components. It encompasses all the activities that re-

sult in software products and involves a series of steps known as the software development lifecycle (SDLC).

- **Environment Setup** is the process of preparing and configuring the necessary hardware and software tools required to build and run software applications. This setup is crucial to provide a consistent, controlled, and efficient workspace for developers to code, test, and deploy their applications. The environment can be set up on an individual's local machine, on a remote server, or in a containerized environment.
- **Implementation** is when developers write code according to the software design documents, using programming languages and tools suitable for the repository.

A.3 Quality Assurance

Quality Assurance (QA) is the systematic process of ensuring that the software being developed meets the specified quality standards and requirements before it is released.

Software testing is an integral part of QA; involves the execution of a software component or system component to evaluate one or more properties of interest. Software testing typically includes:

- **Unit Testing** is the process of testing individual units or components of a software application to ensure their behaviors. A unit is the smallest testable part of any software and usually has one or a few inputs and usually a single output. In procedural programming, a unit could be an entire module, but it is more commonly an individual function or procedure.
- **Acceptance Testing** is a level of software testing where a system is tested for acceptability. It provides the final assurance that the software meets the PRD and is ready for use by end-users.

Code Listing 1 is part of the unit test suite for the Actor Relationship Game Repository, specifically designed to validate the functionality of the `Actor` class. Using the JUnit framework, it defines two test cases: `testActorIdAndName` and `testMovieIds`. The first test, `testActorIdAndName`, instantiates an `Actor` object with a specific ID and name

("101" and "John Doe", respectively) and asserts that the `getId()` and `getName()` methods correctly return these values, ensuring the actor's identity is accurately stored and retrievable. The second test, `testMovieIds`, creates another `Actor` object and adds two movie IDs ("201" and "202") to the actor's list of movie IDs. It then verifies that these movie IDs are indeed associated with the actor by checking if the actor's `getMovieIds()` set contains the added IDs. Together, these tests check the integrity of the `Actor` class's basic functionalities: maintaining an actor's identity and managing their associated movie IDs.

Listing 1: Example Unit Test

```
package Actor_relationship_game;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class ActorTest {
    @Test
    void testActorIdAndName() {
        Actor actor = new Actor("101", "John Doe");
        assertEquals("101", actor.getId());
        assertEquals("John Doe", actor.getName());
    }

    @Test
    void testMovieIds() {
        Actor actor = new Actor("102", "Jane Smith");
        actor.getMovieIds().add("201");
        actor.getMovieIds().add("202");
        assertTrue(actor.getMovieIds().contains("201"));
        assertTrue(actor.getMovieIds().contains("202"));
    }
}
```

In Code Listing 2, the example acceptance test is designed to verify the functionality of generating and comparing actor lists from graph data. It employs the `runGradleTask` method to execute specific Gradle tasks for creating graph data and generating actor lists into specified file paths, using parameters for file names to differentiate between reference and test data. The test first runs the `runGraphCreation` task with paths for both reference and test graphs, followed by the `runActorGraphUtil` task to generate actor lists from these graphs into specified file paths. Once the actor lists are generated, the test reads lines from both the reference and test actor list files and then iterates through each line in the reference actor list, followed by an assertion that each actor from the reference list is also present in the test list. This process effectively checks the integrity and consistency of the actor list generation feature by ensuring that the test actor list replicates the reference list accurately, thereby validating the application's capability to process and output graph-

related data correctly.

Listing 2: Example Acceptance Test

```
@Test
public void testActorList() throws IOException,
    InterruptedException{
    runGradleTask("runGraphCreation -PfileName="+
        referenceGraphPath);
    runGradleTask("runGraphCreation -PfileName="+
        testGraphPath);
    runGradleTask("runActorGraphUtil -PgraphPath="+
        referenceGraphPath+" -PfilePath="+
        referenceActorPath);
    runGradleTask("runActorGraphUtil -PgraphPath="+
        testGraphPath+" -PfilePath="+testActorPath)
        ;

    List<String> referenceLines = Files.readAllLines
        (Paths.get(referenceActorPath));
    List<String> testLines = Files.readAllLines(
        Paths.get(testActorPath));

    for (String referenceLine:referenceLines){
        assertTrue(containsLine(testLines,
            referenceLine));
    }
}
```

B Dataset

B.1 Dataset Construction

The data preparation process in our work consists of three distinct phases: repository preparation, code cleanup, and document preparation.

