THEMISTO: JUPYTER-BASED RUNTIME BENCHMARK

Konstantin Grotov & Sergey Titov
JetBrains Research
{konstantin.grotov,sergey.titov}@jetbrains.com

ABSTRACT

In this work, we present a benchmark that consists of Jupyter notebooks development trajectories and allows measuring how large language models (LLMs) can leverage runtime information for predicting code output and code generation. We demonstrate that the current generation of LLMs performs poorly on these tasks and argue that there exists a significantly understudied domain in the development of code-based models, which involves incorporating the runtime context.

1 INTRODUCTION

Recent developments in code completion and generation have been significant. Over the past several years, the field has progressed from generating relatively simple programs (Chen et al., 2021) to solving real-world issues within software repositories (Jimenez et al., 2023). However, most studies in this area are based on static snapshots of code (Jiang et al., 2024), with only a small body of research exploring the potential of leveraging dynamic code properties, such as runtime information and memory state, for code generation (Chen et al., 2024). A key reason for this limitation is that common programming environments rarely allow code generation during execution, which is when runtime information can be gathered. Jupyter notebooks offer a unique opportunity in this regard—they enable code generation while providing access to runtime information and the current state of the environment.

In this paper, we present a benchmark designed to measure how models can utilize runtime and environment information, using development trajectories of Jupyter notebooks. A development trajectory is a sequence of Jupyter notebook cell executions in the order performed by a human developer. Each operation includes the cell's content and the runtime state after execution. We propose evaluating a model's ability to predict the code of the next cell to be executed and the output of a given executed cell.

We believe that by providing these benchmark and baseline results, we can advance the field of incorporating this type of information into code language models. We also make benchmark data available on Zenodo.[1]

2 BENCHMARK

The benchmark consists of a set of Jupyter notebook development trajectories. Each development trajectory consists of all prior cell executions with the given cell's execution context (*e.g.,* cell content or runtime snapshot). The order of executions was recorded by the authors of the original dataset (see Section 2.2 below) from the notebook development process and is preserved in our benchmark. Optionally, for each trajectory one can append the description of the initial task that the notebook was developed for. Using the given trajectory, we evaluate the model's ability to predict the next piece of code to be executed and the output that will be produced by the cell.

2.1 TASKS AND METRICS

For the benchmark, we have selected two tasks.

[1] Benchmark data available here: https://zenodo.org/records/14861889

Next cell prediction. In this task, we ask the model to predict the code of the next cell to be executed in our trajectory. This task offers an interesting perspective on code generation, as it requires a significant understanding of the given trajectory to determine what needs to be done next.

Cell output prediction. In this task, we ask the model to predict the output text for the cells with such output type. This task can be challenging for language models in a default snapshot setup, as it requires a strong understanding of the code and effective modeling of the runtime behavior (Gu et al., 2024). We hypothesize that providing runtime information should improve scores across the entire set of test trajectories. We suggest that this task demonstrates both the model's capabilities in code modeling and its ability to leverage runtime context.

To measure performance on this task, we use the exact match, ROUGE-L (Lin, 2004), and ChrF (Popović, 2015) metrics in line with recommendations from Evtikhiev et al. (2023).

2.2 Data

To acquire the trajectories, we used the JuNE dataset from the paper by Titov et al. (2025), where the authors tracked the notebook development process for over 8 hours with a small number of participants. They collected more than 14,000 user events, including more than 9,000 cell executions during these experiments across 29 notebooks for two original tasks. While there are datasets with more notebooks available, such as those from Kaggle (Quaranta et al., 2021), we believe that the JuNE dataset provides more information about the development process. It includes not only the environment and final version of the notebook but also intermediate and debugging steps within the notebook setting, which are the most crucial stages where models should support developers.

