ProjectEval: A Benchmark for Programming Agents Automated Evaluation on Project-Level Code Generation

Kaiyuan Liu[1,2], Youcheng Pan[2✉], Yang Xiang[2], Daojing He[1✉], Jing Li[1]
Yexing Du[1,2], Tianrun Gao[3],
[1]Harbin Institute of Technology, Shenzhen, China
[2]Pengcheng Laboratory, Shenzhen, China
[3]Harbin Institute of Technology, Harbin, China
1171000408@stu.hit.edu.cn, daojinghehit@163.com, panych@pcl.ac.cn,

Abstract

Recently, LLM agents have made rapid progress in improving their programming capabilities. However, existing benchmarks lack the ability to automatically evaluate from users' perspective, and also lack the explainability of the results of LLM agents' code generation capabilities. Thus, we introduce ProjectEval, a new benchmark for LLM agents project-level code generation's automated evaluation by simulating user interaction. ProjectEval is constructed by LLM with human reviewing. It has three different level inputs of natural languages or code skeletons. ProjectEval can evaluate the generated projects by user interaction simulation for execution, and by code similarity through existing objective indicators. Through ProjectEval, we find that systematic engineering project code, overall understanding of the project and comprehensive analysis capability are the keys for LLM agents to achieve practical projects. Our findings and benchmark provide valuable insights for developing more effective programming agents that can be deployed in future real-world production.[1]

1 Introduction

The field of programming has seen significant advances with the rise of large language models (LLMs), today the LLM agents can do many programming works without the help of humans (Liu et al., 2024). To evaluate the programming ability of GPT-2, Chen et al. (2021) raised the first programming benchmark named HumanEval in 2021. HumanEval was also the first of the HumanEval-based benchmarks. In the last three years, more than 20 benchmarks had been raised manually or automatically based on LLM. After HumanEval, MBPP (Austin et al., 2021) came out which also was a base for many benchmarks. MBPP concentrating on algorithm realization. In 2023, DS-1000 (Lai et al., 2023) was raised and represented a new series of benchmarks, evaluating LLM agents programming abilities of accessing the third-party libraries or packages. In 2024, LLM-based programming agents developed rapidly, researchers noticed that many of them can do project-level programming (Hong et al., 2024, Nguyen et al., 2024). Therefore, project-level benchmarks came out, they were SoftwareDev (Hong et al., 2024), ProjectDev (Nguyen et al., 2024), SRDD (Qian et al., 2024), CASSD (Zhang et al., 2024a), and DevBench (Li et al., 2024a). These benchmarks provided the methods evaluated agent's code generation capabilities at the granularity level project. **However, only DevBench achieved automated evaluation using in-project test units, and the test-unit-evaluation was NOT a realistic method in a production environment used by human users. And, the other benchmarks still relied on natural language tests, which usually required human judgment for correctness and efficiency. Moreover, only DevBench provided a table of evaluation scores, but it lacked explainability.**

To bridge this gap, we propose **ProjectEval**, a novel benchmark tailored for automated evaluating project-level programming tasks(missions) in this field. ProjectEval is designed to assess the ability of agents to tackle complex user-driven tasks with precision and adaptability. Unlike existing benchmarks, ProjectEval emphasizes real-world usability by **integrating automated test suites of <u>user interaction simulation</u>, parameter analysis, and canonical solutions** into a cohesive evaluation pipeline. ProjectEval contains 20 real-world tasks with totally 284 testcases, and supports two task types: website-based projects and batch/console-based programs, with theoretical scalability to even more complex, custom UI-based tasks. By leveraging LLMs for generation, supplemented with

✉ Corresponding authors
[1]Dataset, code and constructed evaluation machine are available at https://github.com/RyanLoil/ProjectEval/.

Findings of the Association for Computational Linguistics: ACL 2025, pages 20205–20221
July 27 - August 1, 2025

Benchmarks	**Subject**	**Test Type**	**Construction** Automated	**Input Level** Checklist	Skeleton	**Evaluation** Automated	LLM-less	**Pass@k** Available	**#Tasks**	**#Tests**	**#LOC**	**#Tokens**
SoftwareDev(2024)	Execution	-	✗	✗	✗	-	-	✗	70	-	191.6	6218.0
ProjectDev(2024)	Execution	Manual Checklist Reviewing	✗	✓	✗	✗	✓	✓	14	19.10	-	36818.0
SRDD(2024)	Checklist	LLM-Rating	✓	✓	✗	✓	✗	✗	1200	-	-	-
CASSD(2024a)	Execution	Manual Checklist Reviewing	✗	✓	✗	✗	✓	✓	72	5.25	≈240	21993.0
DevBench(2024a)	Execution	Inside Project Test Units	✗	✗	✗	✓	✓	✓	**22**	10.18	377.8	1298.3
ProjectEval(ours)	Execution	**User Interaction Simulation**	●	✓	✓	✓	✓	✓	20	**14.2**	**402.20**	**2972.0**

Table 1: Summary of Existing Project Level Benchmarks. #Tasks: number of tasks, #Tests: average number of testcase in each task, #LOC: average lines of code in the canonical answer, #Tokens: average number of tokens of code in the canonical answer. ●: Our benchmark is constructed by GPT-4o with human reviewing and editing, which is semi-automated. -:SoftwareDev does not contain any evaluation.

manual reviewing, ProjectEval ensures robust and detailed evaluation metrics.

We introduce the core structure and evaluation process of ProjectEval, highlight the challenges faced by state-of-the-art models such as Gemma-2 and GPT-4o, and demonstrate the benchmark's ability to test comprehensive capabilities through pass rate (Pass@K) and existing algorithms. Our goal is to establish a more effectively, and more explainable new standard for automated evaluating project code quality. We summarize the contributions of this work as follow:

- User-Centric Project-level Benchmark: ProjectEval fills the gap in existing benchmarks by offering a user-focused framework with real-world applicability with comprehensive and project-level metrics. It supports website and batch/console-based projects.
- Enhancing the Agent Code Generation Explainability by Three Different Level Inputs: Three different level inputs integrating three kinds of objective indicators and pass rate, ProjectEval ensures precise, adaptable evaluation and enhances the result explainability.
- Automated Evaluation Testsuites: ProjectEval realizes the automated evaluation from user perspective in website tasks and batch/console tasks through simulating user interaction. This is a new low-cost method to evaluate the code generation capabilities of agents.

2 Related Works

2.1 LLM-Based Coding Agents

There are many code LLMs to date, e.g. StarCoder, InCoder, WizardCoder, CodeGen and etc. However, only the NL-i/o-available models could become programming agents as the instruction of an agent-level mission used to be natural language. Without NL input ability, the LLMs are really hard to become agents through agent designs of CoT, ReAct or Reflextion. Some researchers have developed LLM agents. ChatDev (Qian et al., 2024) presents a diffusion-based model combining large language models with image decoders, advancing text-to-image generation. AgileCoder (Nguyen et al., 2024) introduces multi-agent Agile role assignments for efficient, collaborative software development, while MetaGPT (Hong et al., 2024) encodes workflows into prompt sequences for structured multi-agent task management. Besides some large multimodal reasoning models also have the abilities to generate codes (Li et al., 2025).

2.2 Benchmarks for Code Generation

Table 2 provides a detailed summary of existing benchmarks in the programming domain, and categorize them into six groups: exploratory benchmarks before 2020, HumanEval-based benchmarks (Chen et al. 2021,Liu et al. 2023, Hao et al. 2022, Peng et al. 2024, Athiwaratkun et al. 2023), MBPP-based benchmarks (Austin et al. 2021, Peng et al. 2024, Hendrycks et al. 2021, Huang et al. 2024, Jain et al. 2024, Li et al. 2023, Li et al. 2022), DS-based benchmarks (Lai et al. 2023, Du et al. 2023, Zhang et al. 2024b), problem understanding benchmarks, and those focused on specific domains or methods. With the development of LLM and agents, the benchmarks are gradually moving toward higher granularity level. The project-level is the last and highest level of programming. The project-level benchmarks, which are the primary focus of our research, comprehensively evaluate the process of transforming an initial idea into a complete project.

