

System Architecture for Agentic Large Language Models

Tianjun Zhang



Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2025-5

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-5.html>

January 9, 2025

Copyright © 2025, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

System Architecture for Agentic Large Language Models

By

Tianjun Zhang

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Associate Professor Joseph E. Gonzalez, Chair

Associate Professor Sergey Levine

Assistant Professor Jiantao Jiao

Dr. Yuandong Tian

Fall 2024

System Architecture for Agentic Large Language Models

Copyright 2024
by
Tianjun Zhang

Abstract

System Architecture for Agentic Large Language Models

by

Tianjun Zhang

Doctor of Philosophy in Computer Science

University of California, Berkeley

Associate Professor Joseph E. Gonzalez, Chair

Large Language Models (LLMs) are evolving from serving knowledge passively in chatbots to actively interacting with applications and services. This thesis addresses the integration of Large Language Models (LLMs) as autonomous agents—hereafter referred to as Agentic LLMs, in terms of both training and deployment. Although recent advancements in LLM research have enabled sophisticated reasoning and natural language comprehension, their effective deployment in autonomous systems remains challenging. Some of the Key issues includes understanding the dynamics of the environment, ensuring seamless interaction with the environment, and maintaining rigorous standards of operational safety.

This thesis investigates four critical aspects of Agentic LLMs, aligning with the following dimensions: (1) Simulation Environments: It explores methods for creating generalizable and realistic training environments, such as R2E, enabling LLMs to engage with complex, dynamic tasks like software development in a controlled yet practical setting. (2) Advanced Reflection and Planning Capabilities: It delves into frameworks that empower LLM-based agents to iteratively refine their outputs and develop adaptive planning strategies, as demonstrated in LEAP and TEMPERA. (3) Adaptive Action Spaces: It examines techniques to enable LLMs to flexibly integrate and utilize external tools in dynamic environments, leveraging approaches like Gorilla and RAFT to navigate and adapt to evolving action spaces. (4) System Reliability and Safety: Finally, it proposes strategies to enhance system-level reliability, including rollback mechanisms and post-facto verification, ensuring safe and accountable autonomous operations.

The findings presented herein advance the state of the art in AI system architecture and offer valuable insights for future research and development. They also hold significant potential for real-world applications, particularly in fields such as robotics, autonomous vehicles, and intelligent infrastructure.

Contents

Contents	i
List of Figures	iv
List of Tables	x
1 System Architecture for Agentic Large Language Models	1
1.1 Motivation	1
1.2 Outline	3
I Gym Environment for LLM Agents	5
2 Build Gym Environment for Coding Agents	6
2.1 Introduction	6
2.2 Background	9
2.3 R2E Framework	10
2.4 R2E-Eval Benchmark	13
2.5 Towards R2E Programming Agents	15
2.6 Related Work	20
2.7 Limitations	21
2.8 Examples of R2E	21
2.9 Acknowledgement	23
II LLM Agents Planning	24
3 LLM Planning from Learning Mistakes	25
3.1 Background: In-Context Learning	29
3.2 LEAP: Learning Principles from Mistakes	30
3.3 Evaluation	31
3.4 Qualitative Analysis	36
3.5 Related Works	36

3.6 Conclusion	37
4 TEMPERA: Test-Time Prompt Editing Via Reinforcement Learning	38
4.1 Introduction	38
4.2 Related Work	40
4.3 Test-Time Prompt Editing	41
4.4 Test-Time Editing via Reinforcement Learning	43
4.5 Experiments	46
4.6 Conclusion	52
III LLM Interacting with Tools	53
5 Gorilla: Exploring Tool Usage of LLM Agents	54
5.1 Introduction	54
5.2 Related Work	56
5.3 Method	57
5.4 Evaluation	61
5.5 conclusion	66
5.6 Acknowledgement	67
6 RAFT: Document Understanding of LLM Agents	68
6.1 Introduction	68
6.2 LLMs for Open-Book Exam	70
6.3 RAFT	71
6.4 Evaluation	73
6.5 RAFT Generalizes to Top-K RAG	78
6.6 Related Works	79
6.7 Conclusion	81
6.8 Acknowledgement	81
IV Deploying LLM Agents	82
7 GOEX: Towards A Runtime For Autonomous LLM Applications	83
7.1 introduction	83
7.2 Evolution of LLM powered Agents	86
7.3 Navigating the New Frontier: Challenges in Ubiquitous LLM Deployments	88
7.4 Designing a Runtime	89
7.5 GoEx: LLM Runtime	91
7.6 Discussion	98
7.7 Related Work	99
7.8 Conclusion	99

8 Conclusion	101
Bibliography	103
A R2E	120
B In Context Principle Learning	140
B.1 Additional Results on Text Reasoning	140
B.2 Additional Results on Mathematical Reasoning	142
B.3 Additional Results on Text Reasoning	142
B.4 Additional Results on Mathematical Reasoning	144
B.5 Additional Results on BBH	144
B.6 Additional Qualitative Analysis	145
B.7 Learned Principles	150
C Tempera	186
C.1 Training Detail	186
C.2 Network Architecture	187
C.3 Additional Experiments	187
C.4 Natural Instructions and Promptsources	187
C.5 Dataset Detail	187
C.6 Comparison of Different Method	188
D Gorilla	193
D.1 Dataset Details	193
D.2 Gorilla Details	194
D.3 Performance Comparison	196

List of Figures

1.1	System Architecture of LLM Agents	2
2.1	An overview of our R2E framework that takes any GitHub repository and converts it into a programming agent test environment. Given a repository, we first scan for <i>interesting</i> functions and collect corresponding in-file and external-file contexts from the repository. Next, we use our test generation approach to develop high-quality <i>testing harnesses</i> for the function. Our key insight is decoupling the test outputs from inputs by relying on the ground truth implementation to get the expected outputs. Next, we perform specification refinement that improves the natural language docstring, making it amenable for code generation. Our framework yields problem instances comprising docstrings, test harnesses, and repository context (instantiated in the form of R2E-Eval benchmark). Our benchmark can be used to evaluate code generation systems, either static ones that directly generate code or programming agents that interact with the test harness and interpreter to improve code generation performance.	7
2.2	An example problem (left) in the R2E-Eval benchmark. The problem contains a function <code>indexer</code> from the <code>Torchsig</code> ¹¹ GitHub repository. <code>TorchSig</code> is an open-source signal processing machine learning toolkit based on the PyTorch data handling pipeline. The function <code>indexer</code> has dependencies within its file (<code>_parse_sigmf</code>) and from external files (<code>SignalDesc</code> and <code>SignalCap</code> from the file <code>torchsig/utils/types.py</code>). On the right is the generated equivalence test harness from our R2E framework. It contains a complex test setup where files expected by the function <code>indexer</code> are created and added to the file system. Then, the test cases generated perform functional equivalence checks for various granular properties of the returned output. Particularly, we create equivalence tests which instead of predicting the output behavior of the program, check it directly against the <i>ground truth</i> program available on GitHub – a simpler problem that requires constructing <i>diverse</i> inputs to test the function on.	8

2.3	Functional correctness (Pass1) of various models (GPT and CODELLAMA families) on our R2E-Eval. First, we note that, overall, models perform worse on our benchmark against HUMAN-EVAL, highlighting the challenging nature of real-world code generation tasks. GPT-4 performs particularly well, achieving a Pass1 close to 50%, much better than other models. Next, we study two retrieval settings—dependency context and full context and find a trade-off between the two (discussed in Section 2.5).	16
2.4	Pass1 of models as a function of the number of unique dependencies (functions and global variables) used in the original function. We find that models struggle to solve problems that require orchestrating multiple existing functionalities in the file and only perform well when a few dependencies are involved.	18
2.5	We measure whether self-repair using test harnesses and interpreter feedback can help the models correct mistakes and improve performance. We collect problems on which GPT-4 and GPT-3.5-TURBO fail and ask the models to iteratively correct by providing some error messages. We find that models improve performance from black-box feedback (33% and 21% respectively for GPT-4 and GPT-3.5-TURBO after 5 iterations).	19
3.1	Examples for learned principles using LEAP, with key idea of each principle highlighted.	26
3.2	An illustration of LEAP: Given a few input-output examples, Chain-of-Thought (left) generates a response to the test question by directly learning from the (correct) examples. In contrast, Learning Principles (LEAP, right) first (a) generates <i>mistaken</i> zero-shot Chain-of-Thought response for each given input-output example by sampling with a non-zero temperature; (b) generates explicit <i>principles</i> by providing the LLM with the mistaken CoT along with the correct output; and finally (c) generates a response to the test question, by providing the LLM with both the given input-output examples <i>and the learned principles</i> . Note that steps (a) and (b) are performed once per task.	27
3.3	LEAP prompt to help LLM evaluate its own generated reasoning and answers, contrasting them with the correct reasoning and answers. The LLM is prompted to identify errors in its reasoning and extract key insights for improvement. This figure specifically represents the ‘GenerateExplanation’ step in the LEAP algorithm (Chapter 1).	28

3.4	Accuracy in BBH tasks, across GPT-3.5-TURBO-0613, GPT-4-0613, and GEMINI-PRO. The figure presents the results using a scatter plot, where the y-axis represents scores achieved with LEAP, and the x-axis represents the baseline scores from CoT. Each task is represented by a point on the plot, with different shapes assigned to different models for easy distinction. Tasks above the $y = x$ line are those where LEAP leads to an improvement in performance. Chapter B.5 shows the detailed results for all 27 Big-Bench hard tasks. We find that in 37 out of 42 combinations of task and LLM, one of LEAP _{LOW-LEVEL} or LEAP _{HIGH-LEVEL} outperforms the baseline Few-shot CoT.	33
3.5	Examples from the Boolean Expressions (left) and Object counting (right) tasks from BBH. The learned principle is highlighted in yellow, the mistaken step of the baseline is highlighted in red, and the correct use of the principle by LEAP is highlighted in green. This demonstrates how the learned principles guide LEAP in generating a better answer.	35
4.1	Data Efficiency for TEMPERA: We compare the data efficiency of TEMPERA and standard fine-tuning in a few-shot setting. Results are averaged across four tasks: SST2, AG News, RTE and MR. It shows that our method achieves comparable performance using 4x fewer examples.	40
4.2	Test-Time Editing via RL: The RL agent is trained to optimize the performance of a downstream task. At test-time, given a query, the agent adopts an attention-based policy to edit the instructions, in-context exemplars and verbalizers for T rounds.	42
4.3	Data Efficiency for TEMPERA: We compare data efficiency between TEMPERA and few-shot finetuning. Results show that we can achieve a good performance with significantly less data (varying from 4x to 8x).	50
5.1	Examples of API calls. Example API calls generated by GPT-4 [130], Claude [8], and Gorilla for the given prompt. In this example, GPT-4 presents a model that doesn't exist, and Claude picks an incorrect library. In contrast, our Gorilla model can identify the task correctly and suggest a fully-qualified API call.	55
5.2	Accuracy (vs) hallucination in four settings, that is, <i>zero-shot</i> (i.e., without any retriever), and <i>with retrievers</i> . BM25 and GPT are commonly used retrievers and the <i>oracle</i> retriever returns relevant documents at 100%, indicating an upper bound. Higher in the graph (higher accuracy) and to the left is better (lower hallucination). Across the entire dataset, our model, Gorilla, improves accuracy while reducing hallucination.	56
5.3	Gorilla: A system for enabling LLMs to interact with APIs. The upper half represents the training procedure as described in Sec 3.2. This is the most exhaustive API data-set for ML to the best of our knowledge. During inference (lower half), Gorilla supports two modes - with retrieval, and zero-shot. In this example, it is able to suggest the right API call for generating the image from the user's natural language query.	58

5.4	Accuracy with GPT-retriever. Methods to the left of the dotted line are closed source. Gorilla outperforms on Torch Hub and Hugging-Face while matching performance on Tensorflow Hub for all existing SoTA LLMs - closed source, and open source.	62
5.5	Gorilla’s retriever-aware training enables it to react to changes in the APIs. The second column demonstrates changes in model upgrading FCN’s ResNet-50 backbone to ResNet-101. The third column demonstrate changes in model registry from <code>pytorch/vision</code> to <code>NVIDIA/DeepLearningExamples:torchhub</code>	65
6.1	How best to prepare for an Exam? (a) Fine-tuning based approaches implement ”studying” by either directly ”memorizing” the input documents or answering practice QA without referencing the documents. (b) Alternatively, in-context retrieval methods fail to leverage the learning opportunity afforded by the fixed domain and are equivalent to taking an open-book exam without studying. While these approaches leverage in-domain learning, they fail to prepare for open-book tests. In contrast, our approach (c) RAFT leverages fine-tuning with question-answer pairs while referencing the documents in a simulated imperfect retrieval setting — thereby effectively preparing for the open-book exam setting.	69
6.2	Overview of our RAFT method. The top-left figure depicts our approach of adapting LLMs to <i>reading</i> solution from a set of positive and negative documents in contrast to standard RAG setup where models are trained based on the retriever outputs, which is a mixture of both memorization and reading. At test time, all methods follow the standard RAG setting, provided with a top-k retrieved documents in the context.	71
6.3	RAFT prompt to help LLM evaluate its own generated reasoning and answers, contrasting them with the correct reasoning and answers. The LLM is prompted to identify errors in its reasoning and extract key insights for improvement. This figure specifically represents the ‘GenerateExplanation’ step in the RAFT algorithm (Chapter 6.3).	73
6.4	Comparison of RAFT and DSF: We prompt RAFT and DSF fine-tuned models on the HotpotQA dataset. We can see that the DSF model extracts the wrong information from the context. For the question, who is the screenwriter, it responds with a film name. RAFT manages to get the result correctly	77
6.5	How many golden documents to involve? We study the hyperparameter $P\%$ which indicates what fraction of the training data contains the oracle document(s) in its context. Results on NQ, TQA and HotpotQA suggest that mixing a fraction of data that does not have the oracle document in its context is helpful for in-domain RAG.	78
6.6	Test-Time Documents Varying: We study how robust RAFT is to varying numbers of test-time documents that a retriever might provide. In NQ, we find that training with 4 documents leads to the best performance, but training with 2 documents is optimal for HotpotQA. However, across both datasets, training with all datasets consisting of <i>oracle</i> documents hurts performance.	80

7.1	Evolution of LLMs powered applications and services from chatbots, to decision-making agents that can interact with applications and services with human-supervision, to autonomous LLM-agents interacting with LLM-powered apps and services with minimal and punctuated human supervision.	85
7.2	GoEx’s runtime for executing RESTful API calls. Upon receiving the user’s prompt, GoEx presents two alternatives. First, an LLM can be prompted to come up with the (Action, Undo-Action) pair. Second, the application developer can provide tuples of actions and their corresponding undo-actions (function calls) from which the LLM can pick amongst.	92
7.3	Runtime for executing actions on a database. We present two techniques to determine if a proposed action can be undone. On the left, for non-transactional databases like MongoDB, and for flexibility, we prompt the LLM to generate (Action, Undo-Action, test-bed) tuples, which we then evaluate in a isolated container to catch any false (Action, Undo-Action) pairs. On the right, we can provide a deterministic undo with guarantees by employing the transaction semantics of databases.	95
7.4	Runtime for executing actions on a filesystem. GoEx presents two abstractions. On the left, the LLM is prompted to come up with an (Action, Undo-Action, test-bed) which GoEx evaluates in a isolated container to catch any false (Action, Undo-Action) pairs. On the right presents deterministic guarantees by using versioning control system like Git or Git LFS.	97
A.1	Varying number of lines	123
A.2	Varying number of dependencies	123
A.3	Varying number of branches	123
A.4	Varying number of arguments	123
A.5	Varying number of dependencies	139
A.6	Varying number of context tokens	139
A.7	Varying number of ground truth tokens	139
A.8	Varying File usage	139
C.1	Data Efficiency for TEMPERA: We plot all the finetuning performance for 8 tasks we tested. We see that TEMPERA often achieves the better few-shot performance except for MRPC and QNLI.	188
C.2	Comparison of Different Prompting Methods: We compare the different property of different algorithms. We can see that TEMPERA is gradient-free, the resulting prompt is interpretable and query-dependent.	189
D.1	Domain names: Domain names with the three dataset. Tensor Hub is the smallest dataset while the other two hubs contain many more models.	195

D.2	Example of the Dataset: Two examples of the dataset, the above one is zero-shot (without information retrievers) and the bottom one is with information retriever.	196
D.3	Example of the Dataset: Two examples of the dataset, the above one is zero-shot (without information retrievers) and the bottom one is with information retriever.	197
D.4	Hallucination Examples: GPT-4 incurs serious hallucination errors in HuggingFace. We show a couple of examples in the figure.	198
D.5	Performance: We plot each model's performance on different configurations. We see that Gorilla performs extremely well in the zero-shot setting. While even when the oracle answer is given, Gorilla is still the best.	199
D.6	Accuracy vs Hallucination: We plot each model's performance on different configurations. We found that in the zero-shot setting, Groilla has the most accuracy gain while maintaining good factual capability. When prompting with different retrievers, Gorilla is still capable to avoid the hallucination errors.	200
D.7	The API calls that the Gorilla model is trained on is able to call the right API, but the supporting zip() code has a bug.	201

List of Tables

2.1	Test generation evaluation results across 3 strategies—Naïve, Full, and Sliced—for prompt context creation. The results are compared on 2 settings: In-File where the function under test only depends on the context within its file, and Out-File where it depends on external files in the repository. The metrics used are Validity (Val) and Coverage (Cov), for which higher is better.	12
2.2	Comparing R2E-Eval with other NL-to-code generation benchmarks, in terms of test execution-based support (Exec?), use of repository context (Repo?), and the number of lines in the ground truth function (LOC). R2E-Eval is the only executable benchmark, has repository context, and is automated, enabling scalability. Additionally, our benchmark contains more tests (harnesses) per function with diverse input types and quality assurance.	14
2.3	Statistics for problems instances in our R2E-Eval.	14
3.1	Textual Reasoning results: Accuracy in textual reasoning benchmarks. The best approach for each base LLM in each dataset is in bold ; the second-best approach is <u>underlined</u> . We see a good performance boost by adopting LEAP with high-level feedback and low-level feedback. Almost all the models can benefit from the principles learned and fix their previous mistakes.	31
3.2	Math Reasoning Results: Accuracy in MATH and GSM8K. The best approach for each LLM and base task is in bold ; the second-best approach is <u>underlined</u> . LEAP _{HIGH-LEVEL} and LEAP _{LOW-LEVEL} in both GSM8K and MATH datasets consistently improve the performance over the CoT baseline. We also observe that the learned mathematical principles can be generalized to different test questions.	32
3.3	Llama-2-chat-70B Results: Accuracy in MATH and GSM8K using Llama-2-chat-70B as the base LLM, while generating the principles either with Llama-2-chat-70B or GPT-4. Even when the principles are generated by GPT-4 (GPT-4 Critic), Llama-2-chat-70B does not leverage the learned principles, and does not improve over the baseline Few-shot CoT.	34
4.1	Effect of different editing techniques. For instruction, we tokenize it into phrases and perform swapping, addition or deletion. We also allow swapping in-context exemplars or changing different verbalizers.	44

4.2	Few-shot classification results. We compare against different baselines in this setting. Results show that TEMPERA surpasses various baselines including fine-tuning, prompt tuning and discrete prompt search. The standard deviations are shown in brackets.	48
4.3	We compare our method against different methods which do not perform test-time editing. Results show that test-time editing is mostly helpful in harder tasks like AG News.	49
4.4	Ablation on different editing techniques. Results show that adding verbalizer-edits helps all the tasks (especially MR and AG News). Adding instruction-edits marginally helps the performance in SST-2 and MR.	49
4.5	Qualitative results on the effect of the learned policy. We see that our method both enables the flexibility of various edits and interpretability of the final results. On the contrary, many prior methods produce non-readable prompts. Red text is prior to editing and blue text are the changes.	50
4.6	Ablation on the number of in-context exemplars. Results show that increasing the number of examples results in a consistent increase of performance except for AG News (which is due to the length limit).	51
4.7	Ablation on the size of the prompt pool to select from. We see that the performance does not change too much when changing the size of the pool, indicating that the performance is relatively stable.	51
5.1	Evaluating LLMs on Torch Hub, HuggingFace, and Tensorflow Hub APIs .	63
5.2	Gorilla 0-shot with GPT 3-shot incontext examples	63
5.3	Comparison of retrieval techniques	64
5.4	AST evaluation metric has strong correlation with Accuracy	64
5.5	Evaluating LLMs on constraint-aware API invocations	66
6.1	RAFT improves RAG performance for all specialized domains: Across PubMed, HotpotQA, HuggingFace, Torch Hub, and Tensorflow Hub, we see that domain specific Finetuning improves significantly of the performance of the base model, but RAFT consistently outperforms the existing domain specific finetuning method with or without RAG. This suggests the need to train the model with context. We compare our model with LLaMA finetuning recipes, and provide GPT-3.5 for reference.	74
6.2	Ablation on Chain-of-Thought: The numbers of RAFT and RAFT without CoT. Results on various datasets show that adding CoT can significantly improve the performance of the finetuned model. With a gain of 9.66% and 14.93% on the Hotpot QA and HuggingFace datasets respectively.	76
A.1	Interface Understandability Scores	125
A.2	Problem Intent Scores	125
A.3	List of models	125

A.4	Effect of CoT on code generation on a subset of our R2E benchmark	139
B.1	Text Reasoning Results: Accuracy in textual reasoning datasets. The best approach for each base LLM in each dataset is in bold ; the second-best approach is <u>underlined</u>	141
B.2	Math Reasoning Results: Accuracy in MATH and GSM8K. The best approach for each LLM and base task is in bold ; the second-best approach is <u>underlined</u>	142
B.3	Text Reasoning Results: Accuracy in textual reasoning datasets. The best approach for each base LLM in each dataset is in bold ; the second-best approach is <u>underlined</u>	143
B.4	Math Reasoning Results: Accuracy in MATH and GSM8K. The best approach for each LLM and base task is in bold ; the second-best approach is <u>underlined</u>	144
B.5	BBH Results: Accuracy in BBH tasks, across GPT-3.5-turbo, GPT-4, and Gemini Pro. The best approach for each base LLM in each tasks is in bold ; the second-best approach is <u>underlined</u> . Each number represents the average across 3 identical runs with a temperature of zero.	151
B.6	Additional results on Big-Bench Hard tasks. The best approach for each base LLM in each tasks is in bold ; the second-best approach is <u>underlined</u> . Each number represents the average across 3 identical runs with a temperature of zero.	152
B.7	High-level Principles 1	153
B.8	High-level Principles 2	154
B.9	High-level Principles 3	155
B.10	High-level Principles 4	156
B.11	High-level Principles 5	157
B.12	High-level Principles 6	158
B.13	High-level Principles 7	159
B.14	High-level Principles 8	160
B.15	High-level Principles 9	161
B.16	High-level Principles 10	162
B.17	High-level Principles 11	163
B.18	High-level Principles 12	164
B.19	High-level Principles 13	165
B.20	High-level Principles 14	166
B.21	High-level Principles 15	167
B.22	High-level Principles 16	168
B.23	High-level Principles 17	169
B.24	High-level Principles 18	170
B.25	High-level Principles 19	171
B.26	High-level Principles 20	172
B.27	High-level Principles 21	173

B.28 High-level Principles 22	174
B.29 High-level Principles 23	175
B.30 High-level Principles 24	176
B.31 High-level Principles 25	177
B.32 High-level Principles 26	177
B.33 High-level Principles 27	178
B.34 High-level Principles 28	178
B.35 High-level Principles 29	179
B.36 High-level Principles 30	180
B.37 High-level Principles 31	181
B.38 Low-level Principles 1	182
B.39 Low-level Principles 2	183
B.40 Low-level Principles 3	184
B.41 Low-level Principles 4	185
C.1 Hyperparameters used for TEMPERA in all the tasks.	186
C.2 Few-shot classification results. We compare against different baselines in this setting. Results show that TEMPERA surpasses various baselines including finetuning, prompt tuning and discrete prompt search. The standard deviations are shown in brackets.	187
C.3 Natural instructions used for TEMPERA in all the tasks.	190
C.4 Verbalizers used for TEMPERA in all the tasks.	191
C.5 Scaling results for TEMPERA in 512 training data per class. Results show that TEMPERA also scales and achieves better results comparing to finetuning.	191
C.6 Details for the dataset including the type, size of training, evaluation and test. Note that here all the sizes are few-shot dataset.	192
D.1 Hyperparameters for training Gorilla	194

Acknowledgments

Firstly, I would like to extend my deepest gratitude to my advisor, Joseph E. Gonzalez, for providing me with good food and lab amusement every day. Joey's unconditional support for me to explore different research topics has truly shaped my independence and critical thinking. His guidance, encouragement, and belief in my potential have been instrumental in my academic journey, and for that, I am forever thankful.

I am also grateful to my committee members, Sergey Levine, Jiantao Jiao, and Yuandong Tian, for their great advice and generous feedback to improve this thesis. Their insights, critiques, and support have been invaluable, pushing me to refine my work and strive for excellence.

Over the years of my PhD, I have had the pleasure of collaborating with many brilliant minds who have greatly contributed to my growth as a researcher. I would like to thank Paras Jain, Ajay Jain, Daniel Rothchild, Zhewei Yao, Amir Gholami, Yifan Yang, Qijing Huang, Bichen Wu, Pieter Abbeel, Ion Stoica, Michael W. Mahoney, George Biros, John Wawrzynek, Kurt Keutzer, Huazhe Xu, Yi Wu, Kevin Yang, Yiyang Zhao, Linnan Wang, Benjamin Eysenbach, Ruslan Salakhutdinov, Zhengyao Jiang, Robert Kirk, Tim Rocktäschel, Edward Grefenstette, Paria Rashidinejad, Stuart Russell, Xuezhi Wang, Denny Zhou, Dale Schuurmans, Tongzheng Ren, Chenjun Xiao, Na Li, Zhaoran Wang, Bo Dai, Fangchen Liu, Justin Wong, Pieter Abbeel, Yueying Li, Yi Zhang, Vibhav Vineet, Neel Joshi, Xin Wang, Xiao Liu, Jiaxin Ge, Naman Jain, Wei-Lin Chiang, Lianmin Zheng, Sheng Shen, Shijia Yang, Trevor Darrell, Manish Shetty, and Aman Madaan. Your collaboration, support, and friendship have made this journey not only intellectually stimulating but also incredibly enjoyable.

I would also like to extend my appreciation to the administrative and technical staff who have supported me throughout my PhD journey. Your assistance behind the scenes has been crucial in allowing me to focus on my research.

To my friends and fellow lab members, your camaraderie and encouragement have been a constant source of motivation. The shared moments of joy, frustration, and triumph have created memories that I will cherish forever.

Lastly, I am deeply grateful to my family for their unwavering support and love. Your belief in me has been my anchor throughout this journey. To my parents, thank you for always encouraging me to pursue my dreams and for your sacrifices that have made this possible. To my partner, your patience, understanding, and support have been my rock.

Chapter 1

System Architecture for Agentic Large Language Models

1.1 Motivation

A promising direction for leveraging and deploying large language models (LLMs) involves conceptualizing them as general-purpose decision-making entities—often termed AI agents. Numerous publications and frameworks have explored potential agent designs and capabilities, yet the field still grapples with fundamental questions: What truly constitutes an LLM agent? Some define agents by their ability to use tools, others emphasize planning and iterative refinement, and still others highlight the presence of latent state.

The notion of AI agents arose shortly after World War II, and the term itself dates back to 1956. Early formative work, such as that of Russell and Norvig (1995), provided a lasting conceptual framework:

“We define AI as the study of agents that receive percepts from the environment and perform actions. Each such agent implements a function that maps percept sequences to actions, and we cover different ways to represent these functions, such as reactive agents, real-time planners, and decision-theoretic systems.” – (Russell and Norvig, p. 25)

From the outset, researchers aimed to develop agents capable of autonomous decision-making in complex domains. Initial progress centered largely on reinforcement learning, where agents learned optimal strategies through iterative trial and error within a given environment, guided by a reward structure. Classical milestones in this area often emerged from well-defined games and control tasks, culminating in achievements such as DeepMind’s AlphaGo. In 2016, AlphaGo’s victory over one of the world’s top human Go players was a watershed moment, demonstrating that reinforcement learning, combined with deep neural networks and strategic search techniques, could surpass human-level performance in complex decision-making challenges.

Despite these successes, early agents were typically confined to narrowly defined domains and depended on carefully engineered training protocols and reward functions. They excelled where goals and rules were explicit, but faltered in more open-ended, ambiguous environments that demanded linguistic interaction and flexible reasoning. Over time, improvements in training methodologies and computational power ushered in a new class of AI agents rooted in large language models. While these LLM-based agents inherit foundational concepts from their reinforcement learning predecessors, they differ in essential ways. Rather than primarily relying on reinforcement signals, these agents learn from massive corpora of human-generated text, internalizing statistical patterns, contextual cues, and nuanced semantics. They handle tasks once considered outside the scope of narrow AI: engaging in natural dialogue, reasoning about abstract concepts, integrating knowledge across domains, and adapting strategies without explicit instruction.

This transition—from specialized, reward-driven agents like AlphaGo to broader, LLM-based agents—reflects an evolution in the underlying design, training, and deployment paradigms. Instead of tailoring agents to single-purpose tasks, we now empower them with expansive linguistic and conceptual understanding. Such agents can more easily adapt, collaborate with human partners, and handle open-ended complexity. They serve as a versatile foundation of intelligence, readily augmented with new capabilities and minimal retraining. Moving forward, this progression points toward a future where AI agents are not just adept at solving well-defined problems, but can operate as adaptable, language-fluent collaborators. They will interact seamlessly in complex, unstructured environments, interpret ambiguous objectives, and refine their strategies through an ongoing dialogue with human insight.

These developments motivate a reexamination of LLM agents: situating them within general environments and devising effective algorithms for their training, ultimately guiding the field toward more capable, resilient, and context-aware autonomous systems.