The initial phase, repository preparation, involves selecting high-quality, well-structured candidate repositories from a GitHub dump. Recognizing the impracticality of constructing a repository from scratch, we employed a filtering process to identify suitable candidates. Moreover, we imposed a constraint on the total number of lines of code to ensure the repositories' complexity remains manageable, facilitating the evaluation of current LLMs.

During the code cleanup phase, our postgraduate student annotators were tasked with setting up the required environment as stipulated in the repositories' README files. They then executed the code to verify its functionality. Following this sanity check, the annotators were instructed to meticulously refine the code repositories. This refinement included the removal of unnecessary auxiliary files. To ascertain code quality, the annotators were also required to run existing unit and acceptance tests, or to develop additional tests, ensuring they meet the standards of the oracle test and achieve satisfactory coverage.

The final phase, document preparation, involved the creation of standard software design documents, namely, UML class and sequence diagrams, and

architecture designs, for each repository. We provide annotators with specific guidelines and templates for these documents. The annotators were responsible for ensuring that these design documents corresponds accurately and cohesively with the respective code repositories.

B.2 Dataset Statistics

In Table 2, we present an exhaustive statistical breakdown of our datasets. DevEval contains a collection of 22 curated repositories, spanning across four widely-used programming languages (Python, C/C++, Java, JavaScript) and a diverse range of domains. The dataset is characterized by its multi-file structure. The Python repositories in our dataset are relatively straightforward, with each repository comprising approximately two files and an average of 276 lines of code. In contrast, the repositories pertaining to statically-typed programming languages, namely C/C++ and Java, are more complex, featuring an increased count of code files and lines. For JavaScript, our usage of the Vue.js framework necessitates that models adeptly navigate the framework's templates and development paradigms. Consequently, JavaScript repositories exhibit the highest number of code files and lines, posing a substantial challenge for LLMs. Additionally, we have prepared extensive reference acceptance and unit tests for each repository to facilitate rigorous evaluation of the implementation task.

C The Baseline System

We introduce our baseline system formulated for DevEval, building upon the foundations of ChatDev (Qian et al., 2023a,b). ChatDev is a virtual, chat-powered software development system that adheres to the conventional waterfall model. It bifurcates the development process into four primary tasks (dubbed as phrases in ChatDev): design, coding, testing, and documentation. Within this system, multiple LLM agents assume diverse roles such as programmers, reviewers, and testers, pertinent to each phase. ChatDev is characterized by its utilization of a chat chain mechanism, which segments each phase into smaller, atomic tasks. This approach enables context-sensitive, multi-turn dialogues between two distinct roles, facilitating the proposal and validation of solutions for individual tasks.

In contrast to ChatDev, our development of baseline system incorporates several features and en-

hancements. We have restructured the task design to align closely with the evaluation criteria of DevEval. Specifically, this includes the integration of comprehensive input to the system, exemplified by well-structured PRDs. This integration is crucial in addressing and examining the issue of hallucination in controlled experimental settings. Moreover, our baseline system expands upon the capabilities of ChatDev, supporting a wider range of tasks, including standard Object-Oriented programming designs (UML class and sequence diagrams), repository planning (architecture design), environment setup, and acceptance testing. A significant advancement in our baseline system is its compatibility with multiple programming languages and their corresponding runtime environments. This feature is coupled with the provision of comprehensive execution feedback to the system.