There are currently five benchmarks in this category, see Table 1 for the differences. In summary, all project-level benchmarks can't evaluate automatically except DevBench and our ProjectEval. Except ours, all benchmarks don't provide a multi-

Benchmark	Language	Construction	Evaluation	Source	Granularity Level	#Tasks	#Tests	#LOC	#Tokens	Input Information
HumanEval-based										
HumanEval (2021)	Python	Manual	Automated	Original	Function	164	7.7	11.5	24.4	NL+ Signature
Multi-HumanEval(2023)	Multiple	Manual	Automated	HumanEval & Original	Function	164	7.7	11.5	24.4	NL+ Signature
MBPP-based										
MBPP(2021)	Python	Manual	Automated	Original	Function	974	3.0	6.8	24.2	NL
CodeContests(2022)	Python, C++	Automated	Automated	Contest Sites	Competitive	165	203.7	59.8	184.8	NL + Example I/O
DS-based										
DS-1000(2023)	Python	Automated	Automated	Stack Overflow	Statement	1000	1.6	3.8	12.8	NL
CoderEval(2024b)	Python, Java	Automated	Automated	Github	Function	230	N/A	30.0	108.2	NL + Signature
SWEBench(2024)	Python	Automated	Automated	PyPI	Commit	2294	120.8	32.8	≈200	Issue + Repository
ClassEval (2023)	Python	Manual	Automated	PyPI + Original	Class	100	33.1	45.7	123.7	Class Skeleton
Granularity Level - Project										
SRDD (2024)	Python	Automated	Automated	Original	Project	1200	N/A	N/A	N/A	NL
CASSD (2024a)	Python	Manual	Manual	Original	Project	72	5.25	≈240	21993.0	NL
SoftwareDev(2024)	Multiple	Manual	N/A	Original	Project	70	N/A	191.6	6218.0	NL
ProjectDev(2024)	Multiple	Manual	Manual	Original	Project	14	19.1	N/A	36818.0	NL
DevBench(2024a)	Multiple	Manual	Automated	Original	Project	22	10.18	377.8	1298.3	NL
ProjectEval (ours)	**Python**	**Semi-automated**	**Automated**	**SoftwareDev & ProjectDev & Origin**	**Project**	**20**	**14.2**	**402.2**	**2972.0**	**NL+ Class Skeleton /Function Skeleton**

Table 2: Summary of Existing Benchmarks for Code Generation. #Tasks: number of tasks, #Tests: average number of testcase in each task, #LOC: average lines of code in the canonical answer, #Tokens: average number of tokens of code in the canonical answer. N/A: This benchmark doesn't involve this item. Part of this table is referred from Du et al.'s (2023). Other categories are in Appendix C.

level inputs. Compared with DevBench, we have more testcases, and our tasks (missions) are complicate than DevBench as more lines of code and more tokens. Besides, existing project-level benchmarks use the test units, manual checklist reviewing or LLM-scoring directly from the tested library or framework to complete the evaluation, rather than actually compiling and executing the project and checking in users' perspective.

For other based benchmarks, see Table 2 for brief and Appendix C for full version.

3 ProjectEval Benchmark

3.1 Benchmark Format

A standard ProjectEval mission will have three parts: Inputs, Test Suite and Canonical Solution. Figure 1 shows an example structure of a standard ProjectEval mission.

As for the inputs, there are three different input types named Level for the test in each mission for the agent to achieve the target in ProjectEval (See example in Figure 1 purple part):

- **Level 1 - Natural Language Prompt (NL Prompt)**: In this level, the agent will receive one or several natural language sentences to describe the target of the project. The agent will create the entire project ONLY based on these sentences.
- **Level 2 – Natural Language Checklist (NL Checklist)**: In this level, the agent will receive a standard natural language checklist describing the project through the abilities and functions that the project should have.
- **Level 3 – Skeleton**: In this level, the agent will receive a skeleton of the standard answer. This skeleton contains doc-strings and comments to describe the project inside.

A mission test suite will contain two parts (See example in Figure 1 orange part):

- **Testcodes**: a mission contains several automated evaluation Python functions similar to HumanEval testcases. But, these testcodes are prohibited using test unit inside the technical stack but using user simulation by operating UI to test the project generated by agents.
- **Parameter Description (PD)**: usually, every testcode has a matching parameter descriptions. PD is used for a special kind of parameter alignment. These parameters are required by the matching testcode to achieve the established test goal(s), *e.g.* in Figure 1, the "test_url" is the URL of the page which can show all the "tasks" that are required by the testcodes. PD is similar with a user manual given by developers to guide users to accomplish what they want to do - that is, the main evaluation concept we designed: evaluation based on the user's perspective.

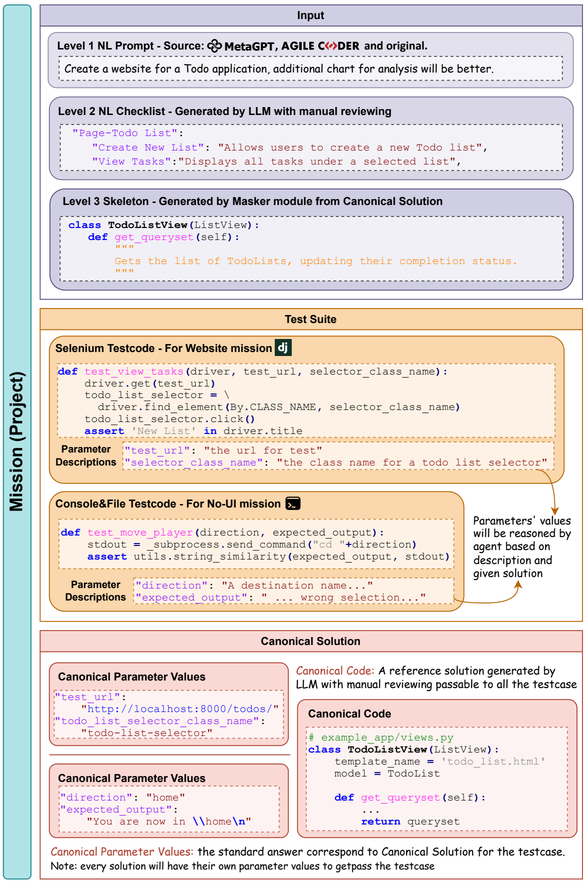

Figure 1: A typical ProjectEval website mission, including three different levels of input, a test suite, and a canonical solution. Notice that the upper test suite is the test suite used in website mission while the lower one is an example of console/file mission test suite.

Finally, every mission we constructed has a canonical solution, beside the canonical code, we also build every PD's standard answer matching to the canonical code called canonical parameter values (See example in Figure 1 red part). In addition, we categorized tasks into "easy", "medium", "hard", and "human" based on code volume and human-reviewed complexity. However, since results show no distinction, we won't elaborate further.

We give the agent missions by JSON format directly embedded into their prompt and asked the same format output, which we consider as a very important ability of code generation.

3.2 Construction Process

The construction process of ProjectEval is relatively complex (See Appendix A for the complete version ProjectEval process and structure diagram).

Level 1 NL Prompt & Level 2 NL Checklist: There are initial 20 tasks (missions) in ProjectEval that are manually edited into concise natural language descriptions, which is Level 1 NL Prompt. 7 of them are sourced from SoftwareDev (Hong et al., 2024) and ProjectDev (Nguyen et al., 2024) while others are created originally by us. Figure 2 purple part shows that these descriptions are sent to an LLM, which generates a list of more detailed natural language task descriptions. After manual review and modification, the refined version is referred as the Level 2 NL Checklist.

Testsuite: Figure 2 orange part shows that the NL Checklist is given into the LLM, which, from a user testing perspective, generates test code. **For website missions, the test code is mostly implemented using the open-source testing library Selenium, which simulates user behavior in a browser to interact with websites. For batch/console tasks, the test code typically uses Python's subprocess module to mimic user interactions such as running commands and entering keyboard input.** If the task involves file generation, the test code utilizes dedicated open-source libraries to read and compare the similarity of the generated file against a canonical file. For example, programs that generate Excel files are validated using the Openpyxl library to compare with the reference files. The test code often requires one or more parameters to execute because the specifics of the code generated by an agent—such as variable names, function names, class names, and output file names—are unpredictable. To address this, the test code is input into the LLM to generate an additional Parameter Description (PD), which provides a natural language explanation of the parameters needed by the test code. The PD, along with the test code, constitutes the Test Suite.

