EXECUTION-EVAL: CAN LANGUAGE MODELS EXECUTE REAL-WORLD CODE?

Anonymous authors
Paper under double-blind review

ABSTRACT

As language models (LLMs) advance, traditional benchmarks face challenges of dataset saturation and disconnection from real-world performance, limiting our understanding of true model capabilities. We introduce EXecution-Eval (EXE), a benchmark designed to assess LLMs' ability to execute code and predict program states. EXE attempts to address key limitations in existing evaluations: difficulty scaling, task diversity, training data contamination, and cost-effective scalability. Comprising over 30,000 tasks derived from 1,000 popular Python repositories on GitHub, EXE spans a wide range of lengths and algorithmic complexities. Tasks require models to execute code, necessitating various operations including mathematical reasoning, logical inference, bit manipulation, string operations, loop execution, and maintaining multiple internal variable states during computation. Our methodology involves: (a) selecting and preprocessing GitHub repositories, (b) generating diverse inputs for functions, (c) executing code to obtain ground truth outputs, and (d) formulating tasks that require models to reason about code execution. This approach allows for continuous new task generation for as few as 1,123 tokens, significantly reducing the risk of models "training on the test set." We evaluate several state-of-the-art LLMs on EXE, revealing insights into their code comprehension and execution capabilities. Our results show that even the best-performing models struggle with complex, multi-step execution tasks, highlighting specific computational concepts that pose the greatest challenges for today's LLMs. Furthermore, we review EXE's potential for finding and predicting errors to aid in assessing a model's cybersecurity capabilities. We propose EXE as a sustainable and challenging testbed for evaluating frontier models, offering insights into their internal mechanistic advancement.

1 INTRODUCTION

Language model benchmarks are facing challenges of rapid saturation (Ott et al., 2022) and an increasing disconnect from real-world performance perceived by end-users (Zheng et al., 2023). Due to this, benchmarks are being continually created to address failure modes; e.g. SuperGLUE targeting GLUE's low problem difficulty (Wang et al., 2019), BIG-bench targeting general low eval diversity (Srivastava et al., 2022) and Auto-Arena-Hard targeting training-set contamination and data diversity in Chatbot-Arena (Li et al., 2024)(Chiang et al., 2024). These failure modes all demonstrate the challenge in linking the mechanistic improvements within language models to human understandable tasks.

Hence, to maximise an eval's utility we aim to minimise the common failure modes of; a) difficulty, not ensuring an unbound scale of small trivial problems to complex multi-step problems, b) diversity, not ensuring a representative distribution across a large space of problems, c) novelty, not ensuring continually fresh, out-out-training data samples can be generated and, d) scalability, not ensuring tasks are cost-effective to generate in the thousands and beyond.

Motivated by these challenges we introduce EXecutionEval (EXE), an evaluation replicating one of the primary tasks humans perform while coding; predicting and comparing a final program state for a given set of inputs - seen in Figure 1. EXE is designed to avoid the aforementioned failure modes; emphasising difficulty (smooth scale from trivial 1-step, one-line functions to difficult 100s-of-step, multi-layer functions), diversity (unbound number of test cases generatable for tasks from

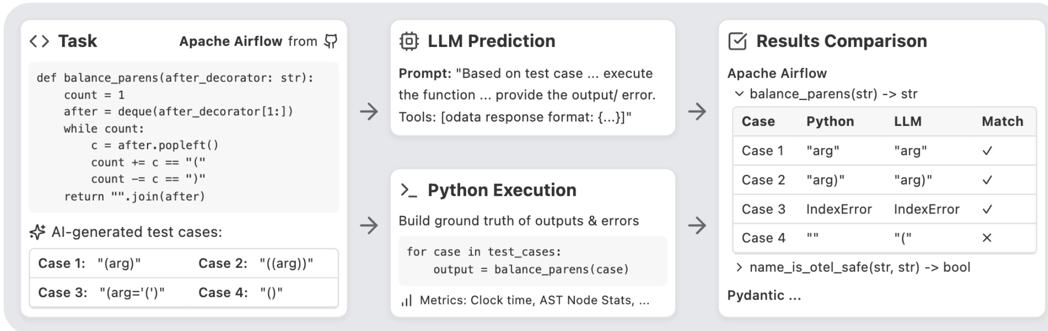

Figure 1: An example task from Apache Airflow's Github repository (code simplified to fit within diagram). EXE sources tasks from 1,000 Python repositories, generates test cases for them, and compares the LLM's ability to execute code against python's interpreter.

1,000 GitHub Repos), novelty (program inputs can be continually generated) and scalability (initial release containing 30,000+ problems at a cost of \$11).

EXE also holds theoretical inspiration. (Fowler et al., 2022) et al have replicated positive pedagogical correlations found by (Lopez et al., 2008) between the abilities of CS1 students to "trace" programs (i.e. manually predict outputs and write the internal state out line by line) and their abilities to pass code writing and explanation exams. This is mirrored in CRUX-Eval's (Gu et al., 2024) findings, where they observe a moderate correlation between a model's ability to execute a block of code and a model's HumanEval (Chen et al., 2021) code writing Pass@1 rate.

2 EVALUATION FRAMEWORK

As seen in Figure 1, an EXE task is to predict a function's return value or error from: a) a code snippet and b) a set of input arguments. Code snippets are extracted from PyPi's most popular 1,000 python projects hosted on GitHub, we select our snippets to be pure (i.e. deterministic, no side effects), language model generatable (i.e. arg types of `ints, lists, ...`) and to only require builtins (local imports and external libraries are inlined for the snippet). To realise this we follow the following three stage pipeline:

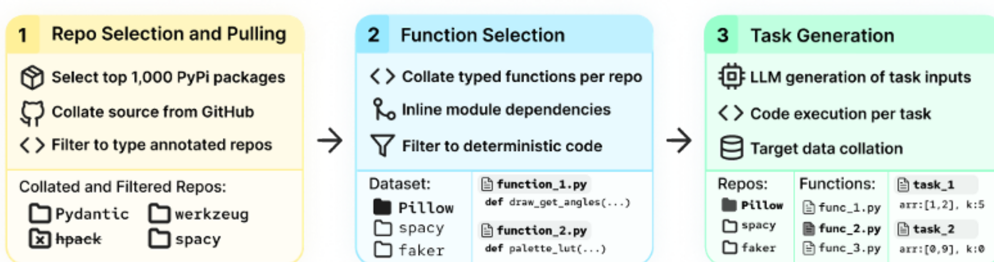

Figure 2: Three stage EXE task generation pipeline. Detailed example tasks and generated inputs can be found in Appendix A.1.