Implementation Details We utilize LMDeploy for the deployment of CodeLlama and DeepSeek-Coder models.⁷ Acknowledging the potential for extensive input context in DevEval tasks, we configure the context length to 32K for these models. For the Software Design task, we set the temperature parameter to 0.2, while for the remaining four tasks, we use a temperature of 0. Other hyperparameters in the experiment are maintained at default settings. All code-related tasks are rigorously evaluated in an isolated sandbox environment, utilizing Docker technology.

D Software Design Evaluation

We follow previous work (Zheng et al., 2023a) to conduct a pairwise comparison to determine which response is better, focusing on the metrics of *general principles* and *faithfulness* (see the corresponding prompts in Figures 4, 5, 6). To reduce the expenditure of the OpenAI GPT API and human effort, the scope of our evaluation was confined to a subset of our dataset. This process involves 192 pairs across eight repositories, eight models, and three sub-tasks. Regarding the LLM judge, we use GPT-4-Turbo as the judge and GPT-3.5-Turbo as the baseline model. To mitigate the issue of position bias (Zheng et al., 2023a; Shi et al., 2024), i.e., LLM judges preferring response at a certain position regardless of the content, the evaluation was executed in a *dual* mode, evaluating each pair twice in different orders (384 pairs in total), with inconsistent decisions being considered as a tie. For

⁷<https://github.com/InternLM/lmdeploy>

the human evaluation, we shuffle the order of two responses and annotate each pair thrice to obtain the *human majority*.

The customized LLM-as-a-Judge prompt for evaluating software design is detailed in Fig. 7, structured to facilitate pairwise comparisons in accordance with predefined evaluation guidelines. The LLM’s judgments are extracted by employing regular expressions to identify the selection specified after "Choice:" within the judges’ responses.

E Experimental Discussions

E.1 Model Capacity

Challenges in Creating C/C++ Makefile and Java Gradle LLMs often face challenges in generating accurate Makefile for C/C++ and Gradle files for Java. Frequently, the generated files are deficient in critical components like source code files, necessary dependencies and essential tasks. In C/C++ and Java repositories, approximately 90% of compilation and execution errors can be attributed to these issues. We find that even GPT-4-Turbo occasionally fail in basic syntax errors on compilation files. This is potentially caused by insufficient training data related to these compilation tools.

Function Redefinition in Multi-file Repositories Models face significant challenges in multi-file repositories contexts, particularly with function redefinitions. Specifically, if a global function is required for the entire repository, it can be defined in any file, and other files just need to correctly reference it. But they tend to redundantly implement the same function across multiple files, suitable for single-file repositories but erroneous in multi-file scenarios. In C/C++, models incorrectly handle header (.h) and implementation (.cpp) files, leading to redundant declarations and conflicting implementations. These issues highlight a gap in the models’ understanding of file-specific roles in programming languages.

File Reference and Linkage Errors Correct file referencing is essential in multi-file programming repositories. Models, especially those with larger parameters, generally perform well in establishing basic reference logic, such as using “import” in Python and “#include” in C++. However, without review, reference errors are common, likely due to the models’ sequential code generation approach,

Evaluation Guidance for UML Class

General Principles

- **Cohesion and Decoupling:** The design should aim for high cohesion within individual classes and low coupling between different classes. High cohesion ensures that each class is dedicated to a singular task or concept, enhancing clarity and functionality. Low coupling reduces dependencies among classes, facilitating easier maintenance and scalability.
- **Complexity:** Utilize metrics such as the total number of classes, the average number of methods per class, and the depth of the inheritance tree to evaluate complexity. It's important to discern between conceptual classes and attributes; not every noun should become a class. The complexity level should be appropriately balanced, aligning with the specific requirements detailed in the repository's Product Requirement Document (PRD).
- **Practicability:** A practical design should be readable and understandable, offering a clear and comprehensive representation of the software's structures, functionalities, and behaviors. This enhances ease in programming, testing, and maintenance. Modularity should be evident, with each component serving a distinct function, streamlining the development process. Interfaces need to be designed for simplicity, facilitating smooth interactions within the software and with external environments. The design must also support robust testing strategies, enabling thorough validation through unit and acceptance tests, ensuring the design's viability in real-world applications.