Canonical Solution: Simultaneously, in Figure 2 red part, the NL Checklist is put into another LLM thread to generate a temporary project skeleton, which is then fed back into the LLM to infer and generate Canonical Code (CNC). Practical results show that while most of the code can't be use directly, a little of the LLM-generated code is mostly correct, but it often requires manual corrections to form the true canonical code. This process aligns with the findings of AgileCoder experiments partially. A human reviewer is asked for checking the CNC to confirm that it is runnable and meets the requirements of the Checklist. By inputting the PD and CNC into the LLM and applying minimal manual adjustments, Canonical Parameter Values (CPV) are obtained. Together, CPV and CNC are the Canonical Solution (CNS). When the CNS

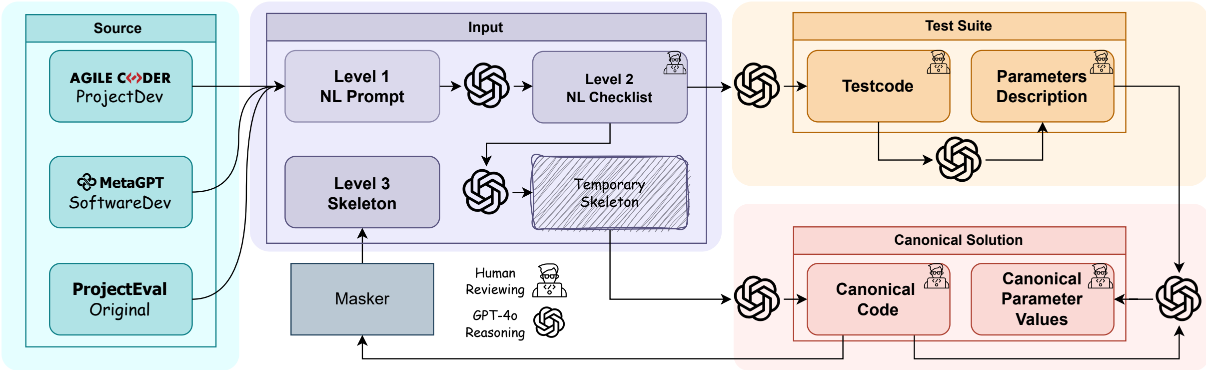

Figure 2: Construction of ProjectEval. Testcode is aligned with Checklist. Parameter Description is aligned with Testcode and Canonical Parameter Values. Canonical Parameter Values is aligned with Canonical Code and use for testcode to get passed.

is input into the ProjectEval testing controller, it achieves a perfect score, *i.e.*, Pass@K = 100%.

Level 3 Skeleton: Finally, the CNC is processed through a Masker (which could be a regex-based program or an LLM) to replace function bodies, class bodies, and critical HTML tag content with functional description comments. This produces a test skeleton that can evaluate LLMs without natural language generation capabilities, referred to as the Level 3 Skeleton.

All CNC are programmed in Python but **ProjectEval theoretically supports any programming language** as we evaluate the LLM through users' perspective. It may need researchers compile the LLM-generated program in advance. The total cost of construction process with GPT-4o is $2.95 and the human reviewing cost is $420 by hiring a third-party company with contract.

3.3 Evaluation Process

The evaluation process begins by selecting a specific level from the input and presenting it to the agent (See Figure 3). The agent can use any designs or methods to solve the inputs and generate Solution Code (Code). The Code is then fed back into the same agent along with the PD. The agent is tasked with answering the parameter description based on its own generated Code to produce Parameter Values (PV). The Code is then converted into an executable file, creating a tangible project in a sandbox environment. This project, together with the testcode with PV substituted, is integrated into the ProjectEval evaluation machine to obtain the evaluation results which is also done in a sandbox.

Since we have different level inputs, we can compare the similarity of the generated results at each level and obtain the score for each step. This process is equivalent to disassembling the CoT of LLM agents to a certain extent, thereby enhancing the explanabilities of the pass rate results.

Parts	Method	Type
Level 2 Checklist	Sentence Transformer (2020) + Jonker Volgenant (1987)	Maximum
Level 3 Skeleton	CodeBLEU (2020) + Jonker Volgenant	Maximum
Code	CodeBLEU + Jonker Volgenant	Maximum
Parameters Values	Levenshtein Distance (1966)	Average

Table 3: ProjectEval Objective Indicators. Four additional objective similarity evaluation methods to evaluate the performance of each parts individually.

Therefore, we introduces four additional objective similarity evaluation methods to evaluate the performance of four parts individually (See Table 3). As a Level 2 Checklist consists of multiple independent natural language sentences, which cannot be considered a cohesive document, after calculating the similarity between each sentence in the canonical Checklist and the test Checklist using Sentence Transformers (Reimers and Gurevych, 2020), the Jonker-Volgenant algorithm (1987) is employed to determine the optimal matching scheme, from which an overall matching score is derived. Since both the Level 3 Skeleton and the answer are written as code, existing code evaluation tools like CodeBLEU (Ren et al., 2020) are used to compute BLEU scores by considering structure similarity. The Skeleton and Code have the rectangular linear sum assignment problems same as Checklist, so Jonker-Volgenant is also used in these parts. Parameter Values are typically short, often

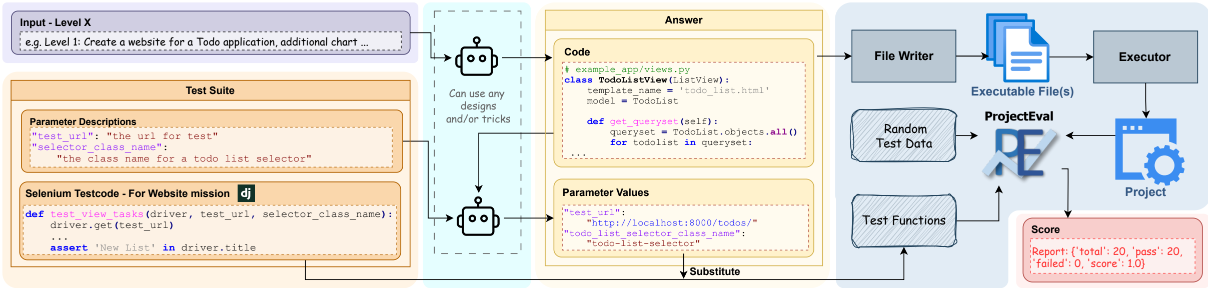

Figure 3: Evaluation Process of ProjectEval. The evaluation process begins by selecting a specific level from the input and presenting it to the agent. The agent generates solution code. The solution code is then fed back into the same agent along with the parameter description. The agent is tasked with answering the parameter description based on its own solution to produce parameter values (PV). The code is then converted into an executable file, creating a tangible project. PV is substitute to testcode, and testcode is integrated into the ProjectEval evaluation machine to obtain the evaluation results.

consisting of single words, compound words, or simple URLs. Therefore, strings cosine similarity is directly used to measure their similarity.

4 Experiments

4.1 Research Questions

Our experiments intend to answer the following research questions:

- **RQ1 (Overall Correctness)**: How do LLM agents perform on ProjectEval benchmark?
- **RQ2 (Cascade Generation & Direct Generation)**: Do LLM agents performs better when they generated level by level till answer code (*i.e.* cascade) than directly generate?
- **RQ3 (Basic LLM Selection)**: Which basic LLM perform the best in the experiments and where it does better than the others?
- **RQ4 (Step by Step Performance)**: How do LLM agents performs on each part of ProjectEval benchmark?

For the basic LLM selection and settings part, see Appendix B for the details.

4.2 Evaluation Metric

Same as many benchmarks of HumanEval-based and MBPP-based, we adapt the pass rate (Pass@K) for every LLM. We have average 14.2, totally 284 testcases (including runnable as a testcase) to evaluate the correctness of Code generated by LLM (See Table 1 for all statistics). The percentage of test cases that passed is the final score that an LLM gains from ProjectEval. Notice that some of the testcases are chain-reacted, as if the former one fails, the followings will never get passed.

We also added 4 objective indicators mentioned in Section 3.3 for each part which agents generated mentioned in Table 3. ProjectEval will compute these metrics in parallel with Pass@K.

The total evaluation cost of Pass@5 with GPT-4o is $28.02, average $5.60 for each round.

