POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

CodeAgent Project Report

Literature Review & Experiment Replication

Authors:
Acquadro Patrizio
Yu Zheng Maria

Course: Large Language Models: Applications, Opportunities and Risks

Professors:
Prof. Carman Mark James
Prof. Brambilla Marco
Prof. Pierri Francesco

Academic Year: 2024-25

Contents

1 | Introduction

The advent of large language models (LLMs) has promoted significant progress in the field of code generation, which consists in producing runnable code from natural language requirements, boosting productivity in academic and industrial activities.

In this project, we have explored this topic with the paper *"CodeAgent: Enhancing Code Generation with Tool-Integrated Agent Systems for Real-World Repo-level Coding Challenges"* [26]. It proposes the first LLM-based agent framework for repository-level code generation tasks, leveraging multiple tools with different strategies to provide useful information to the model in the optimal way.

1.1. Goal and Motivation

Our decision to consider the CodeAgent paper is motivated by the recognition of the complexity of real-world programming tasks and the paper's complete approach to managing them through agent-based systems.

We are particularly motivated by the current relevance of agents and wanted to explore this trend using the LLM foundations from the course. As programmers, we are also driven by the potential of tools to boost development efficiency, and CodeAgent is a valuable resource in this direction.

Reproducing 5 complex tools without access to the original code posed a challenging task, offering deep insights into agent-tool integration. We also replicated the CodeAgentBench dataset to understand the structure of repo-level evaluation environments, one of the paper's most innovative contributions.

Thus, our goal is to faithfully implement the CodeAgent system - including both agent and tool suite - while designing a realistic dataset for empirical evaluation.

1.2. Workflow

We started by analyzing the existing literature related to agents for code generation, including guides and benchmarks, to acquire fundamental knowledge of the subject.

Then, we thoroughly examined the CodeAgent paper and its dataset, in order to design MiniTransformers, a minimal repo-level benchmark that is statistically representative of the CodeAgentBench. Following the paper, we also replicated the experiments by implementing the five tools in combination with the four agent strategies, for each of the tested LLMs.

Finally, our observations and execution results are reported and discussed in the evaluation and conclusion sections.

2 | Literature Review

2.1. Code LLMs

Well-known general-purpose LLMs like GPT [16], Gemini [7], and DeepSeek [4] are trained on a huge variety of data including programming code, and they can be applied in code generation tasks indeed. In parallel with them, numerous models tailored for coding have been released recently. For example, AlphaCode [14] is pre-trained on a collection of selected GitHub code and fine-tuned on competitive programming problems. It obtained remarkable results in simulated Codeforces contests, outperforming a large part of active users. StarCoder [13] is trained on code in 86 programming languages, and fine-tuned on billions of Python tokens.

Some coding-specific models are built upon generalist ones, leveraging the transfer learning abilities developed during pretraining. The *"Qwen Technical Report"* [2] notes in fact that using both text and code in pretraining improves the versatility of models fine-tuned on code, like Code-Qwen. Similarly, Code Llama [19] builds on Llama 2, with additional fine-tuning on instruction data and long-context inputs to enhance performance on coding tasks.

2.2. LLM Agents: Reasoning and Tools

Training LLMs is heavily resource-demanding, due to the large amount of parameters and data. In fact, much effort has been put on enhancing the performance without re-training them, and studies have revealed that prompts heavily affect.

To enhance the effectiveness of LLMs, J. Wei et al. propose the chain-of-thought (CoT) prompting technique [22], which consists of a series of intermediate reasoning steps, leading to the solution of more complex problems. The model can effectively emulate the logic behind the given sample to reach the final goal step by step. We have seen during the course that it works impressively also in zero-shot settings, by simply incorporating the sentence "let's think step by step" into the user prompt. However, the model only relies on its static internal representation, without leveraging external feedback. Subsequently, ReAct [23] devises a paradigm that induces the LLM to generate interleaved reasoning traces and actions, allowing the model to dynamically adjust plans for acting (reason to act), while interactions with the external environment provide further details for reasoning (act to reason). The disadvantage of ReAct is that the framework could be prone to getting stuck in loops, if the observations are not relevant enough to convince the model to accept the results. CoTs also help in developing the concept of self-planning approach [10]: in the planning phase, LLMs are asked to generate plans for the problem through few-shot prompting with samples of (intent, plan), where the plan consists in the decompo-

sition into subtasks of the intent. Then, in the implementation phase, the model can proceed to generate the code step by step, guided by the plan. The framework suffers from the problem of weak verification, since errors in the plan could directly lead to the generation of wrong results. Despite the issues, these works successfully boost the reasoning ability of the models, preparing them for more complex scenarios.