Faithfulness

- Ensure that the design aligns with the given PRD strictly, achieving all the functionalities based on the requirements without making any hallucinations and additions. Ensure that the conceptual classes and their relationships accurately represent the essentials outlined in the PRD. This includes a detailed focus on the associations between classes, their cardinalities, and the types of relationships such as inheritance, aggregation, and composition. Clarity in class names and the optional inclusion of attributes are key for aligning with the repository's vision.

Figure 4: Evaluation Guidance for UML Class.

Evaluation Guidance for UML Sequence

General Principles

- **Uniformity and Integration:** The design should demonstrate a consistent style and integrated approach, ensuring all components work seamlessly together.
- **Cohesion and Decoupling:** Evaluate the sequence diagram for its cohesion within sequences and coupling between different parts of the system. The goal is to ensure each sequence is focused, with minimal dependencies between different system components. Strive for high cohesion within sequences and low coupling between them.
- **Interaction complexity:** This metric assesses the interaction complexity of the sequence diagram, focusing on the number of messages, depth of nested calls, and the number of participating objects. It also examines how the sequence of messages and the roles of key objects are portrayed in these interactions. The ideal level of complexity should be in line with the specific requirements detailed in the repository's PRD.
- **Practicability:** This comprehensive metric includes aspects of readability, understandability, class and method representation, and the overall clarity in depicting system interactions and functionalities. Evaluate the diagram's ease of interpretation for development, testing, and maintenance, its ability to represent the functionality and purpose of each class, document object creation instances, and demonstrate the modularity and interface simplicity that support efficient and reliable system operation.

Faithfulness

- Evaluate how accurately and comprehensively the sequence diagram reflects the system's intended behavior and requirements specified in the PRD. This includes how well it captures system events, both with and without parameters, and the accuracy with which it reflects the impact of these events on the system's behavior. Also Evaluate how accurately and comprehensively the sequence diagram reflects the structural design outlined in the given UML class diagrams, ensuring a coherent and consistent development process.

Figure 5: Evaluation Guidance for UML Sequence.

which limits their ability to correct earlier mistakes. More complex reference issues also arise. Models often struggle to differentiate between global functions and class methods, leading to reference errors when attempting to access class methods directly. These errors are challenging to rectify

through review. This indicates a need for models to better grasp the intricacies of programming language structures and conventions.

Evaluation Guidance for Architecture Design

General Principles

- **Uniformity and Integration:** The design should demonstrate a consistent style and integrated approach, ensuring all components work seamlessly together, ensuring high cohesion and decoupling.
- **Distinction Between Design and Coding:** Recognize that the design process is distinct from coding; good design lays the groundwork for effective coding but is not synonymous with it.
- **Practicability:** Evaluate the architecture’s practicability by assessing its organization, readability and modularity, and efficiency. The design should feature a logical and clear structure, evidenced by a well-organized file tree and distinct class locations in proper directories.
- **Conformance:** Evaluate the architecture for its conformance to community and industry standards. The file tree structure, coding practices including naming conventions, documentation and other structural elements should adhere to the widely accepted conventions by the open-source community and best practices of the programming language used, such as C/C++, Python, Java and JavaScript.

Faithfulness

- The architecture must be in strict accordance with the given PRD and UML class diagrams. It should accurately reflect the requirements specified in the PRD and the structural design outlined in the UML diagrams, ensuring a coherent and consistent development process.

Figure 6: Evaluation Guidance for Architecture Design.

Please evaluate the two responses (Response 1, Response 2) based on the provided scoring criteria.

Scoring criteria: <Evaluating Guidance>

- If the response is incomplete or misses any required key component, regard it as a bad one.
- If the response is verbose and/or repetitive, consider it negatively based on the extent.
- If the response is well-formatted and clearly-structured, give it extra credit.

Important: You should act as an IMPARTIAL judge and be as OBJECTIVE as possible. AVOID ANY POSITION BIASES and ensure that the ORDER in which the responses were presented DOES NOT influence your decision.