Model	Cascade			Direct				All Avg.
	Level 1	Level 2	Avg.	Level 1	Level 2	Level 3	Avg.	
Open-source AGI LLMs								
Llama-2-7B	0.00	0.07	0.04	0.28	0.00	0.07	0.12	0.08
Llama-3.1-7B	0.28	0.28	0.28	0.14	0.28	0.42	0.28	0.28
Llama-3.2-3B	0.21	0.14	0.18	0.14	0.00	0.00	0.05	0.10
Phi-3-14B	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Phi-4-14B	0.14	0.56	0.35	**1.76**	**1.13**	**2.04**	**1.64**	**1.13**
Gemma-7B	0.99	1.06	1.02	0.56	0.63	0.49	0.56	0.75
Gemma-2-9B	**1.69**	**1.06**	**1.37**	1.34	0.56	0.63	0.85	1.06
Mistral-7B-v0.3	1.48	1.06	1.27	0.92	0.99	0.56	0.82	1.00
Code Generation LLMs								
StarCoder-2-7B	-	-	-	-	-	0.00	-	-
CodeGemma	-	-	-	-	-	**1.20**	-	-
CodeLlama	-	-	-	-	-	0.77	-	-
Close-source AGI LLMs								
GPT-3.5-turbo	2.39	2.46	2.43	1.97	2.39	5.28	3.22	2.90
GPT-4o	<u>**8.52**</u>	<u>**12.32**</u>	<u>**10.42**</u>	<u>**16.06**</u>	<u>**15.42**</u>	<u>**10.14**</u>	<u>**13.87**</u>	<u>**12.49**</u>
Gemini 1.5 pro	7.82	7.39	7.61	5.28	4.51	8.24	6.01	6.65
Gemini 2.0-flash	3.24	3.59	3.42	3.52	3.45	7.75	4.91	4.31
Avg.	2.23	2.50	2.37	2.66	2.45	2.51	2.69	2.56

Table 4: ProjectEval Result Pass@5. ProjectEval is hard for recent LLM agents to get pass. GPT-4o has the best score. See Appendix F for Pass@K.

5 Results

5.1 RQ1: Overall Correctness

Table 4 shows the overall correctness is low. The results are very similar to CoderEval(2024b) and Dev-Bench(2024a) as all agents are very unlikely to make the project runnable (only 17.91% projects in CoderEval's result) and almost impossible to make every details correct in the project (lower than 10% passed in DevBench's result).

Model	Cascade							Direct					
	Level 1				Level 2			Level 1		Level 2		Level 3	
	CL	SK	Code	PV	SK	Code	PV	Code	PV	Code	PV	Code	PV
Open-source AGI LLMs													
Llama-2-7B	1.13	0.43	0.32	0.00	0.00	0.49	0.00	0.13	0.00	0.30	0.00	0.40	0.00
Llama-3.1-7B	3.61	2.60	1.41	0.33	1.92	1.99	0.68	1.92	0.68	1.84	0.39	4.74	0.54
Llama-3.2-3B	1.00	0.27	1.61	0.00	0.55	0.00	0.00	0.17	0.00	0.00	0.00	0.17	0.00
Phi-3-14B	5.62	1.69	1.37	0.44	1.14	0.88	0.00	0.30	0.00	0.00	0.00	0.29	0.00
Phi-4-14B	**41.92**	3.74	1.71	1.37	2.42	3.82	4.74	**10.87**	**10.76**	**6.87**	**8.15**	**13.32**	**9.70**
Gemma-7B	38.08	5.70	5.12	0.00	6.98	6.37	0.00	1.95	0.00	2.38	0.00	5.38	0.00
Gemma-2-9B	40.25	**7.90**	7.53	9.32	7.70	**9.05**	6.62	5.50	8.93	5.97	6.59	8.07	5.19
Mistral-7B-v0.3	4.20	7.12	**8.73**	**9.74**	**7.03**	7.48	**7.14**	6.37	6.16	6.80	7.45	7.81	7.23
Code Generation LLMs													
StarCoder-2-7B	-	-	-	-	-	-	-	-	-	-	-	0.00	0.00
CodeGemma	-	-	-	-	-	-	-	-	-	-	-	**9.99**	**14.75**
CodeLlama	-	-	-	-	-	-	-	-	-	-	-	5.44	0.99
Production Code Agent													
OpenHands*†	-	-	-	-	-	-	-	22.96	35.51	-	-	-	-
Close-source AGI LLMs													
GPT-3.5-turbo	38.33	8.82	13.73	38.46	12.56	13.55	42.30	13.27	37.73	13.91	41.19	34.19	39.21
GPT-4o	<u>**55.73**</u>	<u>**16.57**</u>	<u>**36.37**</u>	<u>**54.75**</u>	<u>**15.46**</u>	<u>**36.42**</u>	<u>**53.62**</u>	<u>**35.18**</u>	<u>**51.75**</u>	<u>**33.10**</u>	<u>**50.16**</u>	<u>**53.01**</u>	<u>**62.69**</u>
Gemini 1.5 pro	49.48	14.01	31.96	18.15	15.22	31.04	25.62	15.97	9.99	24.32	22.05	46.51	27.90
Gemini 2.0-flash	51.85	16.08	20.63	6.61	17.63	22.02	11.53	26.39	10.46	24.99	13.39	41.89	19.40
Gemini 2.0-pro*	49.44	13.69	16.86	5.09	19.93	24.09	10.91	2.61	0.00	30.19	16.22	36.95	10.39
Average	29.28	7.59	11.34	11.10	8.35	12.09	12.55	9.28	10.50	11.59	12.74	16.76	12.37

Table 5: ProjectEval Result Objective Indicators. Phi-4, the Gemmas and all close-source LLM agents have abilities to generate Checklist well, but only close-source LLM agents can do the Skeleton and Code well. CL: Checklist, SK: Skeleton, PV: Parameter Value. * We only test the Gemini-2.0-pro and OpenHands pass@1. †We used GPT-4o as Openhands' based LLM.

Table 5 shows that most of the opensource-model agents cannot generate compilable project. We examine through Phi-4 and Gemma-2, they have only 1 to 3 simple projects can be compiled and run. Even if the close-source LLMs can hardly reach the 10% of ProjectEval standard, but the close-source LLM agents do better than the open-source ones which is equivalent to 1 or 2 simple projects get almost full scores.

However, Table 5 shows that the Gemma, Gemma-2 and Phi-4 do have the abilities on Checklist generating as they have approximate score to GPT-3.5-turbo, lower than GPT-4o and Geminis, while the Llama series, Mistral and Phi-3 have very low scores. The latter is caused by the lack of JSON format adaptability.

Both tables shows that Gemma-2 and Phi-4 may have the same capability to GPT-3.5-turbo, but far more way to go for GPT-4o and Gemini-1.5-pro.

We examine the logs of the LLMs and they shows that the typical failure cases are:

- **Invalid Output Format**: LLMs sometimes produce outputs that do not conform to the expected format. This is more common with the Llama series, potentially due to a lack of training data on formatted code and JSON.
- **Missing Essential Files**: Essential files, such as manage.py in Django projects, are often omitted. This may be because these files are considered basic and are therefore underrepresented in training corpora which may more be focusing on function files and their problem.
- **Omitted Content**: LLMs frequently leave sections blank or only comments. This may be due to the prevalence of community corpora where answerers often leave custom parts blank for the questioner to fill in.

Additionally, the Code LLMs agents have almost no effective results can be produced. The reason may be that Skeleton only has natural language descriptions, and it is difficult to fill in the whole framework without the context code.

We also test the current SOTA production coding agent, the OpenHands (Wang et al., 2025), by using GPT-4o as its based LLM. It shows that there were 8 tasks that OpenHands agent did not finish (Task 2, 8, 10, 11, 14, 20 of AgentStuckInLoopError and Task 16, 17 of Not Complete) and lacks a lot of optional features and pages. As ProjectEval's tasks are origin and have no user interaction while the agents generate the code, it is acceptable that OpenHands got lower scores than the source GPT-4o (See Appendix F), but it still beat the Gemini-1.5-pro which indicates that coding agents may have a better chance to pass ProjectEval.

In summary, ProjectEval is hard for nowadays agents as only GPT-4o reach the Pass@5 of 15%. Open-source LLM agents are doing worse than the close-source ones, and Code LLMs agents do not have the abilities to pass ProjectEval.

5.2 RQ2: Cascade Generation & Direct Generation

The average scores of objective indicators in Table 5 show that agents are doing better on cascade generation than the direct generation with 2.06% higher at Level 1 input. The cascade generation in a way mimics the CoT process and the ReAct design of an agent. It allows agents to re-examine the project development procedure and correct some errors. As for the core scores (Pass@K) of all LLMs, they are too low to analyze. But we notice that the Gemmas, Mistral, and Gemini-1.5-pro are doing better in cascade generation.