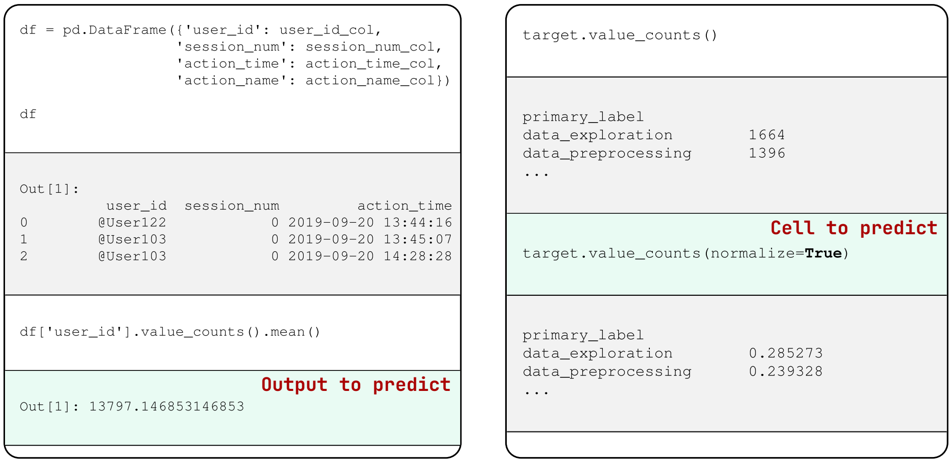
To develop our benchmark, we replicated the environment and re-executed four notebooks from the dataset, resulting in a total of 1,453 code executions. Moreover, we collected additional information, such as memory load and execution time of the cell. We also collected and serialized the state of the environment for each step to incorporate it into trajectories. A full list of available context features is given in Table 1. At the end of this process, we obtained the complete trajectory of prior development for each cell execution in the dataset. The example slice of the trajectory content is shown in Figure 1 and in Appendix A.4.

Feature	Description
`kernel_id`	Unique identifier for the execution kernel
`code`	Code executed in the cell
`output`	Output produced by the executed code
`execution_time`	Time taken to execute the code in seconds
`memory_bytes`	Memory usage during execution in bytes
`runtime_variables`	Dictionary of runtime variables in the execution environment. We store each runtime variable's name, size in bytes, and its `repr` representation.
`hash_index`	Unique hash representing the execution state

Table 1: Descriptions of trajectory step features

The next step involved selecting the cells and outputs to predict. First, we selected all cells with at least five actions in their trajectories. For each task, we filtered out empty examples and extremely long examples, specifically those beyond the 0.99 quantile of the cell length or output length distribution, respectively.

To ensure a diverse set of examples in the benchmark, we selected 200 examples using the following sampling method. First, we randomly chose 180 instances from the second and third quartiles of the output length distribution. Then, we added ten long instances from the fourth quartile and ten short instances from the first quartile. Additionally, we ensured that no sample in the benchmark to be predicted contains an exception, since the foundational models struggle with stack traces (Gehring et al., 2024). For more information on the statistics of the trajectories and the diversity of examples, please refer to Appendix A.1.



```
df = pd.DataFrame({'user_id': user_id_col,
                   'session_num': session_num_col,
                   'action_time': action_time_col,
                   'action_name': action_name_col})

df
```

```
Out[1]:
          user_id  session_num         action_time
0        @User122            0 2019-09-20 13:44:16
1        @User103            0 2019-09-20 13:45:07
2        @User103            0 2019-09-20 14:28:28
```

```
df['user_id'].value_counts().mean()
```

```
Out[1]: 13797.146853146853
```

```
target.value_counts()
```

```
primary_label
data_exploration        1664
data_preprocessing      1396
...
```

```
target.value_counts(normalize=True)
```

```
primary_label
data_exploration        0.285273
data_preprocessing      0.239328
...
```

Figure 1: A sample of code-output trajectory pairs for the output prediction task (on the left side) and next cell prediction task (on the right side). The gray and white rows represent the content of the trajectory, including the cell content and the cell output, while the green indicates the entity we aim to predict.

2.3 BASELINES

To provide an initial baseline for the benchmark, we selected a set of popular language models: GPT-4o (Hurst et al., 2024), GPT-4o-mini (Hurst et al., 2024), Claude 3.5 Sonnet (Anthropic, 2024), Gemini 1.5 Pro (Team et al., 2024), and DeepSeek-V3 (Liu et al., 2024). We report the benchmark in two settings: using runtime information during inference and without using it. Additionally, we carried out further post-processing of the model outputs: we removed cell language identifiers and trimmed all redundant spaces and tabulations. Also, we report benchmark results without additional post-processing of model outputs in Appendix A.3. Table 3 presents the results for the two tasks of our benchmark, and in Appendix A.2, you can find details about the inference setup for these models.