1. Repo Selection and Code Scraping. We first select the top 1,000 most popular pypi packages and collate the corresponding github repos where possible, similar to (Jimenez et al., 2023). These repos are then pulled down locally and filtered based on a static ast analysis determining which repositories contain type annotated code.

2. Function Selection and Dependency Collation. We perform a static ast analysis to filter to functions with LLM generatable argument and return type annotations. Further ast analysis then recursively identifies dependent elements (modules, functions, classes, variables, ...) across files, builds a dependency graph, and inlines them into a base task. Finally, base tasks containing side effects or non-deterministic code such as environment variables, process calls, randomness or network requests are filtered out. See Appendix A.2 for detail on acceptable type annotations and filtering.

3. Test Case Generation. Using the argument type annotations we construct a LLM function calling schema that generates a diverse set of inputs. The base task code is then executed with each generated input and the result with runtime statistics are logged. This forms the test case (base task code + generated input), output (returned result or error from executed code) and statistics (runtime statistics + static ast analysis statistics).

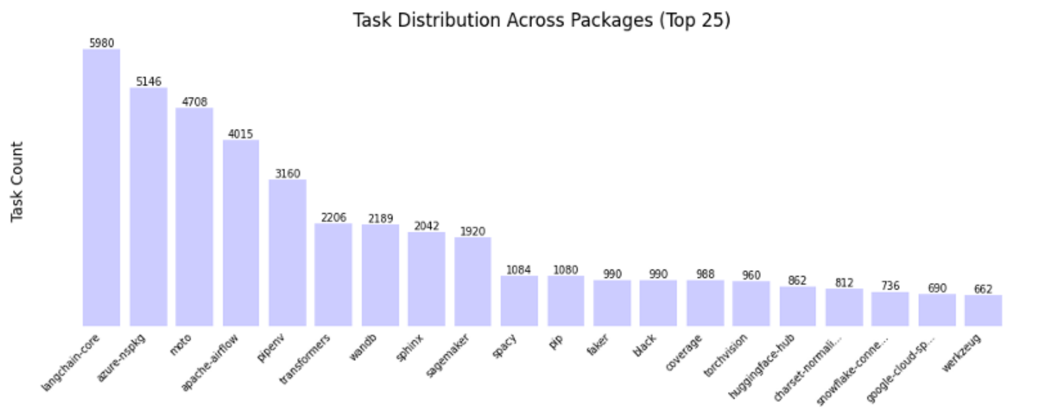

Figure 3: We observe task counts per repository to have a near logarithmic falloff. Note: manual removal of several bad offender repositories was required as they contained thousands of nearly identical functions with only url changes.

Through these stages of filtering, the original top 1,000 repositories are filtered down to the 33,950 task instances which comprise EXE. A high level breakdown of these task instances across repositories is presented in Figure 3. We note some repositories are overrepresented primarily due to being more modern (using typing) and the style of code (shorter deterministic pieces).

2.1 Task Formation

Model input. The model is given a complete snippet of code alongside the input state to be executed. The model is then tasked to predict the resulting return value, or in the case that an exception is raised the model is instructed to generate an exception type and value. In practice, we prompt models with an odata json representation and use a parser to ensure valid generations. We do append one additional user reply with the parsing error if the model's response fails to parse. Examples of input instances can be found in Appendix A.1.

Evaluation metrics. To evaluate a proposed solution, we use the pass@k metric (Chen et al., 2021), comparing the ground truth and the generated prediction as json objects (`set` and `frozenset` are sorted before conversion to json lists). If the original code produced an exception, we compare the type and message (excluding stacktrace) using a language model comparison. Examples of generated outputs can be seen in Appendix A.1.

2.2 Features of EXE

Diversity of inputs and outputs. Unlike many benchmarks focused on a particular subject matter area, a task in this eval may require a model to perform mathematical reasoning, logical inference, bit manipulation, string operations, loop execution, or to maintain multiple internal variables during computation. Furthermore, these may only form part of an algorithm that the model has to execute. Our random human inspection has uncovered algorithmic time complexities spanning from $O(1)$ to $O(x^n)$ and structured analysis has found tasks with code context lengths ranging from 440 to 311,000 tokens. Ensuring this broad diversity reduces the risk of hitting a local maxima and increases our opportunity to measure internal capabilities across a range of difficulties.

Continually updatable. Both our code collection and task input generation processes can create new tasks with minimal human oversight. Simply re-running our code collection to pull the latest commits or directing it towards an uncollected Python GitHub repository will create new task instances. Furthermore we can continue to generate new test cases for existing tasks, our test case generator automatically avoids generating seen inputs. Hence, EXE can be extended continually with new task instances, ensuring answers were not included in training corpuses of models for evaluation.

Cost effective scalability. With generation of new tasks requiring an average of 1,112 input tokens (batch of 15) and evaluation of tasks typically requiring 1,123 tokens, ExecEval can be generated, tested and continually updated at a fraction of the cost of human-curated benchmarks. Our initial dataset of 33,950 cases has only incurred an approximate costing of \$11 to produce and \$95 to test on.

Long multi-step problems with smooth difficulty scaling. We provide a continuous spectrum of task difficulties, ranging from 1-step, one-line functions to multi-file, multi-class, multi-100-step tasks. Our most complex tasks include function call depths (non-recursive) of up to 13 levels (median: 2), separate identifier counts (i.e. variable names, function names, ...) of up to 823 (median: 16) and up to 63 if statements (median: 1). This smooth scaling of difficulty allows for a more detailed measurement of model coherence along multi-step problems than what is typically seen in traditional evals. However, as language models continue to advance rapidly, even this wide range of difficulties may eventually face saturation.

To address this, we observe a mechanism inspired by the SKILL-MIX evaluation (Yu et al., 2023) that leverages the typed nature of our function selection process. This approach allows us to create even more complex tasks by chaining functions where the output type of one matches the input type of another, or by combining multiple outputs into a composite input. The number of potential new tasks can be upper bounded by $n^2 \cdot (T_{\text{max}})^k \cdot C,$, where n is the total number of types, $T_{\text{max}} = \max_{i,j} T_{i,j}$ is the maximum number of existing tasks between any two types, k is the number of functions to chain, and C is the average number of test cases per task. While this is an upper bound and the actual number of valid composite tasks would be lower due to specific type compatibility constraints, it still represents a significant expansion of our task space. We view this as an opportunity to trade some of the 'realism' of using 100% real-world code for the ability to probe the upper bounds of model capabilities. For constant compute models, this approach allows us to test their internal mechanistic capabilities in handling increasingly complex, multi-step problems. And for chain-of-thought models, it provides a test of increasingly long-term agentic coherency.