On the other hand, K. Zhang et al. design the ToolCoder [25] framework. API search tools are integrated into code generation models, replacing the role of online search engines or documentation search tools to suggest the most suitable APIs to the user. The work highlights the potential of introducing external programming tools to enhance the code generation process. We find that the idea of tool usage resembles realistic working scenarios, where programmers often feel the need to search for additional information. The adoption of a single tool could be limiting, and in fact, the same group of authors conceive later the key idea of incorporating multiple tools, leading to CodeAgent. A more recent work in 2025 proposes a framework which is named ToolCoder too [5]. It reformulates tool learning as a code generation task, by converting user queries into Python function scaffolds, which are broken into subtasks with comments. Through a rigorous plan, structured code is then generated and executed. The procedure includes informative error tracebacks as a tool to handle the errors, and successful code snippets are stored for experience reuse. However, it is not an iterative process a priori, and it is more complex with respect to ReAct.

The combination of multi-step reasoning ability with the usage of tools leads to LLM agents, which act as an assistant capable of autonomously evaluating the different aspects of a given task and managing the workflow execution [18]. In single-agent frameworks, a single model manages and uses the tools. For instance, S. Jain et al. propose a retrieval augmented generation (RAG) powered agent [9] for code search. After gathering information from the internet and analyzing the code context, the relevant part is retrieved through embedding cosine similarity and used to augment the user query. In this way, more accurate results can be identified.

A system can also employ more types of agents simultaneously. Multi-agent systems admit multiple coordinated agents, and they can be assigned different roles: if a manager agent is present, it coordinates the other ones and summarizes their results. Otherwise, in a decentralized setting, agents with specializations in distinct areas act as peers, and the handoff mechanism allows an agent to delegate tasks to another specialized agent. As an example, AgentCoder [8] is a multi-agent framework with agents specialized in distinct actions: once a piece of code is generated by the programmer agent with chain-of-thought, the test designer agent produces test cases which are run by the test executor agent, and the feedback is returned to the programmer agent for refining the code. The approach enhances the quality and the efficiency of code with its tests, while increasing the complexity.

2.3. Repository-level Code Generation

Despite the success achieved by LLMs in code generation, ambiguous or complex programming tasks being part of a broader project remain hard to solve. Actually, most studies are centered on statement-level and function-level code generations, which involve at most built-in functions and popular third-party APIs to produce standalone code blocks [26]. In real-world software development instead, the functions in a project repository generally share contextual dependencies that need to be analyzed for a better understanding of the working environment.

Different approaches are adopted to leverage repo-level context information. As Repocoder [24] states, the project code can be considered as independent snippets, and the most relevant blocks are retrieved by measuring their similarity to the requirements. The last lines of unfinished code fuel an iterative pipeline, but an excessive number of iterations could lead to undefined behaviour of the model. For a more complete inspection of the context, A^3-CodGen [15] makes distinction among local information (current working file with path of modules), global information (other code files in the same repository, to promote code reuse), and third-party-library information about the pre-installed libraries. The sophisticated framework systematically acquires information from various contexts, which are handled differently, and merges them into a cohesive prompt.

Alternatively, a novel work published in April 2025, CodeRAG [12], proposes a code-oriented agentic reasoning framework: the dependencies and the semantic relationships among the repository are represented in a DS-code graph, which is mapped to the graph of requirements. In addition to the similarities emerged from the bigraph mapping, a set of tools for web search, graph reasoning, and code testing is incorporated into the generation process through the ReAct agent strategy. This represents a logically sound innovation that could be greatly helpful for complex programming tasks.