However, GPT-4o has higher scores when using direct generation mode rather than the cascade mode. So, we study a case, Project 3 – "Create a password generator", of GPT-4o (See Appendix G). We resend all the input and output by order back to GPT-4o and ask the CoT of it. It shows that GPT-4o directly hits the files that need to be generated when using the direct generation mode while it concentrates more on the NL processing and analysis on cascade generation mode. This is an interesting phenomenon, and we suspect that asking the LLM agents to generate according to the thought steps we set induces the LLM to tend to activate parameters about natural language rather than the more important aspects of code generation.

This phenomenon does not affect ProjectEval's evaluating capabilities as the cascade mode is not for ProjectEval core functions as the standard ProjectEval score is evaluating the projects generated by asking the LLM only once.

Also, this finding in ProjectEval Pass@5 is conflicted with the CodeBLEU result, as the latter's cascade scores are higher than the direct scores no matter it uses Level 1 or Level 2 input, both in Code and PV. This means even if CodeBLEU has considered the structure, "details will determine success or failure". For instance, we found that the GPT-4o's cascade generation did has better structure of the code but it just missed filling a path parameter and the result was fatal for the project.

In summary, cascade generation is better than direct generation, and ProjectEval execution pass rate is better than the similarity indicators as the latter cannot reflect the program execution effects.

5.3 RQ3: Basic LLM Selection

The close-source LLM agents have better performance on ProjectEval than the open-source ones (See Table 4). GPT-4o are the SOTA of project generation under ProjectEval evaluation.

The first difference is the ability to generate systematic project code based on natural language. From the objective indicators (Table 5), we find the close-source LLM agents do better on skeleton and code generation, both GPT and Gemini can generate better skeleton reflecting well to the standard code. We used GPT-4o as the ProjectEval's data generation procedure base model but Gemini-1.5-pro reaches almost the same performance of GPT-4o. Thus, the reason may not be the familiarity of GPT's prompt. This suggests that close-source LLM agents may have better ability on reflecting the Checklist into a skeleton or framework. The Checklist and Skeleton's Functions and/or Classes are many-to-many relationship. It is a very complicate mission for open-source LLM agents to solve.

Second, the over-all understanding of the project and comprehensive analysis capabilities are also very important for the LLM agents. Close-source LLM agents are doing well on all parts of the ProjectEval inputs and the open-source LLM agents will have a better chance to get passed ProjectEval when they have better performance on those parts.

Third, already mentioned in Section 5.1, the LLM agents' formatted output capabilities have huge influence. We ask all LLM agents to output JSON format. All the close-source agents can do well compare with only Phi-4, Gemma and Gemma-2 have this capabilities of open-source agents. This finding is important for research or engineering that requires the LLM Agent to be the controller, as they need to guarantee stable and regulated outputs.

In summary, the close-source LLM agents are doing better on ProjectEval. GPT-4o are the SOTA of project generation under ProjectEval evaluation.

5.4 RQ4: Step-by-Step Performance

Objective indicators can also show the step-by-step performance in cascade generation (See Table 5).

Except the Llama series, from the cascade perspective, the LLM agents perform well in the field of NL generation for Checklists, which is also one of its fundamental capabilities. In addition, the Checklist itself includes an understanding of project prompts which means most LLMs agents also possess this capability.

When it comes to the Skeleton, we have already mentioned the problem of many-to-many relationship question of Checklist and Skeleton. This is a challenge for LLM agents to deal with. And Code generation is a traditional topic of LLM agents, Phi-4 does better when it directly generates the code from Level 1 rather than the cascade.

Parameter answering reflects the code understanding capability. Though, in ProjectEval, the score may highly connect with the Code generations but still may indicate that the capability of LLM agents. When the code is effective, both open-source and close-source LLM agents can explain their own code, which confirms that LLM agents have a strong ability to understand code. Specifically, it can identify the key statements needed from the PD as for ProjectEval questions.

In summary, LLM agents are best at generating Checklists than the other parts of ProjectEval.

6 Conclusion

We develop a new benchmark ProjectEval. It fills the gap for the lack of benchmark in the project granularity level code generation field of natural language processing and provides automated evaluation tools for higher-level research in LLM agent. We also leverage additional objective metrics to reveal the effectiveness of LLM agents at different stages of project generation. These metrics are crucial in revealing the capabilities for improvement in the agents' performance, thus offering a deeper understanding of how these models can be further enhanced. We confirmed that GPT-4o is still the SOTA in this field. Our findings and benchmark provide valuable insights for developing more effective programming agents that can be deployed in real-world production environments.

7 Limitation

- Some projects, whether due to complexity or human-related difficulties, may follow common design patterns that do not align with Django, our primary technical stack.
- The canonical answer right now only supports Python language but in theory, any language that can be rendered and simulated can be tested by the Test suite.
- JSON format is not universally compatible with all LLMs. Some models might perform better if they were allowed to generate output in their own format. However, permitting this would compromise fairness and introduce inconsistencies in output standards.
- CPV is based on the CNC, but the CNC may not the only answer for the project. This may lead to some PV is correct for the reflected code but will get lower score on Levenshtein Distance evaluation.

All these limitations will be solved in the future research if it is possible.

Besides, the ProjectEval judge machine will automatically run the project that generated by LLM agents which may contain harmful code. This is a potential risk that we can't fix. There are also some issues that this work does not fully capture like software maintainability, efficiency, or best coding practices.

Acknowledgments

This work is supported by the National Science and Technology Innovation 2030 Major program (Grant No. 2024ZD01NL00101).

References

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, and Bing Xiang. 2023. Multi-lingual evaluation of code generation models. *Preprint*, arXiv:2210.14868.

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.

Mark Chen, Jared Tworek, Heewoo Jun, et al. 2021. Evaluating large language models trained on code. *arXiv:2107.03374*.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *2308.01861*, arXiv:2308.01861.

Google. 2023a. Gemini: Chat to supercharge your idea.

Google. 2023b. Gemma: Introducing new state-of-the-art open models.

Alex Gu, Baptiste Roziere, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. 2024. CRUXeval: A benchmark for code reasoning, understanding and execution.

Yiyang Hao, Ge Li, Yongqiang Liu, Xiaowei Miao, He Zong, Siyuan Jiang, Yang Liu, and He Wei. 2022. AixBench: A code generation benchmark dataset. *arXiv:2206.13179*.

Dan Hendrycks, Steven Basart, and Saurav Kadavath. 2021. Measuring coding challenge competence with apps. In *Advances in Neural Information Processing Systems*.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jurgen Schmidhuber. 2024. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*.

Dong Huang, Yuhao Qing, Weiyi Shang, Heming Cui, and Jie Zhang. 2024. EFFIBENCH: Benchmarking the efficiency of automatically generated code. In *NeurIPS 2024*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv:1909.09436*.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018a. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652, Brussels, Belgium. Association for Computational Linguistics.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018b. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP 2018)*, pages 1643–1652. Association for Computational Linguistics.

Nikita Jain, Ke Han, and Aiden Gu. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv:2403.07974*.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7b. *Preprint*, arXiv:2310.06825.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

R. Jonker and A. Volgenant. 1987. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computers & Operations Research*, 14(5):325–340.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. DS-1000: A natural and reliable benchmark for data science code generation. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 18319–18345. PMLR.

Vladimir I. Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, 10(8):707–710.

Bowen Li, Wenhan Wu, Ziwei Tang, Lin Shi, John Yang, Jinyang Li, Shunyu Yao, Chen Qian, Binyuan Hui, Qicheng Zhang, Zhiyin Yu, He Du, Ping Yang, Dahua Lin, Chao Peng, and Kai Chen. 2024a. Prompting large language models to tackle the full software development lifecycle: A case study. *arXiv:2403.08604*.

Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. 2024b. Evocodebench: An evolving code generation benchmark aligned with real-world code repositories. *arXiv:2404.00599*.

Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. 2023. Taco: Topics in algorithmic code generation dataset. *arXiv:2312.14852*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals.

2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Yunxin Li, Zhenyu Liu, Zitao Li, Xuanyu Zhang, Zhenran Xu, Xinyu Chen, Haoyuan Shi, Shenyuan Jiang, Xintong Wang, Jifang Wang, Shouzheng Huang, Xinping Zhao, Borui Jiang, Lanqing Hong, Longyue Wang, Zhuotao Tian, Baoxing Huai, Wenhan Luo, Weihua Luo, Zheng Zhang, Baotian Hu, and Min Zhang. 2025. Perception, reason, think, and plan: A survey on large multimodal reasoning models. *Preprint*, arXiv:2505.04921.

Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, Li Zhang, Zhongqi Li, and Yuchi Ma. 2024. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv:2404.00971*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, volume 36, pages 21558–21572. Curran Associates, Inc.