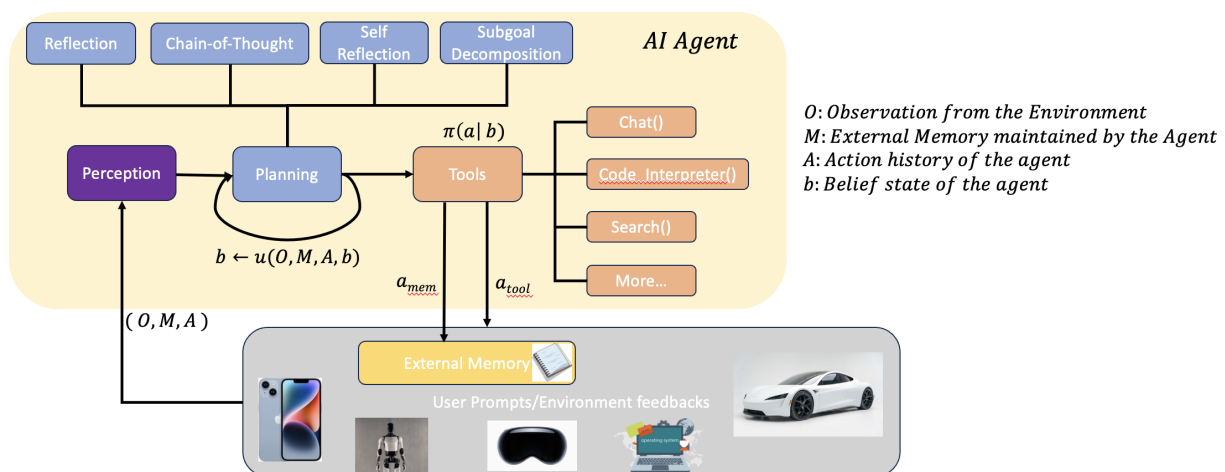


Figure 1.1: System Architecture of LLM Agents

1.2 Outline

This thesis systematically explores four primary areas integral to the development and deployment of large language model (LLM) agents: simulation environments, training algorithms, tool usage, and system assurances. Each chapter identifies limitations in previous work and demonstrates how the proposed contributions address these gaps.

Building Generalizable Training Environment Traditional simulation environments, such as OpenAI Gym and robotics-focused platforms, are largely tailored to physical or constrained task domains. These environments excel in tasks like control and navigation but lack the flexibility to model complex, interactive software systems where coding agents can operate. For instance, they do not accommodate tasks like resolving GitHub pull request issues, where the environment must be equipped with coding interfaces, version control, and contextual understanding.

In the first chapter we present R2E, a novel “gym” environment designed generally for coding agents. Unlike existing platforms, R2E leverages Docker-based containers to create flexible and reproducible coding environments. It enables LLMs to engage in tasks such as analyzing, modifying, and resolving GitHub pull requests in a controlled yet realistic setting. By bridging the gap between simulation and real-world software tasks, R2E provides a robust platform for both training and evaluating coding agents.

LLM Agents have Advanced Reflection and Planning Capabilities Early reinforcement learning methods, exemplified by AlphaGo, demonstrated remarkable success in well-defined domains with explicit rules and objectives. However, these methods struggle with tasks requiring adaptive planning and self-reflection in open-ended or ambiguous environments. Existing LLM training methods often neglect structured planning, focusing instead on task-specific fine-tuning or general language understanding.

The second chapters concludes two works: LEAP (In-Context Principle Learning) shows that LLMs can learn from their mistakes by leveraging in-context feedback, enabling them to iteratively refine their outputs. Unlike traditional guided sampling algorithms, LEAP integrates error histories as part of the learning loop, enhancing the model’s capacity for adaptive reasoning. TEMPERA is a reinforcement learning-based training paradigm that enables LLMs to generalize planning capabilities across unseen scenarios. TEMPERA demonstrates that models can learn to formulate and execute plans without requiring explicit task-specific objectives during training. Together, these contributions significantly enhance the planning and decision-making abilities of LLM agents.

LLM Agents Interact with Adaptive Action Space Previous RL works often pre-define an action space, that greatly limited the generalization of an agent. On another thread, LLM for tool usage often relies on rigid, predefined tool integration, which lacks adaptability to new tools or evolving tasks. Equipped with the text understanding of tool documents, LLMs can now cope with adaptive tooling space, allows more flexibility to cope with complex environment.

In this chapter, Gorilla is a fine-tuning approach that focuses on enabling efficient and contextually appropriate tool usage. By leveraging standard supervised fine-tuning tech-

niques, Gorilla equips LLMs to seamlessly interface with external tools across a range of tasks. RAFT (Retriever-Aware Fine-Tuning) enhances the LLM's ability to retrieve and interpret tool documents effectively. RAFT optimizes document understanding through a retriever-aware fine-tuning process, ensuring the model can dynamically adapt to diverse real-world constraints. These contributions empower LLM agents to handle more complex tasks that require adaptive external tool integration.

Ensuring System Reliability and Safety in Real-World Deployment Deploying autonomous agents in real-world environments introduces significant risks, particularly as LLMs gain the capability to make decisions without continuous human oversight. Existing systems often lack robust mechanisms for rollback or post-facto verification, which are critical for mitigating errors and ensuring safe operation.

GOEX addresses these challenges by proposing a set of system design principles aimed at enhancing reliability and safety. Key features include: (1) Rollback Mechanisms: Allowing agents to reverse potentially erroneous actions based on contextual or human feedback. (2) Post-Facto Verification: Enabling thorough evaluation of agent decisions after execution, ensuring accountability and minimizing risks. These mechanisms establish a framework for deploying LLM agents in practical applications with increased confidence and reduced operational hazards.

Part I

Gym Environment for LLM Agents

Chapter 2

Build Gym Environment for Coding Agents

While Large Language Models' (LLMs) coding capabilities have advanced rapidly, corresponding evaluation benchmarks on real-world programming setups are yet to catch up. Building a scalable and interactive testbed for evaluating general-purpose AI coding agents for real-world code has been challenging, particularly due to a lack of high-quality test suites available. In this paper, we present Repository to Environment (R2E), a framework that can turn any GitHub repository into a test environment to evaluate the performance of code-generating systems, both static and interactive. R2E is powered by a synergistic combination of program analysis and LLMs to construct equivalence test harnesses for any GitHub function. We instantiate our framework to build the first large-scale benchmark, R2E-Eval, for building realistic environments for AI coding assistants. Our results demonstrate that even when SOTA models cannot generate correct solutions with advanced prompting techniques, they can effectively use environment feedback highlighting the need to move from static functional coding to interactive programming paradigm. We hope that our framework (and the instantiated benchmark) can motivate research directions by providing web-scale open-ended coding testbeds.

2.1 Introduction

The rapid improvement of LLMs' performance on code-related tasks has enabled the development of coding assistants deployed in the real world. However, evaluations on such real-world coding setups have not kept pace. Prior benchmarks [27, 204] used for evaluating coding capabilities of LLMs only consist of *short and isolated* functional code completion problems. On the other hand, real-world software engineering requires more complex workflows involving integrating code with existing (large) codebases, using libraries, interacting with the interpreter, debugging errors, etc. In this work, to capture this interactive aspect (in contrast with single-shot code generation), we consider programming agents as AI sys-

tems that can similarly use interpreters and error feedback to improve their own outputs given a specification. *As such programming agents become more powerful, it urges the need to build real-world test environments to evaluate them.*

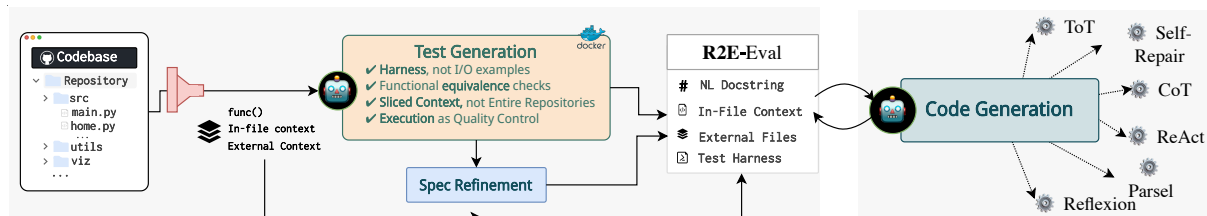


Figure 2.1: An overview of our R2E framework that takes any GitHub repository and converts it into a programming agent test environment. Given a repository, we first scan for *interesting* functions and collect corresponding in-file and external-file contexts from the repository. Next, we use our test generation approach to develop high-quality *testing harnesses* for the function. Our key insight is decoupling the test outputs from inputs by relying on the ground truth implementation to get the expected outputs. Next, we perform specification refinement that improves the natural language docstring, making it amenable for code generation. Our framework yields problem instances comprising docstrings, test harnesses, and repository context (instantiated in the form of R2E-Eval benchmark).

Our benchmark can be used to evaluate code generation systems, either static ones that directly generate code or programming agents that interact with the test harness and interpreter to improve code generation performance.

In this work, we propose **Repository to Environment (R2E)**, a scalable framework for turning any GitHub repository into a test environment to evaluate the performance of code generation systems on real-world scenarios. *We build on a key insight that test suites if synthesized for real-world code, can act as checks as well as orchestrators for execution-guided programming environments.* R2E takes a function (from GitHub), constructs an equivalence test harness – a scaffold consisting of test cases and a setup that establishes the operational conditions and dependencies of the function. R2E further refines the docstring and uses the refined specification along with repository code and test harness as a problem instance for studying code generation. Figure 2.1 (left) provides an end-to-end diagram of our approach.

These environments serve two evaluation purposes: First, a code generation system can be evaluated via the environment in these real-world scenarios. Secondly, even for an interactive programming agent, our environment can provide feedback to the agent using the interpreter (Figure 2.1 right). Notably, R2E framework is *scalable* and can be used to build web-scale open-domain coding datasets. Furthermore, R2E requires minimal human supervision and can be updated in a *live* manner for contamination-free evaluation.

Using this framework, we construct R2E-Eval, the first large-scale benchmark of real-world coding problems consisting of natural-language docstrings, repository contexts, and equivalence test harnesses. Figure 2.2 shows an example of a function and corresponding

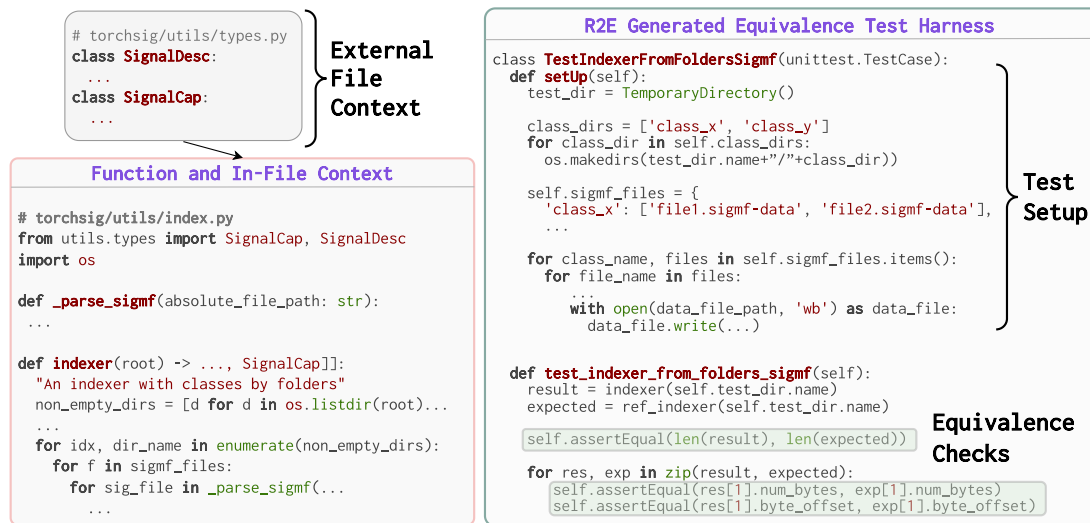


Figure 2.2: An example problem (left) in the R2E-Eval benchmark. The problem contains a function `indexer` from the TorchSig¹ GitHub repository. TorchSig is an open-source signal processing machine learning toolkit based on the PyTorch data handling pipeline. The function `indexer` has dependencies within its file (`_parse_sigmf`) and from external files (`SignalDesc` and `SignalCap` from the file `torchsig/utils/types.py`). On the right is the generated equivalence test harness from our R2E framework. It contains a complex test setup where files expected by the function `indexer` are created and added to the file system. Then, the test cases generated perform functional equivalence checks for various granular properties of the returned output. Particularly, we create equivalence tests which instead of predicting the output behavior of the program, check it directly against the *ground truth* program available on GitHub – a simpler problem that requires constructing *diverse* inputs to test the function on.

synthesized test harness from our dataset. Our dataset comprises of 246 tasks extracted from 137 repositories containing 127.2 code tokens, 11.5 tests, and 3.7 dependencies per problem,

on average.

Finally, we evaluate current LLMs on real-world scenarios from our benchmark. We find that compared to HUMAN-EVAL models perform worse on these problems, highlighting the challenges of real-world programming. Popular techniques like Chain-of-Thought (COT) do not help with performance. On the other hand, LLM agents that interactively program using the test harness and execution feedback greatly improve their performance. We also provide insights into model behavior specific to real-world programs, such as challenges in understanding interfaces to existing functions and reasoning about complex objects.

Overall, we find that real-world programming is complicated, even for SOTA LLMs (GPT-4), motivating the use of better workflows that mimic a typical developer’s programming process. This underscores the need to move from static functional coding to interactive programming, the evaluation of which our framework enables. Finally, the interactions collected in R2E environments can help improve various code-related abilities of LLMs.

2.2 Background

Our R2E pipeline is powered by a synergy of program analysis and LLMs. Here, we provide background on some concepts used in the following sections.

Testing Testing for functional correctness extends beyond mere input-output pairs, encompassing the broader dependencies that real-world software relies on. A **Test Harness** encapsulates this by combining *Test Cases* (defining inputs and expected outputs) and a *Setup* (establishing the operational conditions and dependencies like configuration files). The complexity of test harnesses, as illustrated in Chapter 2.2, surpasses the simple input-output examples in previous benchmarks, like HUMAN-EVAL 27. For instance, in Chapter 2.2, the test harness contains the required setup of files in a directory (i.e., file system dependency) that the program expects to run successfully.

Code Coverage The quality of tests is widely measured by its *coverage*—the fraction of code elements (e.g., statements or branches) it exercises 61. For example, a test that executes all lines of a function is said to have *line coverage* of 100%. A high coverage is desirable to ensure a function is tested thoroughly. We use **branch coverage** to evaluate the quality of our tests as it offers a more fine-grained measure than line coverage.

Program Analysis for Slicing Context To effectively test repository code, we must grasp the function’s operational context, which encompasses the functions and global variables it interacts with. We employ **dependency slicing** to construct this context, defining a slice D_f for function f as the set of functions F' called by f and global variables G' accessed by f , both directly and indirectly. The top-left of Chapter 2.2 shows an example of a dependency slice extracted for a function `indexer`, that serves as a *minimal* context necessary to comprehend the function’s behavior. The resulting slice D_f provides the minimal context for understanding f ’s behavior and indicates its *connectivity* within the repository, quantified by the slice size $|D_f|$.

¹<https://github.com/TorchDSP/torchsig>

2.3 R2E Framework

GitHub is a rich data source for *realistic* code problems, but repositories in the wild can be quite noisy, hard to run, and poorly maintained. We here propose R2E, an automated framework that turns any GitHub repository into a test environment to evaluate the performance of code generation systems on real-world code.

Section 2.3 details our initial problem curation process. Section 2.3 describes our test harness synthesis approach. We evaluate the quality of our synthesized tests in Section 2.3. Finally, we describe how problem specifications can be refined to build a high-quality benchmark in Section 2.3.

Problem Curation

Repository Curation

We scraped PYTHON repositories on GitHub created after July’22 that are non-forks, have at least 40 stars, and contain either a `toml` or `setup.py` file. This date aligns with the reported cutoff data for OpenAI models GPT-3.5-TURBO and GPT-4, thus preventing contamination. Each repository was cloned into a Docker container and built using `pip install`. Repositories with build errors were excluded. However, we are still left with uninstalled packages due to incomplete build files on GitHub, which we resolve by tuning the docker image with the relevant pre-installed packages.

Function Curation

We first extract all functions from the collected repositories to identify functions suitable for natural-language-driven code generation and functional correctness evaluation. We then filter out functions lacking docstrings to ensure we have a natural language prompt equivalent. We apply keyword-based filters to exclude functions associated with GPU and cloud-related tasks since they are not conducive to standard functional correctness evaluations. Lastly, we estimate the complexity of the functions using its *connectivity* (detailed in Chapter 7.2). We filter out functions that do not call other components in the repository.

Through these stages of filtering, we collected candidate 2073 problems from 790 repositories.

Equivalence Test Harness Generation: A Key to Environments

GitHub repositories *lack high-quality tests* necessary for evaluating code generation, thus requiring automated test harness generation to collect problems scalably.

If generated, these tests can act as *checks* and *orchestrators* for execution-guided programming agents. As checks, they can evaluate the functional correctness of generated code. As orchestrators, they can run the generated code, capture compiler feedback, enable repair, and more.

To tackle this, R2E synthesizes tests for arbitrary GitHub code using a novel synergy of *program analysis* with LLM prompting. Below, we summarize some of the key design choices of R2E’s test generation approach.

Harnesses, not I/O pairs R2E generates *equivalence test harnesses* (Chapter 7.2) for each function, which contain the test cases and the required setup, such as database connections, external files, configurations, etc., that makes it possible to run functions in the wild. This is a departure from traditional I/O tests in benchmarks such as HUMANEVAL 27. This is necessary because real-world code often requires more than simple input arguments to run. They may need several dependencies, such as access to files, environment variables, and other functions.

Equivalence Tests, not Output Prediction R2E decouples test outputs from inputs. It instead uses the original function as a reference implementation to generate expected outputs. This key insight dramatically simplifies test generation since it removes the need to predict outputs. Consequently, we generate *equivalence tests*—they check if the outputs of the original function and the generated function is equivalent against a given set of inputs.

Sliced Context, not Entire Repositories Test generation using LLMs has been effective in prior work like HUMANEVAL+ 100 for simple isolated functions. However, in a repository setting, prompting with the function alone is insufficient, and providing the entire repository is expensive. R2E uses a novel *dependency slicing based prompt* to extract the minimal repository context required to understand the functionality of the function under test. As described in Chapter 7.2, it finds functions and global variables on which the function directly or indirectly depends.

Execution and Coverage for Quality Control Finally, recent studies have shown that execution-based benchmarks can be flawed due to low-quality tests 100. To avoid this, we execute the generated test harnesses in the docker container built for the repository. Equivalence tests are run in “self-equivalence” mode, so the function under test and the reference implementation are the same. Inoperative harnesses due to issues like missing packages are excluded. An (equivalence) test harness is deemed *valid* if all the (equivalence) tests pass. We further emphasize the *quality* of test cases by using branch coverage (Chapter 7.2). This check is critical to ensure that the generated tests cover the function’s complete behavior and can be used for checking functional equivalence.

We encode our design decisions as guidelines to prompt GPT-4-TURBO and use the sliced context to generate high-quality test harnesses. Chapter 2.2 shows the resulting harnesses that handle complex data types and unique setups, depending on the function’s requirements. We outline additional guidelines for test generation in the appendix.

Test Harness Evaluation

Experiment Setup

We evaluate equivalence test harness generation on two fronts. First, measure *validity*, i.e., does it execute the original function while passing all equivalence tests? Then, we also

Strategy	In-File		Out-File	
	Val	Cov	Val	Cov
Naïve	41.4%	95.1%	32.1%	93.6%
Full	48.4%	95.1%	39.4%	92.5%
Sliced	50.0%	96.6%	46.3%	95.0%

Table 2.1: Test generation evaluation results across 3 strategies—Naïve, Full, and Sliced—for prompt context creation. The results are compared on 2 settings: **In-File** where the function under test only depends on the context within its file, and **Out-File** where it depends on external files in the repository. The metrics used are Validity (Val) and Coverage (Cov), for which higher is better.

evaluate the *quality* using branch coverage (Chapter 7.2) to identify how well the tests cover the function’s behavior—a critical property for equivalence checking.

We consider three strategies for test generation in a repository context: **Naïve**, **Full**, and **Sliced**. The **Naïve** strategy prompt contains the function and no context. The **Full** strategy provides the file containing the function and all files it imports (until a 6000 token limit). Finally, the **Sliced** strategy implements our proposed dependency slicing to provide the minimal context required for the function. We compare these strategies in 2 problem settings: (1) **In-File**: where the function under test depends only on the context within its file and (2) **Out-File**: where it depends on external files in the repository. We generate all tests using the state-of-the-art GPT-4-TURBO model.

Validity and Quality Results

Chapter 2.1 shows the results of our evaluation.

Focussed context improves coverage. The **Naïve** strategy performs relatively poorly on validity (as low as 32%), but the valid test harnesses it generates have high coverage (93.6%). For example, naïvely generated tests often fail to generate correct input argument types (e.g., schemas or custom classes) due to the lack of necessary context.

Broader context improves validity. On the other hand, the **Full** strategy generates more valid tests (as high as 48.4%) but has lower coverage (92.5%). This indicates that a focused context can be more effective in covering corner cases in the function, but a broader context is necessary to understand the function’s dependencies.

Our sliced strategy strikes a good balance between the two and achieves the best results in validity and coverage. Overall, we observe that **R2E**’s dependency slicing-based strategy generates $\approx 50\%$ valid test harnesses with a high $\approx 95\%$ code coverage.

Failure Modes

We collected and classified invalid Equivalence test harnesses, and study their failure modes. We discovered that 40% of errors were due to `ValueErrors` and `TypeErrors`, reflecting improper key, attribute, or type usage in tests. Additionally, 15% were `DataFormatErrors`, caused by incorrect data formats or schemas, highlighting the complexity of testing GitHub code beyond primitive types.

`AssertionErrors` (expected and actual outputs don't match) accounted for a notable 25% of errors, showing a nuanced aspect of functional correctness in real-world code. Although R2E simplifies this to equivalence tests, assertions often need more granularity than simply checking for equality. For example, checking for class attributes, columns in a dataframe, etc., requires a deeper understanding of code and repository context. Lastly, `EnvironmentErrors` (21%), like OS and File system errors, indicate challenges with test environment configuration.

Refinement of Specifications

Natural language docstrings in GitHub repositories might be ambiguous or under-specified to be used for code generation. Here, we propose an automated approach to refine the natural language docstring of a given function by asking the model to refine the docstring in a self-instruct-like fashion [202]. Distinctly, however, we provide the model with additional context in the form of the original docstring, test harness class, argument types, and serialized input-output arguments available via the test harness.

We note that while we cannot evaluate the quality of refined specifications, we perform rigorous manual evaluations and filter problems with poor or ambiguous specifications.

2.4 R2E-Eval Benchmark

We showed that R2E enables a scalable framework for building execution-based test environments for programming agents. R2E takes a function from a codebase and converts it into a tuple $I = \{D, R, T\}$, where D is a refined docstring for a function, R is the remainder of the codebase, and T is the generated test harness.

We instantiate this framework to construct R2E-Eval, the first large-scale dataset of *real-world* code generation problems with functional correctness tests. Chapter 2.2 compares R2E-Eval against several popular benchmarks used to evaluate code generation capabilities of LLMs. Prior work like HUMAN-EVAL [27] and ODEX [204] support execution-based metrics but for isolated simple problems with no real-world repository setting. Recent work on repository-level code generation like CROSSCODE-EVAL [41], REPOBENCH [103], and REPO-EVAL [238] use repository context, but either forego execution-based evaluation or depend heavily on human-written tests, which are seldom available at scale on GitHub. R2E-Eval is the only executable benchmark that has repository context and is automated, enabling scalability. Following, we describe the construction of R2E-Eval and analysis.

Dataset	Exec?	Repo?	Auto?	LOC	#Tests
HUMANEVAL	✓	✗	✗	6.26	6.6
ODEX	✓	✗	✗	3.05	1.9
CROSSCODEEVAL	✗	✓	✓	1.0	-
REPOBENCH	✗	✓	✓	1.0	-
REPOBENCH-FUNC	✓	✓	✗	10.8	-
R2E-Eval	✓	✓	✓	10.5	11.5

Table 2.2: Comparing R2E-Eval with other NL-to-code generation benchmarks, in terms of test execution-based support (Exec?), use of repository context (Repo?), and the number of lines in the ground truth function (LOC). R2E-Eval is the only executable benchmark, has repository context, and is automated, enabling scalability. Additionally, our benchmark contains more tests (harnesses) per function with diverse input types and quality assurance.

Feature	Value
# Problems (# Repos)	246 (137)
Avg. # lines (# tokens)	10.5 (127.2)
Avg. # tests (coverage)	11.5 (92.2)
Avg. # dependencies	3.7
# Unique APIs	70
# Unique Arg Types	118

Table 2.3: Statistics for problems instances in our R2E-Eval.

Benchmark Construction

Dataset Quality

We emphasize heavily on the quality of problems in this work. Quality, here, means how well the function, docstring, and test cases are written. To ensure this, we only consider functions with high branch coverage. Our final benchmark problems have an average of 11.5 test cases with 92.2% average branch coverage. An additional round of manual inspection helps us select high-quality problems. Notably, in the manual inspection, we avoid very long or complex that is hard to specify precisely using docstrings (like functions with many peculiar corner cases).

Dataset Composition

We also consider the diversity and interestingness of the problems in the benchmark. We identify several properties of code that calibrate interestingness, such as # of dependencies, argument types, lines, libraries used, etc.

Table 2.3 showcases statistics of our benchmark. Our manual analysis also shows that

R2E-Eval problems are diverse in terms of the domains they cover: pythonic operations (`list`, `str`, `dict` manipulations), data manipulation (`JSON`, files, `pandas`, `numpy`), algorithm and protocol implementations (`networkx`, statistics), domain-specific problems (programming languages, networks, quantum computing, formal verification, numerical computing), and more.

We also ensure that the benchmark is diverse in terms of the number of distinct repositories, preventing bias towards a codebase or domain. Overall this process leads to a curated set of 246 problems from 137 repositories in R2E-Eval.

Each problem instance I can be used to evaluate a code generation system by providing docstring D to the system and evaluating its response (in the context of the repository R) against the generated test harness T .

2.5 Towards R2E Programming Agents

We conduct experiments to understand three important problems about LLM performance on real-world coding.

- Q1 How well can current LLMs solve the real-world code generation tasks statically? (Sec. 2.5)
- Q2 What are the typical LLM failure modes? (Sec. 2.5)
- Q3 How do programming agent paradigms (like self-repair) perform against static programming? (Sec. 2.5)

Our results show that the SOTA LLM model (GPT-4) can only achieve $\sim 50\%$ performance in R2E-Eval, despite high accuracy on HUMAN-EVAL. Throughout the analysis, we find that LLMs struggle at *understanding* interfaces to existing functions and reasoning about complex objects. Finally, we compare static coding approaches (e.g., COT) with the proposed interactive programming paradigm, demonstrating significant benefits from the latter.

Static Code Generation

First, we study direct code generation on the R2E-Eval dataset, i.e., using code generation without interaction. Owing to the test harnesses generation approach, we perform functional correctness evaluations for the generated code. This contrasts with prior works [103, 41] that rely on execution-free exact-match metrics to evaluate code completion in the repository setting, which can be unreliable and restrict the scope of the evaluation.

We use Pass1 to evaluate the functional correctness, computed by generating 5 candidate completions for each problem instance and computing the fraction that passes against the

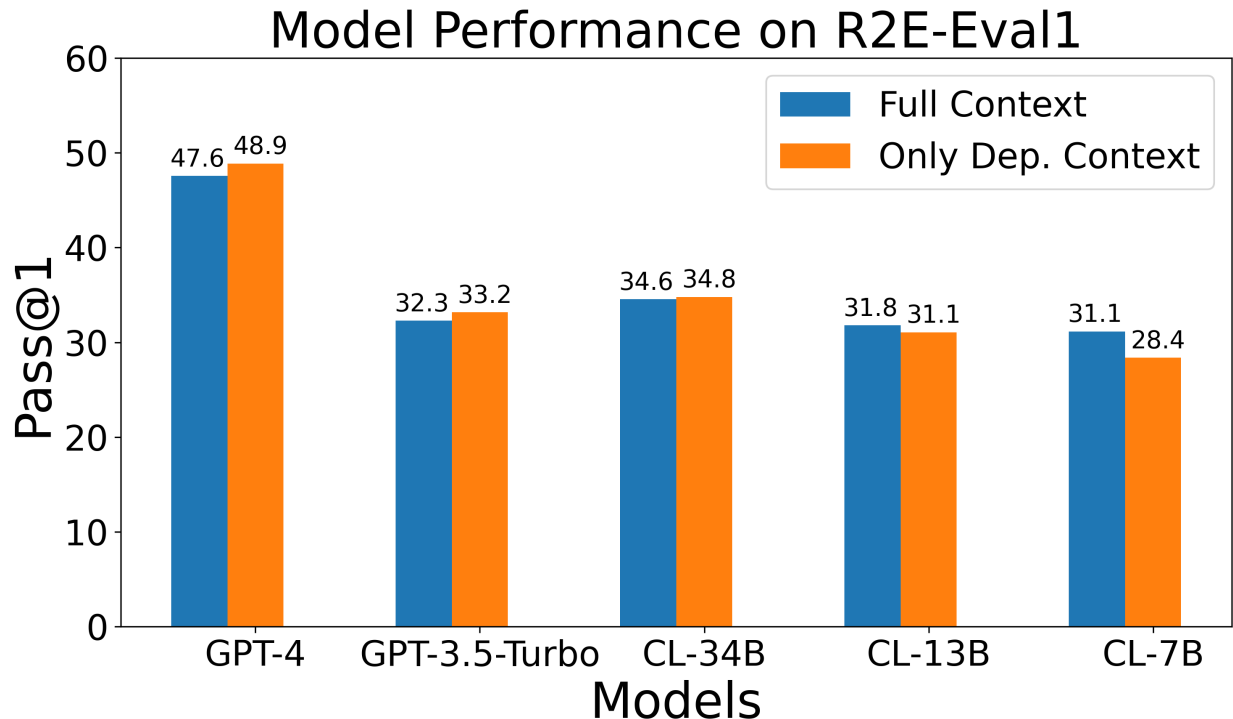


Figure 2.3: Functional correctness (Pass1) of various models (GPT and CODELLAMA families) on our R2E-Eval. First, we note that, overall, models perform worse on our benchmark against HUMAN-EVAL, highlighting the challenging nature of real-world code generation tasks. GPT-4 performs particularly well, achieving a Pass1 close to 50%, much better than other models. Next, we study two retrieval settings—dependency context and full context and find a trade-off between the two (discussed in Section 2.5).

test harness. We consider a mixture of closed access and open access models for our experiments – GPT-4, GPT-3.5-TURBO, CODELLAMA-7B, CODELLAMA-13B, and CODELLAMA-34B². Since GPT-4 and GPT-3.5-TURBO are instruction-tuned models, we use the chat style prompt for them while using the code completion prompt from the CODELLAMA models. We elaborate further on our setup, models, and prompts.

Contamination GPT-4 and GPT-3.5-TURBO have a cut-off date of 2021 and are therefore not contaminated on our benchmark since we curate our problems from repositories created after August 2022 (see Section 2.3).