2.4. Evaluation Benchmarks

Several benchmarks have been constructed to assess the quality of LLMs for code generation. Given its predominance in interactions with LLM-based systems, Python code is considered in our analysis. The HumanEval benchmark [3] is most commonly used for Python code generation nowadays. It contains 164 handwritten programming problems with corresponding test units. MBPP [1] consists of basic programming problems with task descriptions, code solutions and test cases as well. By demanding the implementation of a complete function based on the directives, they enable the evaluation of the functional correctness of the generated code.

However, the effectiveness of repository-level code generation highly depends on the availability of context information, so more structured and sophisticated datasets are required for testing these particular aspects. SWE-bench [11] has the objective of monitoring the resolution of GitHub issues by editing multiple files in the

repository, and the dataset includes software engineering problems with related issue descriptions and codebases. CodeRAG-Bench [21] includes repo-level coding problems from RepoEval [24] and SWE-bench [11], along with retrieval documents collected from programming-related sources. The authors of CodeAgent [26] also build a novel benchmark - CodeAgentBench - containing Python samples from real GitHub repositories, accompanied by the requirements and the documentation.

2.5. CodeAgent

Specifically, our reference paper CodeAgent [26] proposes a *single-agent* repository-level code generation framework, based on the key concept of leveraging programming tools and LLM agents to assist the problem-solving process. In this work, the tools are classified from three different perspectives:

- **Information Retrieval Tools**: the web search tool allows to retrieve useful information from the internet, and the documentation reading tool recovers relevant content from the documentation of the repository.
- **Code Implementation Tools**: a code symbol navigation tool is used for file and module-oriented parsing and for symbol search, contributing to solving intricate dependencies in the repository.
- **Code Testing Tools**: the format checker detects and rectifies format errors in the generated code for correctness and readability, while the code interpreter runs this code and provides the execution feedback to the LLM.

Four agent strategies (ReAct [23], Tool-Planning [20], OpenAIFunc [17], Rule-based) are tested on the system to guide the model and to optimize the tool usage. The framework is then assessed on *repository-level* and *function-level* benchmarks with *nine different LLMs*, and the obtained results are compared with baseline models and established commercial solutions. The authors state that CodeAgent is able to achieve superior performance than existing products in complex coding scenarios.

3 | Experiment Replication

3.1. Project Setup

We worked in the Google Colab environment for a Python implementation of the CodeAgent framework. Specifically, the key libraries are `transformers` for loading and running the model, `bitsandbytes` for the accessibility of LLMs via 4-bit quantization, `langchain` and `langchain_huggingface` for building the agentic system, `tree_sitter`, `black` and `rank_bm25` for implementing the custom agent tools. The versions of all the imported libraries are defined to avoid compatibility issues, and the random seed is fixed to a constant value for reproducibility.

3.2. Analysis of CodeAgentBench

We have accurately analyzed CodeAgentBench for a thorough understanding of the benchmark. The available data is divided into two parts:

- **Codebase:** it includes classes and contextual information from the library `numpy-ml`, representing the environment in which the agent shall operate. The data is collected from high-star GitHub repositories, about topics like machine learning, data structure, information extraction, and networking. It is characterized by large quantities of dependencies involving external libraries and internal modules, and the presence of 192 classes and 1431 functions underlines the need for additional tools for data processing. We successfully used the codebase for reconstructing the repository structure.

	path	content
0	numpy_ml/__init__.py	[# noqa\n, """Common ML and ML-adjacent algori...
1	numpy_ml/gmm/gmm.py	["""A Gaussian mixture model class"""\n, impor...
2	numpy_ml/gmm/__init__.py	[from .gmm import *\n]
3	numpy_ml/nonparametric/kernel_regression.py	[from ..utils.kernels import KernelInitializer...
4	numpy_ml/nonparametric/gp.py	[import warnings\n, import numpy as np\n, from...

Figure 3.1: Structure of Codebase - CodeAgentBench

We analyzed the distribution of file sizes in the codebase, and we noticed that is heavily left-skewed: most of files have a manageable number of lines, while some specific files are very long and impossible to be processed entirely in-context, justifying the need for targeted tools like CodeSymbolNavigationTool.