Minh Huynh Nguyen, Thang Phan Chau, Phong X. Nguyen, and Nghi D. Q. Bui. 2024. Agilecoder: Dynamic collaborative agents for software development based on agile methodology. *arXiv:2406.11912*.

OpenAI. 2023. Openai.

Qiwei Peng, Yekun Chai, and Xuhong Li. 2024. Humaneval-xl: A multilingual code generation benchmark for cross-lingual natural language generalization. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 8383–8394, Torino, Italia. ELRA and ICCL.

Phi. 2023. Phi: A family of powerful, small language models (slms) with groundbreaking performance at low cost and low latency.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15174–15186, Bangkok, Thailand. Association for Computational Linguistics.

Nils Reimers and Iryna Gurevych. 2020. Making monolingual sentence embeddings multilingual using knowledge distillation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 4512–4525, Online. Association for Computational Linguistics.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a method for automatic evaluation of code synthesis. *Preprint*, arXiv:2009.10297.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. Llama: Open and efficient foundation language models. *Preprint*, arXiv:2302.13971.

Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, Dan Roth, and Bing Xiang. 2023. ReCode: Robustness evaluation of code generation models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13818–13843, Toronto, Canada. Association for Computational Linguistics.

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. 2025. Openhands: An open platform for ai software developers as generalist agents. *Preprint*, arXiv:2407.16741.

Yiqing Xie, Alex Xie, Divyanshu Sheth, Pengfei Liu, Daniel Fried, and Carolyn Rose. 2024. Codebenchgen: Creating scalable execution-based code generation benchmarks. *Preprint*, arXiv:2404.00566.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 476–486, New York, NY, USA. Association for Computing Machinery.

Daoguang Zan, Ailun Yu, Wei Liu, Dong Chen, Bo Shen, Wei Li, Yafen Yao, Yongshun Gong, Xiaolin Chen, Bei Guan, Zhiguang Yang, Yongji Wang, Qianxiang Wang, and Lizhen Cui. 2024. CodeS: Natural language to code repository via multi-layer sketch. *Preprint*, arXiv:2403.16443.

Simiao Zhang, Jiaping Wang, Guoliang Dong, Jun Sun, Yueling Zhang, and Geguang Pu. 2024a. Experimenting a new programming practice with llms. *Preprint*, arXiv:2401.01062.

Yakun Zhang, Wenjie Zhang, Dezhi Ran, Qihao Zhu, Chengfeng Dou, Dan Hao, Tao Xie, and Lu Zhang. 2024b. Learning-based widget matching for migrating gui test cases. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, page 1–13. ACM.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M

Rush. 2024. Commit0: Library generation from scratch. *Preprint*, arXiv:2412.01769.

A ProjectEval Structure and Construction Process Full-version

Figure 4 shows the full-version of the structure and construction process of ProjectEval.

B Basic LLM Selection and Settings

Since the ProjectEval Level 1, Level 2 input and PV input are natural language but the Level 3 input, and code output are program language, the basic LLM models should have both aspects abilities. This means that ProjectEval full evaluation must be run under AGI LLM models.

We select three types of models: open-source AGI LLMs, close-source AGI LLMs, and code generation LLMs. Among the open-source AGI models, we include Mistral-7B-v0.3(2023), Gemma(2023b), Phi(2023), and Llama(2023), which are known for their advancements in general AI capabilities. In the close-source AGI LLM category, we consider models such as GPT(OpenAI, 2023) and Gemini(Google, 2023a), which represent cutting-edge proprietary models excelling in a variety of tasks.

Additionally, the Code generation LLMs category features CodeLlama and Starcoder2, which are specialized in code generation and will be used only in Skeleton input evaluation.

For Gemma series, we include Gemma-7B, Gemma2-9B; For Phi series, we include Phi-4, Phi-3-14B; For Llama series we include Llama3.2-3B, Llama3.1-8B, Llama2-7B; For GPT series, we include GPT-4o and GPT-3.5-turbo; For Gemini we include Gemini-1.5-pro and Gemini-2.0. All the models are running under temperature zero with all settings default in their releases.

C Related Benchmarks

Table 6 is the full-version of the related benchmarks.

HumanEval-based Benchmarks: These benchmarks are similar to or extensions of OpenAI's HumanEval, primarily emphasizing functional tasks with general-purpose functions.

MBPP-based Benchmarks: These benchmarks are similar to or derived from Google's MBPP, focusing on algorithmic problem-solving functions.

DS-based Benchmarks: DS-based benchmarks involve the use of external libraries or classes. However, the specific contents and documentation of these libraries/classes are not included within the benchmark dataset itself.

Program Understanding Benchmarks: Unlike other benchmarks that focus on whether the code is written correctly, these benchmarks assess the ability of an agent or LLM to thoroughly understand the provided code.

Project-level Benchmarks: Project-level benchmarks, which are the primary focus of this paper, comprehensively evaluate the process of transforming an initial idea into a complete program. There are currently five benchmarks in this category, see Table 1 for the differences.

D Prompts & Outputs

Here is the prompt of the Reasoning:

ProjectEval Reasoning Prompt

Based on this {description}, give a {technical_stack} Project of its all files (including the essential files to run the project) to meet the requirement in JSON format of `[ {"file":"answer.something","path": "somepath/somedir/answer.something", "code":"the_code_in_the_file"},{...}, ... ]` with **NO other content**. Recommend adding an `id` attribute to each HTML element and adding classes for them too.

The reasoning phase will eventually come to an JSON file. Here is an example of the output:

An Example of ProjectEval Output

```
{
    "19": [
        {
            "file": "19-stdanswer.py",
            "path": "19-stdanswer.py",
            "code": "import pandas as pd\n
                ..."
        }
    ]
}
```

E Human Annotators Instruction

Our human annotators have an 8-page instruction guideline for them to annotate the code in both the Canonical Solutions and the Test Suites. It is impossible to show it in the paper and in the appendix as it is in Chinese and it is too long with instruction images and an assisting small information web