Given a problem instance $I = \{D, R, T\}$ in our benchmark, we need to use the remaining repository context to generate the code. Since the entire repository context can be very large, we retrieve content to provide the model (detailed ahead). We first evaluate how current models hold up on our benchmark and then study how the choice of retrieval impacts

²We use the Python variants of the CODELLAMA models.

performance. Next, we study the effect of using chain-of-thought prompting (COT) [206] for improving model performance on harder tasks.

Model Performance Figure [2.3] compares the performance of various models on our benchmark using the Pass1 (CL used for brevity in the figure instead of CODELLAMA). We find that the performance of various models is relatively lower than other benchmarks like HUMANEVAL. This is expected since our benchmark represents more challenging real-world problems collected from GitHub, which require *understanding* existing context from the repository before generating the code. We find that GPT-4 performs significantly better than all other models with a Pass1 close to 50% whereas other models only achieve Pass1 in the vicinity of 30%.

Effect of retrieval We study the effect of function-definition retrieval vs. function-usage retrieval using dependency slicing (Chapter [7.2]) on the ground-truth function. Specifically, dependency-only-context only provides the necessary definitions, while the full context setting adds the remainder of the file and other files until a 6000 token limit. Chapter [2.3] compares the two settings.

The two retrieval methods perform similarly, achieving $\pm 1\%$ of each other’s performance across most models. On a closer look, we find non-overlapping problems with a Pearson correlation coefficient of 0.48. We find that dependency-only-context vs full-context provides an interesting trade-off. On the one hand, dependencies provide a more focused view of relevant function implementations to the model. At the same time, function usage (present in full context) is often reused and enables models to *copy it directly*. See Chapter [A] in the Appendix for a more detailed discussion and examples of this trade-off. Finally, we believe that R2E-Eval provides a unique opportunity to study this problem in the future with execution enabled.

Effect of COT We study better-prompting strategies and look at both zero-shot and two-shot COT prompts that sketch a *plan* for the function implementation before generating the code. We study this for the instruct GPT-3.5-TURBO and GPT-4 models but find that COT like setup does not improve performance over direct prompt (Table [A.4] in Appendix).

Model Behaviour & Failure Analysis

Performance with problem-complexity We measure the complexity of a problem instance using (1) the number of tokens in the ground-truth implementation, (2) the number of dependencies used by the ground-truth implementation³. We find that both these measures are (inversely) correlated with the Pass1 of the models. In Figures [2.4] and [A.7], we plot the Pass1 of the models against the number of dependencies and the number of tokens used in the ground-truth implementation showing a downward trend.

Single File vs Multi-File Context We compare how models perform on problems that require only a single file to be generated against problems that require multiple files to be generated. Model performance is significantly better on single-file problems than multi-

³counted using the number of unique functions or global variables used in the function body.

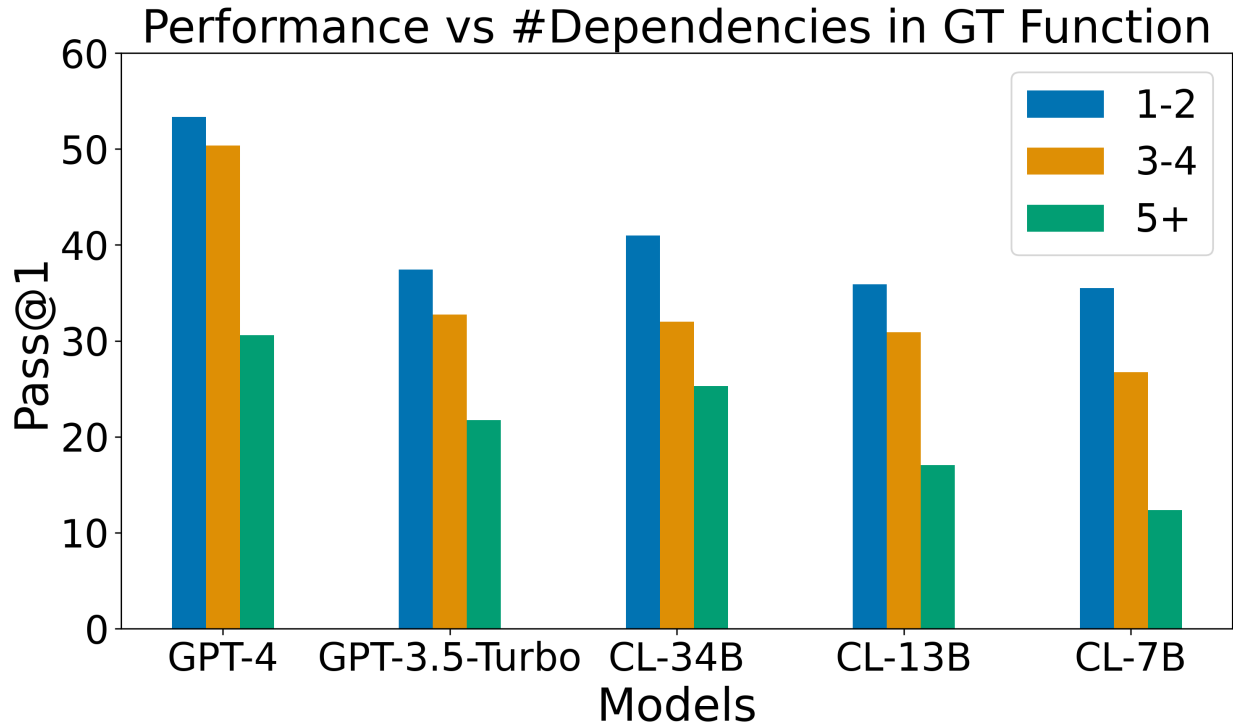


Figure 2.4: Pass1 of models as a function of the number of unique dependencies (functions and global variables) used in the original function. We find that models struggle to solve problems that require orchestrating multiple existing functionalities in the file and only perform well when a few dependencies are involved.

file problems (Figure A.8). This suggests that a.) models struggle with multi-file contexts compared to single-file contexts and b.) problems in the multi-file category are more complex than single-file problems in our benchmark, also observed in practice.

Do not *understand* the interface to provided functions We find that when provided with complex functions in the context, LLMs do not understand the right input-output behavior of such functions and pass in wrong inputs or expect wrong outputs. Thus, even strong LLMs like GPT-4 make mistakes when provided with complex functions in the context. See Listings A.1 for reference. This motivates that if provided access to execution context, programming agents can *understand* such interfaces and perform better.

Repeat vs Reuse Code Abstractions are an integral part of writing good code. LLMs, however, tend to duplicate code instead of using existing context. Specifically, when provided with some existing function in the context, models re-implement the same functionality instead of directly using it. Listings 2.1 provide examples. This aligns with findings on how copilot affects code quality [16].

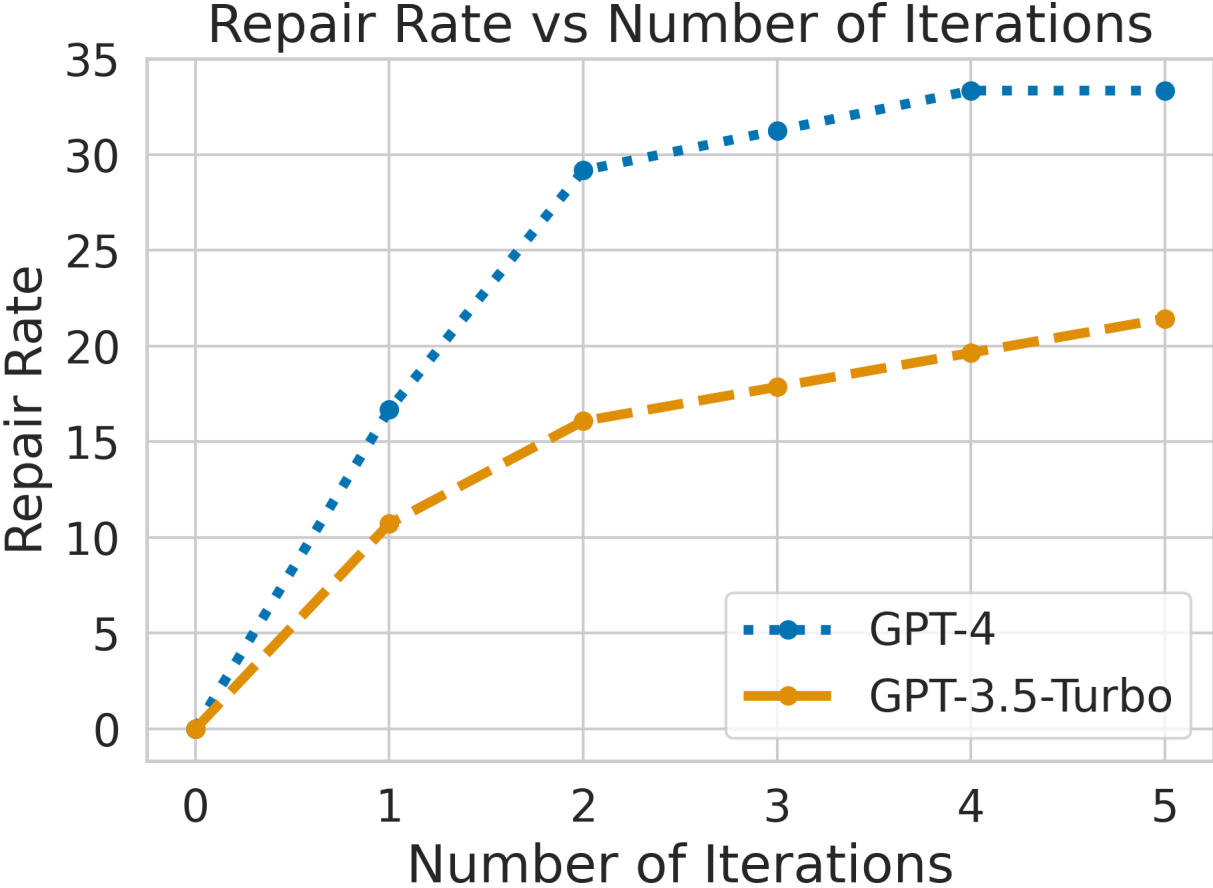


Figure 2.5: We measure whether self-repair using test harnesses and interpreter feedback can help the models correct mistakes and improve performance. We collect problems on which GPT-4 and GPT-3.5-TURBO fail and ask the models to iteratively correct by providing some error messages. We find that models improve performance from black-box feedback (33% and 21% respectively for GPT-4 and GPT-3.5-TURBO after 5 iterations).

Self-Repair Agent

So far, we described model evaluations on our benchmark using the direct code generation approach. However, testing harnesses and access to the interpreter allow us to evaluate programming agents that can interact with the interpreter and get feedback. Specifically, we instantiate a self-repair agent that uses the test harness

We study that when provided with feedback from (oracle) testing harnesses (present in our benchmark instances), can LLMs correct their own mistakes? We sample 56 and 48 instances from our benchmark for GPT-4 and GPT-3.5-TURBO on which the models do not generate a correct solution (detailed experiment setup in Section A in the Appendix). We consider the incorrect programs generated by the models as the initial programs and then provide the models with error feedback using the harness iteratively for 5 iterations. Figure 2.5 shows the self-repair rate of the models on our benchmark as a function of the number of iterations.

First note that since we subsample only the failing instances where models do not generate correct solutions, the 0-th iteration score is 0% for both models. Next, we find that GPT-4 attains a maximum self-repair rate of 33% while GPT-3.5-TURBO only attains a maximum self-repair rate of 20%. This highlights that using execution, interpreter, and test cases, programming *agents* can improve code generation. Note that while advanced prompting techniques do not improve performance (Table A.4), using an interpreter enables programming agents to achieve strong results.

2.6 Related Work

Code Generation Benchmarks. Code generation is primarily evaluated using functional-correctness and has been explored in multiple domains. HUMAN-EVAL [27] and MBPP [10] study code generation on isolated single-function problems. APPS [56] and CODE-CONTESTS [91] benchmarks are primarily used for evaluating algorithmic code generation capabilities. DS-1000 [83], ARCADE [228], NUMPY-EVAL [235], and PANDAS-EVAL [64] study data science API code generation. More recently, [204] proposed ODEX that evaluates coding on APIs with human-written input-output examples. These works evaluate code generation capabilities in isolated settings devoid of surrounding context or dependencies from other files. In contrast, R2E coding problems are curated directly from GitHub thus more similar to real-world setups. INTERCODE and WEBARENA provide general environments for domain-specific interactive programming and web tasks respectively. We provide a framework and environments for interactive general-purpose programming tasks extract from GitHub.

For the repository setting, prior works have primarily focused on execution-free evaluation metrics like exact-match and BLEU due to absence of test harnesses. CONALA [227] curated a large dataset from STACKOVERFLOW with paired natural language and program snippets. [174, 175] study different context selection methods for prompting and training LLMs for repository-level code generation.

REPOEVAL [233], REPOBENCH [103], and CROSSCODEEVAL [41] study repository-level code *completion*. However, these works only evaluate short context code generation capabilities without execution or functional correctness restriction to short completions. In contrast, we synthesize function-level test harnesses using our novel test generation approach and use them for performing function correctness checks on repository code. Recently, [69] proposed SWEBENCH to evaluate whether LLMs can solve GitHub issues. However, they assume test cases availability from pull requests preventing scalable collection of problems. Our test harness synthesis in contrast allows collecting problems from diverse set of repositories (137 repositories vs 12 repositories). Finally, [42] proposed, CLASSEVAL, manually curated for evaluating LLMs.

Other code-related tasks. Beyond codegen, tasks like self-repair [29, 128, 116, 137, 234], test generation [191, 205], execution [10, 96, 52], and optimization [114] have been studied. These enable various *agentic* setups as CODET [26, 75], PARSEL [230], FUNSEARCH [153], REFLEXION [173], LEVER [125], CODEPLAN [13], ALPHACODIUM [152], REACT [224], and TOT [225].

2.7 Limitations

Limitations Natural language is inherently ambiguous and docstrings might not specify the corner cases properly. We tried to mitigate this effect with our specification refinement approach along with manual filtering. Future work study this ambiguity in more and also look into better interaction mechanisms. Next, we use observational equivalence to check whether the model-generated candidates are correct over a set of inputs. We use branch coverage as a metric for evaluating tests but it is still a softer check. Future work can apply mutation testing and oversampling to provide further confidence on generated tests.

Conclusion We propose R2E, a scalable framework to convert GitHub repositories to programming agent test environments. R2E-Eval constructed via this framework can evaluate both static and interactive code generation systems, offering valuable insights into model behaviors and the need for better programming workflows. Prior work has applied rejection sampling and reinforcement learning to improve coding capabilities of LLMs [177, 65, 85]. We believe R2E can enable such attempts for real-world programs.

2.8 Examples of R2E

```
...
## paco/evaluation/utils/paco_query_utils.py
import torch

def _product_attr(vision, text, alter):
    """
    Args:
```



```

        vision: N x D
        text: M x D
        alter: N x M, to replace results in some cases, see details
in Returns
    Returns: N x M.
        For (n, m) element, set J_m = {j : text[m, j] == 1}.
        - if |J_m| > 0, it equals to (prod_{j in J_m} vision[n, j])
**(1/|J_m|)
        - if |J_m| == 0, it equals to alter[n, m]
    """
    vision = vision.unsqueeze(1)
    text = text.unsqueeze(0)
    num_attr = text.sum(-1)
    queried_attr = vision * text
    queried_attr = queried_attr.masked_fill(text == 0, 1)
    queried_attr = torch.float_power(queried_attr.prod(dim=2), 1 /
torch.max(num_attr, torch.ones_like(num_attr))).float()
    no_attr_queries = num_attr.squeeze(0) == 0
    queried_attr[:, no_attr_queries] = alter[:, no_attr_queries]
    return queried_attr

def obj_with_attributes(input_embeddings, query_embeddings, n_obj,
n_part, n_attr):
    """Compute the similarity between object embeddings and query
embeddings based on attributes.

    This function calculates the similarity score between each pair
of object and query embeddings.
    ...

    Args:
        ...
    Returns:
        ...

    Example:
        >>> ..."""
    ...

    ## CodeLLaMa-34B-Python completion

    obj_type = type(obj)
    if obj_type is property:
        obj = obj.fget

```

```

    if not hasattr(obj, '__module__') or not hasattr(obj, '
__qualname__'):
        return None
    module = inspect.getmodule(obj)
    if module is None:
        return None
    if not hasattr(module, '__file__'):
        return None
    if obj_type is type:
        obj_source = obj.__qualname__
    elif obj_type is types.ModuleType:
        obj_source = obj.__name__
    else:
        obj_source = f'{obj.__module__}.{obj.__qualname__}'
    return f'File "{module.__file__}", line {inspect.getsourcelines(
obj)[1]}.replace('\n', '/')

```

Listing 2.1: CODELLAMA-34B repeats the code from .

2.9 Acknowledgement

This work was supported in part by NSF grants CCF:1900968, CCF:1908870 and by SKY Lab industrial sponsors and affiliates Astronomer, Google, IBM, Intel, Lacework, Microsoft, Mohamed Bin Zayed University of Artificial Intelligence, Nexla, Samsung SDS, Uber, and VMware. A. Gu is supported by the NSF Graduate Research Fellowship under Grant No. 2141064. A. Solar-Lezama is supported by the NSF and Intel Corporation through NSF Grant CCF:2217064. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and do not necessarily reflect the position of the sponsors.

Part II

LLM Agents Planning

Chapter 3

LLM Planning from Learning Mistakes

In-context learning (ICL, also known as *few-shot prompting*) has been the standard method of adapting LLMs to downstream tasks, by learning from a few input-output examples. Nonetheless, all ICL-based approaches only learn from *correct* input-output pairs. In this paper, we revisit this paradigm, by learning *more* from the few given input-output examples. We introduce Learning Principles (LEAP): First, we intentionally induce the model to *make mistakes* on these few examples; then the model itself reflects on these mistakes, and learn explicit task-specific “principles” from them without any human supervision, which help solve similar problems and avoid common mistakes; finally, we prompt the model to answer unseen test questions using the original few-shot examples and these learned general principles. We evaluate LEAP on a wide range of benchmarks, including multi-hop question answering (Hotpot QA), textual QA (DROP), Big-Bench Hard reasoning, and math problems (GSM8K and MATH); in all these benchmarks, LEAP improves the strongest available LLMs such as GPT-3.5-turbo, GPT-4, GPT-4-turbo and Claude-2.1. For example, LEAP improves over the standard few-shot prompting using GPT-4 by 7.5% in DROP, and by 3.3% in HotpotQA. Importantly, LEAP does not require any more input or examples than the standard few-shot prompting settings.

The rise of large language models [LLMs; 145, 33, 236, 91, 6, 189, 188] that are too costly to finetune for downstream tasks has led to the growing popularity of *in-context learning* (ICL), also known as few-shot prompting [18, 102, 209]. In in-context learning, the LLM is provided with a few (e.g., three) input-output task-specific examples in its prompt, along with an unseen test input. Using this emergent ability [207], the LLM is then expected to generate output for the test input. The LLM generates this output by implicitly learning the task from the few given examples, at inference time.

ICL was shown to be extremely effective and data-efficient across a variety of tasks and domains [119, 3, 98, 111], mainly because it allows for downstream task-adaptation without training. Further, ICL enables generalization using only a few annotated examples. Nonetheless, all ICL-based approaches only learn from *correct* input-output pairs [120]; in

Learned Principle	Benchmark
When solving a problem involving multiple sources of income and expenses, it is crucial to keep track of each component separately and calculate the total accurately.	GSM8K
When simplifying complex numbers raised to powers, it is important to remember the following rules: (1) $i^2 = -1$ (2) $i^3 = -i$ (3) $i^4 = 1$ (4) $i^{-1} = \frac{1}{i} = -i$	MATH
Perform calculations using the full precision available and only round as a final step , if necessary.	DROP
When answering questions about commonalities between two entities, it is important to consider all relevant aspects and not just the most obvious or prominent one.	HotpotQA
Double negation, as in ‘not not’ , cancels out and returns the original value.	Boolean Expressions (BBH)
Sarcasm often involves saying the opposite of what is meant ... Paying attention to the incongruity between the literal meaning of the words and the intended meaning can help in accurately identifying sarcasm.	Snarks (BBH)

Figure 3.1: Examples for learned principles using LEAP, with key idea of each principle highlighted.

contrast, learning from *mistakes* is a foundational aspect of the way humans learn [46, 31, 45]. In fact, learning from mistakes is also a fundamental concept of machine learning, which goes back to classical work such as [211, 154] and [121], but is not utilized by current ICL and prompting methods.

Differently from machines, humans can often *verbalize* their mistakes and articulate ex-

Input

Q: Evaluate $\log_{\sqrt{5}} 125\sqrt{5}$

Few-shot Chain-of-Thought (Wei et al., 2022)

Model Output

A: We can write $125\sqrt{5}$ as $5^3 \cdot 5^{1/2} = 5^{7/2}$.
 The logarithm base $\sqrt{5}$ of $5^{7/2}$ is the exponent we need to raise $\sqrt{5}$ to get $5^{7/2}$.
 Since $\sqrt{5} = 5^{1/2}$, we have $(5^{1/2})^{2n} = 5^n$ for any integer n .
 So, we need to find n such that $5^n = 5^{7/2}$
 Clearly, $n = 7/2$.

Therefore, $\log_{\sqrt{5}} 125\sqrt{5} = \boxed{\frac{7}{2}}$ ❌

Learning Principles (LEAP; this work)

Generating Mistakes Intentionally (performed once)

Q: Evaluate $\log_{1/3} 9$
 A: To evaluate $\log_{1/3} 9$, you want to find the exponent to which $\frac{1}{3}$ must be raised to get 9 as the result.

...
 $\frac{1^x}{(1/3)^x} = \frac{3^2}{1}$
 So $\log_{1/3} 9 = \boxed{2}$ ❌

Learning Principles (performed once)

It is crucial to accurately calculate the powers of the base in logarithm problems to correctly determine the exponent. Additionally understanding the definition of a logarithm is key: the logarithm base b of a number x is the exponent to which b must be raised to get x.

Inference with Learned Principles:

A: First we can write $125\sqrt{5}$ as $5^3 \cdot 5^{1/2} = 5^{7/2}$.
 The logarithm base $\sqrt{5}$ of $5^{7/2}$ is asking to which power we must raise $\sqrt{5}$ to get $5^{7/2}$.
 Since $\sqrt{5} = 5^{1/2}$, we can see that $(5^{1/2})^7 = 5^{7/2}$.

Therefore, $\log_{\sqrt{5}} 125\sqrt{5} = \boxed{7}$ ✅

Figure 3.2: An illustration of LEAP: Given a few input-output examples, Chain-of-Thought (left) generates a response to the test question by directly learning from the (correct) examples. In contrast, Learning Principles (LEAP, right) first (a) generates *mistaken* zero-shot Chain-of-Thought response for each given input-output example by sampling with a non-zero temperature; (b) generates explicit *principles* by providing the LLM with the mistaken CoT along with the correct output; and finally (c) generates a response to the test question, by providing the LLM with both the given input-output examples *and the learned principles*. Note that steps (a) and (b) are performed once per task.

licit *principles*, or “lessons”; these principles can further help humans avoid these and similar mistakes in the future. Inspired by this ability and the benefits it provides, we

```

Question: {question}

Generated Reasoning: {response}

Generated Answer: {generated_answer}

Correct Reasoning: {correct_reasoning}

Correct Answer: {correct_answer}

Instruction: Conduct a thorough analysis of the generated answer in comparison to the
correct answer. Also observe how the generated reasoning differs from the correct
reasoning. Identify any discrepancies, misunderstandings, or errors. Provide clear
insights, principles, or guidelines that can be derived from this analysis to improve
future responses. We are not focused on this one data point, but rather on the general
principle.

Reasoning: <discuss why the generated answer is wrong>
Insights: <what principle should be looked at carefully to improve the performance in
the future>

```

Figure 3.3: LEAP prompt to help LLM evaluate its own generated reasoning and answers, contrasting them with the correct reasoning and answers. The LLM is prompted to identify errors in its reasoning and extract key insights for improvement. This figure specifically represents the ‘GenerateExplanation’ step in the LEAP algorithm (Chapter [1](#)).

propose Learning Principles (LEAP): A prompting approach for learning principles from mistakes, and then conditioning on these principles when responding to *other* inputs. Instead of providing the model *only* with the given (correct) few-shot examples, LEAP begins by (a) inducing the LLM to *make mistakes* on each of these given few-shot inputs, in a zero-shot fashion, by sampling outputs with a non-zero temperature; (b) generating explicit *principles* by providing the same LLM with the mistaken outputs along with the correct output; and finally (c) generating a response to the test question as in standard few-shot ICL, while providing the LLM with both the given input-output examples *and the learned principles*. LEAP is illustrated in Chapter [3.2](#); examples for some of the learned principles are provided in Chapter [3.1](#).

Crucially, LEAP does not require any more input than the standard few-shot prompting settings. Further, the principle learning step is applied *once* for a task, and then the same learned principles are used for the entire test set. If the few-shot examples are given as Chain-of-Thought [CoT; [206](#)], LEAP is applied seamlessly by generating *mistaken* chains-of-thought, contrasting them with the correct chains-of-thought, and generating principles.

We demonstrate the effectiveness of LEAP across a wide range of reasoning benchmarks,

Algorithm 1 LEAP Algorithm

Require: Few-shot examples $\mathcal{P} = \{\langle x_i, y_i \rangle\}_{i=1}^k$, a pretrained LLM, number of outputs per input n , high-temperature setting T

- 1: **for** each input-output pair $\langle x_i, y_i \rangle$ in \mathcal{P} **do**
- 2: $\mathcal{S}_i \leftarrow \text{ZeroShotCoT}(LLM, x_i, n, T)$ \triangleright Generate solutions using zero-shot chain-of-thought prompting
- 3: $\mathcal{M}_i \leftarrow \{(x_i, y_i, \hat{y}_i^j) \in \mathcal{S}_i : \hat{a}_i^j \neq a_i^j\}$ \triangleright Identify incorrect solutions
- 4: **for** each x_i, y_i, \hat{y}_i in \mathcal{M}_i **do**
- 5: $\mathcal{L}_{\text{LOW-LEVEL},i} \leftarrow \text{GenerateLowLevelPrinciples}(LLM, x_i, \hat{y}_i, y_i)$ \triangleright Generate principles for each mistake
- 6: **end for**
- 7: **end for**
- 8: $\mathcal{L}_{\text{LOW-LEVEL}} \leftarrow \bigcup_{i=1}^k \mathcal{L}_{\text{LOW-LEVEL},i}$ \triangleright Aggregate low-level principles
- 9: $\mathcal{L}_{\text{HIGH-LEVEL}} \leftarrow \text{GenerateHighLevelPrinciples}(LLM, \mathcal{L}_{\text{LOW-LEVEL}})$ \triangleright Generate high-level principles
- 10: $p_{\text{LOW-LEVEL}} \leftarrow \text{Concatenate}(\mathcal{L}_{\text{LOW-LEVEL}}, \mathcal{P})$ \triangleright Create enhanced prompt with low-level principles
- 11: $p_{\text{HIGH-LEVEL}} \leftarrow \text{Concatenate}(\mathcal{L}_{\text{HIGH-LEVEL}}, \mathcal{P})$ \triangleright Create enhanced prompt with high-level principles
- 12: **return** $p_{\text{LOW-LEVEL}}, p_{\text{HIGH-LEVEL}}$

including mathematical reasoning in GSM8K [35] and MATH [56], multi-hop question answering tasks in HotpotQA [223], textual reasoning in DROP [43], and 27 Big-Bench Hard [183] tasks. LEAP outperforms the standard practice of few-shot prompting using strong models such as GPT-3.5-turbo, GPT-4, GPT-4-turbo and Claude-2.1, with and without Chain-of-Thought. For example, LEAP improves over the standard few-shot CoT using GPT-4 by 7.5% in DROP, by 3.3% in HotpotQA, and on 16 tasks in Big-Bench Hard. These results suggest that LEAP revolutionizes the “traditional” concept of few-shot ICL, by leveraging the recently emerged abilities of LLMs to follow instructions [122, 208, 132] and explain mistakes given the correct answer or feedback [116, 29].

3.1 Background: In-Context Learning

In-context learning, also known as few-shot prompting, uses a few (typically between 3 and 8) input-output task-specific examples for teaching a pre-trained LLM to solve a downstream task [18, 98, 102]. These input-output pairs $\mathcal{P} = \{\langle x_i, y_i \rangle\}_{i=1}^k$ are concatenated to form a prompt $p = \langle x_1 \cdot y_1 \rangle \oplus \langle x_2 \cdot y_2 \rangle \oplus \dots \oplus \langle x_k \cdot y_k \rangle$, where “ \cdot ” denotes the concatenation of each input with its corresponding output, and “ \oplus ” denotes the sequential combination of these pairs. Then, every new test input x_{test} is appended to this prompt, resulting in $p \oplus \langle x_{\text{test}} \cdot \cdot$, which is then provided to the LLM. The LLM completes this input, and generates the corresponding output \hat{y}_{test} .

Typically, the answer y additionally contains an explanation (or *thought*) for deriving the

answer [206]. In these settings, each solution y contains a thought t and the final answer a , forming: $y_i = t_i \cdot a_i$, and the LLM is expected to generate the test-thought before generating the final test-answer: $t_{test} \cdot a_{test}$. In this work, we focus on this *chain-of-thought* setup, because of its widely demonstrated effectiveness for reasoning tasks [200, 206, 244, 202].

3.2 LEAP: Learning Principles from Mistakes

In LEAP, our goal is to learn general *principles* that help the model avoid potential mistakes in a downstream task. Given the few-shot examples $\mathcal{P} = \{\langle x_i, y_i \rangle\}_{i=1}^k$ for a certain task, we start by generating *mistakes* to the few given examples.

Generating Mistakes For each input-output pair $\langle x_i, y_i \rangle \in \mathcal{P}$, we generate a diverse set of solutions in a zero-shot fashion. Specifically, we follow [78], and create a zero-shot chain-of-thought prompt using x_i and the phrase *Think step-by-step*. For each input x_i , we sample $n = 15$ outputs with a non-zero temperature, producing a varied set of potential solutions $\{\hat{y}_i^j\}_{j=1}^n$ for each example x_i , such that $\hat{y}_i^j = \hat{t}_i^j \cdot \hat{a}_i^j$, where \hat{t}_i^j represents the intermediate reasoning steps (thoughts), and \hat{a}_i^j denotes the final answer.

We identify *incorrect* solutions by comparing each \hat{a}_i^j with the ground-truth answer a_i (which is given as part of the task), forming a set of mistakes for each $\langle x_i, y_i \rangle \in \mathcal{P}$ pair: $\mathcal{M}_i = \left\{ \langle x_i, y_i, \hat{y}_i^j \rangle \right\}_{j=1}^n$, such that $\forall j : \hat{a}_i^j \neq a_i$.