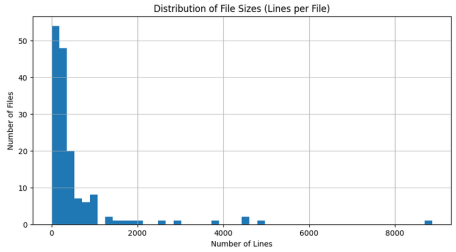

Figure 3.2: File size distribution of CodeAgentBench

- **Tasks:** each entry defines a task along with other metadata necessary to guide the generative process. The 57 samples are split into class implementation and function-level tasks: the creation of stateful objects with multiple methods can help testing software engineering skills and multi-step reasoning ability.

	title	class_annotation	comment	class_name	class_link	test_file_path
0	BallTree	numpy_ml.utils.d...	"BallTree"\n\n**...	numpy_ml.utils.d...	numpy_ml/utils/d...	numpy_ml/tests/t...
1	BatchNorm2D	numpy_ml.neural_...	"BatchNorm2D"\n\...	numpy_ml.neural_...	numpy_ml/neural_...	numpy_ml/tests/t...
2	RandomForest	numpy_ml.trees.R...	"RandomForest"\n...	numpy_ml.trees.R...	numpy_ml/trees/r...	numpy_ml/tests/t...
3	MLENGram	numpy_ml.ngram.M...	"MLENGram"\n\n**...	numpy_ml.ngram.M...	numpy_ml/ngram/n...	numpy_ml/tests/t...
4	BidirectionalLSTM	numpy_ml.neural_...	"BidirectionalLS...	numpy_ml.neural_...	numpy_ml/neural_...	numpy_ml/tests/t...

Figure 3.3: Structure of Tasks - CodeAgentBench

Despite the fact that the dataset reflects the overall characteristics of repo-level code evaluation, its structural complexity and the critical lack of actual testing files pose a serious inconvenience for putting into practice the assessment of our models.

3.3. Creation of MiniTransformers

The above-mentioned problems lead us to create a smaller-scale benchmark named MiniTrasformers, which replicates the essential structure of large repositories. We took inspiration from CodeAgentBench to realize a reproduction of the `miniformer` repository: it is a minimal but architecturally sound Python library providing core components for building and experimenting with transformers.

The generation of all data is based on the model Gemini 2.5 Pro 06-05, and it is fully guided by the statistics obtained from the analysis of CodeAgentBench. Our generator notebook produces the following set of artifacts:

- Codebase: the codebase includes the core `miniformer` library with additional sub-packages, for a total of 6 classes and 23 functions. It is engineered to be statistically representative of real-world projects: we managed to replicate the long-tail left-skewed distribution of file size in CodeAgentBench in a scaled version, maintaining the presence of large files hard to be processed entirely by the

LLM. We also reproduced the dependency structure by incorporating internal external library dependencies such as `torch` and `pytest`, and internal dependencies like `miniformer/config.py` and `miniformer/layers/attention.py`.

	path	content
0	main.py	[# Main entry point for agent tasks. Initially...
1	README.md	[# Miniformer\n, A minimal, educational librar...
2	requirements.txt	[numpy\n, torch\n, scipy\n, pydantic]
3	miniformer/__init__.py	[from .models.block import TransformerBlock]
4	miniformer/config.py	[from pydantic import BaseModel, Field\n, from...

Figure 3.4: Structure of Codebase - MiniTransformers

Lines per File	count	mean	min	25%	50%	75%	max
CodeAgentBench	156.00	542.49	2.00	113.50	265.50	512.50	8866.00
MiniTransformers	22.00	41.27	1.00	2.00	29.00	40.50	345.00

Table 3.1: Comparison of file size distribution between benchmarks

From the codebase, we can physically reconstruct the `miniformer` repository, which represents a realistic file system that our system is designed to work in. Through parsing with `tree-sitter` and creating all the directories and files, a markdown API guide is generated for function and class definitions, and the structure of the `miniformer` repository is reported:

- `layers/` containing `__init__.py`, `attention.py`, `feedforward.py`.
- `models/` containing `__init__.py`, `block.py`.
- `utils/` containing `__init__.py`, `tensor_ops.py`, `testing.py`.
- `__init__.py`.
- `activations.py`.
- `config.py`.

An example of function definition in the API guide is shown in Figure 3.5.

- Tasks: the benchmark defines 15 programming tasks, with difficulty ranging from simple file modifications to code refactoring and file creation. The prompt for each task is written in documentation-style format to simulate realistic specifications. The raw task list is processed by Python's `ast` (Abstract Syntax Tree) module, which generates metadata fields automatically, ensuring that its format matches the reference dataset CodeAgentBench.