Please choose from the following two options based on the scoring criteria:

- A. Response 1 is better than Response 2.
- B. Response 2 is better than Response 1.
- C. Tie. [Only for 3-option cases]

Question:

<Question Start> {question} <Question End>

Response 1:

<Response 1 Start> {response 1} <Response 1 End>

Response 2:

<Response 2 Start> {response 2} <Response 2 End>

Reference answer [If provided]:

<Reference Answer Start> {reference answer} <Reference Answer End>

Please provide detailed reasons for your choice. Also, you should pay adequate and the same attention to both responses. Your output should be in the following format:

```
Choice: A
Reason:
1. xxxxx
2. xxxxx
.....
```

Figure 7: LLM-as-a-Judge prompt for software design evaluation

E.2 Instruction Following

Naming Errors Proper naming in code is crucial for readability and maintainability. Larger models, like GPT-4-Turbo, while capable of generating syntactically correct code, exhibit deficiencies in this

area. It inaccurately modifies function and variable names, disrupting the code’s functionality. For instance, in “Graph BFS DFS”, we instruct the use of the “top()” method for stack access, yet GPT-4-Turbo incorrectly labels it as “getTop()”. Fur-

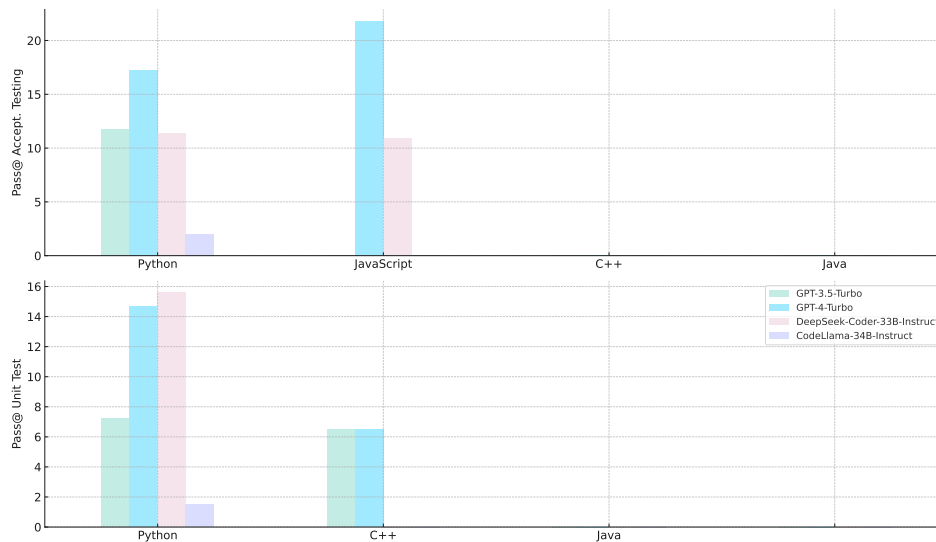


Figure 8: Performance Break Down on Different Languages. The results are averaged across all repositories and weighted by the number of code lines

thermore, the model’s lack of attention to plurals and capitalization aggravates these errors. Despite reviews, this issue remains inadequately addressed.

Function Parameters and Overloading Errors

We observe that even some large models, such as GPT-4-Turbo, neglect the correct number of function parameters, missing critical ones. Function overloading is another similar nuanced aspect that many models mishandle. They often overlook the necessity of multiple constructors or methods with varying parameters. For example, in a task like “area_calculation”, models fail to create both parameterized and parameterless constructors, focusing solely on the former. This oversight is not significantly rectified in the review stages, as models mistakenly attribute the errors to the test program. They stubbornly resist correcting their generated code, even when provided with clear instructions.

Type Errors Models demonstrate a lack of sensitivity to type conversions, especially in strongly-typed languages like C/C++. Errors in matching const types and misusing pointer types are common, and these missteps are not readily resolved in the review process. However, in weakly-typed languages like Python, such issues are less critical but still present a concern for code accuracy.