Generating Low-Level Principles Then, for each such mistake in \mathcal{M} , we prompt the LLM to generate a natural language explanation of the mistake. In this step, the LLM is provided with the ground truth answer y_i , to articulate the rationale behind the solution’s inaccuracy, as illustrated in Chapter 3.3. The insights of how to avoid these mistakes are aggregated across all examples to form a set of low-level principles, denoted as $\mathcal{L}_{\text{LOW-LEVEL}}$.

Generating High-Level Principles Subsequently, we use the LLM to condense the low-level principles into approximately 5 key bullet points, thus creating *high-level* principles, denoted as $\mathcal{L}_{\text{HIGH-LEVEL}}$. The motivation for this step is generating *generic*, example-agnostic, principles for solving the downstream task, that do not depend on mistakes made for any specific example.

Final Inference on Unseen Examples These principles, either low or high-level, are then appended to the prompt p , forming enhanced prompts: $p_{\text{LOW-LEVEL}} = \mathcal{L}_{\text{LOW-LEVEL}} \oplus p$ for low-level feedback, and $p_{\text{HIGH-LEVEL}} = \mathcal{L}_{\text{HIGH-LEVEL}} \oplus p$ for high-level feedback, where p is the prompt constructed using the standard concatenation of the few-shot examples, as described in Chapter 7.2.

Finally, we use the enhanced prompts $p_{\text{LOW-LEVEL}}$ or $p_{\text{HIGH-LEVEL}}$ for answering all unseen examples in the test set. In effect, the one-time process of generating principles helps learn and *articulate* insights from the model’s previous mistakes, potentially improving its accuracy and reasoning capabilities in future responses for *other* inputs.

The complete algorithm is summarized in Chapter 1. Although different LLMs can be used for each step, in all our experiments, we fixed the LLM across all steps: generating

Table 3.1: **Textual Reasoning results:** Accuracy in textual reasoning benchmarks. The best approach for each base LLM in each dataset is in **bold**; the second-best approach is underlined. We see a good performance boost by adopting LEAP with high-level feedback and low-level feedback. Almost all the models can benefit from the principles learned and fix their previous mistakes.

		GPT-3.5-turbo	GPT-4	GPT-4-turbo	Gemini Pro
HotpotQA	Few-shot CoT	29.10	36.35	<u>38.10</u>	28.25
	LEAP _{LOW-LEVEL}	32.60	<u>39.30</u>	37.85	23.70
	LEAP _{HIGH-LEVEL}	<u>30.35</u>	39.65	38.75	<u>25.50</u>
DROP	Few-shot CoT	<u>63.20</u>	72.05	<u>83.40</u>	64.60
	LEAP _{LOW-LEVEL}	63.35	79.55	83.60	<u>67.15</u>
	LEAP _{HIGH-LEVEL}	63.00	<u>78.60</u>	80.00	67.60

mistakes, generating principles from those mistakes, and testing using these self-generated principles. This ensures that the difference in results arises only from LEAP rather than any kind of model ensemble or teaching.

3.3 Evaluation

We evaluated LEAP across various reasoning tasks, including HotpotQA [223], DROP [43], MATH [56], GSM8K [35], and Big-Bench Hard [183]. We follow the standard few-shot Chain-of-Thought (CoT); we apply LEAP on top of few-shot CoT using the exact same number of labeled examples. Importantly, principles are generated *once* for every LLM and benchmark.

Experiment Setup

Compared Approaches Our baseline is the standard Few-shot prompting with CoT. We compare this baseline with two variants of our proposed approach LEAP:

- **LEAP_{high-level}** is our proposed approach, as described in Chapter 3.2.
- **LEAP_{low-level}** is similar to LEAP_{HIGH-LEVEL}, except that we skip the step of “Generating High-Level Principles”, and test the models on the downstream task using the few-shot examples *and the low-level principles*, using $\mathcal{L}_{\text{LOW-LEVEL}}$ rather than $\mathcal{L}_{\text{HIGH-LEVEL}}$ (Chapter 3.2).

Benchmarks We used diverse reasoning benchmarks:

Table 3.2: **Math Reasoning Results:** Accuracy in MATH and GSM8K. The best approach for each LLM and base task is in **bold**; the second-best approach is underlined. LEAP_{HIGH-LEVEL} and LEAP_{LOW-LEVEL} in both GSM8K and MATH datasets consistently improve the performance over the CoT baseline. We also observe that the learned mathematical principles can be generalized to different test questions.

		GPT-3.5-turbo	GPT-4	Claude-2	Gemini Pro
GSM8K	Few-shot CoT	76.4	93.6	84.3	77.8
	LEAP _{LOW-LEVEL}	77.4	94.2	82.7	77.3
	LEAP _{HIGH-LEVEL}	<u>76.6</u>	<u>93.8</u>	<u>83.8</u>	78.7
MATH	Few-shot CoT	55.6	63.5	<u>43.2</u>	31.1
	LEAP _{LOW-LEVEL}	<u>56.1</u>	64.5	42.7	29.7
	LEAP _{HIGH-LEVEL}	56.5	<u>64.0</u>	43.4	30.3

- **Textual Reasoning:** HotpotQA [222] is a question-answering dataset of computational questions that require multi-hop reasoning. DROP [44] is a reading comprehension dataset that requires numerical and logical reasoning over textual paragraphs; for evaluation, we randomly sampled 2000 questions from its dev set.
- **Mathematical Reasoning:** GSM8K [35] comprises a test set of 1,319 diverse grade school math word problems, curated by human problem writers. In MATH [56], there are 5,000 diverse examples consisting of problems from mathematics competitions. These are the two most common mathematical reasoning benchmarks.
- **Big-Bench Hard [183]:** contains 27 challenging tasks that test various reasoning capabilities of LLMs. We repeated every run 3 times with a temperature of zero and report the average.¹

Models We evaluated LEAP across a wide range of base models, including GPT-3.5-turbo (version -0613), GPT-4 (version -0613), GPT-4-turbo (version -1106), Claude-2.1, and Gemini Pro [186].

Few-shot examples In Big-Bench Hard, we used the textscCoT prompts from [183] with three given examples (3-shot) in each benchmark. In MATH and GSM8K, we used the standard training examples from each dataset, with 3 examples for each. In DROP, we used 3 given examples that we adopted from Least-to-Most [244]. In HotpotQA we used 6 examples from ReAct [224] (“closed-book”). Importantly, in each benchmark, the exact same few-shot examples were used across all evaluated approaches, including the baseline and LEAP.

¹Internal non-determinism causes different outputs even with a temperature of zero, but the variance was negligible.

Textual Reasoning Results

Chapter 3.1 shows the results on DROP and HotpotQA. As shown, LEAP improves over the Few-shot CoT baseline by up to 3.5% on Hotpot QA and 7.5% on DROP. In HotpotQA, GPT-3.5-Turbo and GPT-4 are consistently improved when using LEAP. In DROP, GPT-4 is significantly improved by LEAP (an absolute gain of 7.5%), Gemini Pro is improved by 3%, while the improvement for GPT-3.5-Turbo and GPT-4-turbo are more minor.

In most tasks and base models, both $\text{LEAP}_{\text{LOW-LEVEL}}$ and $\text{LEAP}_{\text{HIGH-LEVEL}}$ improve over the Few-shot CoT baseline. The only case where the Few-shot CoT baseline performs better than both $\text{LEAP}_{\text{LOW-LEVEL}}$ and $\text{LEAP}_{\text{HIGH-LEVEL}}$ is in HotpotQA using Gemini Pro. Observing the low-level principles that Gemini Pro learned in HotpotQA (Chapter B.39), we believe that the learned principles are correct and useful, but they are overly focused on the examples they were generated for, more verbose, and similar to each other. These hinder the principles (and the high-level principles generated from them) from generalizing to other examples. For *zero-shot* prompting using principles learned from few examples, additional results are shown in Chapter B.3.

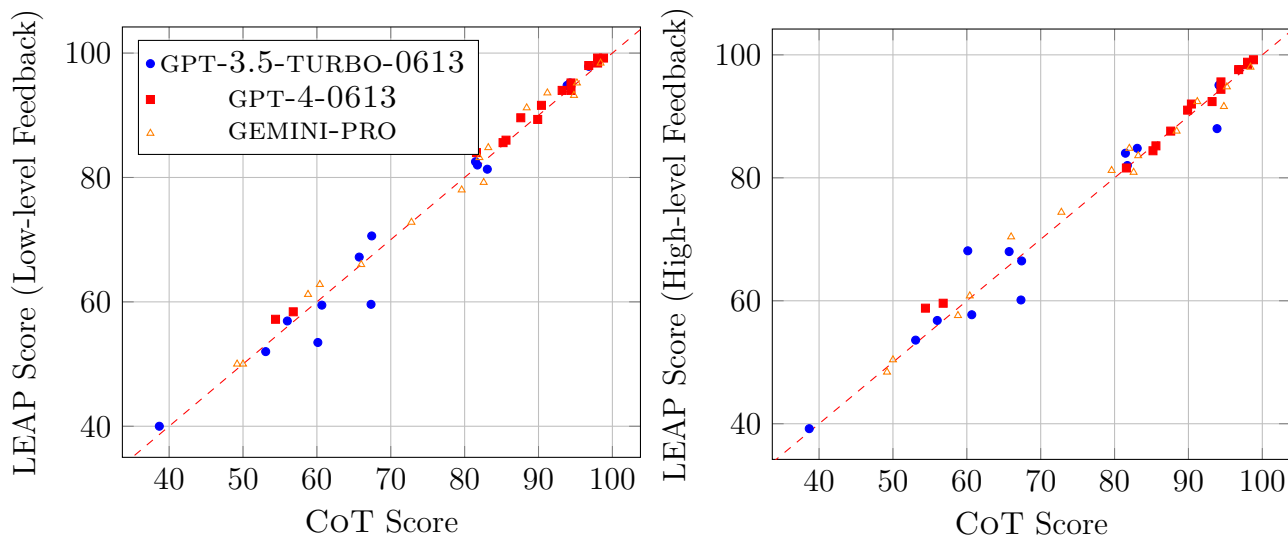


Figure 3.4: Accuracy in BBH tasks, across GPT-3.5-TURBO-0613, GPT-4-0613, and GEMINI-PRO. The figure presents the results using a scatter plot, where the y-axis represents scores achieved with LEAP, and the x-axis represents the baseline scores from CoT. Each task is represented by a point on the plot, with different shapes assigned to different models for easy distinction. Tasks above the $y = x$ line are those where LEAP leads to an improvement in performance. Chapter B.5 shows the detailed results for all 27 Big-Bench hard tasks. We find that in 37 out of 42 combinations of task and LLM, one of $\text{LEAP}_{\text{LOW-LEVEL}}$ or $\text{LEAP}_{\text{HIGH-LEVEL}}$ outperforms the baseline Few-shot CoT.

Table 3.3: **Llama-2-chat-70B Results:** Accuracy in MATH and GSM8K using Llama-2-chat-70B as the base LLM, while generating the principles either with Llama-2-chat-70B or GPT-4. Even when the principles are generated by GPT-4 (GPT-4 Critic), Llama-2-chat-70B does not leverage the learned principles, and does not improve over the baseline Few-shot CoT.

Llama-2-chat-70B as the base model, with:		Llama-2 Critic	GPT-4 Critic
	Few-shot CoT	52.5	52.5
GSM8K	LEAP _{LOW-LEVEL}	50.6	49.8
	LEAP _{HIGH-LEVEL}	47.0	51.0
	Few-shot CoT	16.2	16.2
MATH	LEAP _{LOW-LEVEL}	13.8	12.9
	LEAP _{HIGH-LEVEL}	14.2	13.8

Math Reasoning Results

Chapter 3.2 shows the results on MATH and GSM8K. As shown, in GPT-3.5-turbo and GPT-4, both LEAP_{LOW-LEVEL} and LEAP_{HIGH-LEVEL} outperform the Few-shot CoT baseline. Claude-2 shows inconclusive results: in GSM8K, Few-shot CoT performs better than LEAP; in MATH, LEAP_{HIGH-LEVEL} achieves slightly higher accuracy than Few-shot CoT. Gemini Pro also shows inconclusive results, where both LEAP_{LOW-LEVEL} and LEAP_{HIGH-LEVEL} outperform the baseline in GSM8K, but perform slightly worse in MATH. Additional results, for zero-shot prompting using principles learned from few examples, are shown in Chapter B.4.

Does leap work with open-source models? In our preliminary experiments with open-source models, LEAP did not improve over the few-shot CoT baseline. While the open-source models *did* produce useful principles, the open-source models did not follow these principles at test time.

Chapter 3.3 shows some of these experiments with Llama-2-chat-70B: we used Llama-2-chat-70B as the base model, but generated the principles using either Llama-2-chat-70B (“LLama-2 Critic”) or with GPT-4 (“GPT-4 Critic”). As shown, even when the principles are generated by GPT-4 (which we assume to be useful), Llama-2-chat-70B does not manage to leverage them to generate better final responses with either LEAP_{LOW-LEVEL} or LEAP_{HIGH-LEVEL}.

In general, we believe that LEAP requires a base LLM with strong enough instruction following and reflection capabilities; we believe that, unfortunately, most open-source models are not as powerful as proprietary models yet.

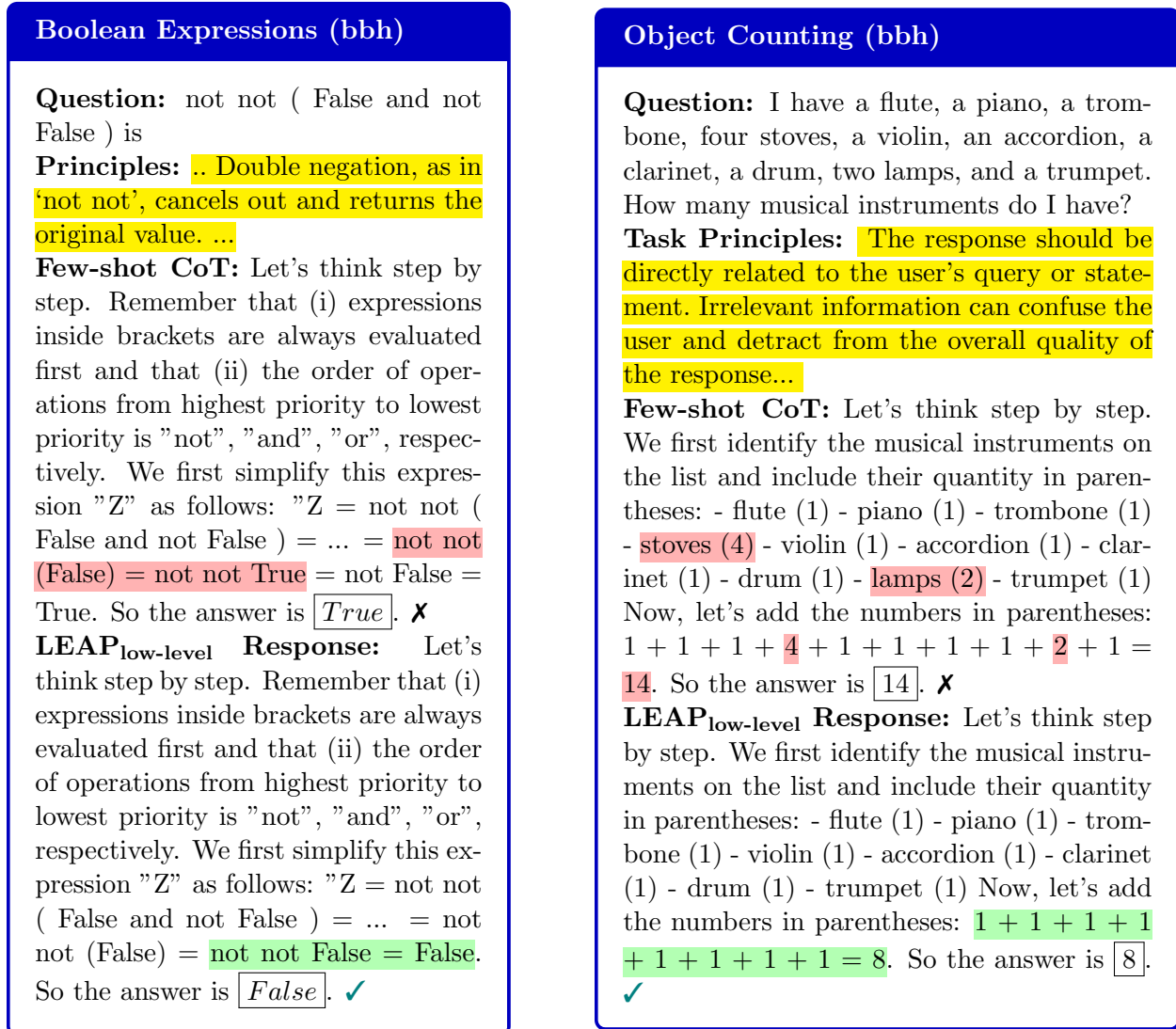


Figure 3.5: Examples from the Boolean Expressions (left) and Object counting (right) tasks from BBH. The learned principle is highlighted in yellow, the mistaken step of the baseline is highlighted in red, and the correct use of the principle by LEAP is highlighted in green. This demonstrates how the learned principles guide LEAP in generating a better answer.

Big-Bench Hard Results

Chapter 3.4 shows the results on selected BBH tasks; results for the rest of the BBH tasks are shown in Chapter B.6 in Chapter B.5. We selected tasks to Chapter B.5 where the difference in results was the most meaningful: there were some tasks where GPT-4-0613 achieved 100% accuracy with *any* of the approaches, including both LEAP and the baseline.

As shown in Chapter B.5, in 37 out of 42 combinations of task and LLM, one of

LEAP_{LOW-LEVEL} or LEAP_{HIGH-LEVEL} outperforms the baseline Few-shot CoT. In 24 of the cases, both LEAP_{LOW-LEVEL} and LEAP_{HIGH-LEVEL} outperform the Few-shot CoT baseline. In two cases, the Few-shot CoT baseline performs equally to LEAP_{LOW-LEVEL}; and in 3 cases the Few-shot baseline performs better than both LEAP_{LOW-LEVEL} and LEAP_{HIGH-LEVEL}.

We could not identify any particular pattern as to which method should be used: LEAP_{LOW-LEVEL} or LEAP_{HIGH-LEVEL}; it seems that this depends on the reasoning complexity of the task, the diversity across questions in the benchmark, and the quality of the principles that the LLM had managed to generate from the 3 given examples for that task. We thus suggest that in real-life scenarios, both approaches should be tested, and selected using a validation set. Additional results, for zero-shot prompting using principles learned from a few examples, are shown in Chapter B.5.

3.4 Qualitative Analysis

Chapter 3.5 shows examples of questions from BBH, along with the task’s learned principles, the baseline few-shot CoT response, and the LEAP_{LOW-LEVEL} response, all generated by GPT-3.5-turbo. The learned principle in each case is highlighted, along with the mistake in the CoT output and the correct reasoning generated by LEAP.

Additional examples are shown in Chapter B.6. The actual learned principles for some of the tasks and LLMs are provided in Tables B.7-B.41 in Chapter B.7.

3.5 Related Works

This section focuses on the few works most related to ours. For a complete survey of relevant papers, see [242, 142, 72, 217, 240].

Comparison to [116] A related work to ours is Self-Refine [116], which, similarly to LEAP, uses the LLM to reflect on its own previously generated outputs. However, a significant difference from Self-Refine is that applying Self-Refine to a test example requires feedback *for that specific test example*; contrarily, in this work, we show that principles that were learned from some examples *generalize to other* test examples which we do not have feedback for. Further, Self-Refine uses “intrinsic self-correction”, without any external feedback; as found by recent work [60, 192], such intrinsic self-correction does not benefit *reasoning* tasks. In contrast, in this work, we show that the external signal provided by as few as 3 examples is sufficient for effective self-reflection in reasoning if these are used to articulate explicit learned principles. Another related work is MemPrompt [115], which stores feedback on mistakes in a memory, and retrieves relevant feedback for a new query; in contrast, LEAP is a one-time process that does not require continuous storage and retrieval.

Comparison to [221] Optimization by PROMpting (OPRO) [221] uses LLMs to improve zero-shot prompts, and both OPRO and LEAP can be thought of as ways of improving a prompt for a specific task. However, applying OPRO for prompt optimization requires a

training set, which can be as large as ~ 260 examples for GSM8K; in contrast, LEAP requires only **3** labeled examples. Further, OPRO calls the LLM for each of these ~ 260 examples in each step; while the number of steps can be as low as 6, it can also be as high as 107. This results in between ~ 1500 calls and up to $107 \times 260 \approx 27,000$ inference calls to the LLM. In contrast, LEAP uses only about **34** inference calls: 3 calls to generate mistakes; one call for each mistake to generate low-level principles, which results in about 30 calls; and a final call to generate high-level principles. Moreover, like OPRO, LEAP can also improve zero-shot prompting by learning principles from a few examples and applying them in a zero-shot fashion, as we show in Chapter [B.3](#), Chapter [B.4](#), and Chapter [B.5](#). For example, Zero-shot-LEAP improves the strong GPT-4 model by 1% on GSM8K (Chapter [B.4](#)). Similarly, EvoPrompt [\[53\]](#) requires around **10,000** LLM inference calls, which is also several orders of magnitude more than LEAP. Other related approaches are AutoPrompt [\[171\]](#), STaR [\[231\]](#), LMSI [\[59\]](#), and Self-Align [\[180\]](#) but these are based on *training* the LLM, which is often either inaccessible or computationally infeasible.

Comparison to [\[30\]](#) Contrastive Chain-of-Thought [\[30\]](#) shares a similar motivation with our work: learning from negative in-context examples instead of learning only from positive examples. However, when we reproduced their results, we found that simple improvements to the post-processing code of extracting the final answer out of the LLM’s raw output led to differences of more than 10 absolute points from their reported results in GSM8K, which made the proposed Contrastive CoT approach perform similarly or worse than the baseline few-shot CoT. We thus believe that generating *explicit* principles, as in LEAP, is the key ingredient in learning from negative in-context examples.

3.6 Conclusion

In this paper, we introduce Learning Principles (LEAP), a novel approach that allows LLMs to learn *more* out of given few-shot examples, by intentionally making mistakes on these examples; reflecting on the mistakes; and finally articulating explicit task-specific principles, which helps avoid similar mistakes in the future. LEAP requires exactly the same number of labeled examples as few-shot prompting, and allows improving a variety of strong LLMs (GPT-3.5-turbo, GPT-4, GPT-4-turbo and Gemini Pro) across a broad range of reasoning tasks (DROP, HotpotQA, GSM8K, MATH, and Big-Bench Hard). We believe that LEAP unlocks new possibilities from learning in the traditional concept of few-shot in-context learning, by learning from *mistakes*, rather than learning from positive examples only.

Chapter 4

TEMPERA: Test-Time Prompt Editing Via Reinforcement Learning

Careful prompt design is critical to the use of large language models in zero-shot or few-shot learning. As a consequence, there is a growing interest in automated methods to design optimal prompts. In this work, we propose **TEst-tiMe Prompt Editing using Reinforcement leArning** (TEMPERA). In contrast to prior prompt generation methods, TEMPERA can efficiently leverage prior knowledge, is adaptive to different queries, and provides an interpretable prompt for every query. To achieve this, we design a novel action space that allows flexible editing of the initial prompts covering a comprehensive set of commonly-used components like instructions, few-shot exemplars, and verbalizers. The proposed method achieves significant gains compared with recent SoTA approaches like prompt tuning, AutoPrompt, and RLPrompt, across a variety of tasks, including sentiment analysis, topic classification, natural language inference, and reading comprehension. Our method achieves 5.33x on average improvement in sample efficiency when compared to the traditional fine-tuning methods. Our code is available at <https://github.com/tianjunz/TEMPERA>.

4.1 Introduction

With the recent advances in pre-training large language models [18, 47, 148, 33], prompting, or in-context learning provides a data-efficient framework for performing NLU [90, 171, 50]. Such methods achieve impressive zero-shot and few-shot performance in many downstream tasks.

However, the prompt often has to be carefully tuned to achieve consistent performance for each task [113]. For example, prompt tuning aims to optimize a continuous prefix embedding via gradient descent and directly takes generated output from the frozen pre-trained language model [86, 104, 105]. On the contrary, discrete prompt optimization focuses on constructing meaningful instructions, in-context exemplars and verbalizers [18, 50]. Prior work often performs black-box optimization or applies RL-based methods for direct generation [37,

[179, 141]. Recent works in the prompt tuning field have shown that, performing instance-dependent prompt tuning [216, 67] can improve the performance of some downstream tasks. The corresponding concept in the discrete prompt optimization domain is intriguing since it allows users to provide different instructions for different inputs and task. Unlike prompt tuning, such instructions can be more human interpretable. However, finding such query-dependent prompts is often overlooked and is not feasible given the inefficiency of black-box optimization.

In this paper, we investigate the importance of providing query-dependent discrete prompts and demonstrate how this can be achieved via efficient search. To this end, we propose the concept of *test-time editing* through reinforcement learning (RL) that allows the agent to perform different editing techniques at *test time* to construct query-dependent prompts efficiently.

We formulate discrete prompt optimization as an RL problem by sequentially editing an initial prompt, which only requires high-level guidance on which part to edit and what tools to use. Different from prior work, this formulation strikes a good balance between human prior knowledge, flexibility, feasibility and interpretability. The method allows easy incorporation of human knowledge since one can provide a manually chosen initial prompt and allow RL to perform editing on it. It also achieves a balance between search flexibility and feasibility because by enabling different editing techniques, the prompt can be transformed to very different forms but the search space is more feasible compared to direct generation. The final prompt is also more interpretable since the editing tools we adopted usually do not change the semantic meaning of the sentence.

To summarize, we propose to construct query-dependent prompts through test-time editing and formulate this as an RL problem. We carefully design the action space, enabling the agent to flexibly edit the instructions, in-context exemplars and verbalizers. To better train the RL agent, we propose using the score difference between consecutive prompts before and after editing as rewards and developing a set of techniques that help improve the final performance (e.g., reward normalization). We also adopt an attention-based policy architecture to attend over possible candidates or design choices, and show this can be effective for RL training.

Following the standard few-shot text classification setting, we benchmark our algorithm extensively on multiple tasks (including those from GLUE [195] and SuperGLUE [196]). We show that TEMPERA can achieve SoTA performance (e.g., 1.8% better in SST-2 and 3.9% better in CR) compared to few-shot finetuning, prompt tuning and discrete prompt optimization. We also show that TEMPERA is on 4x more data efficient (over the average of 4 tasks SST2, MR, AG News and RTE) compared with traditional finetuning methods (Figure 4.1). In addition, we perform extensive ablations on different aspects of the proposed algorithm. We demonstrate that TEMPERA is robust to the prompt pool size and the number of few-shot exemplars.

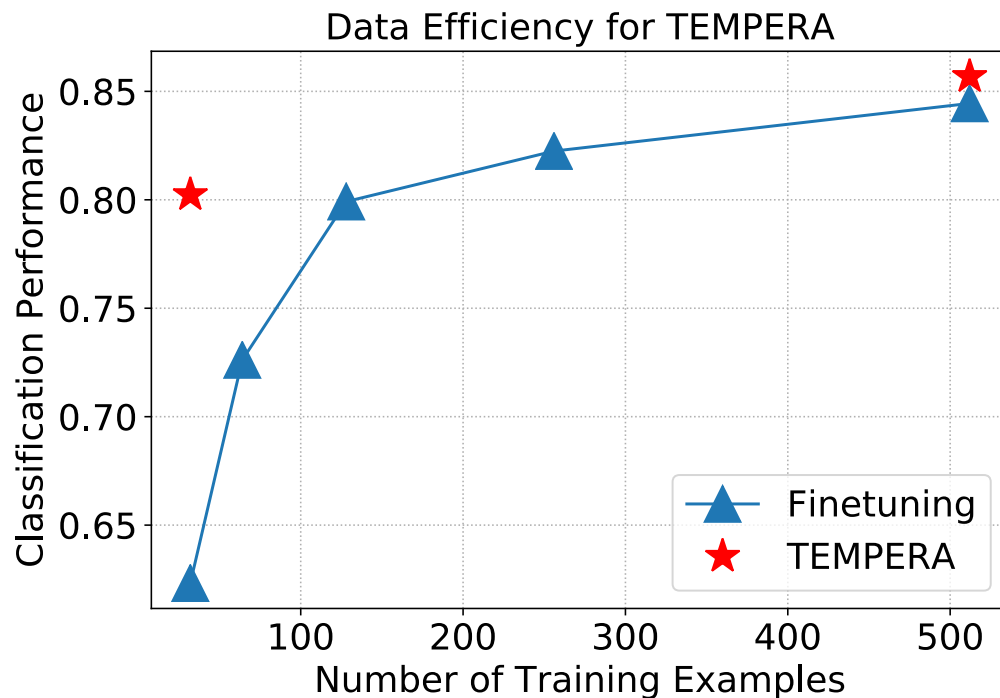


Figure 4.1: **Data Efficiency for TEMPERA:** We compare the data efficiency of TEMPERA and standard fine-tuning in a few-shot setting. Results are averaged across four tasks: SST2, AG News, RTE and MR. It shows that our method achieves comparable performance using 4x fewer examples.

4.2 Related Work

Prompting in language models and sensitivity to prompts. Recent research has shown that as language models scale up, new capabilities could be unlocked such as in-context learning [18], where the language model is prompted with a few in-context demonstrations and learns to perform a certain task in a sample-efficient way. However, several works have studied the in-context learning ability more closely and found that the task performance can be highly sensitive to how the in-context prompt is written. For example, [112] found that the prompt order can have a large effect on the final task performance; [241] show that the choice of prompt format, training examples, and prompt order can cause the performance to vary quite significantly.

Automatic prompt generation and search. To address such sensitivity in language models, multiple approaches have been proposed for better prompt generation. In the continuous space, [86] propose prompt-tuning to add tunable tokens for each task during the fine-tuning stage to improve task performance. [243] propose OptiPrompt that optimizes

the prompts in the input embedding space directly for factual probing. More recently, [216] found performing instance-independent prompt-tuning can further boost the performance. In the discrete space, [49] propose prompt-based fine-tuning and utilize pre-trained models to automatically generate prompt templates. [162] and [160] use a small amount of training data to automatically identify the best label words to use for few-shot classification. [172] propose AutoPrompt to perform gradient-guided search to find the best tokens in the prompt, although the best prompts found are usually not interpretable by humans. [68] propose mining-based and paraphrasing-based methods to generate meaningful and diverse prompts for factual knowledge probing. Related to our work, [37] propose an RL-based framework to directly generate better prompts via black-box optimization. Different from existing work, our approach frames the problem as test-time prompt editing with an RL-based framework to perform efficient search in the editing space.