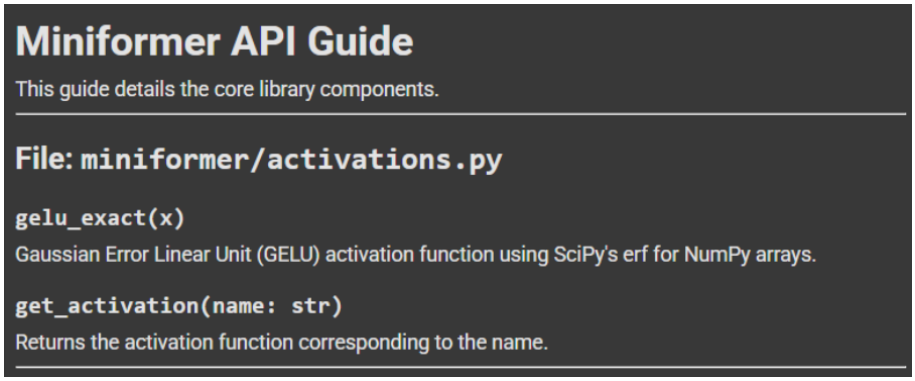

Figure 3.5: Function definition in API guide

	title	class_annotation	comment	class_name	class_link	test_file_path	task_id
0	Transformer...	miniformer....	"Add Bias C...	miniformer....	miniformer/...	tests/test_...	miniformer-01
1	to_numpy	miniformer....	"Verify Ten...	N/A	miniformer/...	tests/test_...	miniformer-02
2	Transformer...	miniformer....	"Implement ...	miniformer....	miniformer/...	tests/test_...	miniformer-03
3	get_activation	miniformer....	"Add Swish ...	N/A	miniformer/...	tests/test_...	miniformer-04
4	PositionalE...	miniformer....	"Create Pos...	miniformer....	miniformer/...	tests/test_...	miniformer-05

Figure 3.6: Structure of Tasks - MiniTransformer

- Downloadable archive: after passing the sanity check, the generated data are saved in .jsonl for its easily interpretable syntax, with each line representing a valid JSON value. Then they are packaged into a .zip file for easier distribution.

Notably, the tasks are designed to be verified by an executable test suite, enabling an automatic evaluation of the models. For each new task, the initial state of the codebase is restored to avoid overwriting conflicts with previous tasks. There are tasks asking the same LLM to output an entire code file and its corresponding tests too, resulting in a potential drop in the reliability of the solutions: in these cases, multi-agent approaches could perform better. The challenging creation of the benchmark allows us to better comprehend the working principles of code generation in concrete use cases. We shared the generator notebook and all the datasets publicly on GitHub, facilitating experiments and collaboration.

3.4. LLM Integration

Models of various sizes were considered during the development: for the initial baselines, smaller models like basic versions of Qwen3 [2] (8B, 4B, 1.7B, 0.6B) and CodeLlama-Instruct [19] were preferred. 4-bit (nf4) quantization is applied to the model to reduce the memory usage and speed up inference, and the data type bfloat16 is used for computation during inference for its balance of range and precision on compatible hardware. Hence, the model in Hugging Face Transformers

format [6] is loaded with the appropriate tokenizer, and a Hugging Face Pipeline is created for execution with the LangChain Wrapper.

Considering that the limitation of computational resources causes runtime disconnection, we switched to the use of large LLMs to more accurately simulate the paper setup and to achieve better performance with agents. Through APIs, we used as base LLMs the state-of-the-art open source model Deepseek V3 [4], and proprietary LLMs from the smaller GPT-4.1 nano [16] to GPT-4.1 mini [16] and Gemini 2.5 Flash [7]. We chose these models since they were available in the APIs' free-tier, not really expensive, and their similar dimensions allowed a more direct comparison.

3.5. Tool Implementation

Before defining the tools, we introduce a small set of safe–I/O helpers that confine every operation to the project's working directory and work indistinctly on Colab or a local file system.