Error prediction. To test the full spectrum of code execution we further generate test cases designed to trigger exceptions. Many of these require in-depth analysis to see ahead of time, for example predicting an invalid array index through multiple functions. While debugging exceptions is one of the more challenging software engineering tasks, we are yet to see it commonly evaluated in benchmarks.

3 RESULTS

We report our evaluation results across different SOTA models alongside our findings across different task statistics below.

Table 1: EXE Pass@1 results

Model	EXE dataset (Pass@1)	Errors (Pass@1)
gpt4o	72.4	49.5
gpt4o-mini	60.9	32.0

LLMs can execute real-world code, achieving results in-line with code generation benchmarks. We find EXE shows similar relative model performance between models as seen in coding benchmarks such as HumanEval (Chen et al., 2021) and as seen in benchmarks requiring logical inference such as (Lu et al., 2023). Furthermore we find a similar diversity of performance across packages as seen in agentic benchmarks such as (Jimenez et al., 2023). We show our findings in Figure 4.

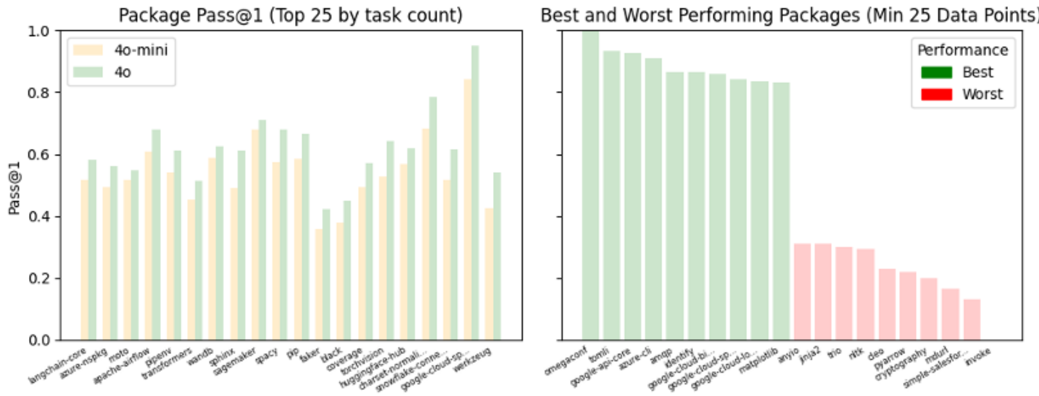

Figure 4: Left - We show the relative accuracy of different models across the top 20 packages by task count. Both the relative differences between models and the relative differences between packages are within expectations from other coding benchmarks (Jimenez et al., 2023). Right - We show the magnitude of diversity across packages (mean performance across all models).

Prior works such as Learning To Execute (Zaremba & Sutskever, 2014) and CRUX-Eval (Gu et al., 2024) have placed justifiable limitations on code complexity; removing mathematical operations, limiting line count, disallowing custom classes and only having one singular function to name a few. We hypothesised that these are no longer necessary, and to understand the true internal capabilities of a constant compute model (i.e. no Chain of Thought) we must test on real-world code, only applying limitations where forced (i.e. no arbitrary object inputs, as LLMs can't generate them). Our results as seen in table 1 provide initial evidence towards our hypothesis.

ExecEval provides a smooth curve of task difficulties. We set out to ensure a) our eval does not induce saturation from a bounded distribution of task difficulties, b) our eval does not induce an "AI overhang" by not having a smooth transition between difficulties and, c) the correlated factors affecting difficulty are human interpretable.

As shown in Figure 5 several task statistics such as "lines of code", "processing time" and "number of function calls" all correlate log-linearly with a model's achieved pass@1 score. These correlations provide preliminary evidence towards c) as they align with simplistic human intuition, i.e. more lines of code, more compute cycles, higher difficulty. Furthermore, we view the log-linear relationships as evidence towards b), i.e. EXE provides a smooth transition between difficulties. And finally, we view the relationships as a demonstration of difficulty being affected by factors within our control, i.e. number of function calls - providing empirical evidence towards a).

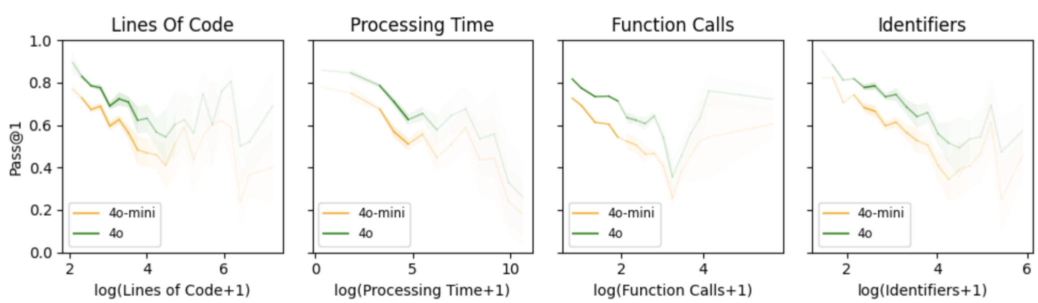

Figure 5: Pass@1 for all tasks across four of our code metrics. The shaded area represents variance, and the opacity is scaled with count of samples. Processing time is measured in microseconds.

Stylistic coding patterns shape the metrics. As can be seen in Figure 5 the pass@1 rate of function calls hits an elbow and then surprisingly improves as the call count increases. During our investigation we found several of these occurrences, and not only with call count. These were found to be largely driven by specific coding patterns and complex tasks that LLMs excel at. We show in Figure 6 below three example tasks, and more specifically coding patterns driving this anomaly.