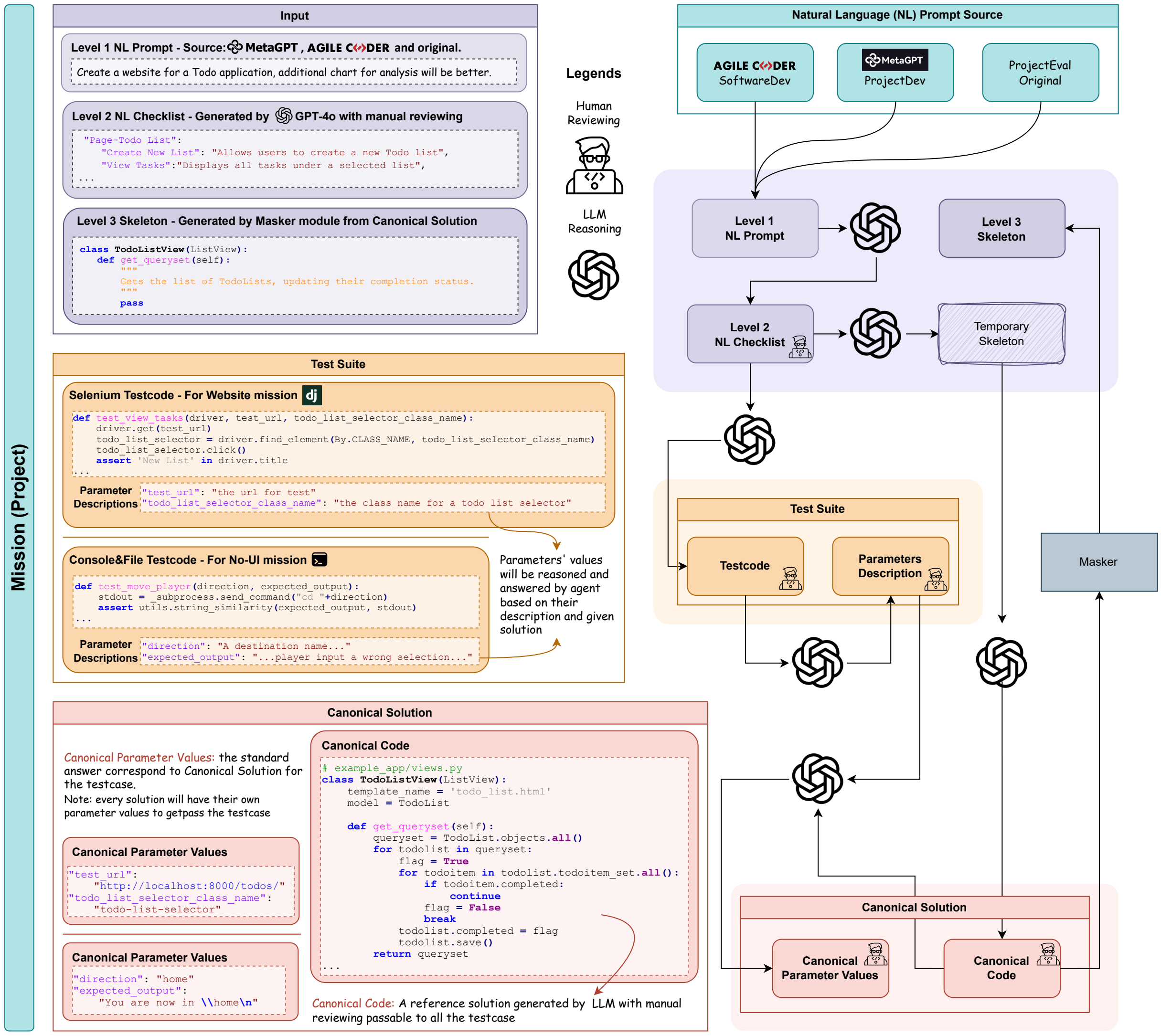

Figure 4: ProjectEval Structure and Construction Process

system. As compensation, we provide the following annotation key steps (listed in the instruction guideline):

1. Review the checklist, see if there are any inappropriate requirements and remove them or edit them, and add the missing general requirements;
2. Review the answer code in render mode, and make sure it can meet all the requirements of the annotated checklist;
3. Run the answer code and review the test code and the parameter values in render mode, make sure it can test the checklist requirements, and those common-sense requirements (such as login authorization check), since the common-sense requirements may not be added in Step 2, you may edit the answer code at the same time;
4. Review the parameter description, add the missing ones, and remove the redundant ones;
5. Run the entire answer code again, make sure that all the tests can be aligned with the answer.

Figure 5 shows a screenshot of the assisting small information web system.

F ProjectEval Result Pass@K

Table 7 is the full result of the ProjectEval.

Benchmark	Language	Construction	Evaluation	Source	Granularity Level	#Tasks	#Tests	#LOC	#Tokens	Input Information
Early Years Research										
Concode (2018a)	Java	Automated	Automated	GitHub	Function	2000	N/A [1]	N/A [1]	26.3	NL
CoNaLA (2018)	Python	Automated	Automated	Stack Overflow	Statement	500	N/A [1]	1.0	4.6	NL
BLEU-based										
Django (2018b)	Python									
CodeBLEU(2020)	Multiple	N/A[2]	Automated	N/A[2]	Project	N/A[2]				Code
SketchBLEU(2024)	Multiple									
HumanEval-based										
HumanEval (2021)	Python	Manual	Automated	Original	Function	164	7.7	11.5	24.4	NL+ Function Signature
AixBench (2022)	Java	Manual	Automated	HumanEval & Original	Function	175		N/A[3]		NL+ Function Signature
Multi-HumanEval(2023)	Multiple	Manual	Automated	HumanEval & Original	Function	164	7.7	11.5	24.4	NL+ Signature
HumanEval+	Python	Manual	Automated	Original	Function	164	774.8	11.5	24.4	NL+ Signature
MBPP-based										
MBPP(2021)	Python	Manual	Automated	Original	Function	974	3.0	6.8	24.2	NL
APPS(2021)	Python	Automated	Automated	Contest Sites	Competitive	5000	13.2	21.4	58	NL+ Examples
MBXP(2023)	Multiple	Manual	Automated	MBPP & Original	Function	974	3.0	6.8	24.2	NL
CodeContests(2022)	Python, C++	Automated	Automated	Contest Sites	Competitive	165	203.7	59.8	184.8	NL + Example I/O
DS-based										
DS-1000(2023)	Python	Automated	Automated	Stack Overflow	Statement	1000	1.6	3.8	12.8	NL
CoderEval(2024b)	Python, Java	Automated	Automated	Github	Function	230	N/A[1]	30.0	108.2	NL + Function Signature
ClassEval (2023)	Python	Manual	Automated	PyPI + Original	Class	100	33.1	45.7	123.7	Class Skeleton
EvoCodeBench (2024b)	Python	Semi-automated	Automated	GitHub	Function	275	N/A [1]	20.40	185.57	Repository
SWEBench(2024)	Python	Automated	Automated	PyPI	Commit	2294	120.8	32.8	≈ 200	NL + Repository
Commit0(2024)	Python	Automated	Automated	PyPI	Commit	54	≈ 500	≈ 25000	≈ 1250	NL + Testunits
Program Understanding										
ReCode(2023)	Python	Automated	Automated	HumanEval & MBPP	Function	30	10.0	N/A [1]	N/A [1]	Code
CRUXEval (2024)	Python	Automated	Automated	Python Standard Libs	Function	800	10.0	5.49	N/A [1]	Code
CodeBenchGen(2024)	Python	Automated	Automated	CodeSearchNet(2020) (GitHub)	Function	1931	8.79	60.5	491.9	Code + NL Statements
Granularity Level - Project										
SRDD (2024)	Python	Automated	Automated	Original	Project	1200	N/A [1]	N/A [1]	N/A [1]	NL
CASSD (2024a)	Python	Manual	Manual	Original	Project	72	5.25	≈240	21993.0	NL
SoftwareDev(2024)	Multiple	Manual[1]	N/A	Original	Project	70	N/A [1]	191.6	6218.0	NL
ProjectDev(2024)	Multiple	Manual	Manual	Original	Project	14	19.1	N/A [1]	36818.0	NL
DevBench(2024a)	Multiple	Manual	Automated	Original	Project	22	10.18	377.8	1298.3	NL
ProjectEval (ours)	**Python**	**Semi-automated**	**Automated**	**SoftwareDev & ProjectDev & Origin**	**Project**	**20**	**14.2**	402.2	2972.0	**NL+ Class/Function Skeleton**

[1]This benchmark doesn't involve this item.

[2]The Bleu-based benchmark doesn't involve construction, source and number items.

[3]Since Aixbench wrote 175 Java files for testing and wrote the test samples directly into the code, it is very difficult to count its details.

Table 6: Summary of Existing Benchmarks for Code Generation. #Tasks: number of tasks, #Tests: average number of testcase in each task, #LOC: average lines of code in the canonical answer, #Tokens: average number of tokens of code in the canonical answer. Part of this table is referred from Du et al.'s (2023).