Since the interaction between LLMs and the tools is based on LangChain, and to maintain coherence with the reference paper, each tool is a subclass of LangChain's `BaseTool`. For their definition we followed the practices in *"A practical guide to building agents"* [18] of OpenAI, so each tool has a concise and action-oriented *name*, a detailed and focused *description*, and a *Pydantic input schema*, for argument sanity. Moreover, all tools log explicit error messages instead of raising raw exceptions, which lets the agent chain use them as self-repair signals.

- `WebsiteSearchTool`: it queries the free DuckDuckGo API, filters risky domains, and summarizes the retrieved content, returning the final results. By default, this summarizer is one of the available LLMs (not a distinct one due to limited Colab resources), but if it is unavailable, we simply retrieve the first 60 tokens.
- `DocumentationSearchTool`: it pre-splits project docs on `--` delimiters; then it leverages BM25-Okapi to rank these chunks against the user query. When the best chunk exceeds a configurable length, the same summarizer used above produces a tightened abstract, guaranteeing a bounded response size.
- `CodeSymbolNavigationTool`: Instead of Python's built-in `ast`, the tool compiles a Tree-Sitter grammar *once* (located under `/content/grammars`) and then reuses the resulting parser for sub-millisecond symbol queries. It takes a single string argument, and if it is a class or function, it returns its definition, but if it is a file, it lists all top-level symbols contained in it.
- `FormatCheckTool`: The input string is first run through `compile()` so that only syntactically valid Python code reaches `black`. Therefore, the tool emits exactly one of two messages: *"SyntaxError:..."* or the fully re-formatted source. The tests cover Unicode, wrong whitespace, and nested edge cases.
- `CodeInterpreterTool`: The snippet is written to a temp file and executed

in a `subprocess` with a **20 s timeout**. STDOUT, STDERR and the return code are captured and returned back in a string. The temp file is removed after the timeout, protecting the host environment from clutter or malicious persistence.

Robust unit testing is performed on all the tools, with at least five test cases covering different application scenarios, such as nominal, edge-case and failure scenarios.

3.6. Agent Strategy Integration

Once the tools are defined, we constructed the agentic system through LangChain. The set of tools is provided during the invocation of the agent, and integration tests are performed for each tool, model and strategy to ensure their proper functioning. Different agent strategies are developed:

- **ReAct:** The agent iterates thought–action–observation loops and emits JSON tool calls that are parsed automatically by the core LLM. This acts as our *baseline* because it is model-agnostic and requires no special prompting.
- **Tool Calling:** For closed models, we use their function-calling formats, which follow structured prompting patterns seen during training. We did not provide a system message, since native schema inference builds the plan implicitly and includes it in a single response with tool calls, for shorter conversations.
- **Rule-Based:** the sequence of tool usage is fixed as defined in the original CodeAgent paper—WebSearch, DocReader, Code Symbol Navigator, Format Checker, and Code Interpreter.
- **Tool-Planning:** the LLM defines a complete action plan before execution and attempts to follow it, separating the planning and tool-use skills.

A fact to notice is that experiments highlighted the decisive role of tool descriptions in generating reliable code, so we carefully tuned them for prompting.

4 | Results and Conclusion

The major problems that we encountered during the executions are:

- A timer of a few seconds should be set between the execution of each task, due to the API's latency extended by the presence of context information.
- The agent could get stuck in a loop of self-doubt without accepting the result. The reason could be that the internal stop condition was not fulfilled. Larger models were observed to be less affected by this complication.
- Given the same prompt, specific agents could fail in the execution. It could be solved by turning into a few-shot learning scenario, providing an example.

4.1. Evaluation

The framework is evaluated on the MiniTransformers benchmark for repository-level testing, and on HumanEval for function-level testing. The evaluation metric Pass@1 is adopted to measure if the model generates a right solution on the first try, which has strong practical meaning in realistic coding scenarios. The models considered here are DeepSeekV3, GPT-4.1-nano, GPT-4.1-mini, and Gemini 2.5 Flash. While a specific Tool Calling agent is available for the GPT models and Gemini, DeepSeekV3 has no default support for this approach.

We first report the repository-level pass rates achieved on MiniTransformers.