Efficient training exemplar retrieval as prompts. In addition, existing work has shown the choice of the exemplars can also be critical to the final performance. For example, [99] propose to retrieve exemplars from a training pool that are semantically similar to a test example, and show it can significantly boost the performance. [156] trained a dense retriever to efficiently retrieve good training examples as prompts during test time. In this work, we show that an attention-based exemplar selection process over the embedding space can effectively choose performant training examples within our RL framework.

4.3 Test-Time Prompt Editing

We formulate the task of test-time editing in this section. We give some background on the few-shot text classification and how to use prompts for downstream NLP tasks. Then we formalize a new setting called *test-time editing* where users are allowed to perform editing over a given prompt, depending on the given input and task during test time.

Background

Few-Shot Text Classification. Following the standard few-shot language model classification setting [18], we assume that we are given a pretrained language model \mathcal{L} and wish to perform classification on dataset \mathcal{D} with label space \mathcal{Y} . Assume we are given K samples per class from the training set, the new few-shot training set is given as $\mathcal{D}_{\text{train}} = \{x_i, y_i\}_{i=1}^{K \times |\mathcal{Y}|}$. In addition, there is a hold-out test dataset $\mathcal{D}_{\text{test}}$ that we use for evaluation on downstream NLP tasks.

Optimizing Discrete Prompts. Prompt-based few-shot learning considers the following problem: given a piece of text \mathbf{p} as a prompt, we use the generative distribution of the language model $P_{\mathcal{L}}(y|\mathbf{p}, \mathbf{x})$ to perform various NLP tasks without fine-tuning the model. In particular, for a given objective R , we propose to perform the desired optimization over

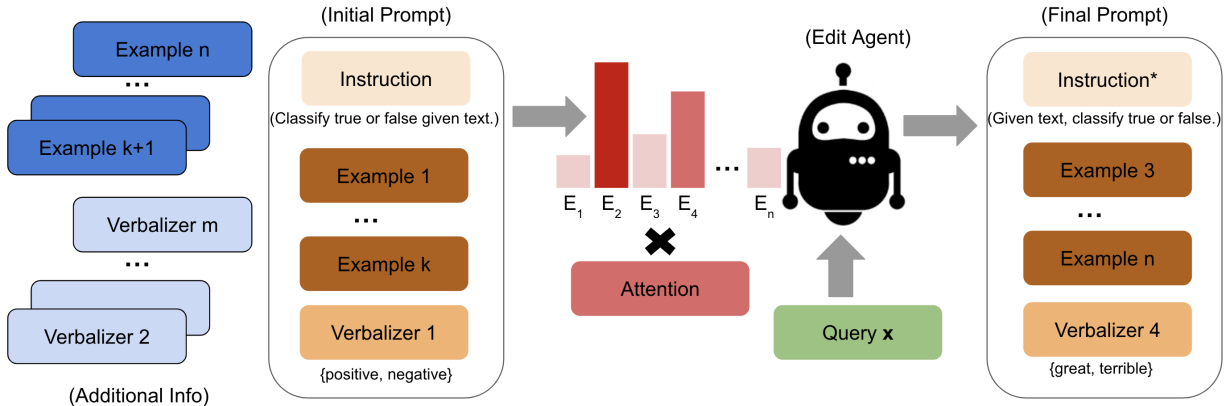


Figure 4.2: **Test-Time Editing via RL:** The RL agent is trained to optimize the performance of a downstream task. At test-time, given a query, the agent adopts an attention-based policy to edit the instructions, in-context exemplars and verbalizers for T rounds.

the prompt by finding an optimal $\mathbf{p}^* = \operatorname{argmin}_{\mathbf{p} \in \mathcal{V}} R(P_{\mathcal{L}}(y|\mathbf{p}, \mathbf{x}))$. In this paper, we focus on restricting the prompt \mathbf{p} as a piece of text instead of letting \mathbf{p} to be any vector in the latent space. This not only provides more interpretability of the prompt, but also allows us to use existing natural language tools (e.g., NLTK [15]) to perform a discrete search for constructing better prompts.

Different Forms of Discrete Prompts. We consider three popular forms of discrete prompts: (1) Instructions, which provide a segment of text describing how the task is performed, usually put at the beginning. (2) In-Context Demonstrations $\{e_0, e_1, \dots, e_k\}$, which selects several examples and their corresponding labels, usually placed before the query. (3) Verbalization, which aims to design how the task is asked and which keywords to select as labels. See Figure 4.2 for an example of different transformations that we perform when editing in our RL-based framework.

Test-Time Editing

Prior works have often attempted to identify a query-agnostic prompt or attempted to directly generate a query-dependent prompt via hyper-networks learning. However, query-agnostic prompting fails to incorporate any query-related information into the prompts and directly generating prompts for each individual query is challenging (due to its difficulty to incorporate human prior knowledge or feedback). In addition, by permuting the order of in-context exemplars $\{e_0, e_1, \dots, e_k\}$ or searching for the k nearest neighbors of the current test instance as in-context exemplars yields better performance. These reveal the importance of constructing query-dependent prompts.

Algorithm 2 Test-Time Prompt Editing with TEMPERA

- 1: **Input:** Language Model \mathcal{L} , Initial Prompt p_0 , Training set $\mathcal{D}_{\text{train}}$, Evaluation set $\mathcal{D}_{\text{eval}}$, Iteration N , Fix rounds T
 - 2: Initialize $\pi_\theta(\cdot | s)$ to be uniform;
 - 3: **for** episode $n = 1, \dots, N$ **do**
 - 4: Random sample batch $\mathcal{B} \sim \mathcal{D}_{\text{train}}$, Set p_0
 - 5: **for** step $t = 1, \dots, T$ **do**
 - 6: Get $s_t = \mathcal{L}(\mathcal{B}, p_t)$
 - 7: Run editing policy $a_t = \pi_\theta(s_t)$, Get new prompt p_{t+1}
 - 8: Get new state $s_{t+1} = \mathcal{L}(\mathcal{B}, p_{t+1})$
 - 9: Add transition (s_t, a_t, s_{t+1}) to replay buffer
 - 10: **end for**
 - 11: Update policy parameter θ of π_θ with the PPO loss
 - 12: **end for**
 - 13: **Evaluate** policy π_θ on evaluation dataset $\mathcal{D}_{\text{eval}}$
-

Unlike prior methods, we perform prompt editing at test-time. The procedure works as follows: at test time, one is given an initial prompt \mathbf{p}_0 . We want to learn a function f that takes the initial prompt \mathbf{p}_0 , query \mathbf{x} and a pool of examples/verbalizers \mathbf{p}' , and outputs a final prompt: $\mathbf{p}_f = f(\mathbf{p}_0, \mathbf{x}, \mathbf{p}')$. The overall framework of our algorithm is shown in Fig. 4.2. We allow f to make edits (e.g., editing verbalizers and/or swapping examples) over the original prompt to make it more suitable for the downstream task and query \mathbf{x} . Since the editing function f can depend on the query \mathbf{x} , we call it the test-time editing function. Note that we train the function f in a fixed training dataset and directly deploy it at test time without any addition training. This is different from the test-time optimization since we don't have access to the ground truth label or a surrogate objective. Please see Algorithm.2 for details.

4.4 Test-Time Editing via Reinforcement Learning

In order to learn the test-time editing function f , we present a novel RL-based framework that naturally maps the editing process to an MDP. We will present our framework and discuss how we design the state space, action space and reward in this section.

Reinforcement Learning Formulation. We formulate test-time editing as a Markov Decision Process (MDP). Given an initial state, $\mathbf{s} = (\mathbf{p}_0, \mathbf{x})$, consisting of an initial prompt and a query, at each time step t , the RL agent selects one of the editing methods from the action space A . We can then define the transition function $\mathcal{T} : S \times A \rightarrow S$ to be the state of prompt before and after editing $(\mathbf{p}_t, \mathbf{x}) \times \mathbf{a}_t \rightarrow (\mathbf{p}_{t+1}, \mathbf{x})$. That is, the transition dynamics are deterministic given the editing action. We can either define a fixed horizon H or design

Table 4.1: Effect of different editing techniques. For instruction, we tokenize it into phrases and perform swapping, addition or deletion. We also allow swapping in-context exemplars or changing different verbalizers.

Before Editing	After Editing
“Given text, classify whether it is good or bad.”	“Classify whether it is good or bad, given text.”
“Given text, classify whether it is good or bad.”	“Given text, given text, Classify whether it is good or bad.”
“Given text, classify whether it is good or bad.”	“Classify whether it is good or bad.”
{Example 1, Example 2, ..., Example k }	{Example k , Example 3, ..., Example 1 }
{Example 1, Example 2, ..., Example k }	{Example $k + 1$, Example n , ..., Example 1 }
{ “positive”, “negative” }	{ “great”, “terrible” }

a termination function to stop editing and get the final prompt. The goal is to maximize the expected reward $R = \mathbb{E}[\sum_{k=0}^T \gamma^k r_k]$ where r_t is the reward and γ is the discount factor. We introduce in detail each component of the state representation, action space and rewards in the following subsections.

State Representation. The RL framework is general and flexible about the representation of states. The only requirement is that such representation contains text information. Instead of directly using the raw text representation, we use the last hidden states of the pretrained language model $\mathbf{s}_t = \mathcal{L}(\mathbf{p}_t, \mathbf{x})$ as the state representation and feed it into the policy network.

Action Space Design. We include most of the editing actions in our action space. At each stage, the RL agent can choose the editing objects from instruction, in-context exemplars or verbalizer. For editing the instruction, we provide the initial instruction from natural instructions [201]. Then we tokenize the instruction into phrase level using NLTK [15] and perform swapping, deletion or addition of different phrases. Suppose we have l phrases, the action space size will become $(l \times (l - 1))/2 + 2l$.

For the in-context exemplars, we keep an example pool of N , initialize our prompt by randomly choose n of them as the initial prompt. We then allow the agent to directly perform swapping one example from the current prompt with either another one from the current prompt or from the pool of examples that are not currently used. This results in an action space for the RL agent of $n \times N - (n \times (n - 1))/2$ since we do not allow swapping with the same example.

For the verbalizer, we allow the RL agent to freely choose which verbalizer to use for each in-context example from PromptSource [12]. We also will enable the agent to freely choose which verbalizer to use for each query \mathbf{x} . Interestingly we found that this helps boost the performance of our algorithm. We provide some examples of the editing process in Tab. [4.1].

Reward Design. We adopt the step reward proposed in RLPrompt [37]. For each query \mathbf{x} , we get the log probability of the output label from the language model $\log P_{\mathcal{L}}(\hat{y}|\mathbf{x}, \mathbf{p}_t)$ given the proposed prompt \mathbf{p}_t with the correct label c , and we define the score difference $s(c)$ as:

$$s(c, \mathbf{x}, \mathbf{p}_t) = \lambda_1 \log P_{\mathcal{L}}(\hat{y}_c|\mathbf{x}, \mathbf{p}_t) - \lambda_2 \operatorname{argmax}_{c' \neq c} \log P_{\mathcal{L}}(\hat{y}_{c'}|\mathbf{x}, \mathbf{p}_t) \quad (4.1)$$

where we have introduced the two balancing hyperparameters $\lambda_1 > 0$ and $\lambda_2 > 0$ for the positive and negative terms respectively. Intuitively, this score gives a negative reward when the prediction is not correct and a positive reward otherwise. The goal is to optimize the score for the final prompt.

However, RL aims to optimize the accumulated reward during the MDP process while prompt design only cares about the performance of the final prompt. Thus, we propose to use the score difference between successive edits as the immediate reward:

$$r_t = s(c, \mathbf{x}, \mathbf{p}_t) - s(c, \mathbf{x}, \mathbf{p}_{t-1}) \quad (4.2)$$

Ignoring the discounting factor γ , this makes the accumulated reward from time 0 to T correspond to the score difference between the final and the initial prompt $s(c, \mathbf{x}, \mathbf{p}_T) - s(c, \mathbf{x}, \mathbf{p}_0)$. Now the objective of RL is to maximize the score difference.

Attention-Based Policy Architecture. We adopt an attention-based policy architecture for the reinforcement learning agent. We put attention over a graph of possible candidates and let the agent choose which editing technique to perform. We find that the attention-based architecture helps the agent to emphasize the important examples (e.g., examples that are more semantically similar to the test instance).

We use the PPO [165] algorithm in our experiments. The detailed hyperparameter used can be found in Appendix. C.1. We list here a couple of very important techniques we used in our experiments. We found these techniques are crucial to the success of our RL-based framework.

Observation Normalization: Since we take the last hidden states of the language model as observation, it might have very small variances between different samples. We keep a running mean and standard deviation for the observation and normalize it before feeding it to the policy and value network. This is commonly used in RL and we found this boosts the performance of our method.

Reward Normalization: For different training samples, performing editing over prompts may result in significantly different reward scales. For some of the samples, different prompts might have very marginal effects on the final prediction, either due to the fact that the model is already confident about the prediction since it is too easy, or the task sample is too hard to predict and the model is confused regardless of what prompt it is fed. On the other hand, for other training samples, editing prompts might bring a huge difference in terms of the accuracy. Thus, we perform sample-wise reward normalization to ensure that the reward scale between samples is relatively consistent.

Conditioning Policy on Action History: Directly taking the observation from the language model can be inefficient since the policy has no clue about how it has reached the current state. This will bring a loop that the policy will edit prompts $\mathbf{p}_A \rightarrow \mathbf{p}_B$ and then $\mathbf{p}_B \rightarrow \mathbf{p}_A$. To mitigate this effect, we build a policy that not only takes in the current hidden state, but also conditioned on the action history on how it gets to the current state. Thus, we break the loop between two prompts by considering how each state is reached.

4.5 Experiments

Our experiments first reveal the effectiveness of TEMPERA in the few-shot setting. We compare TEMPERA with prior baselines like Finetuning [39], Soft Prompt Tuning [86], Black-Box Tuning [179], RLPrompt [37] and other manually tuned prompt methods. On various tasks from GLUE [195] and SuperGLUE [196], our method achieves impressive performance comparing to prior baselines. This shows that only using a small amount of training examples is sufficient for RL and TEMPERA is sample efficient. We also illustrate the data efficiency of our method compared to finetuning, showing that TEMPERA can achieve same performance with 5.33x less data.

In addition to the performance gains, we aim to understand our method from different aspects. In Sec. 4.5, we study how much test-time editing helps compared to query-agnostic prompts. Our experiments demonstrate the importance of test-time editing and the necessity of query-dependent prompts. In Sec. 4.5, we show that how different editing techniques (e.g, instruction, in-context demonstration and verbalization) affect the final performance of the downstream task. We also ablate the number of in-context demonstrations used and the size of the example pool in Sec. 4.5 and Sec. 4.5. Finally, we show some example prompts after editing to illustrate the editing policy.

Tasks. We conduct our experiments from different categories including single-sentence tasks (e.g., sentiment analysis including SST-2, Yelp reviews, MR, CR, topic classification including AG News). For one-sentence tasks, the goal is to make a prediction based on the sentence. We also include tasks from different types like NLI (e.g., SST-2) and multiple choices (e.g., AG News). Most of the tasks are from the standard GLUE [195].

Task Settings. To ensure a fair comparison, we follow the same setting from LM-BFF [50] and RLPrompt [37], we test TEMPERA on few-shot text classification tasks. The setting is devised as follows: We randomly sample 16 training samples per class from the training dataset of each task and use them as the few-shot dataset. This will result in a total of $16 \times |\mathcal{Y}|$ training samples (please refer to Appendix C.5 for the number of classes in each task). We also randomly sample 16 samples per class as the validation dataset. For reporting the final performance, we use the standard test set and the detailed information can be found at Appendix C.5. In addition to the common setup, we also randomly select n examples from the training dataset as the in-context exemplar pool. We average our runs for 4 random seeds

and report the average performance and corresponding standard deviation. For the language model, we use $\mathcal{L} = \text{RoBERTa-large}$ [108]. For the details of these settings and tasks, please refer to Appendix. C.5. The initial instruction is taken from the Natural Instructions [122]. The initial in context demonstrations are randomly sampled from a fixed example pool of size 16 and the example pool is also randomly sampled from the training dataset, different from the few-shot dataset that used for training the RL policy.

Baselines. We compare TEMPERA with several SoTA prompt tuning and discrete prompt optimization baselines (including finetuning).

- **Finetuning:** it finetunes the entire language model with a classification head using the few-shot dataset.
- **Manual Prompt:** we take the handcrafted prompt from [12].
- **Black-Box Tuning:** it is a mixture of discrete and soft prompt. The soft part is trained using gradient descent and the discrete part is optimized using gradient-free tuner.
- **AutoPrompt:** it adds the discrete trigger token and updates the prompts by iterative gradient search.
- **In-Context Demonstration:** it randomly selects one training example and concatenates them with the input query.
- **Instructions:** Following Natural Instructions [201], prompts are manually created instruction for each task. Each prompt is concatenated with inputs. Details are in Appendix. C.4.
- **GrIPS:** it performs phrase level editing on the instructions and selects the best one.
- **RLPrompt:** it generates discrete prompts using RL framework.

Few-Shot Text Classification

Following the settings in existing work, we evaluate our model on some few-shot text classification tasks. In Tab. 4.2. We compare our method with various baselines including RLPrompt. We can see that on most tasks we tested, TEMPERA outperforms previous baselines by a large margin. For example, we have a 1.8% absolute gain on the SST-2 task (over RLPrompt), 3.9% gain on the CR task and the performance is almost comparable to finetuning the language model on the AG News task. We also see that our method results in a much smaller variance between runs than Soft Prompt Tuning and AutoPrompt, indicating that it is more stable across different few-shot datasets. Comparing to search-based methods (e.g., Black-Box Tuning or GrIPS), our method avoids the expensive run-time search if one

wants to perform test-time editing using one of the black-box optimization methods with a surrogate reward. Note since the original Black-Box Tuning or GrIPS paper didn't perform query-dependent search, this is our conjecture. Thus, our method achieves both test-time efficiency and good performances on downstream tasks.

Table 4.2: Few-shot classification results. We compare against different baselines in this setting. Results show that TEMPERA surpasses various baselines including finetuning, prompt tuning and discrete prompt search. The standard deviations are shown in brackets.

		SST-2	Yelp P.	MR	CR	AG News
Finetuning	Finetuning (few-shot)	80.6 (3.9)	88.7 (4.7)	67.4 (9.7)	73.3 (7.5)	84.9 (3.6)
Continuous Prompt	Soft Prompt Tuning	73.8 (10.9)	88.6 (2.1)	74.1 (14.6)	75.9 (11.8)	82.6 (0.9)
	Black-Box Tuning	89.1 (0.9)	93.2 (0.5)	86.6 (1.3)	87.4 (1.0)	83.5 (0.9)
	AutoPrompt	75.0 (7.6)	79.8 (8.3)	62.0 (0.8)	57.5 (5.8)	65.7 (1.9)
Discrete Prompt	Manual Prompt	82.8	83.0	80.9	79.6	76.9
	In-Context Demo.	85.9 (0.7)	89.6 (0.4)	80.6 (1.4)	85.5 (1.5)	74.9 (0.8)
	Instructions	89.0	84.4	85.2	80.8	54.8
	GrIPS	87.1 (1.5)	88.2 (0.1)	86.1 (0.3)	80.0 (2.5)	65.4 (9.8)
	RLPrompt	90.1 (1.8)	93.9 (1.8)	86.7 (2.4)	87.2 (1.7)	77.2 (2.0)
Discrete Prompt	TEMPERA (ours)	91.9 (2.0)	92.6 (1.7)	88.0 (1.1)	91.1 (1.6)	85.5 (1.5)

Importance of Test-time Prompt Editing

To illustrate the importance of test-time prompt editing, we compare our method with various baselines that do not perform test-time editing. In addition, we also construct another baseline where we create a RL based method where the policy is not dependent on the input query \mathbf{x} , denoted as “TEMPERA (No TTE)”. Results in Tab. 4.3 show that TEMPERA even without test-time editing can find better query-agnostic prompts comparing to manually construct prompts, in-context demonstration and GrIPS. However, adding test-time editing can further improve the performance when the task is harder: we got 0.8% improvement on MR task and 3.0% improvement at AG News task. On SST-2, the effect of test-time editing is not significant as we suspect that the task is too easy. We found on harder tasks like AG News, the gain of test-time editing is huge.

Data Efficiency for TEMPERA

To illustrate the data efficiency of our method, we compare the performance of TEMPERA with some few-shot standard finetuning results in Fig. 4.3. We see that in SST-2, we achieve similar performance using almost 8x fewer training data. In tasks like Yelp, the gain is about 4x. We see that with fewer examples, TEMPERA strictly dominates fine-tuning methods.

Table 4.3: We compare our method against different methods which do not perform test-time editing. Results show that test-time editing is mostly helpful in harder tasks like AG News.

	SST-2	MR	AG News
Manual Prompt	82.8	80.9	76.9
In-Context Demo.	85.9	80.6	74.9
Instructions	89.0	85.2	54.8
GrIPS	87.1	87.1	65.4
TEMPERA (No TTE)	92.0	87.4	81.3
TEMPERA	91.9	88.2	84.3

Table 4.4: Ablation on different editing techniques. Results show that adding verbalizer-edits helps all the tasks (especially MR and AG News). Adding instruction-edits marginally helps the performance in SST-2 and MR.

	SST-2	MR	AG News
TEMPERA (No Inst & Verb)	91.2	87.2	82.2
TEMPERA (No Inst)	91.9	88.2	84.3
TEMPERA	92.4	88.4	85.5

This is critical when applying TEMPERA in the real-world application since labeled data is expensive to get.

Qualitative Analysis of the Edits

We also visualize our policy by taking a few examples from the final prompts after editing in Tab. 4.5. We see that our method mostly does example selection, verbalizer swapping and phrase-level instruction editing. Our editing techniques are flexible and the final prompt may take different combinations for each query. In addition, the resulting final prompt is still interpretable by human, showing that our method achieves flexibility and interpretability at the same time. Note that in the examples provided in Tab. 4.1, our policy choose to modify the example selection and verbalization.

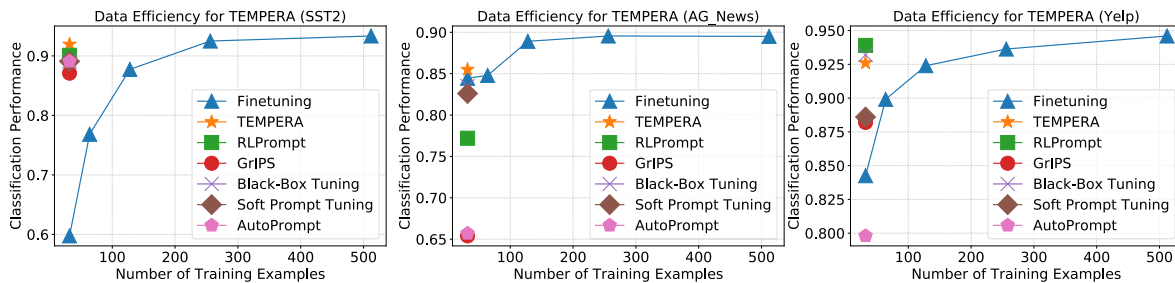


Figure 4.3: **Data Efficiency for TEMPERA:** We compare data efficiency between TEMPERA and few-shot finetuning. Results show that we can achieve a good performance with significantly less data (varying from 4x to 8x).

Table 4.5: Qualitative results on the effect of the learned policy. We see that our method both enables the flexibility of various edits and interpretability of the final results. On the contrary, many prior methods produce non-readable prompts. Red text is prior to editing and blue text are the changes.

SST-2	Before Edit	“In this task, you are given sentences from movie reviews. The task is to classify a sentence as “ positive ” if the sentiment of the sentence is positive or as “ negative ” if the sentiment of the sentence is negative. Review: of saucy. Sentiment: positive . Review: cold movie. Sentiment: negative . Review: heroes. Sentiment: <mask>.”
	After Edit (<i>better verbalizer</i>)	“In this task, you are given sentences from movie reviews. The task is to classify a sentence as “ great ” if the sentiment of the sentence is positive or as “ terrible ” if the sentiment of the sentence is negative. Review: of saucy. Sentiment: great . Review: cold movie. Sentiment: terrible . Review: heroes. Sentiment: <mask>.”
AG News	Before Edit	“Classify the news articles into the categories of World, Sports, Business, and Technology. Article: What’s in a Name? Well, Matt Is Sexier Than Paul (Reuters) Reuters - As Shakespeare said, a rose by any other name would smell as sweet. Right? Answer: Technology. Article: Wall St. Bears Claw Back Into the Black (Reuters) Reuters - Short-sellers, Wall Street’s dwindling band of ultra-cynics, are seeing green again. Answer: <mask>.”
	After Edit (<i>better exemplar selection</i>)	“Classify the news articles into the categories of World, Sports, Business, and Technology. Article: Expansion slows in Japan Economic growth in Japan slows down as the country experiences a drop in domestic and corporate spending. Answer: Business. Article: Wall St. Bears Claw Back Into the Black (Reuters) Reuters - Short-sellers, Wall Street’s dwindling band of ultra-cynics, are seeing green again. Answer: <mask>.”

Ablation: Different Editing Techniques

We ablate on the different editing techniques and study how adding or removing them can affect the performance. The results are shown in Tab. 4.4. We can see that adding each component (e.g., verbalizer, instruction) is helpful in terms of the final performance. We also find that verbalizer is especially helpful in some tasks like AG News, resulting in a 1.2%

Table 4.6: Ablation on the number of in-context exemplars. Results show that increasing the number of examples results in a consistent increase of performance except for AG News (which is due to the length limit).

	SST-2	MR	AG News
TEMPERA (2 Examples)	91.6	87.9	84.0
TEMPERA (4 Examples)	91.9	88.2	84.3
TEMPERA (8 Examples)	92.4	88.4	82.2

difference in the final performance. This indicates that adding more flexibility to some extent can help the performance.

Ablation: Number of Shots

We also ablate on the number of examples used in the in-context demonstration part of our algorithm. We choose the size of 2, 4 and 8 for the analysis. We see that from Tab. 4.6, in all the tasks we tested (SST-2, MR and AG News), increasing the number of examples consistently improves the performance. However, the performance improvement is relatively limited. In addition, due to the input length limit constraint by the language model (512 for RoBERTa), longer sequences of input will be truncated. This results in the performance decrease when increasing the number of examples from 4 to 8 for AG News, where the input length is longer than 512.

Table 4.7: Ablation on the size of the prompt pool to select from. We see that the performance does not change too much when changing the size of the pool, indicating that the performance is relatively stable.

	SST-2	MR	AG News
TEMPERA (Pool Size 8)	91.6	87.9	84.1
TEMPERA (Pool Size 16)	91.9	88.2	84.3
TEMPERA (Pool Size 32)	92.2	88.4	84.7

Ablation: Size of the Prompt Pool

We also ablate on the example size of the prompt pool where we keep the number of exemplars of 4. Intuitively, allowing our method to choose in-context demonstrations from a large range of example pool can provide better prompts. From Table. 4.7, we can see that increasing the

example pool size gives the algorithm more flexibility to choose in-context demonstrations, resulting in a slightly better final performance.

4.6 Conclusion

In this paper we present TEMPERA, a test-time prompt editing method for large language models via reinforcement learning. We found that perform test-time editing can greatly improve the performance of downstream tasks for a pretrained language model. The proposed method only requires little guidance on high-level search space design and can easily incorporate prior human knowledge. It achieves SoTA performance on multiple benchmarks including those from GLUE. This intersection area of research between NLP and RL can inspire future research on designing better test-time editing algorithms for practical usage.

Part III

LLM Interacting with Tools

Chapter 5

Gorilla: Exploring Tool Usage of LLM Agents

Large Language Models (LLMs) have seen an impressive wave of advances recently, with models now excelling in a variety of tasks, such as mathematical reasoning and program synthesis. However, their potential to effectively use tools via API calls remains unfulfilled. This is a challenging task even for today’s state-of-the-art LLMs such as GPT-4 largely due to their unawareness of what APIs are available and how to use them in a frequently updated toolset. We develop Gorilla, a finetuned LLaMA model that surpasses the performance of GPT-4 on writing API calls. When combined with a document retriever, Gorilla demonstrates a strong capability to adapt to test-time document changes, enabling flexible user updates or version changes. It also substantially mitigates the issue of hallucination, commonly encountered when prompting LLMs directly. To evaluate the model’s ability, we introduce APIBench, a comprehensive dataset consisting of HuggingFace, TorchHub, and TensorHub APIs. The successful integration of the retrieval system with Gorilla demonstrates the potential for LLMs to use tools more accurately, keep up with frequently updated documentation, and consequently increase the reliability and applicability of their outputs. Gorilla’s code, model, and data will be open-sourced.

5.1 Introduction

The use of API’s and Large Language Models [33, 18, 159, 19, 130, 131] has changed what it means to program. Previously, building complex machine learning software and systems required extensive time and specialized skills. Now with tools like the HuggingFace API, an engineer can set up a deep learning pipeline with a few lines of code. Instead of searching through StackOverflow and documentation. Developers can ask models like GPT for solutions and receive immediate, actionable code with docstrings. However, using off-the-shelf LLM’s to generate API calls remains unsolved because there are millions of available API’s which are frequently updated.

“Help me find an API to convert the spoken language in a recorded audio to text using Torch Hub.”

Correct (Gorilla)	<pre> <domain>: Speech-to-Text <api_provider>: TorchHub <code>: asr_model = torch.hub.load('snakers4/silero-models', 'silero_sst') result = asr_model.transcribe(audio_path) </pre>
API Argument Hallucination (GPT-4)	<pre> <domain>: Speech-to-Text <api_provider>: TorchHub <code>: asr_model = torch.hub.load('snakers4/silero-models', 'asr', source='local') result = asr_model.transcribe(audio_path) </pre>
Incorrect API Call (Claude)	<pre> <domain>: Audio-Translation <api_provider>: Pytorch <code>: import torchaudio translation = torchaudio.pipelines.WAV2VEC2_ASR_PIPELINE("audio.wav") </pre>

Figure 5.1: **Examples of API calls.** Example API calls generated by GPT-4 [130], Claude [8], and Gorilla for the given prompt. In this example, GPT-4 presents a model that doesn’t exist, and Claude picks an incorrect library. In contrast, our Gorilla model can identify the task correctly and suggest a fully-qualified API call.

We connect LLM’s and massive API’s with Gorilla, a system which takes in an instruction like “build me a classifier for medical images” and outputs the corresponding imports and API calls, along with a step-by-step explanation of the pipeline. Gorilla uses self-instruct, fine-tuning, and retrieval to enable LLMs to accurately select from a large, overlapping, and changing set tools expressed using their APIs and API documentation. The model trained with the our method demonstrates good adaptation behavior to test-time API document changes such as API version and arguments.