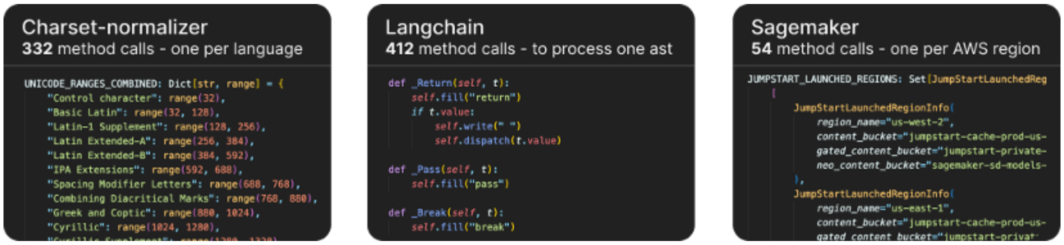

Figure 6: Three examples of high pass@1 rate tasks that contain large amounts of function calls. Left - Charset-normaliser performs 300+ function calls to define ranges of unicode characters upon initialisation; this constant has little effect on task difficulty but is used frequently and hence appears in many tasks. Middle - Langchain's Unparser class traverses an AST and regenerates source code. The calling method in our dataset is "add_last_line_print(str) → str" which takes in code, parses it and then uses Unparse(...) to unparse it; this is a prime example of a "directly predictable task", i.e. one not requiring line by line code execution to predict a result. Right - Similar to Charset-normaliser, AWS's Sagemaker has a module level constant with 10s of calls; not creating a large impact on task difficulty but frequent in its use.

LLMs struggle with certain coding features. As EXE contains a diverse set of tasks, we are able to observe model performance differing greatly based on coding features used in any task. To illustrate: floating point math operations such as multiplications (gpt4o: 43 mean Pass@1) significantly increase task difficulty, however bit manipulation and boolean operations only showed a minor negative impact. Iterative operations such as compound assignment operations i.e. "i += 1" (56 Pass@1), list slicing (65 Pass@1) and list comprehensions (68 Pass@1) all increased difficulty, however for loops on (73 Pass@1) on average did not have a significant impact.

With the above metrics, and those seen in Figure 6, their mean Pass@k decreases as their count increases. To reduce the risk of our metrics being a proxy for longer problems we show the effects can still be seen below in Figure 7 after normalisation by lines of code (only lines with executable syntax tokens are counted).

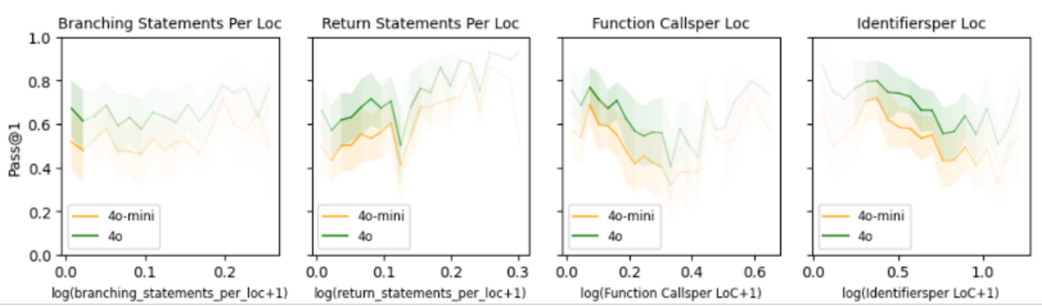

Figure 7: Pass@1for all tasks across four of our code metrics normalised by line of code count. All four of the above metrics previously showed a negative impact as they increased, interestingly we now observe branching statements having little to no impact and return statements surprisingly driving an increase in Pass@1 score. Our strong negative factors i.e. function calls and identifiers created, still are seen increasing task difficulty as they take up ever greater percentages of the task.

4 Related Work

There is a rich history of work on evaluating language models' abilities in reasoning, execution, and multi-step problem-solving across various domains. These efforts span from natural language processing to mathematical reasoning, and from code generation to program execution. Our work, EXecution-Eval (EXE), builds upon this foundation while addressing key challenges in benchmark design and evaluation.

Code generation benchmarks have been the foundation of evaluating the coding abilities of language models. Works like HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) established standardized datasets for assessing code synthesis from natural language descriptions. These efforts have expanded to cover multiple programming languages (Cassano et al., 2022; Khan et al., 2023) and more complex domains such as algorithmic problem solving (Huang et al., 2023). While these benchmarks focus primarily on the task of code generation, we believe additional focus on the tasks of code execution and error prediction have been overlooked and may offer additional insight into the internal capabilities of frontier models.

The concept of ”learning to execute” itself has a long history, Zaremba & Sutskever (2014) explored neural networks' ability to learn and execute simple programs. Graves et al. (2014) constructed the first Neural Turing Machines with (Kaiser & Sutskever, 2015; Reed & de Freitas, 2015; Dehghani et al., 2018) all building further into this domain. This line of research has evolved, with recent works like Bieber et al. (2020); Nye et al. (2021) and Gu et al. (2024) applying graph and language models to execute synthetic or simplistic Python programs. EXE builds upon these foundations by evaluating execution capabilities on complex, messy, real-world code from diverse GitHub repositories, providing a more challenging, scaleable and realistic test bed.

Recent trends in benchmark design have emphasized the importance of diverse, multi-step problems and agentic capabilities. Works like Jimenez et al. (2023) have introduced benchmarks that require solving real world software engineering problems while Zhou et al. (2023) has enabled evaluation of complex instruction following and performing multi-step reasoning. In the mathematical domain, benchmarks like those by Hendrycks et al. (2021) and Lu et al. (2023) have pushed models to solve intricate, multi-step problems.

The challenge of benchmark saturation and the need for continually updated evaluations has been recognized in recent works (Ott et al., 2022). Live benchmarks such as those proposed by Li et al. (2024), (Chiang et al., 2024) and Kiela et al. (2021) aim to address this issue. Skill-Mix (Yu et al., 2023) takes a novel approach, combining separate skills required to solve a problem they are able to increase task difficulty non-linearly with k skills. EXE has been inspired by both these concepts, hence the focus on enabling continual generation of new coding tasks and test cases, as well as the potential extension into chaining functions.

While many existing benchmarks use curated or synthetic datasets, EXE leverages real-world code from popular Python repositories. This approach is inspired by works like CodeNet (Puri et al., 2021) and The Stack (Kocetkov et al., 2022) which demonstrated the value of diverse, real-world data in training and evaluating language models.

5 Extensions

Expanding the scope and diversity We believe scaling EXE to include more repositories by as much as 100x would significantly reduce the noise seen in our coding metrics and provide a more resilient baseline for future frontier models. By incorporating additional Python functions — potentially using language models to predict missing type annotations — and including a diversity of other programming languages such as C++, Go and JavaScript, we believe there is even further opportunity to scale. This would offer further insights into the generalisability of a model's code understanding, pose new challenges for analysis such as pointers, macros and type-free codebases.