AgentStrategy	DeepSeekV3	GPT-4.1-mini	GPT-4.1-nano	Gemini 2.5 Flash
NoAgent	0.333	0.333	0.333	0.533
ReAct	0.533	0.800	0.400	0.667
Tool-calling	-	0.867	0.467	0.733
Rule-based	0.600	0.667	0.400	0.667
Tool-Planning	0.467	0.733	0.333	0.600

Table 4.1: Pass@1 on MiniTransformers

We can observe that the DeepSeek model and both GPTs gain a low success rate in the NoAgent case, indicating that the models are not able to complete autonomously most of the assigned tasks. The main reasons for failure consist of missing imports and calls to inexistent functions. However, GPT-4.1-mini succeeds in achieving outstanding performance once the agent strategy is incorporated. It proves that the tools have provided useful context information to the LLM for generating better code. On the other hand, DeepSeek and GPT-4.1-nano did not realize so much improvement with the tools. GPT-4.1-nano may be limited by the small size of the

model, affecting its problem-solving capabilities. DeepSeek is not used with specific function-calling method and prompt structures: it may require a different approach or few-shot learning setting. We found the higher NoAgent success rate of Gemini 2.5 Flash insightful, and we speculated that this could be due to the fact that the benchmark has been generated by a model of the same family (Gemini 2.5 Pro 06-05). The affinity of models may have influenced its behaviour in evaluation. Overall, the effectiveness of the agent approach and the tools is demonstrated by a marginal positive trend, especially for powerful models.

Function-level testing is performed on HumanEval [3] in accordance with the paper CodeAgent. It offers a reference implementation and a ready-to-use test suite for each task, where the model is asked to complete a function given its declaration and the requirements. In this case, the documentation reading tool and the code symbol navigation tool are disabled because not meaningful in the absence of a coding context. The obtained pass rates are reported below.

AgentStrategy	DeepSeekV3	GPT-4.1-mini	GPT-4.1-nano	Gemini 2.5 Flash
NoAgent	0.781	0.805	0.701	0.793
Tool-calling	-	0.848	0.768	0.842
ReAct	0.817	0.829	0.756	0.829
Rule-based	0.823	0.835	0.762	0.829
Tool-Planning	0.811	0.823	0.744	0.817

Table 4.2: Pass@1 on HumanEval

The HumanEval benchmark, being simpler structurally, is shown to be easier to handle. In fact, most tasks can be completed without the need for additional context information. We noticed that all models show remarkable performance, with GPT-4.1-mini being the best once more. The positive trend is still observable, indicating the effectiveness of the tools and the agent strategies.

By examining both testing parts, it seems that Tool-Calling was the best-performing agent for the problems we considered, in the case of GPT models and Gemini, probably due to the fact that we are using the tool-calling agent specifically tuned for them. We observed that the results could moderately differ among the executions, due to the intrinsically non deterministic nature of LLMs. More structured tests could be run in the future to further investigate the issue.

4.2. Conclusion

In this project, we have reached all the objectives described previously. Although the reference paper did not include practical code to take into consideration, we succeeded in implementing complex tools, such as DocumentationReadingTool, in a custom manner. Besides, we properly constructed and ran the agent system without previous experience. The assessment proves the validity of the framework, and the designed MiniTransformers benchmark effectively benefits the development process.

Bibliography

[1] J. Austin, A. Odena, M. I. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. J. Cai, M. Terry, Q. V. Le, and C. Sutton. Program synthesis with large language models. *CoRR*, abs/2108.07732, 2021. URL https://arxiv.org/abs/2108.07732.

[2] J. Bai, S. Bai, Y. Chu, Z. Cui, K. Dang, X. Deng, Y. Fan, W. Ge, Y. Han, F. Huang, B. Hui, L. Ji, M. Li, J. Lin, R. Lin, D. Liu, G. Liu, C. Lu, K. Lu, J. Ma, R. Men, X. Ren, X. Ren, C. Tan, S. Tan, J. Tu, P. Wang, S. Wang, W. Wang, S. Wu, B. Xu, J. Xu, A. Yang, H. Yang, J. Yang, S. Yang, Y. Yao, B. Yu, H. Yuan, Z. Yuan, J. Zhang, X. Zhang, Y. Zhang, Z. Zhang, C. Zhou, J. Zhou, X. Zhou, and T. Zhu. Qwen technical report, 2023. URL https://arxiv.org/abs/2309.16609.