With the development of API generation methods comes a question of how to evaluate, as many APIs will have overlapping functionality with nuanced limitations and constraints. Thus, we construct APIBench (~ 1600 APIs) by scraping a large corpus of ML APIs and developing an evaluation framework that uses AST sub-tree matching to check functional correctness.

Using APIBench, we finetune Gorilla, a LLaMA-7B-based model with document retrieval and show that it significantly outperforms both open-source and closed-source models like Claude and GPT-4 in terms of API functionality accuracy as well as a reduction in API argument hallucination errors (Fig. 5.1). Finally, we demonstrate Gorilla’s ability to understand and reason about constraints.

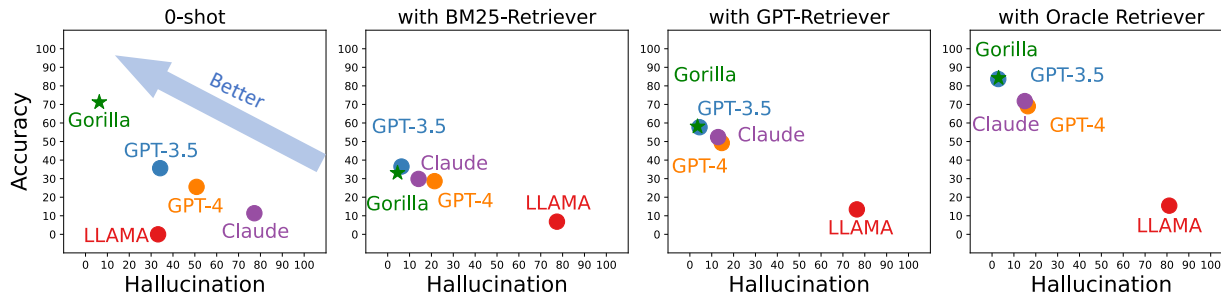


Figure 5.2: **Accuracy (vs) hallucination** in four settings, that is, *zero-shot* (i.e., without any retriever), and *with retrievers*. BM25 and GPT are commonly used retrievers and the `oracle` retriever returns relevant documents at 100%, indicating an upper bound. Higher in the graph (higher accuracy) and to the left is better (lower hallucination). Across the entire dataset, our model, Gorilla, improves accuracy while reducing hallucination.

5.2 Related Work

By empowering LLMs to use tools [163], we can grant access to vastly larger and changing knowledge bases and accomplish complex computational tasks. By providing access to search technologies and databases, [124, 187, 176] demonstrated that we can augment LLMs to address a significantly larger and more dynamic knowledge space. Similarly, by providing access to computational tools, [187, 4, 245, 178, 182] demonstrated that LLMs can accomplish complex computational tasks. Consequently, leading LLM providers [130], have started to integrate plugins to allow LLMs to invoke external tools through APIs.

Large Language Models Recent strides in the field of LLMs have renovated many downstream domains [33, 189, 236, 232], not only in traditional natural language processing tasks but also in program synthesis. Many of these advances are achieved by augmenting pre-trained LLMs by prompting [206] and instruction fine-tuning [34, 158, 203, 62]. Recent open-sourced models like LLaMa [189], Alpaca [185], and Vicuna [32] have furthered the understanding of LLMs and facilitated their experimentation. While our approach, Gorilla, incorporates techniques akin to those mentioned, its primary emphasis is on enhancing the LLMs’ ability to utilize millions of tools, as opposed to refining their conversational skills. Additionally, we pioneer the study of fine-tuning a base model by supplementing it with information retrieval - a first, to the best of our knowledge.

Tool Usage The discussion of tool usage within LLMs has seen an upsurge, with models like Toolformer taking the lead [163, 79, 84, 124]. Tools often incorporated include web-browsing [161], calculators [35, 187], translation systems [187], and Python interpreters. While these efforts can be seen as preliminary explorations of marrying LLMs with tool usage, they generally focus on specific tools. Our paper, in contrast, aims to explore a vast

array of tools (i.e., API calls) in an open-ended fashion, potentially covering a wide range of applications.

With the recent launch of Toolformer [163] and GPT-4 [130], the importance of API calls has been highlighted, encouraging many works in employing API calls as tooling [166, 92]. Moreover, the application of API calls in robotics has been explored to some extent [193, 1]. However, these works primarily aim at showcasing the potential of “prompting” LLMs rather than establishing a systematic method for evaluation and training (including fine-tuning). Our work, on the other hand, concentrates on systematic evaluation and building a pipeline for future use.

LLMs for Program Synthesis Harnessing LLMs for program synthesis has historically been a challenging task [91, 27, 219, 64, 40, 82]. Researchers have proposed an array of strategies to prompt LLMs to perform better in coding tasks, including in-context learning [206, 78, 27], task decomposition [77, 224], and self-debugging [29, 173]. Besides prompting, there have also been efforts to pretrain language models specifically for code generation [126, 89, 127].

DocPrompting [245] looked at choosing the right subset of code including API along with a retriever. Gorilla presents distinct advancements over DocPrompting. First, the way the data-sets are constructed are different, leading to interesting downstream artifacts. Gorilla focuses on model usages where we also collect detailed information about parameters, performance, efficiency, etc. This helps our trained model understand and respond to finer constraints for each API. Docprompting focuses on generic API calls but not on the details within an API call. Second, Gorilla introduces and uses the AST subtree-matching evaluation metric that helps measure hallucination which we find are more representative of code structure and API accuracy compared to traditional NLP metrics. Finally, Gorilla focuses on instruction-tuning method and has “agency” to interact with users while DocPrompting focuses on building an NLP-to-Code generative model. On equal footing, we demonstrate that Gorilla performs better than DocPrompting in Appendix D.3.

5.3 Method

We first describe APIBench, a comprehensive benchmark constructed from TorchHub, TensorHub, and HuggingFace API Model Cards. We begin by outlining the process of collecting the API dataset and how we generated instruction-answer pairs. We then introduce Gorilla, a novel training paradigm with an information-retriever incorporated into the training and inference pipelines. Finally, we present our AST tree matching evaluation metric.

Dataset Collection

To collect the dataset, we recorded all online model cards for HuggingFace’s “The Model Hub”, PyTorch Hub, and TensorFlow Hub Models. Throughout the rest of the paper, we

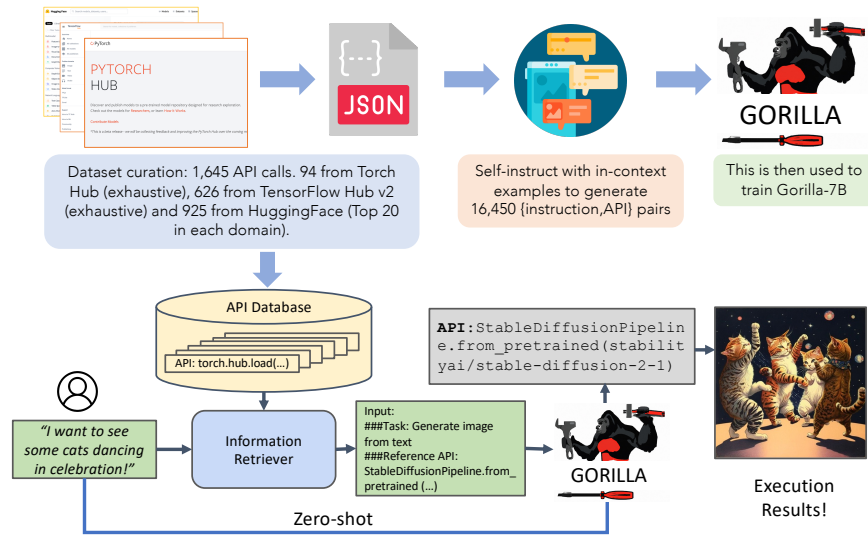


Figure 5.3: **Gorilla: A system for enabling LLMs to interact with APIs.** The upper half represents the training procedure as described in Sec 3.2. This is the most exhaustive API data-set for ML to the best of our knowledge. During inference (lower half), Gorilla supports two modes - with retrieval, and zero-shot. In this example, it is able to suggest the right API call for generating the image from the user’s natural language query.

call these HuggingFace, Torch Hub, and TensorFlow Hub respectively for brevity.

API Documentation The HuggingFace platform hosts and servers about 203,681 models. However, many of them have poor documentation, lack dependencies, have no information in their model card, etc. To filter these out, we pick the top 20 models from each domain. We consider 7 domains in multimodal data, 8 in CV, 12 in NLP, 5 in Audio, 2 in tabular data, and 2 in reinforcement learning. Post filtering, we arrive at a total of 925 models from HuggingFace. TensorFlow Hub is versioned into v1 and v2. The latest version (v2) has 801 models in total, and we process all of them. After filtering out model cards with little to no information, we are left with 626 models. Similar to TensorFlow Hub, we extract 95 models (exhaustive) from Torch Hub. We then converted the model cards for each of these 1,645 API calls into a JSON object with the following fields: {domain, framework, functionality, api_name, api_call, api_arguments, environment_requirements, example_code, performance, description}. We provide more information in D.1. These fields were chosen to generalize beyond API calls within the ML domain, to other domains, including RESTful, SQL, and other potential API calls.

Instruction Generation Guided by the self-instruct paradigm [202], we employ GPT-4 to generate synthetic instruction data. We provide three in-context examples, along with reference API documentation, and task the model with generating real-world use cases that

call upon the API. We specifically instruct the model to refrain from using any API names or hints when creating instructions. We constructed 6 examples (Instruction-API pairs) for each of the 3 model hubs. These 18 examples were the only hand-generated or modified data. For each of our 1,645 API datapoints, we generate 10 instruction-API pairs by sampling 3 of 6 corresponding instruction examples in each pair (illustrated in Figure 5.3).

API Call with Constraints API calls often come with inherent constraints. These constraints necessitate that the LLM not only comprehend the functionality of the API call but also categorize the calls according to different constraint parameters. Specifically, for machine learning API calls, two common sets of constraints are parameter size and a lower bound on accuracy. Consider, for instance, the following prompt: *“Invoke an image classification model that uses less than 10M parameters, but maintains an ImageNet accuracy of at least 70%.”* Such a prompt presents a substantial challenge for the LLM to accurately interpret and respond to. Not only must the LLM understand the user’s functional description, but it also needs to reason about the various constraints embedded within the request. This challenge underlines the intricate demands placed on LLMs in real-world API calls. It is not sufficient for the model to merely comprehend the basic functionality of an API call; it must also be capable of navigating the complex landscape of constraints that accompany such calls. We also incorporate these instructions in our training dataset.

Gorilla

Our model Gorilla, is a retrieval-aware finetuned LLaMA-7B model, specifically for API calls. As shown in Figure 5.3, we employ self-instruct to generate {instruction, API} pairs. To finetune LLaMA, we convert this to a user-agent chat-style conversation, where each datapoint is a conversation with one round each for the user and the agent. We then perform standard instruction finetuning on the base LLaMA-7B model. For our experiments, we train Gorilla with and without the retriever.

Retriever-Aware training For training with retriever, the instruction-tuned dataset also appends to the user prompt, *“Use this API documentation for reference: API_doc_JSON”*. Through this, we aim to teach the LLM to parse the second half of the question to answer the first half. We demonstrate that this (1) makes the LLM adapt to test-time changes in API documentation, (2) improves performance from in-context learning, and (3) reduces hallucination error.

Surprisingly, we find that augmenting a LLM with retrieval, does not always lead to improved performance, and can at-times hurt performance.

Gorilla Inference During inference, the user provides the prompt in natural language (Figure 5.3). This can be for a simple task (e.g, *“I would like to identify the objects in an image”*), or they can specify a vague goal, (e.g, *“I am going to the zoo, and would like to*

track animals”). Gorilla, similar to training, can be used for inference in two modes: zero-shot and with retrieval. In zero-shot, this prompt (with *no* further prompt tuning) is fed to the Gorilla LLM model when then returns the API call that will help in accomplishing the task and/or goal. In retrieval mode, the retriever (either of BM25 or GPT-Index) first retrieves the most up-to-date API documentation stored in the API Database. Before being sent to Gorilla, the API documentation is concatenated to the user prompt along with the message “*Use this API documentation for reference.*” The output of Gorilla is an API to be invoked. Besides the concatenation as described, we do *no* further prompt tuning in our system. While we also implemented a system to execute these APIs, that is not a focus of this paper.

Verifying APIs

Inductive program synthesis, where a program is synthesized to satisfy test cases, has found success in several avenues [14, 117]. However, test cases fall short when evaluating API calls, as it is often hard to verify the semantic correctness of the code. For example, consider the task of classifying an image. There are over 40 different models that can be used for the task. Even if we were to narrow down to a single family of Densenet, there are four different configurations possible. Hence, there exist multiple correct answers and it is hard to tell if the API being used is functionally equivalent to the reference API by unit tests. Thus, to evaluate the performance of our model, we compare their functional equivalence using the dataset we collected. To trace which API in the dataset is the LLM calling, we adopt the AST tree-matching strategy. Since we only consider one API call in this paper, checking if the AST of the candidate API call is a sub-tree of the reference API call reveals which API is being used in the dataset.

Identifying and even defining hallucinations can be challenging. We use the AST matching process to directly identify the hallucinations. We define a hallucination as an API call that is not a sub-tree of any API in the database – invoking an entirely imagined tool. This form of hallucination is distinct from invoking an API incorrectly which we instead define as an error.

AST Sub-Tree Matching We perform AST sub-tree matching to identify which API in our dataset is the LLM calling. Since each API call can have many arguments, we need to match on each of these arguments. Further, since, Python allows for default arguments, for each API, we define which arguments to match in our database. For example, we check `repo_or_dir` and `model` arguments in our function call. In this way, we can easily check if the argument matches the reference API or not. We first build the tree, and verify that it matches a subtree in our dataset along nodes `torch.hub.load`, `pytorch/vision`, and `densenet121`. We do not check for match along leaf node `pretrained=True` since that is an optional argument.

5.4 Evaluation

When evaluating Gorilla (finetuned on APIBench), we aim to answer the following questions:

1. How does Gorilla compare to other LLMs on API Bench? (5.4)
2. How well does Gorilla adapt to test-time changes in API documentation? (5.4)
3. How well can Gorilla handle questions with constraints? (5.4)

We show that Gorilla in the specific domain we evaluated on can outperform both open-source and close-source models. With our retriever-aware training techniques, our model can generalize to APIs that are outside of its training data. Finally, Gorilla also handles the constraints pretty well.

Baselines We primarily compare Gorilla with state-of-the-art language models in a zero-shot setting and with 3-shot in-context learning. The models under consideration include: GPT-4 by OpenAI with the `gpt-4-0314` checkpoint; GPT-3.5-turbo with the `gpt-3.5-turbo` checkpoint, both of which are RLHF-tuned models specifically designed for conversation; Claude with the `claude-v1` checkpoint, a language model by Anthropic, renowned for its lengthy context capabilities; and LLaMA-7B, a state-of-the-art open-source large language model by Meta.

Retrievers The term *zero-shot* (abbreviated as 0-shot in tables) refers to scenarios where no retriever is used. The sole input to the model is the user’s natural language prompt. For BM25, we consider each API as a separate document. During retrieval, we use the user’s query to search the index and fetch the most relevant (top-1) API. This API is concatenated with the user’s prompt to query the LLMs. Similarly, GPT-Index refers to the embedding model `text-embedding-ada-002-v2` from OpenAI where each embedding is 1,536 dimensional. Like BM25, each API call is indexed as an individual document, and the most relevant document, given a user query, is retrieved and appended to the user prompt. Lastly, we include an Oracle retriever, which serves two purposes: first, to identify the potential for performance improvement through more efficient retrievers, and second, to assist users who know which API to use but may need to help invoking it. In all cases, when a retriever is used, it is appended to the user’s prompt as follows: *user_prompt*; *Use this API documentation for reference: retrieved_API_doc_JSON*. The dataset for these evaluations is detailed in Section 3.2. We emphasize that we have maintained a holdout test set on which we report our findings. The holdout test set was created by dividing the self-instruct dataset’s instruction, API pairs into training and testing sets.

AST Accuracy on API call

We first demonstrate the results for the AST accuracy for different models. We present the results in Table 5.1. We test each model for different retriever settings defined above. We

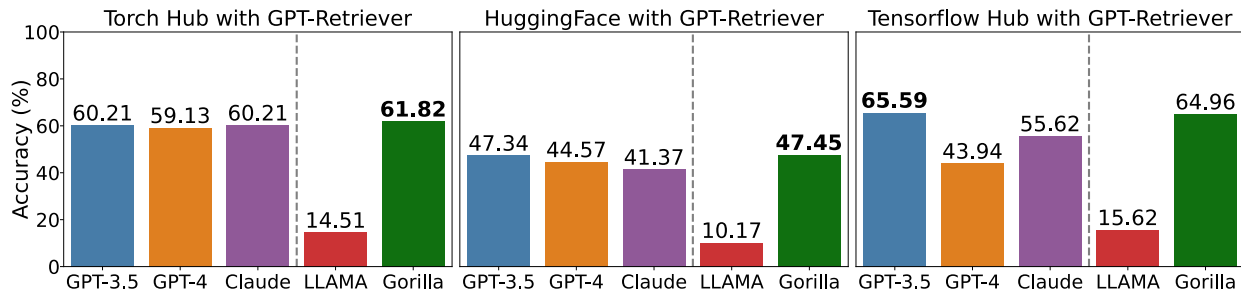


Figure 5.4: **Accuracy with GPT-retriever.** Methods to the left of the dotted line are closed source. Gorilla outperforms on Torch Hub and Hugging-Face while matching performance on Tensorflow Hub for all existing SoTA LLMs - closed source, and open source.

report the overall accuracy, the error by hallucination and the error by selecting wrong API call. Note that for TorchHub and TensorHub, we evaluate all the models using AST tree accuracy score. However, for HuggingFace, since the dataset is not exhaustive, for all the models except Gorilla, we only check if they can provide the correct domain names. So this problem reduces to picking one of multiple choices.

Finetuning without Retrieval In Table 5.1 we show that lightly fine-tuned Gorilla is able to match, and often surpass performance in the zero-shot over all the models, 20.43% better than GPT-4 and 10.75% better than ChatGPT. When compared to other open-source models LLAMA, the improvement is as big as 83%. This suggests quantitatively, that finetuning is better than retrieval, at-least in our scope.

In addition, we found that finetuning without the retriever and putting a ground truth retriever during evaluation improves the performance: 0.88% worse in TensorHub and 0.97% better in HuggingFace. If we put BM25 or GPT-Index as retriever, results will be significantly dropped: 21.50% in Torch Hub and 47.57% in HuggingFace. The result illustrates that adding a non-optimal retriever at test time sometimes misguide the model and result in more errors.

Finetuning with Retrieval We now discuss an interesting experiment on how finetuning language with retriever incorporated is helping the performance. The settings for this experiment are finetuning the base LLAMA with the prompt (instruction generated), reference API document (from golden-truth oracle), and the example output generated by GPT-4. In Table 5.3, incorporating ground truth retriever in the finetuning pipeline achieves significantly better results 12.37% better than training without retriever in Torch Hub and 23.46% better in HuggingFace. However, we found that at evaluation time, current retrievers still have a big gap between the ground truth retriever: using GPT-Index at evaluation results in 29.20% accuracy degradation and using BM25 results in a 52.27% accuracy degradation. Nevertheless, we can still conclude that with a good retriever, finetuning an LLM to incorporate retrieved results is preferred than zero-shot finetuning.

Table 5.1: Evaluating LLMs on Torch Hub, HuggingFace, and Tensorflow Hub APIs

LLM (retriever)	TorchHub			HuggingFace			TensorFlow Hub		
	overall \uparrow	hallu \downarrow	err \downarrow	overall \uparrow	hallu \downarrow	err \downarrow	overall \uparrow	hallu \downarrow	err \downarrow
LLAMA (0-shot)	0	100	0	0.00	97.57	2.43	0	100	0
GPT-3.5 (0-shot)	48.38	18.81	32.79	16.81	35.73	47.46	41.75	47.88	10.36
GPT-4 (0-shot)	38.70	36.55	24.7	19.80	37.16	43.03	18.20	78.65	3.13
Claude (0-shot)	18.81	65.59	15.59	6.19	77.65	16.15	9.19	88.46	2.33
Gorilla (0-shot)	59.13	6.98	33.87	71.68	10.95	17.36	83.79	5.40	10.80
LLAMA (BM-25)	8.60	76.88	14.51	3.00	77.99	19.02	8.90	77.37	13.72
GPT-3.5 (BM-25)	38.17	6.98	54.83	17.26	8.30	74.44	54.16	3.64	42.18
GPT-4 (BM-25)	35.48	11.29	53.22	16.48	15.93	67.59	34.01	37.08	28.90
Claude (BM-25)	39.78	5.37	54.83	14.60	15.82	69.58	35.18	21.16	43.64
Gorilla (BM-25)	40.32	4.30	55.37	17.03	6.42	76.55	41.89	2.77	55.32
LLAMA (GPT-Index)	14.51	75.8	9.67	10.18	75.66	14.20	15.62	77.66	6.71
GPT-3.5 (GPT-Index)	60.21	1.61	38.17	29.08	7.85	44.80	65.59	3.79	30.50
GPT-4 (GPT-Index)	59.13	1.07	39.78	44.58	11.18	44.25	43.94	31.53	24.52
Claude (GPT-Index)	60.21	3.76	36.02	41.37	18.81	39.82	55.62	16.20	28.17
Gorilla (GPT-Index)	61.82	0	38.17	47.46	8.19	44.36	64.96	2.33	32.70
LLAMA (Oracle)	16.12	79.03	4.83	17.70	77.10	5.20	12.55	87.00	0.43
GPT-3.5 (Oracle)	66.31	1.60	32.08	89.71	6.64	3.65	95.03	0.29	4.67
GPT-4 (Oracle)	66.12	0.53	33.33	85.07	10.62	4.31	55.91	37.95	6.13
Claude (Oracle)	63.44	3.76	32.79	77.21	19.58	3.21	74.74	21.60	3.64
Gorilla (Oracle)	67.20	0	32.79	91.26	7.08	1.66	94.16	1.89	3.94

Table 5.2: Gorilla 0-shot with GPT 3-shot incontext examples

	HF (Acc \uparrow)	TH (Acc \uparrow)	TF (Acc \uparrow)
GPT-3.5 (0s)	16.81	41.93	41.75
GPT-4 (0s)	19.80	54.30	18.20
GPT-3.5 (3i)	25.77	73.11	71.82
GPT-4 (3i)	26.32	75.80	77.37
Gorilla (0s)	58.05	75.80	83.79

Hallucination with LLM One phenomenon we observe is that zero-shot prompting with LLMs (GPT-4/GPT-3.5) to call APIs results in dire hallucination errors. These errors, while diverse, commonly manifest in erroneous behavior such as the model invoking the `AutoModel.from_pretrained(dir_name)` command with arbitrary GitHub repository names. Surprisingly, we also found that in TorchHub, HuggingFace and TensorFlow Hub, GPT-3.5 has less hallucination errors than GPT-4. This finding is also consistent for the settings when various retrieving methods are provided: 0-shot, BM25, GPT-Index and the

Table 5.3: Comparison of retrieval techniques

	Gorilla without Retriever				Gorilla with Oracle retriever			
	zero-shot	BM25	GPT-Index	Oracle	zero-shot	BM25	GPT-Index	Oracle
Torch Hub (overall) \uparrow	59.13	37.63	60.21	54.83	0	40.32	61.82	67.20
HuggingFace (overall) \uparrow	71.68	11.28	28.10	45.58	0	17.04	47.46	91.26
TensorHub (overall) \uparrow	83.79	34.30	52.40	82.91	0	41.89	64.96	94.16
Torch Hub (Hallu) \downarrow	6.98	11.29	4.30	15.59	100	4.30	0	0
HuggingFace (Hallu) \downarrow	10.95	46.46	41.48	52.77	99.67	6.42	8.19	7.08
TensorHub (Hallu) \downarrow	5.40	20.43	19.70	13.28	100	2.77	2.33	1.89

	Accuracy
Gorilla AST metric (proposed)	0.78
Eval by Human	0.78
Code Executable (Eval by Human)	0.72

Table 5.4: AST evaluation metric has strong correlation with Accuracy

oracle. This might suggest that RLHF plays a central role in turning the model to be truthful. Additional examples and discussion are in [D.3](#).

AST as a Hallucination Metric We manually executed Gorilla’s API generations to evaluate how well AST works as an evaluation metric. Executing every code generated is impractical within academic setting—for example, executing the HuggingFace model needs the required library dependencies (e.g., transformers, sentencepiece, accelerate), correct coupling of software kernels (e.g., torch vision, torch, cuda, cudnn versions), and required hardware support (e.g., A100 40G gpus). Hence, to make it tractable, we sampled 100 random Gorilla generations from our evaluation set. The accuracy using AST subtree matching is 78%, consistent with human evaluation of 78% accuracy in calling the right API. All the generations that AST flagged as incorrect, were the same ones that were manually flagged as incorrect. Additionally, Gorilla also generates supporting code to call the API which includes installing dependencies e.g., `pip install transformers[sentencepiece]`, environment variables, etc. When we manually attempted to execute the code, 72% of all codes generated were executed successfully. It’s worth noting that the 6% discrepancy are not semantic errors, but errors that arose due to factors external to the API, and in the supporting code. We have included the full example to illustrate this further in [D.3](#). Considering the significant time and effort required for manual validation of each generation, the strong correlation between human evaluation and the AST evaluation further reinforces our belief in the efficiency of using AST as a robust offline metric.

Default Response	Update the model	Update the model repository
<p>USER: I want to automatically remove the background from an input image. <Retrieval: ..fcn_resnet50..></p> <p>GORILLA: api_call: "torch.hub.load('pytorch/vision', 'fcn_resnet50', pretrained=True)"</p>	<p>USER: I want to automatically remove the background from an input image. <Retrieval: ..fcn_resnet101..></p> <p>GORILLA: api_call: "torch.hub.load('pytorch/vision', 'fcn_resnet101', pretrained=True)"</p>	<p>USER: I want to automatically remove the background from an input image. <Retrieval: ..NVIDIA/DeepLearningExamples:torchhub..></p> <p>GORILLA: api_call: "torch.hub.load('NVIDIA / DeepLearningExamples:torchhub', 'fcn_resnet50', pretrained=True)"</p>

Figure 5.5: Gorilla’s retriever-aware training enables it to react to changes in the APIs. The second column demonstrates changes in model upgrading FCN’s ResNet-50 backbone to ResNet-101. The third column demonstrate changes in model registry from `pytorch/vision` to `NVIDIA/DeepLearningExamples:torchhub`

Test-Time Documentation Change

The rapidly evolving nature of API documentation presents a significant challenge for the application of LLMs in this field. These documents are often updated at a frequency that outpaces the re-training or fine-tuning schedule of LLMs, making these models particularly brittle to changes in the information they are designed to process. This mismatch in update frequency can lead to a decline in the utility and reliability of LLMs over time.

However, with the introduction of Gorilla’s retriever-aware training, we can readily adapt to changes in API documentation. This novel approach allows the model to remain updated and relevant, even as the API documentation it relies on undergoes modifications. This is a pivotal advancement in the field, as it ensures that the LLM maintains its efficacy and accuracy over time, providing reliable outputs irrespective of changes in the underlying documentation.

For instance, consider the scenario illustrated in Figure 6, where the training of Gorilla has allowed it to react effectively to changes in APIs. This includes alterations such as upgrading the FCN’s ResNet-50 backbone to ResNet-101, as demonstrated in the second column of the figure. This capability ensures that the LLM remains relevant and accurate even as the underlying models and systems undergo upgrades and improvements. Furthermore, the third column in Figure 6 shows how Gorilla adapts to changes in the model registry from `pytorch/vision` to `NVIDIA/DeepLearningExamples:torchhub`. This reflects the model’s ability to adjust to shifts in API sources, which is vital as organizations may change their preferred model registries over time.

In summary, Gorilla’s ability to adapt to test-time changes in API documentation offers numerous benefits. It maintains its accuracy and relevance over time, adapts to the rapid pace of updates in API documentation, and adjusts to modifications in underlying models and systems. This makes it a robust and reliable tool for API calls, significantly enhancing its practical utility.

Table 5.5: Evaluating LLMs on constraint-aware API invocations

	GPT-3.5				GPT-4				Gorilla			
	0-shot	BM25	GPT-Index	Oracle	0-shot	BM25	GPT-Index	Oracle	0-shot	BM25	GPT-Index	Oracle
Torch Hub (overall)	73.94	62.67	81.69	80.98	62.67	56.33	71.11	69.01	71.83	57.04	71.83	78.16
Torch Hub (Hallu)	19.01	30.98	14.78	14.08	15.49	27.46	14.08	9.15	19.71	39.43	26.05	16.90
Torch Hub (err)	7.04	6.33	3.52	4.92	21.83	16.19	14.78	21.83	8.45	3.52	2.11	4.92
Accuracy const	43.66	33.80	33.09	69.01	43.66	29.57	29.57	59.15	47.88	30.28	26.76	67.60
	LLAMA				Claude							
	0-shot	BM25	GPT-Index	Oracle	0-shot	BM25	GPT-Index	Oracle				
Torch Hub (overall)	0	8.45	11.97	19.71	29.92	81.69	82.39	81.69				
Torch Hub (Hallu)	100	91.54	88.02	78.87	67.25	16.19	15.49	13.38				
Torch Hub (err)	0	0	0	1.4	2.81	2.11	2.11	4.92				
Accuracy const	0	6.33	3.52	17.60	17.25	29.57	31.69	69.71				

API Call with Constraints

We now focus on the language model’s capability of understanding constraints. For any given task, which API call to invoke is typically a tradeoff between a multitude of factors. In the case of RESTful APIs, it could be the cost of each invocation (\$), and the latency of response (ms), among others. Similarly, within the scope of ML APIs, it is desirable for Gorilla to respect constraints such as accuracy, number of learnable parameters in the model, the size on disk, peak memory consumption, FLOPS, etc. We present the underlying ablation study evaluating the ability of different models in zero-shot and with retrievers settings to respect a given accuracy constraint. This setting is best understood with an example. If the user were to ask for an Image classification model that achieves at least 80% top-1 accuracy on the Imagenet dataset, then while both are classification models hosted by Torch Hub, ResNeXt-101 32x16d with a top-1 accuracy of 84.2% would be the right model whose API to call and not, say, MobileNetV2 which has a top-1 accuracy of 71.88%.

For Table 5.5, we filtered a subset of the Torch Hub dataset that had accuracy defined for at least one-dataset in its model card (65.26% of TorchHub dataset in Table 5.1). We notice that with constraints, understandably, the accuracy drops across all models, with and without a retriever. Gorilla is able to match performance with the best-performing model GPT-3.5 when using retrievals (BM25, GPT-Index) and has the highest accuracy in the Zero-shot case. This highlights Gorilla’s ability to navigate APIs while considering the trade-offs between different constraints.