	Model	**Output Prediction**			**Next Cell Prediction**		
		Exact Match	RougeL	ChrF	Exact Match	RougeL	ChrF
No Runtime	GPT-4o	0.16	0.32	0.47	0.10	0.28	0.39
	GPT-4o-mini	0.16	0.31	0.43	0.06	0.25	0.38
	Claude-3.5	**0.18**	**0.38**	0.50	0.12	0.30	0.42
	Gemini Pro	0.17	0.35	**0.54**	0.12	0.34	0.43
	DeepSeek-V3	0.18	0.35	0.49	**0.13**	**0.34**	**0.46**
Runtime	GPT-4o	0.16	0.34	0.46	0.10	0.26	0.37
	GPT-4o-mini	0.15	0.30	0.43	0.07	0.27	0.36
	Claude-3.5	0.09	0.34	0.48	0.11	0.30	0.42
	Gemini Pro	0.16	**0.35**	**0.55**	0.13	0.33	0.42
	DeepSeek-V3	**0.19**	0.33	0.48	**0.14**	**0.35**	**0.47**

Table 2: Performance comparison of different foundation models on *Output Prediction* and *Next Cell Prediction* tasks. The metrics shown are Exact Match (higher is better), RougeL F1 score (higher is better), and ChrF score (higher is better).

All tested models were able to produce a significant number of exact matches for the output prediction task without leveraging the runtime information. The best results were given by Claude-3.5, with 18% of cases achieving an exact match. All other models achieved very similar results, even though the set of correctly predicted examples differs from model to model. This indicates that the task is equally challenging for different models. The models scored higher on Rouge-L and ChrF, suggesting that even in non-exact matches, the models can still produce outputs close to the original. In the setup with runtime information, the scores for the models remained very similar to those without runtime information, except for Claude-3.5, which saw its score drop by half. This indicates that the models are not yet able to effectively leverage the runtime context for this task, highlighting significant potential for further post-training improvements.

The results for next cell prediction show poor overall performance across different types of models, particularly from the perspective of exact match. The best results are produced by DeepSeek-V3, achieving an exact match in 13% of cases. Similarly to output prediction, these results are compensated by higher scores on ROUGE-L and ChrF, indicating that the models at least produce outputs relevant to the next cell prediction. Despite these numbers, we still consider the results significant—we tasked the models with predicting code for an extremely open-ended task and measured their ability to guess very specific data points, and in some cases, they were able to correctly predict next user actions.

After runtime inclusion experiments for code predictions, we found that the performance cannot be improved by simply adding all available information in the context and needs to be carefully curated. Although this information is surely valuable for accurate prediction and understanding of the current program state, the actual implementation is an interesting question for the research community.

3 Related work

In recent years, there have been multiple attempts to leverage runtime information in the training process (Ding et al., 2024; Liu et al., 2023a; Ni et al., 2023). Many approaches mainly focus on using the outputs of programs, as seen in recent work by Gehring et al. (2024); Dou et al. (2024); Liu et al. (2023b), where they demonstrated that models poorly respond to compiler feedback and suggested a reinforcement learning approach to improve code generation results in a multi-shot setup. However, other works, such as TRACED (Ding et al., 2024), show that the addition of runtime information can improve the behavior of the model to predict execution states or locate bugs.

There are two notable benchmarks that assess a model's ability to simulate code execution: CruxEval (Gu et al., 2024) and REval (Chen et al., 2024). CruxEval proposes triplets of code, input, and output, asking models to predict the input or output given the other two. They demonstrate that a chain of thought setup is more efficient, highlighting that reasoning plays a key role in modeling the execution process. REval builds on CruxEval by adding runtime data to the test set and introduces novel tasks like program state prediction and execution path prediction, in addition to output prediction. They show that reasoning capabilities vary significantly among models. For example, in execution path prediction, even the strongest tested model, GPT-4-Turbo, only achieves an accuracy of 57.7%.

4 Threat to validity and Conclusion

We believe that our benchmark can provide strong momentum toward the utilization of runtime information in code-based models. We hope that the unique environment of Jupyter notebooks can leverage newly fine-tuned models, making the notebook development process more pleasant and productive. However, the current version of the benchmark has several significant issues.

The main problem is low variability. The original data was collected from only two tasks and 20 participants, and we used only a subsample of this data. This leads to a severely underrepresented space of notebook trajectories. This limitation makes it difficult to draw conclusions about the generalizability of approaches that work with runtime context. Given the community's interest in the benchmark, one may use the tooling provided in the original JuNE dataset to gather more data for additional tasks.