Variable Scope and Lifecycle Mismanagement

Models frequently misuse variables beyond their intended scope or lifecycle. For example, they might attempt to use a loop control variable outside its loop. Another issue is the misunderstanding of pri-

vate and public members in classes, where models inappropriately access private elements from outside the class. This indicates a gap in the models’ understanding of encapsulation and scope management in object-oriented programming.

E.3 Hallucination

Fabrication of Variables A significant challenge is the models’ propensity to fabricate non-existent local variables, a problem typically rectified during review. This issue suggests a fundamental limitation in the models’ sequential generation process. Unable to retroactively integrate essential variable definitions, the models end up introducing imaginary variables in the code, resulting in apparent inaccuracies.

Misinterpretation of Data Files Models also exhibit a tendency to incorrectly interpret data files as Python libraries. They attempt to import methods from these non-existent libraries, leading to further reference errors. This behavior underscores the intricacy involved in handling file references accurately within code generation tasks.

E.4 Limitations in Testing

In Acceptance Testing task, we employ an execution-based evaluation method for the generated tests, foregoing manual quality assessments. This approach assumes a test is valid if it can accurately assess standard implementation code. However, we observed that smaller models, such as DeepSeek-Coder-6.7B-Instruct, tend to game this

method. They set arbitrary criteria and invariably provide positive feedback, thereby circumventing a genuine evaluation.

Larger models like GPT-4-Turbo fell short of our expectations. They persistently recall and utilize methods in the original repository code, instead of our specially designed versions, leading to frequent import errors. This issue is exemplified in our tests with the “GeoText” and “Stocktrends” repositories. We modified the original repositories by removing “__init__.py” files, expecting models could correctly handle import relationships without them, based on our provided file structures. However, the models continued to follow the import logic of the original repositories, leading to hallucination and inaccurate test generation. This indicates a training data bias, where these models are predisposed to original repository code and show a reluctance to adjust to new circumstance.

To prevent meaningless but executable testing code, structured test templates with explicit instructions and incomplete assertion statements can potentially guide and force models toward meaningful test generation.

F Repositories statistics in DevEval

Table 8 shows the repository statistics within DevEval.

Text	Language	Domain	#code files	#code lines	#code tokens	#acceptance tests	#unit tests	Unit test coverage
TextCNN	Python	DL, NLP	5	403	1566	1	10	99
ArXiv digest	Python	SE, API	1	198	901	4	38	94
chakin	Python	NLP	1	62	225	1	1	86
readtime	Python	ALGO	3	284	920	4	8	95
hone	Python	SE	4	274	844	5	7	90
Stocktrends	Python	ALGO	1	384	1350	2	7	85
GeoText	Python	NLP	2	470	1701	5	4	98
lice	Python	SE	2	376	1329	6	25	88
PSO	Python	ALGO	2	168	578	1	5	93
hybrid images	Python	ALGO, CV	1	144	746	1	19	90
Actor Relationship Game	Java	ALGO, API	8	493	1453	4	16	64.32
Leftist Trees and Fibonacci Heaps Comparison	Java	ALGO	3	632	2009	2	2	45.32
Redis	Java	SE, DB	9	779	2546	1	17	78.6
idcenter	Java	SE	4	333	1140	3	4	54.2
image similarity	Java	SE, CV	3	382	1397	2	2	71.34
xlsx2csv	C/C++	SE	10	476	1440	5	8	95.17
people management	C/C++	SE, DB	6	540	2043	7	9	95.14
Area Calculation	C/C++	SE	7	162	307	3	3	90.48
Graph BFS DFS	C/C++	ALGO	5	667	2828	5	22	/
Logistic Management System	C/C++	SE	7	630	2007	7	17	99.11
listen-now-frontend	JS	Web	6	232	492	1	0	/
register	JS	Web	6	223	741	3	0	/

Table 8: Repository statistics within DevEval