5.5 conclusion

LLMs are swiftly gaining popularity across diverse domains. In our study, we spotlight techniques designed to enhance the LLM’s ability to accurately identify the appropriate API. Since APIs as a universal language enabling diverse systems to communicate effectively, their correct usage can boost the ability of LLMs to interact with tools. In this paper, we proposed

Gorilla, a novel pipeline for finetuning LLMs to call APIs. The finetuned model’s performance surpasses prompting the state-of-the-art LLM in three massive datasets we collected. Gorilla generates reliable API calls to ML models without hallucination, demonstrates an impressive capability to adapt to test-time API usage changes, and can satisfy constraints while picking APIs.

5.6 Acknowledgement

This work was supported in part by NSF grants CCF:1900968, CCF:1908870 and by SKY Lab industrial sponsors and affiliates Astronomer, Google, IBM, Intel, Lacework, Microsoft, Mohamed Bin Zayed University of Artificial Intelligence, Nexla, Samsung SDS, Uber, and VMware. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and do not necessarily reflect the position of the sponsors.

Chapter 6

RAFT: Document Understanding of LLM Agents

Pretraining Large Language Models (LLMs) on large corpora of textual data is now a standard paradigm. When using these LLMs for many downstream applications, it is common to additionally bake in new knowledge (e.g., time-critical news, or private domain knowledge) into the pretrained model either through RAG-based-prompting, or finetuning. However, the optimal methodology for the model to gain such new knowledge remains an open question. In this paper, we present Retrieval Augmented Fine Tuning (RAFT), a training recipe that improves the model’s ability to answer questions in an ”open-book” in-domain setting. In RAFT, given a question, and a set of retrieved documents, we train the model to ignore those documents that don’t help in answering the question, which we call, distractor documents. RAFT accomplishes this by citing verbatim the right sequence from the relevant document that would help answer the question. This coupled with RAFT’s chain-of-thought-style response helps improve the model’s ability to reason. In domain specific RAG, RAFT consistently improves the model’s performance across PubMed, HotpotQA, and Gorilla datasets, presenting a post-training recipe to improve pre-trained LLMs to in-domain RAG.

6.1 Introduction

Trained on vast quantities of public data, Large Language Models LLMs have achieved significant advances in a wide range of general knowledge reasoning tasks [18, 206].

However, increasingly LLMs are being employed in specialized domains to support tasks ranging from code completion for specific software frameworks to question answering on specific document collections (e.g., legal or medical documents). In these settings, general knowledge reasoning is less critical but instead, the primary goal is to maximize accuracy based on a given set of documents. Indeed, adapting LLMs to the specialized domains (e.g., recent news, enterprise private documents, or program resources constructed after the training cutoff) is essential to many emerging applications [194, 84] and is the focus of this

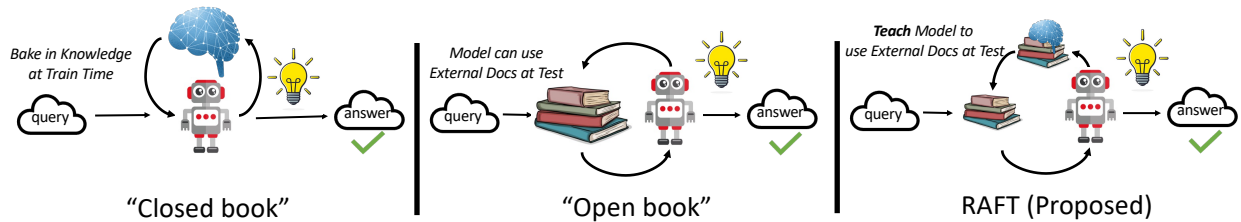


Figure 6.1: **How best to prepare for an Exam?** (a) Fine-tuning based approaches implement "studying" by either directly "memorizing" the input documents or answering practice QA without referencing the documents. (b) Alternatively, in-context retrieval methods fail to leverage the learning opportunity afforded by the fixed domain and are equivalent to taking an open-book exam without studying. While these approaches leverage in-domain learning, they fail to prepare for open-book tests. In contrast, our approach (c) RAFT leverages fine-tuning with question-answer pairs while referencing the documents in a simulated imperfect retrieval setting — thereby effectively preparing for the open-book exam setting.

work.

This paper studies the following question – *How to adapt pre-trained LLMs for Retrieval Augmented Generation (RAG) in specialized domains?*

When it comes to adapting LLMs to specialized domains, we consider the following two candidates: in-context learning through Retrieval-Augmented Generation (RAG) and supervised fine-tuning. RAG-based methods allow the LLM to reference the documents when answering questions. However, these methods fail to leverage the learning opportunity afforded by the fixed domain setting and early access to the test documents. Alternatively, supervised fine-tuning offers the opportunity to learn more general patterns in the documents and better align to end tasks and user preferences. However, existing fine-tuning based approaches either fail to leverage the documents at test time (don't incorporate RAG) or fail to account for the imperfections in the retrieval process during training.

We can draw an analogy to an open-book exam. Existing in-context retrieval methods are equivalent to taking an open-book exam without studying. Alternatively, existing fine-tuning based approaches implement "studying" by either directly "memorizing" [218] the input documents or answering practice questions [202] without referencing the documents. While these approaches leverage in-domain learning they fail to prepare for the open-book nature of test setting.

In this paper, we study how to combine supervised fine-tuning (SFT) with retrieval augmented generation (RAG). We propose a novel adaptation strategy – Retrieval-Augmented Fine Tuning (RAFT). RAFT specifically addresses the challenge of fine-tuning LLMs to incorporate domain knowledge while also improving in-domain RAG performance. RAFT aims to not only enable models to learn domain specific knowledge through fine-tuning, but also to ensure robustness against inaccurate retrievals. This is achieved by training the models to understand the dynamics between the question posed (prompt), the domain specific

documents retrieved, and the appropriate answer. Going back to our analogy, our approach is analogous to studying for an open-book exam by recognizing relevant, and irrelevant retrieved documents.

In RAFT, we train the model to answer the question (Q) from Document(s) (D^*) to generate an answer (A^*), where A^* includes chain-of-thought [206, 7], and in the presence of distractor documents (D_k). We explain the methodology in detail in Section 6.3 and analyze the sensitivity to the number of distractor documents (k) at train- and test-time in Section 6.5. RAFT consistently outperforms Supervised-finetuning both with- and without- RAG across PubMed [38], HotpotQA [223], and HuggingFace Hub, Torch Hub, and Tensorflow Hub Gorilla datasets [136], presenting a novel, yet simple technique to improve pre-trained LLMs for in-domain RAG.

6.2 LLMs for Open-Book Exam

To understand our goal better, we expand on our analogy between training an LLM in the real-world setting of preparing for an exam.

Closed-Book Exam A closed book exam often refers to a scenario where the LLMs do not have access to any additional documents or references to answer the questions during the exam. For LLMs, this is equivalent to the scenario, for example, in which the LLM is used as a chatbot. In this scenario, the LLM draws from the knowledge baked in during pre-training and supervised finetuning to respond to the prompt.

Open Book Exam In contrast, we liken the open-book exam setting to the scenario in which the LLM can refer to external sources of information (e.g., a website or a book chapter). In such scenarios, typically, the LLM is paired with a retriever which retrieves ‘k’ documents (or specific segments of the document) which are appended to the prompt. It is only through these documents retrieved that the LLM gains access to “new knowledge”. As a result, we argue that the LLM’s performance in these settings, where it is trained as a general-purpose LLM is largely dependent on the quality of the retriever and how accurately the retriever can identify the most relevant piece of information.

Domain Specific Open-Book Exam In this paper, we focused on a narrower but increasingly popular domain than the general open book exam, called the domain specific open book exam. In domain specific open book exams, we know apriori the domain in which the LLM will be tested – used for inference. The LLM can respond to the prompt using use any and all information from this specific domain, which it has been fine-tuned on. Examples of domain specific examples include enterprise documents, latest news, code repositories belonging to an organization, etc. In all these scenarios, the LLM will be used to respond to the questions, whose answers can be found within a collection of documents (a small practical domain). The retrieval technique itself has little to no impact on the mechanism (though

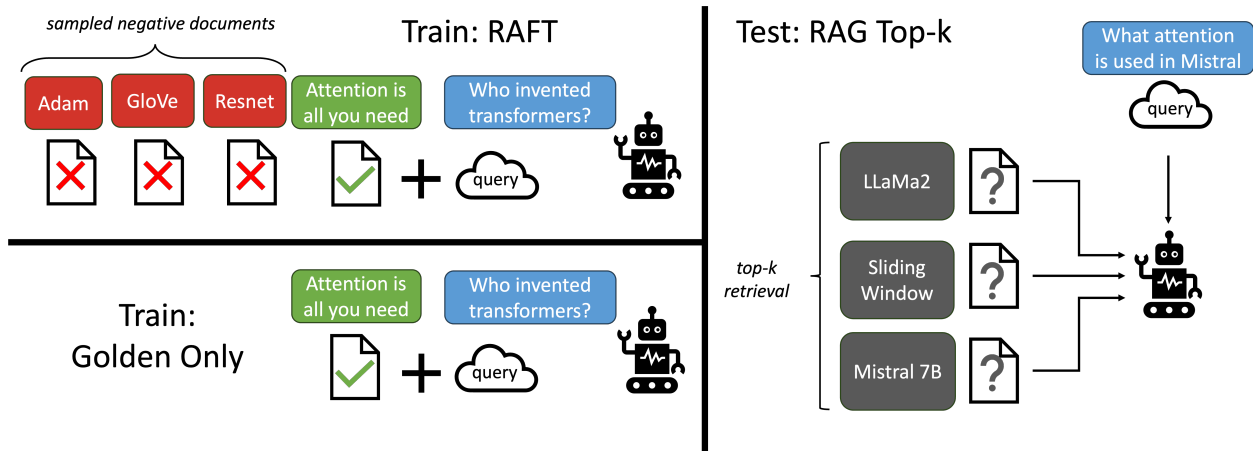


Figure 6.2: **Overview of our RAFT method.** The top-left figure depicts our approach of adapting LLMs to *reading* solution from a set of positive and negative documents in contrast to standard RAG setup where models are trained based on the retriever outputs, which is a mixture of both memorization and reading. At test time, all methods follow the standard RAG setting, provided with a top-k retrieved documents in the context.

it may impact the accuracy). This paper mainly studies this, domain specific open-book setting and how to adapt a pretrained LLM to this specific domain, including how to make it more robust to a varying number of retrieved documents and distractors.

6.3 RAFT

In this section, we present RAFT, a novel way of training LLMs for domain specific open-book exams. We first introduce the classical technique of supervised fine-tuning, followed by the key takeaways from our experiments. Then, we introduce RAFT, a modified version of general instruction tuning. Lastly, we provide an overview of the experiments to expect in the later sections.

Supervised Finetuning

Consider the supervised fine-tuning (SFT) setting for a Question-Answer dataset. The formulation consists of the Dataset (D) from which a set of Question (Q) and corresponding answer (A) pairs are derived or already available. In the classical SFT setting, the model is trained to improve its ability to answer the questions based on its knowledge - obtained either during pre-training, or during the SFT training phase. The model so trained can also be used at test-time with the Retrieval Augmented Generation (RAG) setting, where additional documents can be introduced in the prompt to help the model answer the question. This can be represented as follows:

- Train: $Q \rightarrow A$

- 0-shot Inference: $\mathbf{Q} \rightarrow \mathbf{A}$
- RAG Inference: $\mathbf{Q} + \mathbf{D} \rightarrow \mathbf{A}$

RAFT

Retrieval Aware Fine-Tuning (RAFT), presents a novel recipe to prepare fine-tuning data to tailor the models for domain specific open-book settings, equivalent to in-domain RAG. In RAFT, we prepare the training data such that each data point contains a question (Q), a set of documents (D_k), and a corresponding Chain-of-thought style answer (A^*) generated from one of the document (D^*). We differentiate between two types of documents: ‘oracle’ documents (D^*) i.e. the documents from which the answer to the question can be deduced, and ‘distractor’ documents (D_i) that do not contain answer-relevant information. As an implementation detail, the ‘oracle’ document doesn’t need to be a single document, but can be more than one document, as is the case in HotpotQA [223]. Then, for P fraction of the questions (q_i) in the dataset, we retain the oracle document (d_i^*) along with distractor documents (d_{k-1}). For $(1 - P)$ fraction of the questions (q_i) in the dataset, we include no oracle document and only include distractor documents (d_k). We then fine-tune the language model using the standard supervised training (SFT) technique, training it to generate answers from the provided documents and questions. Fig. 6.2 illustrates the high-level design principal for RAFT.

We demonstrate that our approach trains the model to perform better RAG on the set of documents it is trained on *i.e., in-domain*. By removing the oracle documents in some instances, we are compelling the model to memorize answers instead of deriving them from the context. The training data for RAFT is as follows, and an example of training data can be seen in Fig. 6.3:

- P % of data: $\mathbf{Q} + \mathbf{D}^* + \mathbf{D}_2 + \dots + \mathbf{D}_k \rightarrow \mathbf{A}^*$
- $(1 - P)$ % of data: $\mathbf{Q} + \mathbf{D}_1 + \mathbf{D}_2 + \dots + \mathbf{D}_k \rightarrow \mathbf{A}^*$

Subsequently, for the test scenario, the model is provided with the Q and top- k documents retrieved by the RAG pipeline. Note that RAFT is independent of the retriever used.

A key factor in enhancing training quality is the generation of a reasoning process, such as Chain-of-Thought, to explain the provided answers. RAFT approach is similar: we demonstrate that creating a full reasoning chain and in addition, clearly citing sources enhances the model’s accuracy in answering questions. In Fig. 6.3, we illustrate this set-up. Generating the training data in this fashion, involves presenting the model with a question, context, and verified answers, and then requesting it to form a reasoning chain that appropriately references the original context.

For all the datasets in our experiments, we generate the answers using the technique described above. Note that the Gorilla APIBench dataset, already includes reasoning in the answers. We provide an example of the generation step in Fig. 6.3, the detailed reasoning answer includes a citation from the original context inside `##begin_quote##` and

```

Question: The Oberoi family is part of a hotel company that has a head office in what
city?

context: [The Oberoi family is an Indian family that is famous for its involvement in
hotels, namely through The Oberoi Group]...[It is located in the city center of
Jakarta, near Mega Kuningan, adjacent to the sister JW Marriott Hotel. It is operated
by The Ritz-Carlton Hotel Company. The complex has two towers that comprise a hotel and
the Airlangga Apartment respectively]...[The Oberoi Group is a hotel company with its
head office in Delhi.]

Instruction: Given the question, context, and answer above, provide a logical reasoning
for that answer. Please use the format of: ##Reason: {reason} ##Answer: {answer}.

-----

CoT Answer: ##Reason: The document ##begin_quote## The Oberoi family is an Indian
family that is famous for its involvement in hotels, namely through The Oberoi Group.
##end_quote## establishes that the Oberoi family is involved in the Oberoi group, and
the document ##begin_quote## The Oberoi Group is a hotel company with its head office
in Delhi. ##end_quote## establishes the head office of The Oberoi Group. Therefore, the
Oberoi family is part of a hotel company whose head office is in Delhi. ##Answer: Delhi

```

Figure 6.3: RAFT prompt to help LLM evaluate its own generated reasoning and answers, contrasting them with the correct reasoning and answers. The LLM is prompted to identify errors in its reasoning and extract key insights for improvement. This figure specifically represents the ‘GenerateExplanation’ step in the RAFT algorithm (Chapter [6.3](#)).

##end_quote## as well as the detailed explanation on how to reach the conclusion based on the citations. We demonstrate that adding detailed reasoning paragraphs helps boost the model’s performance in our experiment section.

6.4 Evaluation

We design our experiments to study how well RAFT performs compared to various baselines. We find that the RAFT-7B model (a finetuned version of LLaMA-2) is better at reading and extracting information from in-domain documents, than domain specific finetuned model, and general-purpose model with RAG. As an ablation, we also demonstrate how important it is for the model to learn with Chain-of-Thought responses. In this section, we will first introduce all the datasets we used in the experiments, then all the baseline model/fine-tuning techniques that we benchmark against.

Table 6.1: **RAFT improves RAG performance for all specialized domains:** Across PubMed, HotpotQA, HuggingFace, Torch Hub, and Tensorflow Hub, we see that domain specific Finetuning improves significantly of the performance of the base model, but RAFT consistently outperforms the existing domain specific finetuning method with or without RAG. This suggests the need to train the model with context. We compare our model with LLaMA finetuning recipes, and provide GPT-3.5 for reference.

	PubMed	HotpotQA	HuggingFace	Torch Hub	TensorFlow Hub
GPT-3.5 + RAG	71.60	41.5	29.08	60.21	65.59
LLaMA2-7B	56.5	0.54	0.22	0	0
LLaMA2-7B + RAG	58.8	0.03	26.43	08.60	43.06
DSF	59.7	6.38	61.06	84.94	86.56
DSF + RAG	71.6	4.41	42.59	82.80	60.29
RAFT (LLaMA2-7B)	73.30	35.28	74.00	84.95	86.86

Datasets

In our experiments, we use the following datasets to evaluate our model and all baselines. We selected these datasets to represent both popular and diverse domains including Wikipedia, Coding/API documents, and question-answering on medical documents.

- Natural Questions (NQ) [81], Trivia QA [71] and HotpotQA [223] are the open-domain question-answers based on Wikipedia, mainly focused on common knowledge (e.g., movies, sports, etc).
- HuggingFace, Torch Hub, and TensorFlow Hub are from the APIBench [136] proposed in the Gorilla paper. These benchmarks measure how to generate the correct, functional, and executable API calls based on the documentation.
- PubMed QA [70] is a question-answering dataset tailored only for biomedical-research question-answering. It mainly focuses on answering medical and biology questions based on a given set of documents.

Note that the first category of dataset (NQ, Trivia QA, and HotpotQA) is a relatively general domain whereas the latter two domains are on very domain specific documents.

Baselines We consider the following baselines for our experiments:

- LLaMA2-7B-chat model with 0-shot prompting: this is the commonly used instruction-finetuned model for QA tasks, where we provide clearly written instructions, but no reference documentation.
- LLaMA2-7B-chat model with RAG (Llama2 + RAG): similar to the previous setting, except here we include reference documents. This is a popular technique when dealing with domain specific QA tasks.
- domain specific Finetuning with 0-shot prompting (DSF): Performing standard supervised finetuning, without documents in context. We find that it is mostly useful to align the answering style of the model as well as get familiar with the domain context.
- domain specific Finetuning with RAG (DSF + RAG): Equip a domain specific finetuned model with external knowledge using RAG. So, for the “knowledge” the model does not know, it can still refer to the context.

Results

Using the above datasets and baselines, we evaluate our model RAFT and demonstrate the effectiveness of RAFT in Tab. 6.1. We see that RAFT consistently and significantly outperforms the baselines. Compared with the base Llama-2 instruction-tuned model, RAFT with RAG does much better in terms of extracting information as well as being robust towards distractors. The gain can be as big as 35.25% on Hotpot QA and 76.35% on Torch Hub evaluation. Compared with DSF on the specific dataset, our model does better at relying on the provided context to solve the problem. RAFT does much better on tasks like HotpotQA and HuggingFace datasets (30.87% on HotpotQA and 31.41% on HuggingFace). Note that for PubMed QA, since it is a binary yes/no question, we don’t observe significant gains when we compare our model with DSF + RAG. Even compared with a much larger and better model GPT-3.5, RAFT demonstrates significant advantages.

Overall, the LLaMA-7B model, both with and without the RAG, performs poorly due to its answering style not aligning with the ground truth. By applying domain specific tuning, we significantly enhance its performance. This process enables the model to learn and adopt the appropriate style of answering. However, introducing RAG to a domain-specifically finetuned (DSF) model doesn’t invariably lead to better outcomes. This might indicate that the model lacks training in context processing and extracting useful information from it. By incorporating our method, RAFT, we train the model not only to match its answering style with that required but also to improve its document processing capabilities. Consequently, our approach outperforms all others.

Effect of CoT

We also conduct an analysis to evaluate the effectiveness of the Chain-of-Thought approach in enhancing the model’s performance. As indicated in Table 6.2, simply providing the answer

Table 6.2: **Ablation on Chain-of-Thought:** The numbers of RAFT and RAFT without CoT. Results on various datasets show that adding CoT can significantly improve the performance of the finetuned model. With a gain of 9.66% and 14.93% on the Hotpot QA and HuggingFace datasets respectively.

	PubMed	HotpotQA	HuggingFace	Torch Hub	TensorFlow Hub
RAFT w.o CoT	68.30	25.62	59.07	86.56	83.21
RAFT	73.30	35.28	74.00	84.95	86.86

to a question may not always be adequate. This approach can lead to a rapid decrease in loss, resulting in the training process to diverge. Incorporating a reasoning chain that not only guides the model to the answer but also enriches the model’s understanding can improve the overall accuracy. In our experiments, integrating the Chain-of-Thought significantly enhances training robustness. We employ GPT-4-1106 to generate our Chain-of-Thought prompts and include an example of the prompt we used in Figure 6.3.

Qualitative Analysis

To illustrate the potential advantages of RAFT over the domain-specifically fine-tuned (DSF) approach, we present a comparative example in Figure 6.4. This example qualitatively demonstrates a scenario where the DSF model becomes confused by a question asking for the identity of a screenwriter. Instead of providing the correct name, it mistakenly cites one of the films written by the screenwriter. In contrast, the RAFT model accurately answers the question. This discrepancy suggests that training a model solely with question-answer pairs may impair its ability to derive relevant context from provided documents. The comparison underscores the importance of incorporating both standard instructional tuning and context comprehension into the training dataset to preserve and enhance the model’s ability to process text effectively.

Should we train the LLM always with the oracle context for RAG?

In our exploration of whether large language models (LLMs) should always be trained with the oracle context for Retrieval-Augmented Generation (RAG), we address a key question: what proportion (p%) of the training data should include oracle documents? Intuitively, one might assume that for effective training in reading and extracting information from context (e.g., RAG tasks), the oracle document should always be included during training (P = 100%). However, our findings challenge this assumption: incorporating a portion of the training data without the oracle document in the context (P = 80%) appears to enhance the model’s performance on RAG tasks.

HotPot QA

Question: What screenwriter with credits for “Evolution” [0/1879] is a film starring Nicolas Cage and Téa Leoni?

Documents: ... David Weissman is a screenwriter and director. His film credits include “The Family Man” (2000), “Evolution” (2001), and “When in Rome” (2010).
...
The Family Man is a 2000 American romantic comedy-drama film directed by Brett Ratner, written by David Diamond and David Weissman, and starring Nicolas Cage and Téa Leoni. ...

RAFT: ##Reason: The screenwriter with credits for the film “Evolution,” starring Nicolas Cage and Téa Leoni, is David Weissman. This information is provided in the reference documents which mention David Weissman as a screenwriter with film credits including “The Family Man” (2000), “Evolution” (2001), and “When in Rome” (2010). Therefore, the screenwriter for “Evolution” is David Weissman. ##Answer: David Weissman ✓

DSF: The Family Man ✗

Figure 6.4: **Comparison of RAFT and DSF:** We prompt RAFT and DSF fine-tuned models on the HotpotQA dataset. We can see that the DSF model extracts the wrong information from the context. For the question, who is the screenwriter, it responds with a film name. RAFT manages to get the result correctly .

Fig. 6.5 presents our investigation into the hyperparameter $P\%$, which represents the percentage of training instances that should include oracle documents. Our analysis reveals that the optimal proportion varies across datasets, with figures ranging from 40%, 60%, and 100%. This indicates that training your LLM without the correct corresponding context at times can be beneficial for the downstream task of answering questions related to the documents. In our training setup, we include four distractor documents alongside the oracle document, and at test time, we maintain this format by providing the oracle document with four distractors. Our findings suggest that, for domain specific RAG tasks, including a certain percentage of training data without the oracle documents in the context proves to be advantageous.

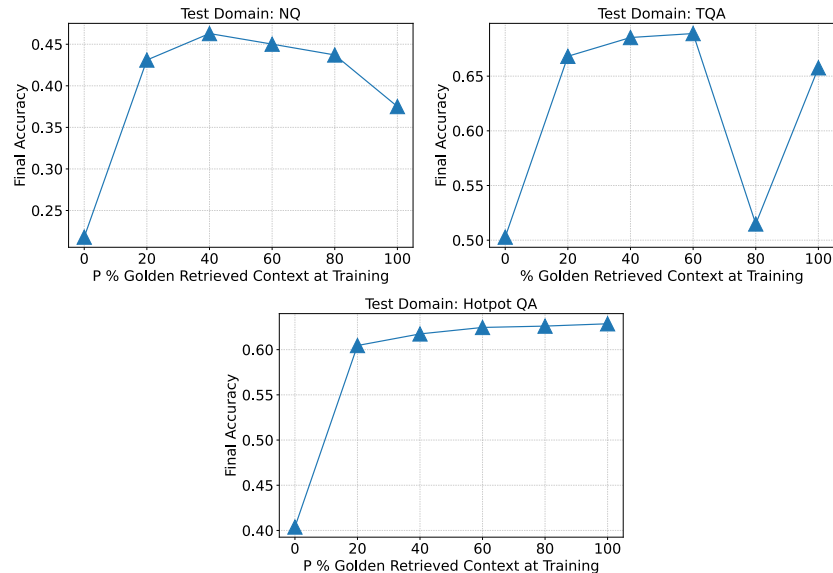


Figure 6.5: **How many golden documents to involve?** We study the hyperparameter $P\%$ which indicates what fraction of the training data contains the oracle document(s) in its context. Results on NQ, TQA and HotpotQA suggest that mixing a fraction of data that does not have the oracle document in its context is helpful for in-domain RAG.

6.5 RAFT Generalizes to Top-K RAG

After demonstrating the performance of RAFT on various benchmarks, we now study another important problem: How does the number of distractor documents in RAFT affect the model’s performance when augmented with top-k retriever augmented generation (RAG) result during the evaluation? Previous research has highlighted the vulnerability of LLMs to irrelevant text (see studies [167, 210, 101]). This issue is particularly critical for LLMs + RAG since top-k RAG is frequently employed at test time to ensure high recall. Such a scenario necessitates the model to have the ability to discern and disregard irrelevant content, focusing solely on pertinent information.

Making Model Robust to top-K RAG

To tackle the challenge of enhancing large language models’ (LLMs) ability to sift through irrelevant text within the retrieval pipeline, our analysis revealed that training solely with oracle (highly relevant) documents can inadvertently diminish the model’s ability to discern and disregard irrelevant information. To address this, our algorithm, RAFT, adopts a strategy that integrates oracle documents with a mix of irrelevant ones. This methodology prompts us to investigate the ideal fraction of negative (irrelevant) documents to incorporate throughout the training process and to assess how well this training approach adapts to

different volumes of documents encountered by the Retrieval-Augmented Generation (RAG) during the test phase. Our aim is to refine the balance between relevant and irrelevant information to strengthen the model’s efficiency in identifying and utilizing pertinent content. Notice that Sec 6.4 looked at what P% of training data should include distractors, while in this section, we study test-time scenarios.

Training with Negative Documents To enhance the robustness of large language models (LLMs) against irrelevant text in retrieved documents, we adopted a finetuning approach that incorporates both golden (highly relevant) documents and distractor (irrelevant) documents. The model was trained with varying numbers of distractor documents, but consistently evaluated using the top-k documents obtained from the retriever - not to be confused with p .

Our findings, detailed in Fig. 6.6, reveal that finetuning with only the oracle document frequently results in inferior performance compared to configurations that include a greater number of distractor documents. As we can see in the figure, the better performance for Natural Questions is training with $D^* + 3D$ and it is $D^* + 1D$ documents with Hotpot QA. This insight has been particularly beneficial for our algorithm, RAFT. In our experiments, we typically employ a training setup consisting of one oracle document alongside four distractor documents. This approach strikes a balance, ensuring the model is not overwhelmed by distractors while still gaining the ability to effectively discern and prioritize relevant information.

Generalization to a variable number of test-time documents. We extended our research to examine the impact of different quantities of test-time documents on the model’s performance. Specifically, our experiments focused on assessing how models, trained with varying numbers of distractor documents, respond to changes in the number of documents presented at test time.

The results, illustrated in Fig. 6.6, confirm that the inclusion of distractor documents during training indeed makes the model more resilient to fluctuations in the number of documents encountered during testing. This ability to maintain consistent performance despite variations in test-time document numbers further validates the robustness of our approach, RAFT. This finding underscores the importance of a well-calibrated training environment to prepare the model for a range of scenarios it may encounter in real-world applications.

6.6 Related Works

Retrieval-Augmented Language Models RAG enhances language models by integrating a retrieval module that sources relevant information from external knowledge bases, significantly improving performance across various NLP tasks, including language modeling [54, 17, 76, 170, 93, 169, 9, 220, 198] and open-domain question answering [63, 88]. This integration follows a “retrieve-and-read” paradigm where the retrieval module provides additional context from external sources, which the LM then uses to generate the final output. The retrieval process involves using the input as a query to fetch documents, which the LM

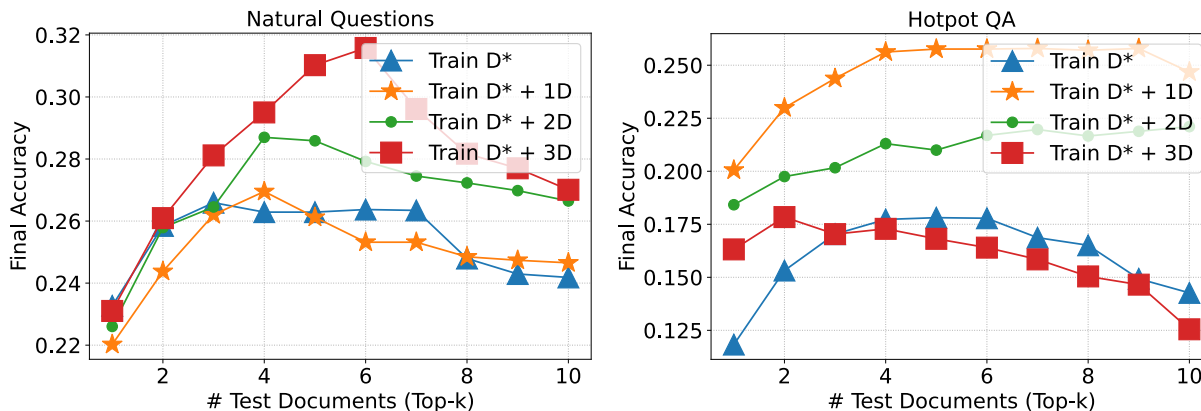


Figure 6.6: **Test-Time Documents Varying**: We study how robust RAFT is to varying numbers of test-time documents that a retriever might provide. In NQ, we find that training with 4 documents leads to the best performance, but training with 2 documents is optimal for HotpotQA. However, across both datasets, training with all datasets consisting of *oracle* documents hurts performance.

incorporates for final predictions. For instance, Atlas [63] fine-tunes T5 models with the retriever, treating documents as latent variables, while RETRO [17] modifies the decoder-only architecture to include retrieved texts and conducts pre-training from scratch. kNN-LM [76] interpolates between the LM’s next token distribution and distributions computed from retrieved tokens at inference. [170, 149] assume black-box access to an LM and combine it with either off-the-shelf or fine-tuned retriever.