[3] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. URL https://arxiv.org/abs/2107.03374.

[4] DeepSeek. Deepseek api docs. URL https://api-docs.deepseek.com/.

[5] H. Ding, S. Tao, L. Pang, Z. Wei, J. Gao, B. Ding, H. Shen, and X. Cheng. Toolcoder: A systematic code-empowered tool learning framework for large language models, 2025. URL https://arxiv.org/abs/2502.11404.

[6] H. Face. Hugging face transformers documentation. URL https://huggingface.co/docs/transformers/en/index.

[7] Google. Google ai studio documentation. URL https://ai.google.dev/gemini-api/docs.

[8] D. Huang, J. M. Zhang, M. Luck, Q. Bu, Y. Qing, and H. Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation, 2024. URL https://arxiv.org/abs/2312.13010.

[9] S. Jain, A. Dora, K. S. Sam, and P. Singh. Llm agents improve semantic code search, 2024. URL https://arxiv.org/abs/2408.11058.

[10] X. Jiang, Y. Dong, L. Wang, Z. Fang, Q. Shang, G. Li, Z. Jin, and W. Jiao. Self-planning code generation with large language models, 2024. URL https://arxiv.org/abs/2303.06689.

[11] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

[12] J. Li, X. Shi, K. Zhang, L. Li, G. Li, Z. Tao, J. Li, F. Liu, C. Tao, and Z. Jin. Coderag: Supportive code retrieval on bigraph for real-world code generation, 2025. URL https://arxiv.org/abs/2504.10046.

[13] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries. Starcoder: may the source be with you!, 2023. URL https://arxiv.org/abs/2305.06161.

[14] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, T. Hubert, P. Choy, C. de Masson d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. Sutherland Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, Dec. 2022. ISSN 1095-9203. doi: 10.1126/science.abq1158. URL http://dx.doi.org/10.1126/science.abq1158.

[15] D. Liao, S. Pan, X. Sun, X. Ren, Q. Huang, Z. Xing, H. Jin, and Q. Li. A3-codgen: A repository-level code generation framework for code reuse with local-aware, global-aware, and third-party-library-aware, 2024. URL https://arxiv.org/abs/2312.05772.

[16] OpenAI. Openai api developer platform documentation. URL https://platform.openai.com/docs/overview.

[17] OpenAI. Function calling and other api updates, 2023. URL https://openai.com/index/function-calling-and-other-api-updates.

[18] OpenAI. A practical guide to building agents, 2025. URL

https://cdn.openai.com/business-guides-and-resources/a-practical-guide-to-building-agents.pdf.

[19] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve. Code llama: Open foundation models for code, 2024. URL https://arxiv.org/abs/2308.12950.

[20] L. Wang, W. Xu, Y. Lan, Z. Hu, Y. Lan, R. K.-W. Lee, and E.-P. Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models, 2023. URL https://arxiv.org/abs/2305.04091.

[21] Z. Z. Wang, A. Asai, X. V. Yu, F. F. Xu, Y. Xie, G. Neubig, and D. Fried. Coderag-bench: Can retrieval augment code generation?, 2025. URL https://arxiv.org/abs/2406.14497.

[22] J. Wei, X. Wang, D. Schuurmans, M. Bosma, E. H. Chi, Q. Le, and D. Zhou. Chain of thought prompting elicits reasoning in large language models. *CoRR*, abs/2201.11903, 2022. URL https://arxiv.org/abs/2201.11903.

[23] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. React: Synergizing reasoning and acting in language models, 2023. URL https://arxiv.org/abs/2210.03629.

[24] F. Zhang, B. Chen, Y. Zhang, J. Keung, J. Liu, D. Zan, Y. Mao, J.-G. Lou, and W. Chen. Repocoder: Repository-level code completion through iterative retrieval and generation, 2023. URL https://arxiv.org/abs/2303.12570.

[25] K. Zhang, H. Zhang, G. Li, J. Li, Z. Li, and Z. Jin. Toolcoder: Teach code generation models to use api search tools, 2023. URL https://arxiv.org/abs/2305.04032.

[26] K. Zhang, J. Li, G. Li, X. Shi, and Z. Jin. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges, 2024. URL https://arxiv.org/abs/2401.07339.