Probing code execution mechanisms with simple functions We believe there is an opportunity to align code execution with mechanistic interpretability, to gain an understanding of how constant compute language models can execute complex multi-step instructions. To illustrate, if we select the simplest function that a language model can not directly predict the outcome of, a hash function for example (one that doesn't use floating point math in this case), one requiring compute at each iteration. This would force the network to perform the computation step by step, and for a constant compute feed-forward network, layer by layer. Hence, performing a single iteration that may not lead to anything interesting, however as we increase the iteration count one by one, the model now must find a repeated circuit to perform the same computation in the later layers. For every increase it must find another circuit or a more optimal way of performing its work until it fails. We believe this would present an interesting approach alongside standard mechanistic interpretability techniques for circuit discovery and understanding of control flow, variable tracking and computational logic at the mechanistic level.

Breakpoint analysis for validating code execution granularly Rather than evaluating the final return value, including multiple evaluation points within code execution may assist verification of if models are performing the step-by-step computations to reach a return value. Furthermore by inserting ’breakpoints’ throughout the execution process, we can transform a single return state prediction task into numerous intermediate state prediction tasks. To illustrate, given a code snippet with a breakpoint at a specific line, a model would be tasked to determine the values of the local variables when the breakpoint is triggered. This mirrors common human debugging practices and may reveal discrepancies between final output accuracy and intermediate state understanding, offering further resistance against tasks where their final outcome can be directly predicted.

Connection to cybersecurity threat model. Software vulnerability research techniques are largely [1] enabled by the ability to predict and reason about expected program outcomes. For example, code injection, path resolution and memory buffer attacks are often found through manual human analysis; tracing inputs through the control flow, predicting output states and reasoning if there are opportunities to exploit. As EXE contains parsers such as seen in Appendix A.1 we see an opportunity to select a subset of EXE where prediction of error would imply language models have the internal capability to comprehend and aid humans with crafting vulnerabilities.

6 Conclusions

In this paper, we introduced EXecution-Eval (EXE), a benchmark designed to evaluate whether language models can execute real-world code. By collecting over 30,000 tasks from 1,000 popular Python repositories, EXE presents a diverse range of problems requiring computational operations such as mathematical reasoning, logical inference, and state maintenance. Our evaluations suggest that while language models demonstrate some capability in executing code, they often struggle with complex, multi-step tasks—particularly those involving many identifiers, function calls and iterative operations. Our findings indicate that although current models have limitations in accurately reasoning about and executing real-world code, they perform surprisingly well on average, prompting several opportunities extending this investigation.

EXE aims to address limitations of existing benchmarks by providing a scalable, diverse, and continually updatable framework. Its design targets a smooth difficulty scale and easy generation of new tasks with minimal human oversight with the goal to reduce the risk of models ”training on the test set.”

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. August 2021.

David Bieber, Charles Sutton, Hugo Larochelle, and Daniel Tarlow. Learning to execute programs with instruction pointer attention graph neural networks. October 2020.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. MultiPL-E: A scalable and extensible approach to benchmarking neural code generation. August 2022.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian,

[1] Some techniques such as random fuzzing may not rely on any internal program knowledge. However, to find actionable results within realistic computational bounds, fuzzers are often augmented based on this knowledge to limit their generatable space.

Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. July 2021.

Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Hao Zhang, Banghua Zhu, Michael Jordan, Joseph E Gonzalez, and Ion Stoica. Chatbot arena: An open platform for evaluating LLMs by human preference. March 2024.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. July 2018.

Max Fowler, David IV, Mohammed Hassan, Seth Poulsen, Matthew West, and Craig Zilles. Reevaluating the relationship between explaining, tracing, and writing skills in cs1 in a replication study. *Computer Science Education*, 32:1–29, 06 2022. doi: 10.1080/08993408.2022.2079866.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. October 2014.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I Wang. CRUXEval: A benchmark for code reasoning, understanding and execution. January 2024.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the MATH dataset. March 2021.

Yiming Huang, Zhenghao Lin, Xiao Liu, Yeyun Gong, Shuai Lu, Fangyu Lei, Yaobo Liang, Yelong Shen, Chen Lin, Nan Duan, and Weizhu Chen. Competition-Level problems are effective LLM evaluators. December 2023.

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? October 2023.

Łukasz Kaiser and Ilya Sutskever. Neural GPUs learn algorithms. November 2015.

Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. XCodeEval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. March 2023.

Douwe Kiela, Max Bartolo, Yixin Nie, Divyansh Kaushik, Atticus Geiger, Zhengxuan Wu, Bertie Vidgen, Grusha Prasad, Amanpreet Singh, Pratik Ringshia, Zhiyi Ma, Tristan Thrush, Sebastian Riedel, Zeerak Waseem, Pontus Stenetorp, Robin Jia, Mohit Bansal, Christopher Potts, and Adina Williams. Dynabench: Rethinking benchmarking in NLP. April 2021.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. The stack: 3 TB of permissively licensed source code. November 2022.

Tianle Li, Wei-Lin Chiang, Evan Frick, Lisa Dunlap, Tianhao Wu, Banghua Zhu, Joseph E Gonzalez, and Ion Stoica. From crowdsourced data to high-quality benchmarks: Arena-Hard and BenchBuilder pipeline. June 2024.

Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the Fourth International Workshop on Computing Education Research*, ICER '08, pp. 101–112, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605582160. doi: 10.1145/1404520.1404531. URL `https://doi.org/10.1145/1404520.1404531`.

Pan Lu, Hritik Bansal, Tony Xia, Jiacheng Liu, Chunyuan Li, Hannaneh Hajishirzi, Hao Cheng, Kai-Wei Chang, Michel Galley, and Jianfeng Gao. MathVista: Evaluating mathematical reasoning of foundation models in visual contexts. October 2023.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. Show your work: Scratchpads for intermediate computation with language models. November 2021.

Simon Ott, Adriano Barbosa-Silva, Kathrin Blagec, Jan Brauner, and Matthias Samwald. Mapping global dynamics of benchmark creation and saturation in artificial intelligence. March 2022.

Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. CodeNet: A large-scale AI for code dataset for learning a diversity of coding tasks. May 2021.

Scott Reed and Nando de Freitas. Neural Programmer-Interpreters. November 2015.

Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, Agnieszka Kluska, Aitor Lewkowycz, Akshat Agarwal, Alethea Power, Alex Ray, Alex Warstadt, Alexander W Kocurek, Ali Safaya, Ali Tazarv, Alice Xiang, Alicia Parrish, Allen Nie, Aman Hussain, Amanda Askell, Amanda Dsouza, Ambrose Slone, Ameet Rahane, Anantharaman S Iyer, Anders Andreassen, Andrea Madotto, Andrea Santilli, Andreas Stuhlmüller, Andrew Dai, Andrew La, Andrew Lampinen, Andy Zou, Angela Jiang, Angelica Chen, Anh Vuong, Animesh Gupta, Anna Gottardi, Antonio Norelli, Anu Venkatesh, Arash Gholamidavoodi, Arfa Tabassum, Arul Menezes, Arun Kirubarajan, Asher Mullokandov, Ashish Sabharwal, Austin Herrick, Avia Efrat, Aykut Erdem, Ayla Karakaş, B Ryan Roberts, Bao Sheng Loe, Barret Zoph, Bartłomiej Bojanowski, Batuhan Özyurt, Behnam Hedayatnia, Behnam Neyshabur, Benjamin Inden, Benno Stein, Berk Ekmekci, Bill Yuchen Lin, Blake Howald, Bryan Orinion, Cameron Diao, Cameron Dour, Catherine Stinson, Cedrick Argueta, and Ramírez. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. June 2022.

Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. SuperGLUE: A stickier benchmark for general-purpose language understanding systems. May 2019.

Dingli Yu, Simran Kaur, Arushi Gupta, Jonah Brown-Cohen, Anirudh Goyal, and Sanjeev Arora. Skill-Mix: a flexible and expandable family of evaluations for AI models. October 2023.

Wojciech Zaremba and Ilya Sutskever. Learning to execute. October 2014.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P Xing, Hao Zhang, Joseph E Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-bench and chatbot arena. June 2023.

Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. Instruction-following evaluation for large language models. November 2023.

A Appendix

You may include other additional sections here.

A.1 Example Input & Output

Below is an example from the eval set. It is split into three components:

1. Code Task. The function `split_email` was found to pass the type requirements, and as such all modules, classes, functions and attributes required to execute it have been recursively inlined.

2. Test Case Inputs. Based on the type definition (used for setting the function calling schema) inputs/ output pairs have been generated with the goal of maximising diversity of control flow paths within the function.

3. Outputs. Based on the type definition (used for setting the function calling schema) inputs/ output pairs have been generated with the goal of maximising diversity of control flow paths within the function.