With this benchmark, we introduce a new dynamic modality for code generation and program analysis, moving beyond static code base snapshots to incorporate complete development trajectories. This approach makes runtime information and development progress available to models, potentially allowing them to better align with developers' workflows and expectations. Our findings demonstrate that this is a challenging problem that remains difficult even for advanced foundation models, opening new horizons for future research in areas such as runtime-aware code completion, dynamic context understanding, and interactive development assistance.

REFERENCES

AI Anthropic. Claude 3.5 sonnet model card addendum. *Claude-3.5 Model Card*, 3:1–8, 2024.

Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. Reasoning runtime behavior of a program with llm: How far are we? *arXiv preprint cs.SE/2403.16437*, 2024.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Yangruibo Ding, Benjamin Steenhoek, Kexin Pei, Gail Kaiser, Wei Le, and Baishakhi Ray. Traced: Execution-aware pre-training for source code. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pp. 1–12, 2024.

Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Wei Shen, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, et al. Stepcoder: Improve code generation with reinforcement learning from compiler feedback. *arXiv preprint arXiv:2402.01391*, 2024.

Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. Out of the bleu: how should we assess quality of the code generation models? *Journal of Systems and Software*, 203: 111741, 2023.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. Rlef: Grounding code llms in execution feedback with reinforcement learning. *arXiv preprint arXiv:2410.02089*, 2024.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I Wang. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*, 2024.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pp. 74–81, 2004.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

Chenxiao Liu, Shuai Lu, Weizhu Chen, Daxin Jiang, Alexey Svyatkovskiy, Shengyu Fu, Neel Sundaresan, and Nan Duan. Code execution with pre-trained language models. *arXiv preprint arXiv:2305.05383*, 2023a.

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. Rltf: Reinforcement learning from unit test feedback. *Trans. Mach. Learn. Res.*, 2023, 2023b. URL `https://api.semanticscholar.org/CorpusID:259501019`.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*, pp. 26106–26128. PMLR, 2023.

Maja Popović. chrf: character n-gram f-score for automatic mt evaluation. In *Proceedings of the tenth workshop on statistical machine translation*, pp. 392–395, 2015.

Luigi Quaranta, Fabio Calefato, and Filippo Lanubile. Kgtorrent: A dataset of python jupyter notebooks from kaggle. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pp. 550–554. IEEE, 2021.

Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*, 2024.

Sergey Titov, Konstantin Grotov, Cristina Sarasua, Yaroslav Golubev, Dhivyabharathi Ramasamy, Alberto Bacchelli, Abraham Bernstein, and Timofey Bryksin. June: Jupyter notebooks executions dataset. `https://huggingface.co/datasets/JetBrains-Research/JuNE`, 2025. Dataset containing logs of code evolution in Jupyter notebooks, comprising over 100 hours of execution logs from 20 participants solving data science tasks.