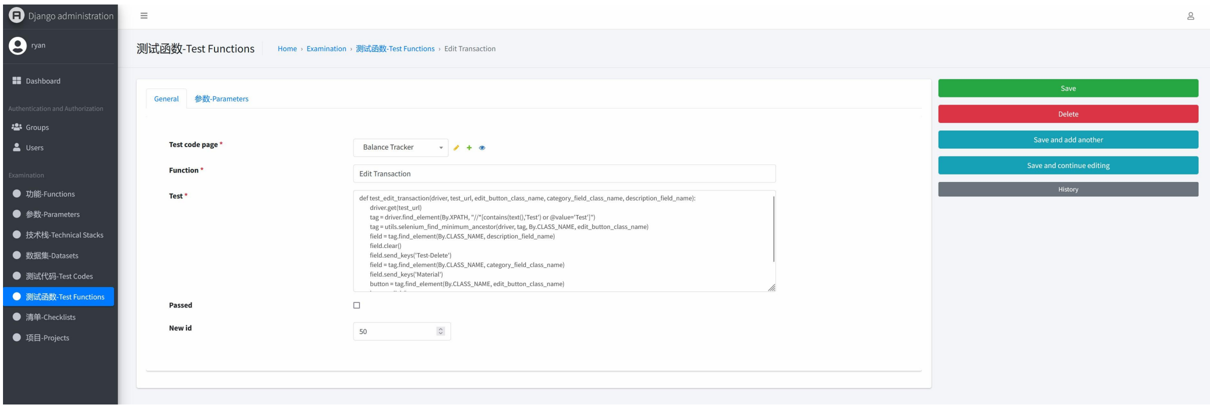

Figure 5: screenshot of the Assisting Websystem

Model	Pass@1					Pass@5							All Avg.
	Cascade		Direct			Cascade			Direct				
	Level 1	Level 2	Level 1	Level 2	Level 3	Level 1	Level 2	Avg.	Level 1	Level 2	Level 3	Avg.	
Open-source AGI LLMs													
Llama-2-7B	0.00	0.35	1.06	0.00	0.35	0.00	0.07	0.04	0.28	0.00	0.07	0.12	0.08
Llama-3.1-7B	0.70	0.70	0.35	0.70	1.06	0.28	0.28	0.28	0.14	0.28	0.42	0.28	0.28
Llama-3.2-3B	0.70	0.35	0.35	0.00	0.00	0.21	0.14	0.18	0.14	0.00	0.00	0.05	0.10
Phi-3-14B	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Phi-4-14B	0.35	1.06	**4.23**	**1.41**	**4.23**	0.14	0.56	0.35	**1.76**	**1.13**	**2.04**	**1.64**	**1.13**
Gemma-7B	1.41	1.41	1.06	0.70	1.06	0.99	1.06	1.02	0.56	0.63	0.49	0.56	0.75
Gemma-2-9B	**2.11**	**2.46**	2.11	1.06	1.06	**1.69**	**1.06**	**1.37**	1.34	0.56	0.63	0.85	1.06
Mistral-7B-v0.3	2.11	1.76	1.41	1.41	0.70	1.48	1.06	1.27	0.92	0.99	0.56	0.82	1.00
Code Generation LLMs													
StarCoder-2-7B	-	-	-	-	0.00	-	-	-	-	-	0.00	-	-
CodeGemma	-	-	-	-	**2.11**	-	-	-	-	-	**1.20**	-	-
CodeLlama	-	-	-	-	1.41	-	-	-	-	-	0.77	-	-
Production Code Agent													
OpenHands	-	-	7.39	-	-	-	-	-	-	-	-	-	-
Close-source AGI LLMs													
GPT-3.5-turbo	2.46	2.46	2.46	2.46	6.69	2.39	2.46	2.43	1.97	2.39	5.28	3.22	2.90
GPT-4o	**10.21**	**15.85**	**19.72**	**17.96**	**12.32**	**8.52**	**12.32**	**10.42**	**16.06**	**15.42**	**10.14**	**13.87**	**12.49**
Gemini 1.5 pro	9.15	8.80	7.39	6.34	9.51	7.82	7.39	7.61	5.28	4.51	8.24	6.01	6.65
Gemini 2.0-flash	5.63	5.28	5.63	6.69	8.80	3.24	3.59	3.42	3.52	3.45	7.75	4.91	4.31
Gemini 2.0-pro*	4.93	4.93	0.00	5.63	7.39	-	-	-	-	-	-	-	-
Avg.	3.06	3.49	3.52	4.03	3.54	2.23	2.50	2.37	2.66	2.45	2.51	2.69	2.56

* We only test the Gemini-2.0-pro pass@1.

Table 7: ProjectEval Result Pass@k

G Casestudy Log of GPT-4o

G.1 Cascade Version

Listing 1: Cascade Verison

```
My chain of thought for each step in the log is as follows:

1. **Instruction to Generate Natural Language Function Checklist**:
   - **Input Reasoning**: A request is made to create a website for a password generator and generate
      a natural language function checklist from the users' perspectives in a JSON format.
   - **Action Taken**: Based on the request, I generate a JSON object representing different pages of
      the proposed website, each containing functions for user interactions, along with
     descriptions of these functions.
   - **Output Reasoning**: I structure the output JSON to clearly separate the different pages (e.g.,
      Home, Password Generator, Settings) and the functions associated with each, providing a
     concise description for each user-facing action.

2. **Generate Django Framework Skeleton**:
   - **Input Reasoning**: Build a Django framework skeleton based on the previously provided
     checklist, using JSON format to outline the necessary files and logic.
   - **Action Taken**: Construct a basic Django app structure, including `views.py` and `urls.py` for
      simplicity, containing placeholder functions that align with the provided natural language
     functions.
   - **Output Reasoning**: The JSON output details files like `views.py` and `urls.py`, including
     paths and basic skeleton code for each function as derived from the JSON function checklist.

3. **Complete Django Project with Essential Files**:
   - **Input Reasoning**: Expand the Django project to include all essential files for a basic
     working project, add ID attributes, and consider adding classes to HTML elements for future
     CSS or JavaScript interactions.
   - **Action Taken**: Create a comprehensive JSON representation of a Django project, including
     files such as `manage.py`, `settings.py`, `urls.py` for the project and the app, HTML
     templates, and CSS files necessary for the application.
   - **Output Reasoning**: The output JSON contains each file's relative path and code content,
     ensuring each crucial component for running a Django application is included. It also adheres
     to the instruction of adding identifiers like IDs and classes to HTML elements.

4. **Generating Required Parameters for Django Project Test**:
   - **Input Reasoning**: Derive and return necessary parameters for testing each function within the
      previously defined Django project based on the requirements outlined, using JSON format,
     specifying example URLs and identifiers.
   - **Action Taken**: I extract specific identifiers (like IDs and URLs) used in the Django project
     HTML and map them to their respective functions in a JSON format that defines parameter names
     and their corresponding examples.
   - **Output Reasoning**: The JSON output specifies each function's parameters for testing purposes,
      using IDs defined in the HTML and expected URL paths. Each function in the project maps back
     to a user scenario described earlier, ensuring the testing parameters align with those
     scenarios.

Throughout each step, I maintained a structured approach to interpret the instructions, generating
    JSON outputs that capture the interaction logic, application skeleton, and testing parameters in
     alignment with the requirements provided.
```

G.2 Direct Version

Listing 2: Direct Verison

```
Given the provided log, here's how I processed the task requests step-by-step:

### Initial Request and Reasoning

1. **Request to Provide a Django Project:**

   - The request was to create a Django project for a password generator. The specific structure was
     requested in a JSON format that included all necessary files for the project.
   - The project was constructed with essential modules and files, typical of a basic Django setup: `
     manage.py`, `__init__.py`, `settings.py`, `urls.py`, `wsgi.py`, etc.
```

```
   - Inside the project folder, two main components were established: `password_generator` for
       project-level configurations and `generator` for the app dedicated to password generation
       logic.

2. **Project Configuration and Application Code:**

   - **`manage.py`**: Standard entry point for running Django commands.
   - **`settings.py`**: Configures the project's settings including installed apps (like `generator`)
       , middleware, database configurations, allowed hosts, static files handling, etc.
   - **`urls.py`** at the project level defines the URL routes and maps them to respective views in
       the `generator` app.
   - The app `generator` has core logic hosted in its `views.py`, handling requests to generate
       passwords with different options (uppercase, numbers, symbols).

3. **HTML Templates for User Interaction:**

   - **`home.html`**: Presents users with options to generate a password, using forms with checkboxes
        and a submit button to include various character types.
   - **`password.html`**: Displays the generated password, allowing users to generate another if
       desired.
   - **`about.html`**: Provides basic project information.
   - IDs and classes were assigned to HTML elements per the requirement to enable easy referencing in
        styles and scripts.

### Parameterized Test Outputs

4. **Parameters Extraction for Test:**

   - A structured JSON was requested to determine specific parameter values within the project for
       automated testing purposes.
   - Each HTML element's IDs, button names, and page URLs were mapped to ensure correct access and
       functionality tests.

5. **Page: Home:**

   - It confirmed the main URL and associated ID for displaying the welcome message (`title`).
   - Navigation from the home page via specific button IDs to the password generator feature was
       detailed.

6. **Page: PasswordOptions:**

   - Confirmed the identification of form elements like length input and checkboxes for including
       character types (uppercase, numbers, symbols) during password generation.

7. **Page: GeneratedPassword:**

   - Ensured the display of generated passwords was accurately linked via the element ID (`password`).

   - Proposed testing for a "Copy to Clipboard" functionality (though the ID `copy` was assumed, this
        component may have been developed later).

8. **Page: About:**

   - Validated navigation to and from the 'About' page. Extracted the ID from the hyperlink (`about`)
        as well as a description ID, although it posed wrong by using `title` where expected a
       unique ID for description was needed.

9. **Errors & Oversights:**

   - Some mismatches were found in the parameter mappings, notably in assuming generic IDs like `
       length` to also be suitable for lowercase inclusion function tests or reusing `title` for `
       description_id`.
   - This indicates that further adjustments or clarification might be needed to align data bindings
       for consistent testing.

This reflection covers the derivation of each function response from the given initial requests and
     input, along with the pathway the project's elements were stored and rationalized for potential
     tests.
```