Code

Note: The top 1,000 PyPI repos are used to form EXE, this function is from email-validator, rank 345

```
from typing import Optional, Tuple
import re
import unicodedata


class EmailNotValidError(ValueError):
      """Parent class of all exceptions raised by this module."""
      pass


class EmailSyntaxError(EmailNotValidError):
      """Exception raised when an email address fails validation
          because of its form."""
      pass


ATEXT = r'a-zA-Z0-9_!#\$%&\'\*\+\-/=\?\^`\{\|\}~'


def safe_character_display(c: str) -> str:
      # Return safely displayable characters in quotes.
      if c == '\\':
         return f"\"{c}\""  # can't use repr because it escapes it
      if unicodedata.category(c)[0] in ("L", "N", "P", "S"):
         return repr(c)


      # Construct a hex string in case the unicode name doesn't exist.
      if ord(c) < 0xFFFF:
         h = f"U+{ord(c):04x}".upper()
      else:
         h = f"U+{ord(c):08x}".upper()


      # Return the character name or, if it has no name, the hex
          string.
      return unicodedata.name(c, h)


ATEXT_RE = re.compile('[.' + ATEXT + ']')  # ATEXT plus dots


def check_unsafe_chars(s: str, allow_space: bool = False) -> None:
      # Check for unsafe characters or characters that would make the
          string
      # invalid or non-sensible Unicode.
      bad_chars = set()
      for i, c in enumerate(s):
         category = unicodedata.category(c)
         if category[0] in ("L", "N", "P", "S"):
            # Letters, numbers, punctuation, and symbols are permitted
                .
```

```
                pass
            elif category[0] == "M":
                # Combining character in first position would combine with something
                # outside of the email address if concatenated, so they are not safe.
                # We also check if this occurs after the @-sign, which would not be
                # sensible because it would modify the @-sign.
                if i == 0:
                        bad_chars.add(c)
            elif category == "Zs":
                # Spaces outside of the ASCII range are not specifically disallowed in
                # internationalized addresses as far as I can tell, but they violate
                # the spirit of the non-internationalized specification that email
                # addresses do not contain ASCII spaces when not quoted. Excluding
                # ASCII spaces when not quoted is handled directly by the atom regex.
                #
                # In quoted-string local parts, spaces are explicitly permitted, and
                # the ASCII space has category Zs, so we must allow it here, and we'll
                # allow all Unicode spaces to be consistent.
                if not allow_space:
                        bad_chars.add(c)
            elif category[0] == "Z":
                # The two line and paragraph separator characters (in categories Zl and Zp)
                # are not specifically disallowed in internationalized addresses
                # as far as I can tell, but they violate the spirit of the non-internationalized
                # specification that email addresses do not contain line breaks when not quoted.
                bad_chars.add(c)
            elif category[0] == "C":
                # Control, format, surrogate, private use, and unassigned code points (C)
                # are all unsafe in various ways. Control and format characters can affect
                # text rendering if the email address is concatenated with other text.
                # Bidirectional format characters are unsafe, even if used properly, because
                # they cause an email address to render as a different email address.
                # Private use characters do not make sense for publicly deliverable
                # email addresses.
                bad_chars.add(c)
            else:
                # All categories should be handled above, but in case there is something new
                # to the Unicode specification in the future, reject all other categories.
                bad_chars.add(c)
        if bad_chars:
            raise EmailSyntaxError("The email address contains unsafe characters: "
```

```
                                    + ", ".join(safe_character_display(c) for c in sorted(bad_chars)) + ".")


def split_email(email: str) -> Tuple[Optional[str], str, str, bool]:
    # Return the display name, unescaped local part, and domain part
    # of the address, and whether the local part was quoted. If no
    # display name was present and angle brackets do not surround
    # the address, display name will be None; otherwise, it will be
    # set to the display name or the empty string if there were
    # angle brackets but no display name.


    # Typical email addresses have a single @-sign and no quote
    # characters, but the awkward "quoted string" local part form
    # (RFC 5321 4.1.2) allows @-signs and escaped quotes to appear
    # in the local part if the local part is quoted.


    # A `display name <addr>` format is also present in MIME messages
    # (RFC 5322 3.4) and this format is also often recognized in
    # mail UIs. It's not allowed in SMTP commands or in typical web
    # login forms, but parsing it has been requested, so it's done
    # here as a convenience. It's implemented in the spirit but not
    # the letter of RFC 5322 3.4 because MIME messages allow newlines
    # and comments as a part of the CFWS rule, but this is typically not
    # allowed in mail UIs (although comment syntax was requested once too).
    #
    # Display names are either basic characters (the same basic characters
    # permitted in email addresses, but periods are not allowed and spaces
    # are allowed; see RFC 5322 Appendix A.1.2), or or a quoted string with
    # the same rules as a quoted local part. (Multiple quoted strings might
    # be allowed? Unclear.) Optional space (RFC 5322 3.4 CFWS) and then the
    # email address follows in angle brackets.
    #
    # We assume the input string is already stripped of leading and trailing CFWS.


    def split_string_at_unquoted_special(text: str, specials: Tuple[str, ...]) -> Tuple[str, str]:
        # Split the string at the first character in specials (an @-sign
        # or left angle bracket) that does not occur within quotes and
        # is not followed by a Unicode combining character.
        # If no special character is found, raise an error.
        inside_quote, escaped, left_part = False, False, ""
        for i, c in enumerate(text):
            # < plus U+0338 (Combining Long Solidus Overlay) normalizes to
            #  U+226E (Not Less-Than), and  it would be confusing to treat
            # the < as the start of "<email>" syntax in that case. Likewise,
```

```
        # if anything combines with an @ or ", we should probably not
        # treat it as a special character.
        if unicodedata.normalize("NFC", text[i:])[0] != c:
            left_part += c


        elif inside_quote:
            left_part += c
            if c == '\\' and not escaped:
                escaped = True
            elif c == '"' and not escaped:
                # The only way to exit the quote is an unescaped quote.
                inside_quote = False
                escaped = False
            else:
                escaped = False
        elif c == '"':
            left_part += c
            inside_quote = True
        elif c in specials:
            # When unquoted, stop before a special character.
            break
        else:
            left_part += c


    if len(left_part) == len(text):
        raise EmailSyntaxError("An email address must have an @-sign.")


    right_part = text[len(left_part):] # The right part is whatever is left.


    return left_part, right_part


def unquote_quoted_string(text: str) -> Tuple[str, bool]:
    # Remove surrounding quotes and unescape escaped backslashes
    # and quotes. Escapes are parsed liberally. I think only backslashes
    # and quotes can be escaped but we'll allow anything to be.
    quoted, escaped, value = False, False, ""
    for i, c in enumerate(text):
        if quoted:
            if escaped:
                value += c
                escaped = False
            elif c == '\\':
                escaped = True
            elif c == '"':
                if i != len(text) - 1:
                    raise EmailSyntaxError("Extra character(s) found after close quote: "
                                           + ", ".join(safe_character_display(c) for c in text[i + 1:]))
                break
            else:
                value += c
        elif i == 0 and c == '"':
```

```
                    quoted = True
                else:
                    value += c

        return value, quoted

    # Split the string at the first unquoted @-sign or left angle bracket.
    left_part, right_part = split_string_at_unquoted_special(email, ("@", "<"))

    # If the right part starts with an angle bracket, then the left part
    # is a display name and the rest of the right part up to the
    # final right angle bracket is the email address, .
    if right_part.startswith("<"):
        # Remove space between the display name and angle bracket.
        left_part = left_part.rstrip()

        # Unquote and unescape the display name.
        display_name, display_name_quoted = unquote_quoted_string(left_part)

        # Check that only basic characters are present in a non-quoted display name.
        if not display_name_quoted:
            bad_chars = {
                    safe_character_display(c)
                    for c in display_name
                    if (not ATEXT_RE.match(c) and c != ' ') or c == '.'
            }
            if bad_chars:
                    raise EmailSyntaxError("The display name contains invalid characters when not quoted: " + ", ".join(sorted(bad_chars)) + ".")

        check_unsafe_chars(display_name, allow_space=True) # Check for other unsafe characters.

        # Check that the right part ends with an angle bracket
        # but allow spaces after it, I guess.
        if ">" not in right_part:
            raise EmailSyntaxError("An open angle bracket at the start of the email address has to be followed by a close angle bracket at the end.")
        right_part = right_part.rstrip(" ")
        if right_part[-1] != ">":
            raise EmailSyntaxError("There can't be anything after the email address.")

        # Remove the initial and trailing angle brackets.
        addr_spec = right_part[1:].rstrip(">")

        # Split the email address at the first unquoted @-sign.
        local_part, domain_part = split_string_at_unquoted_special(addr_spec, ("@",))
```

```python


        # Otherwise there is no display name. The left part is the local
        # part and the right part is the domain.
        else:
            display_name = None
            local_part, domain_part = left_part, right_part


        if domain_part.startswith("@"):
            domain_part = domain_part[1:]


        # Unquote the local part if it is quoted.
        local_part, is_quoted_local_part = unquote_quoted_string(local_part)


        return display_name, local_part, domain_part, is_quoted_local_part
```

Test Case Inputs

```
[
    {
        "input": [["simple@example.com"], {}],
        "output": [null,"simple","example.com", false],
    },
    {
        "input": [["user+name@sub.domain.com"], {}],
        "output": [null,"user+name","sub.domain.com", false],
    },
    {
        "input": [["user.name@domain.co.uk"], {}],
        "output": [null,"user.name","domain.co.uk", false],
    },
    {
        "input": [["\"quoted@local\"@example.com"], {}],
        "output": [null,"quoted@local","example.com", true],
    },
    {
        "input": [["display name <user@domain.com>"], {}],
        "output": ["display name","user","domain.com", false],
    },
    {
        "input": [["user@localhost"], {}],
        "output": [null,"user","localhost", false],
    },
    {
        "input": [["user@[IPv6:2001:db8::1]"], {}],
        "output": [null,"user","[IPv6:2001:db8::1]", false],
    },
    {
        "input": [["\"escaped\\\"quote\"@example.com"], {}],
        "output": [null,"escaped\"quote","example.com", true],
    },
    {
        "input": [["user.name@longsubdomain.example.com"], {}],
        "output": [null,"user.name","longsubdomain.example.com", false],
    },
    {
        "input": [["very.common@example.com"], {}],
        "output": [null,"very.common","example.com", false],
    },
    {
```

```
        "input": [["user@domain-with-dash.com"], {}],
        "output": [null,"user","domain-with-dash.com", false],
    },
    {
        "input": [["user@123.123.123.123"], {}],
        "output": [null,"user","123.123.123.123", false],
    },
    {
        "input": [["\"much.more unusual\"@example.com"], {}],
        "output": [null,"much.more unusual","example.com", true],
    },
    {
        "input": [["user@xn--exmple-cua.com"], {}],
        "output": [null,"user","xn--exmple-cua.com", false],
    },
    {
        "input": [["user@domain_with_underscore.com"], {}],
        "output": [null,"user","domain_with_underscore.com", false],
    }
]
```