A Appendix

A.1 Diversity of the Samples in Benchmark

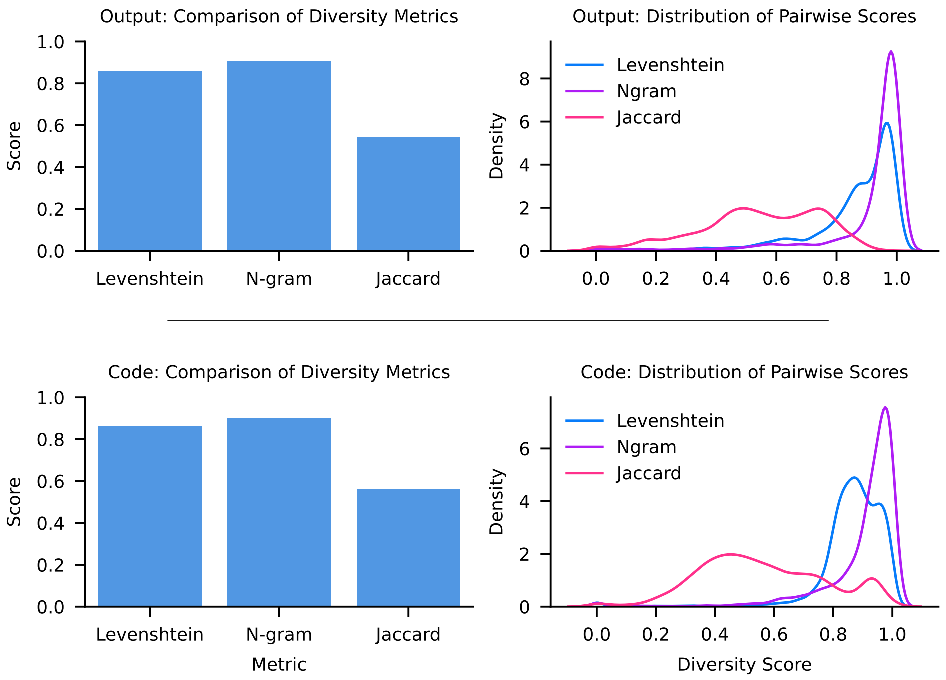


Figure 2: Diversity metrics comparison between output and code.

A.2 Inference Setup

Output Prediction

```
You are a Python REPL interpreter. Given a sequence of executed Python code cells and their outputs,
predict the output of the next executed code cell. Provide only the output, exactly as it would appear
in a Python interpreter. YOU MUST NOT include any additional tags (```python, ```, etc).
Previous code cells and their outputs:

Code: {{code}}
Output: {{output}}
...
Code: {{code}}
Output: {{output}}

Predict the output for this code: {{code_to_predict}}
```

Next Cell Prediction

```
You are an expert Python programmer. Given a sequence of executed Python code cells and their outputs,
predict what the next code cell will be. Provide only the code, exactly as it would be written in a
Jupyter notebook. YOU MUST NOT include any additional tags (```python, ```, etc).
Previous code cells and their outputs:

Code: {{code}}
Output: {{output}}
...
Code: {{code}}
Output: {{output}}

Predict the next code cell that would logically follow:
```

A.3 Baselines Without Output Processing

Model	**Output Prediction**			**Next Cell Prediction**		
	Exact Match	RougeL	ChrF	Exact Match	RougeL	ChrF
GPT-4o$^{\text{Runtime}}$	**0.10**	0.53	0.52	0	0.17	0.26
GPT-4o-mini$^{\text{Runtime}}$	0.08	0.37	0.41	0	0.16	0.23
GPT-4o	**0.10**	0.54	0.55	0	0.23	0.31
GPT-4o-mini	0.09	0.51	0.51	0	0.21	0.31
Claude-3.5	0.06	0.50	0.52	0.01	0.10	0.21
Gemini 1.5 Pro	0.05	**0.55**	**0.57**	0.02	0.27	0.32
DeepSeek-V3	0.08	0.53	0.56	**0.07**	**0.34**	**0.40**

Table 3: Performance comparison of different foundation models on *Output Prediction* and *Next Cell Prediction* tasks **without additional output processing**. The metrics shown are Exact Match (higher is better), RougeL F1 score (higher is better), and ChrF score (higher is better).

A.4 Example of Trajectory

Step 70

Code:

```
df
```

Runtime Variables:

```
action: {'size': 80, 'type': 'str', 'value': 'Action_7 (27/06...)'}
...
action_time: {'size': 68, 'type': 'str', 'value': '27/06/20 | 17:3...'}
df: {'size': 79714318, 'type': 'DataFrame', 'value': '       user_id ...'}
```

Output:

```
   user_id                                              info
0   User92  Action_3 (15/10/19 | 18:08:02) -> Action_1 (15...
1  User140  Action_3 (15/05/20 | 15:37:04) -> Action_8 (15...
2  User105  Action_4 (25/04/20 | 01:08:29) -> Action_7 (25...
...
[87192 rows x 2 columns]
```

Execution Time: 0.01 seconds
Memory Usage: 10250.90 MB

Step 71 (to predict)

Code:

```
df = df.assign(actions=df['info'].str.split('-> ')).explode('actions')
df
```

Runtime Variables:

```
action: {'size': 80, 'type': 'str', 'value': 'Action_7 (27/06...)'}
...
action_time: {'size': 68, 'type': 'str', 'value': '27/06/20 | 17:3...'}
df: {'size': 75310105, 'type': 'DataFrame', 'value': '       user_id ...'}
```

Ground Truth Output:

```
   user_id  ...                           actions
0   User92  ...   Action_3 (15/10/19 | 18:08:02)
0   User92  ...   Action_1 (15/10/19 | 18:54:49)
0   User92  ...  Action_10 (15/10/19 | 20:02:54)
...
[2053521 rows x 3 columns]
```