Generated Outputs


```
[
    {
        "input": [["simple@example.com"], {}],
        "output": [null,"simple","example.com", false],
        "prediction": [null,"simple","example.com",false],
        "result": true,
        "answer_tokens": {"completion": 18,"prompt": 4610,"total": 4628}
    },
    {
        "input": [["user+name@sub.domain.com"], {}],
        "output": [null,"user+name","sub.domain.com", false],
        "prediction": [null,"user+name","sub.domain.com",false],
        "result": true,
        "answer_tokens": {"completion": 21,"prompt": 4614,"total": 4635}
    },
    {
        "input": [["user.name@domain.co.uk"], {}],
        "output": [null,"user.name","domain.co.uk", false],
        "prediction": [null,"user.name","domain.co.uk",false],
        "result": true,
        "answer_tokens": {"completion": 20,"prompt": 4613,"total": 4633}
    },
    {
        "input": [["\"quoted@local\"@example.com"], {}],
        "output": [null,"quoted@local","example.com", true],
        "prediction": ["null","quoted@local","example.com",true],
        "result": false,
        "answer_tokens": {"completion": 20,"prompt": 4615,"total": 4635}
    },
    {
        "input": [["display name <user@domain.com>"], {}],
        "output": ["display name","user","domain.com", false],
        "prediction": ["display name","user","domain.com",false],
        "result": true,
        "answer_tokens": {"completion": 19,"prompt": 4615,"total": 4634}
    },
    {
        "input": [["user@localhost"], {}],
        "output": [null,"user","localhost", false],
        "prediction": [null,"user","localhost",false],
        "result": true,
        "answer_tokens": {"completion": 17,"prompt": 4610,"total": 4627}
```

```
    },
    {
        "input": [["user@[IPv6:2001:db8::1]"], {}],
        "output": [null,"user","[IPv6:2001:db8::1]", false],
        "prediction": "EmailSyntaxError: An email address must have an @-
            sign.",
        "result": false,
        "answer_tokens": {"completion": 24,"prompt": 4620,"total": 4644
        }
    },
    {
        "input": [["\"escaped\\\"quote\"@example.com"], {}],
        "output": [null,"escaped\"quote","example.com", true],
        "prediction": ["null","escaped\"quote","example.com",true],
        "result": false,
        "answer_tokens": {"completion": 20,"prompt": 4615,"total": 4635}
    },
    {
        "input": [["user.name@longsubdomain.example.com"], {}],
        "output": [null,"user.name","longsubdomain.example.com", false],
        "prediction": [null,"user.name","longsubdomain.example.com",false],
        "result": true,
        "answer_tokens": {"completion": 22,"prompt": 4615,"total": 4637}
    },
    {
        "input": [["very.common@example.com"], {}],
        "output": [null,"very.common","example.com", false],
        "prediction": [null,"very.common","example.com",false],
        "result": true,
        "answer_tokens": {"completion": 19,"prompt": 4611,"total": 4630}
    },
    {
        "input": [["user@domain-with-dash.com"], {}],
        "output": [null,"user","domain-with-dash.com", false],
        "prediction": [null,"user","domain-with-dash.com",false],
        "result": true,
        "answer_tokens": {"completion": 21,"prompt": 4614,"total": 4635}
    },
    {
        "input": [["user@123.123.123.123"], {}],
        "output": [null,"user","123.123.123.123", false],
        "prediction": [null,"user","123.123.123.123",false],
        "result": true,
        "answer_tokens": {"completion": 23,"prompt": 4616,"total": 4639}
    },
    {
        "input": [["\"much.more unusual\"@example.com"], {}],
        "output": [null,"much.more unusual","example.com", true],
        "prediction": [null,"much.more unusual","example.com",true],
        "result": true,
        "answer_tokens": {"completion": 20,"prompt": 4615,"total": 4635}
    },
    {
        "input": [["user@xn--exmple-cua.com"], {}],
        "output": [null,"user","xn--exmple-cua.com", false],
        "prediction": [null,"user","xn--exmple-cua.com",false],
        "result": true,
        "answer_tokens": {"completion": 24,"prompt": 4617,"total": 4641}
    },
    {
        "input": [["user@domain_with_underscore.com"], {}],
        "output": [null,"user","domain_with_underscore.com", false],
        "prediction": "EmailSyntaxError: The email address contains unsafe
            characters: ’U+005F’.",
        "result": false,
```

```
        "answer_tokens": {"completion": 28,"prompt": 4614,"total": 4642}
    }
]
```


A.2 Acceptable Types & Filtering Criteria

Acceptable types. To find functions where the inputs and outputs are LLM generatable, we recursively parse both arguements and return types as ast objects i.e. for `list[tuple[str, False]]` we first check `list` is an acceptable type, then recurse down into `tuple`, following that we then check `str` and finally we check `False`. `False` isn't an acceptable type but it is an acceptable constant and hence accepted. Note: certain acceptable types and constants are not allowed as return values, i.e. `None` is not an accepted return constant

acceptable_types = { ’int’, ’str’, ’float’, ’bool’, ’none’, ’list’, ’dict’, ’tuple’, ’set’, ’datetime.date’, ’date’, ’literal’, ’optional’, ’union’, ’sequence’, ’iterable’, ’frozenset’, ’mapping’ }

acceptable_constants = { ’ellipsis’, True, False, None }

Filtering functions. When filtering functions we maintain four separate block lists, 1) a list of banned imports (including direct and aliases), 2) a list of banned functions (some common libraries have a limited set of non-deterministic methods, we don't want to fully exclude them), 3) a list of banned variables (some variables such as `__version__` are likely to be environment based), 4) a list of banned repos (some repos from cloud providers provide thousands of near identical methods with different urls, we remove these as they are not a valuable contribution to the evaluation).