Memorization A key question around large neural language models is whether they truly “understand” text [48, 140] or simply rely on surface pattern memorization [24, 184]. [48, 24, 23] develop methodologies to quantify the extent of memorization in neural models. [18, 140, 110] further explored how memorization impacts the models’ generalization capabilities. Recently, a seminal work by [22, 168] demonstrated the ability of language models to memorize and regurgitate training data, raising significant privacy concerns [73, 133].

Finetuning of LLMs Recent years have seen rapid progress in developing large-scale language models (LLMs) [18, 130, 213, 189, 189, 6]. To adapt these foundation models to downstream tasks, fine-tuning [122, 158, 34, 123, 93, 66] has become a prevalent approach. Traditional supervised fine-tuning may be limited by the cost and compute required for adapting LLMs. Addressing these challenges, research in the realm of parameter-efficient fine-tuning [57], such as Prompt Tuning [87], Prefix-Tuning [90], P-Tuning [106] and Low-Rank based fine-tuning [58], has gained traction. These methods enable LLMs to acquire domain-specific knowledge and adapt to specialized tasks such as question answering, summarization, and dialogue generation. Another branch of finetuning is through RLHF [132, 146, 97, 237], which adopts RL to align LLM’s preference with human.

Finetuning for RAG More recently, several papers have been exploring the idea of finetuning a pretrained LLM to be better at RAG tasks [94, 198, 220, 109]. These works focus on constructing a combination of finetuning dataset for RAG and train a model to perform well on these tasks. In particular, in their settings, at test time, the domain or documents can be different than the training time; whereas our paper studies a slightly opposite scenario where we only care about testing the LLM on the same set of documents.

6.7 Conclusion

RAFT is a training strategy designed to enhance the model’s performance in answering questions within a specific domain, in ”open-book” settings. This technique demonstrates a fine-tuning recipe for LLMs for question-answering tasks based on a selected collection of documents. We have pinpointed several crucial design decisions, such as training the model alongside distractor documents, organizing the dataset so a portion lacks oracle documents in their context, and formulating answers in a chain-of-thought manner with direct quotations from the relevant text. Our evaluations on PubMed, HotpotQA, and Gorilla API Bench underline RAFT’s significant potential. Looking forward, we anticipate that in-domain Retrieval-Augmented Generation (RAG) will continue to gain interest within both industrial and academic spheres. Unlike general-RAG, our work addresses practical scenarios where LLMs are tasked with answering questions using domain-specific knowledge. Aligning with current trends, our findings suggest that smaller, fine-tuned models are capable of performing comparably well in domain-specific question-answering tasks, in contrast to their generic LLM counterparts.

6.8 Acknowledgement

This work was supported in part by NSF grants CCF:1900968, CCF:1908870 and by SKY Lab industrial sponsors and affiliates Astronomer, Google, IBM, Intel, Lacework, Microsoft, Mohamed Bin Zayed University of Artificial Intelligence, Nexla, Samsung SDS, Uber, and VMware. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and do not necessarily reflect the position of the sponsors.

Part IV

Deploying LLM Agents

Chapter 7

GOEX: Towards A Runtime For Autonomous LLM Applications

Large Language Models (LLMs) are evolving beyond their classical role of providing information within dialogue systems to actively engaging with tools and performing actions on real-world applications and services. Today, humans verify the correctness and appropriateness of the LLM-generated outputs (e.g., code, functions, or actions) before putting them into real-world execution. This poses significant challenges as code comprehension is well known to be notoriously difficult. In this paper, we study how humans can efficiently collaborate with, delegate to, and supervise autonomous LLMs in the future. We argue that in many cases, “post-facto validation”—verifying the correctness of a proposed action after seeing the output—is much easier than the aforementioned “pre-facto validation” setting. The core concept behind enabling a post-facto validation system is the integration of an intuitive *undo* feature, and establishing a *damage confinement* for the LLM-generated actions as effective strategies to mitigate the associated risks. Using this, a human can now either revert the effect of an LLM-generated output or be confident that the potential risk is bounded. We believe this is critical to unlock the potential for LLM agents to interact with applications and services with limited (post-facto) human involvement. We describe the design and implementation of our open-source runtime for executing LLM actions, Gorilla Execution Engine (GoEx), and present open research questions towards realizing the goal of LLMs and applications interacting with each other with minimal human supervision.

7.1 introduction

Large Language Models (LLMs) are evolving from serving knowledge passively in chatbots to actively interacting with applications and services. Enabling agents and software systems to interact with one another has given rise to new innovative applications. In fact, it is no longer science fiction to imagine that many of the interactions on the internet are going to be between LLM-powered systems. Agentic systems [199, 214, 135], co-pilots [155], plug-

ins [129], function calling and tool use [136, 143, 134, 224, 130], are all steps towards this direction.

The logical next-step in this evolution is towards autonomous LLM-powered microservices, services, and applications. This paper is a first step towards realizing this goal, and addresses some of the key trustworthiness challenges associated with it.

As a running example, consider an LLM-powered personal assistant that has access to a user’s email account. The user asks the assistant to send an important email to their boss, but instead, the LLM sends a sensitive email to the wrong recipient. In designing such a system, several critical challenges must be addressed:

Hallucination, stochasticity, and unpredictability. LLM-based applications place an unpredictable and hallucination-prone LLM at the helm of a system traditionally reliant on trust. Currently, services and APIs assume a human-in-the-loop, or clear specifications to govern how and which tools are used in an application. In our running example, the user clicks the “Send” button after confirming the recipient and body of the email. In contrast, an LLM-powered assistant may send an email that goes against the user’s intentions, and may even perform actions unintended by the user. LLMs are not only capable of being stochastic, but crucially, are capable of unpredictable and unbounded behavior even when trained not to do so [5].

Unreliability. Given their unpredictability and impossibility to comprehensively test, it is difficult for a user to trust an LLM off the shelf. Consequently, LLM-powered applications are challenging for users to adopt as LLMs are untrusted components that would be running within a trusted execution context. Trivially, one can ensure safety by restricting the LLM to have no credentials, at the expense of losing utility. Given that prior work [130, 136, 163] has surfaced the growing utility of LLM-based systems, mechanisms are needed to express the safety-utility tradeoff to developers and users.

Delayed feedback and downstream visibility. Lastly, from a system-design principle, unlike chatbots and agents of today, LLM-powered systems of the future will not have immediate human feedback. This means that the intermediate state of the system is not immediately visible to the user and often only downstream effects are visible. An LLM-powered assistant may interact with many other tools (e.g., querying a database, browsing the web, or filtering push notifications) before composing and sending an email. Such interactions before the email is sent are invisible to the user.

A Runtime that Enables Autonomous LLMs

The question this paper tries to address is: How do we enable an untrusted agent to take sensitive actions (e.g., code generation, API calls, and tool use) on a user’s behalf and then verify that those actions aligned with the intent of that user’s request?

Traditional methods such as using containers which guarantee isolation through virtualization falter when adjustments to the environment are required within the user’s context. For example, a user might want to modify the state of their operating system. Further, isolation alone cannot ensure the final state aligns with the intended actions, especially under

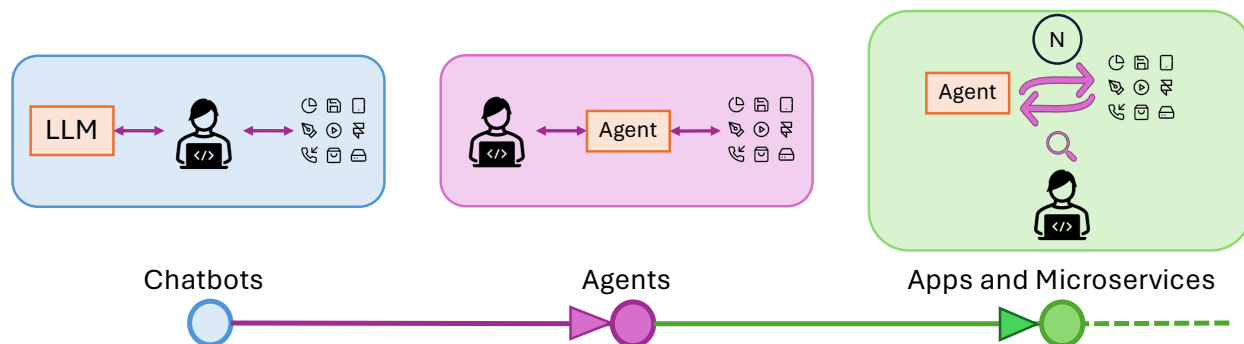


Figure 7.1: Evolution of LLMs powered applications and services from chatbots, to decision-making agents that can interact with applications and services with human-supervision, to autonomous LLM-agents interacting with LLM-powered apps and services with minimal and punctuated human supervision.

ambiguity or execution errors and ambiguity of stated intent. Both are common flaws when intent is being specified in natural-language as opposed to a more precise domain-specific language.

In this paper, we introduce the notion of “**post-facto LLM validation**” as opposed to “pre-facto LLM validation”. While in both scenarios humans are the ultimate arbitrators, in “post-facto validation” human’s arbitrate the output produced by executing the actions produced by the LLM, as opposed to the process or the intermediate outputs. A natural risk arising from “post-facto validation” is that the actions we execute may have unintended consequences. While the benefits of evaluating the output might justify the risks involved with unintended execution, we recognize that this could be a non-starter in many applications. To remedy this, we introduce the intuitive abstractions of “undo” and “damage confinement” or “damage confinement”. The “undo” abstraction allows LLMs to back-track an action that may be unintended, for example, delete a message that was sent in Slack. And for those actions which may not have an “undo”, we present “damage confinement” semantic. “Damage confinement” can be considered as a quantification of the user’s risk appetite. For example, a user may tolerate the risk of the LLM agent delivering pizza to the wrong address, but perhaps they might not want to allow their LLM to interact with their bank.

As a step towards realizing our vision, we developed GoEx, a runtime for executing actions generated by LLMs. By designing GoEx to exclusively utilize readily available off-the-shelf software components, we aim to assess the readiness of current resources and provide an ecosystem to empower developers.

In summary, we make the following contributions:

1. We first make the case for the future of deeply-embedded LLM systems with LLMs powering microservices, applications, etc. In this paradigm, LLMs are not merely used

- for information compression, but also as decision makers (Chapter 7.2). We identify the key challenges associated with integrating LLMs into existing systems, including the inherent unpredictability of LLMs, the lack of trust in their execution, and the difficulty in detecting and mitigating their failures in real-time (Chapter 7.3).
2. We introduce the concept of “post-facto LLM validation” as an approach to ensuring the safety and reliability of LLM-powered systems, focusing on validating the results of LLM-generated actions rather than the process itself (Chapter 7.4).
 3. We introduce “undo” (Chapter 7.4) and “damage confinement” (Chapter 7.4) abstractions as mechanisms for mitigating the risk of unintended actions taken in LLM-powered systems.
 4. We propose the *Gorilla Execution Engine (GoEx)*, a runtime designed to enable the autonomous interactions of LLM-powered software systems and agents by safely executing LLM-generated actions and striking a tradeoff between safety and utility (Chapter 7.5).

7.2 Evolution of LLM powered Agents

We first present the background on the evolution of LLM-powered-systems to mean applications, microservices, and other systems that integrate- or interface- with LLMs. Based on this trend, we then speculate on what such a future would look like.

LLM-Human Interaction: Chatbots, and Search

LLMs have transformed the landscape of human-computer interaction. With early adoption as chatbots, these models were designed to mimic human conversation, allowing users to interact with computers in natural language. This era of LLMs focused on understanding and generating answers, serving as a bridge for humans to interact with vast amounts of web data in a more intuitive way. Early implementations were primarily used in the *read-only* model for information retrieval where the LLM did not make any stateful changes [20], customer service agents in enterprises, and as educational tools, laying the foundation for more sophisticated applications.

From Chatbots to Agents: The Rise of Actionable LLMs

Increasing trustworthiness [197], the availability of adapters, and novel-techniques [163, 136], have expanded the role of LLMs from passive providers of information to active agents capable of executing simple tasks. These agents, powered by LLMs, can interact with applications, services, and APIs to perform actions on behalf of the user. This shift represents a significant leap in the capabilities of LLMs, enabling them to contribute actively to workflows and processes across various domains.

However, this evolution also brings challenges. The complexity of understanding context, intent, and the subtleties of human language make it difficult to ensure the accuracy and appropriateness of the actions taken by LLMs. As a result, *human oversight* remains crucial to manage and validate the set of actions proposed by the LLM.

Towards Ubiquitous LLM Integration

Looking ahead, we expect the integration of LLMs into daily workflows and systems to deepen. This future envisions LLMs not just as tools or assistants but as pervasive agents embedded in a myriad of workflows, enhancing functionality and adaptability across the board. This vision is materialized through the development of advanced LLM-powered microservices, LLM-powered applications, and LLM-powered workflows all interacting with each other, constantly, with limited-to-no human interaction.

From an application developer's perspective, these can be categorized into personalized systems, hosted agents for collective use, and third-party integrations.

Personalized LLM-powered workflows for individuals. In the personal domain, LLMs are anticipated to become deeply integrated with individual user experiences, offering tailored assistance that understands and anticipates the unique preferences and needs of each user. Imagine a personalized LLM-powered version of Siri or Google Voice, not merely responding to queries but proactively managing schedules, filtering information based on user preferences, and even performing tasks across a range of applications and services. Such personalized systems would mark a significant departure from generic assistants to truly personalized digital companions.

Hosted agents for enterprise and group applications. Within professional and enterprise environments, hosted LLM-powered agents will take on specific roles, such as managing database queries or automating routine administrative tasks, tailored to the unique needs of an organization. These specialized agents would be operating within the confines of an organization but would serve a wide set of users.

Third-party agents: expanding the ecosystem of services. The expansion of LLM capabilities is also expected to include extensive collaboration with third-party service providers (e.g., Slack, Gmail, Dropbox, etc.), enabling seamless interactions between users and services through personalized LLM workflows. These third-party agents would allow users to communicate with and through them using customized LLM-powered workflows, and integrate a wide range of services.

7.3 Navigating the New Frontier: Challenges in Ubiquitous LLM Deployments

With advancements in LLM capabilities and their applications come new challenges: How do we contend with the inaccuracies inherent to even the best models currently available? How do we ensure security for client information? How do we handle system reliability and quantify risk tolerance to a user? In the following sections, we elaborate on these new challenges.

Delayed Signals

The challenge of timely feedback. In traditional software development, immediate feedback through error messages or direct outputs enables quick recovery mechanisms to be triggered. However, when embedding LLMs, especially in complex systems or applications interfacing with real-world data and actions, feedback can be significantly delayed. This delay in obtaining relevant signals to assess the performance or correctness of LLM actions (especially with text-in, text-out modality) introduces challenges in rapidly iterating and refining model outputs.

Impact on system development. The lag between action and feedback complicates the identification of errors and the assessment of system performance, potentially leading to additional state being built on top of the system. This necessitates designing systems that can accommodate these delays and implement strategies for asynchronous feedback collection.

Aggregate Signals

In the realm of LLMs, particularly when applied to large-scale systems or microservices, individual LLM actions may not provide clear insights into overall system performance, nor assist in diagnosing the cause of the error. Instead, the true measure of success (or failure) often emerges from aggregated outcomes, necessitating a shift in how developers and stakeholders evaluate and account for, in LLM-driven applications.

The Death of Unit-Testing and Integration-Testing

The integration of LLMs into software systems challenges traditional paradigms of unit testing and integration testing. While closed-source and continuous-pre-trained LLMs from third-parties pose a new challenge of the model changing constantly, in-house LLMs are not a panacea either. Given the dynamic, often unpredictable nature of LLM outputs, establishing a fixed suite of tests that accurately predict and verify all potential behaviors becomes increasingly difficult, if not impossible.

Variable Latency

LLMs’ auto-regressive text-generation by the very nature means inference time may vary as the LLM can either output a long or short response. This is an important consideration for hard-deadline real-time systems (RTS) [144].

Protecting Sensitive Data

In order for LLMs to interact with a user’s accounts across multiple applications, the LLM must be able to reason about having access to credentials granting access to the user’s accounts. When an LLM is hosted by an (untrusted) external service, it is desirable to not directly pass any credentials or sensitive data to the LLM while still preserving functionality of the LLM-powered system. LLMs can also generate (untrusted) code and are susceptible to hallucinations [150, 239], which can result in running potentially malicious code or inadvertently performing actions that were not intended by the user.

7.4 Designing a Runtime

To address the challenges introduced by the new paradigm of LLM-powered applications, we propose an *LLM runtime*—designed specifically to execute actions generated by LLMs—as a compelling solution. This section discusses the necessity of such a runtime, its envisioned properties, and strategies to mitigate the risks associated with LLM-generated actions.

LLMs started the problem, can LLMs solve the problem? Despite the rapid advancements in LLMs, expecting LLMs to self-correct and eliminate all potential errors or unintended actions through existing training techniques—such as pre-training, instruction tuning, DPO [147], or RLHF [246]—is promising (Chapter 7.5) but challenging. The challenges are manifold, primarily due to ill-defined metrics and the inherent complexities of accurately predicting the real-world impacts of actions suggested by LLMs. Thus, while LLMs are at the heart of these challenges, their current evolutionary trajectory suggests they cannot entirely solve the problem without external frameworks to guide their execution.

Post-facto LLM validation

In the realm of LLM-powered-systems, we introduce “post-facto LLM validation,” which contrasts with traditional “pre-facto” methods. In “post-facto validation,” humans evaluate the outcomes of actions executed by the LLM, rather than overseeing the intermediate processes. This approach assumes that validating results over processes, acknowledging that while verifying outcomes is crucial, understanding and correcting processes based on those outcomes is equally important.

Forgoing “pre-facto validation” means execution of actions without prior validation, which introduces risks and potentially leads to undesirable outcomes. We propose two

abstractions to mitigate the risk associated with post-facto validation: undoing an action (Chapter 7.4), and damage confinement (Chapter 7.4).

Reversibility

When possible, actions executed by an LLM should give users the right to *undo* an action. This approach may require maintaining multiple versions of the system state, leading to high costs in terms of memory and computational resources. Furthermore, the feasibility of implementing undoing an action is often dependent on the level of access granted to the system. For instance, in file systems or databases, where root access is available, undoing actions is possible. However, in scenarios where such privileged access is not granted, such as in email clients like Gmail, the ability to undo an action may be limited or require alternative approaches. One potential solution is for the runtime to make a local copy of the email before deleting, which introduces additional state to the runtime but enables *undo* for email deletion.

To account for the resource costs associated with maintaining multiple system states, we adopt the notion of a *commit*, also called a “watermark” in streaming data-flow systems [21, 2]. By grouping together sets of actions based on their associativity, commutativity, and distributive properties, it may be possible to define checkpoints at which the system state can be saved or rolled back. This approach would enable selective undoing of actions within a defined scope, rather than maintaining the ability to undo every individual action.

Atomicity. In agent-driven systems, the option for users to demand atomicity of operations can be crucial. Atomicity ensures that either all of the operations within a task are successfully completed, or, in the event of a failure in any step, the system is reverted to its initial state before any operation was applied. This binary outcome—success or a clean-slate reset—provides a clear, predictable framework for managing tasks, increasing the system’s reliability and user trust in the LLM agent executing complex sequences of actions.

Damage Confinement

Not all applications or tools provide the ability to undo an action. For example, emails currently cannot be unsent after some time has elapsed. In scenarios like these, we fall back to “damage confinement” or “blast-radius confinement”, as it is necessary to provide users with mechanisms to quantify and assess the associated risks of the actions their LLM-powered application may take.

One approach to address this challenge is through the implementation of coarse-grained access control mechanisms. A user could permit their LLM to only read emails instead of sending emails, thus confining the blast radius to an tolerable level. Such permissioning has already been explored preliminarily by [215], in the context of a user authorizing independent LLM applications to interact with one another.

Symbolic Credentials and Sandboxing

As mentioned in Chapter 7.3, the LLM could be (1) hosted by an (untrusted) external provider and (2) susceptible to hallucinations [150, 239], resulting in code that may be unsafe to execute.

In order to protect sensitive user information from and untrusted LLM, the sensitive information in the input prompt can be substituted with a symbolic credential (e.g., a dummy API key), similar to the anonymization approach taken by Presidio [118], and sending this sanitized prompt to the LLM. Then, the LLM will never see the user’s sensitive information in its input.

We can mitigate this risk of running potentially unsafe code by executing the generated code in a sandboxed environment, whether it be a container or a bare-metal VM. With this approach, we only grant the code access to the required dependencies and necessities, such as a specific API key used for service access, and nothing more. If utilizing a container, we only mount the necessary files and impose appropriate network restrictions.

Storing Keys and Access Control

As LLMs are inherently untrusted, a user would feel uneasy permitting the LLM based system to store their credentials. There are two challenges an LLM runtime should address: how to store and manage the user’s credentials; and determining and projecting the minimal set of permissions that an LLM needs in order to accomplish its task. More formally, mapping an action to the least privileges that need to be granted to perform the task. While solutions such as asking a ML model have the benefits of generalizability and scalability, pre-computing this permission set manually provides strong security guarantees. Finding a common ground between these two techniques, remains an interesting open area for research. Further, in an enterprise use case in which an LLM-powered application is managing many user credentials, recording an audit trail of credential access is critical.

7.5 GoEx: LLM Runtime

GoEx represents a first step towards building a runtime for executing LLM-generated actions within a secure and flexible runtime environment. Central to GoEx are its abstractions for “undo” (Chapter 7.4) and “damage confinement” or “blast-radius confinement” (Chapter 7.4), which provide developers of apps and services the flexibility to tailor policies to their specific needs, recognizing the impracticality of a one-size-fits-all policy given the varied contexts in which LLMs are deployed. GoEx supports a range of “actions” including RESTful API requests (Chapter 7.5), databases operation (Chapter 7.5), and filesystem actions (Chapter 7.5). Each action type, while initiated from a unified GoEx interface, are handled uniquely as described below.

RESTful API calls

We first describe how GoEx handles RESTful API calls (illustrated in Chapter [7.2](#)).

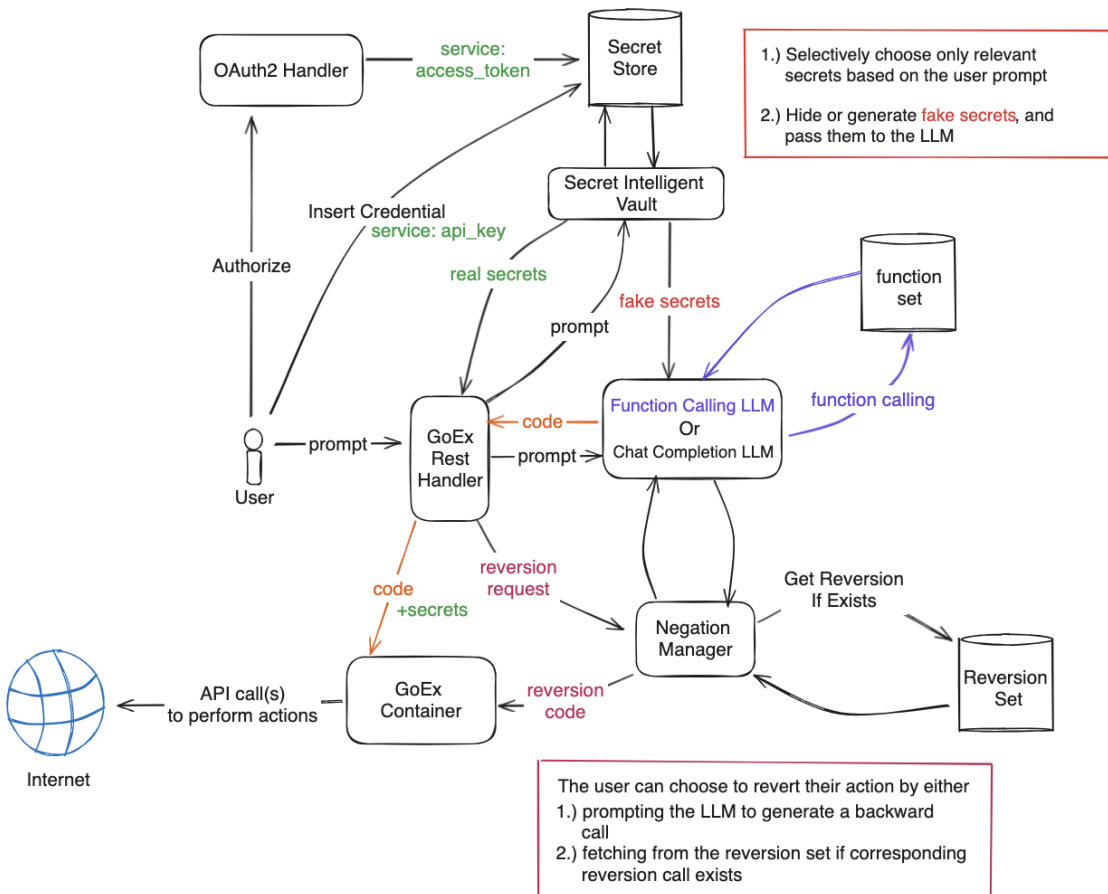


Figure 7.2: GoEx’s runtime for executing RESTful API calls. Upon receiving the user’s prompt, GoEx presents two alternatives. First, an LLM can be prompted to come up with the (Action, Undo-Action) pair. Second, the application developer can provide tuples of actions and their corresponding undo-actions (function calls) from which the LLM can pick amongst.

Authentication. GoEx provides a secure way to handle user secrets, whether using OAuth2 for token-based authentication or API keys for direct service access. GoEx acts as the secure

intermediary to facilitate authenticated actions across various services. For OAuth2, GoEx sits between the user and services, facilitating the necessary relay to retrieve access tokens. These tokens allow users to delegate the GoEx system to perform actions on their behalf. For other services that authenticate accounts through API keys, GoEx provides an interface that allows users to insert and retrieve them.

Storing secrets. User secrets and keys are stored locally on the user’s device in a Secret Intelligent Vault (SIV). SIV maps `service_name` to `key` and `format`. When user wishes to interact with specific service(s), the corresponding keys are requested from the SIV. The `format` specifies how the keys are store, that is, in a file, or as a string, etc. The role of the SIV is to selectively retrieve just the required keys for a given execution. For example, if a user wants to send an email invite to their friend for lunch, the agent only needs their OAuth2 token for their email provider, and not, for example, their bank account’s API keys. The policy used for SIV is user-defined and highly flexible; it could be as simple as parsing through the user prompt to detect which service’s keywords are present, or as complex as a fine-tuned prompt-to-service retrieval model.

Generating actions. The GoEx framework supports two techniques to generate the APIs. In the Chat Completion case, assuming the user prompt is, “send a Slack message to gorilla yahoo.com”, the user must initially authorize GoEx to use their access token through the Slack browser. After receiving the user prompt, GoEx requests the SIV for the necessary secrets from the Secret Store. Slack secrets (OAuth2) are inherently hidden because they are stored as a file, so GoEx pass the file path along with the prompt directly to the LLM. GoEx mounts the Slack secret file and passes the LLM-generated code to be executed in the GoEx container. If the user wishes to revert the execution, the reversion call will be retrieved from the reversion set if it exists; otherwise, the handler prompts the LLM to generate it. If the user chooses Function Calling, instead of asking the LLM to come up with a command to satisfy the user’s prompt, GoEx asks it to select a function from a user-defined function set and populate the arguments. Secrets will be chosen from the SIV similarly, and execution occurs in the GoEx container. If the user wishes to revert, another function from the function set will be chosen by the LLM.

Generating undo actions. Identifying the ‘undo’ action for RESTful APIs, includes the following steps. First, we check if the reverse call for the action API is in the database **Reversion Set** as shown in figure [7.2](#). GoEx presents the systems abstractions, while developers are free to define the policies for mapping. For some APIs it might be critical to check for exact match for all parameters of the API, on the other hand for some other APIs, perhaps just the API name might be sufficient to uniquely identify what the reverse API would be. For example it is *not* sufficient to say the reverse of `send_slack_message` is `delete_slack_message`, since number of messages to be deleted could be one of the arguments.

To populate such a mapping, first, we instruct the LLM to generate a reverse API call whenever the user attempts to perform an action. We recognize that this gives no guarantees, but the philosophy is that we allow the LLM to be wrong at most once. Post each new API, the table is then if the reversion worked or not making this information available for future invocations. For applications that need guarantee, developers can pre-populate this table and combined with function-calling mode of operation, the system can be forced to only use those API's that are 'guaranteed' by the developers to be reversible.

Damage confinement. Often reversibility cannot be guaranteed. For examples sending an email isn't really reversible. For such scenarios, GoEx presents abstraction to bound the worst case. Currently, the way blast-radius-containment is implemented is through coarse-grained access control, and exact string match. First, GoEx looks at the user's prompt to determine the end service that they are then authorized to use. For example, a prompt of *I would like to send a slack message* would only need credentials for slack, and not, say, their bank. GoEx currently does this, through a simple sub-string check of the prompt, while giving developers the flexibility to adopt any mapping they might choose.

Execution. Once the API, and the set of credentials required are determined, the APIs are then executed in a Docker container for isolation.

Database Operations

GoEx leverages the mature transaction semantics offered by databases. This section describes the abstractions available, and the two default policies.

Abstractions

GoEx relies on the LLM to generate database operations, but there are two prerequisites needed to execute database operations: (1) knowledge of the current database state, and (2) knowledge on how to access the database. To provide these, `DBManager` class is used. This allows the database to readily minimally query for the database state (e.g. only the schema) to provide additional info to the LLM during prompting without leaking sensitive data. It also tracks the connection configuration to the database so that connections can be established without leaking credentials to the LLM as an untrusted third-party by asking the user to store the credentials locally, and after the LLM generates the operation, GoEx then executes the operation.

`DBManager` also assists the user store with storing a previous state. Here, the *commit* and *undo* actions are introduced where a *commit* means the user permanently saves the executed changes, and an *undo* reverses the aforementioned changes. Most modern databases also provide ACID guarantees [55], including NoSQL databases like DynamoDB and MongoDB, which we leverage to implement committing and undoing actions.

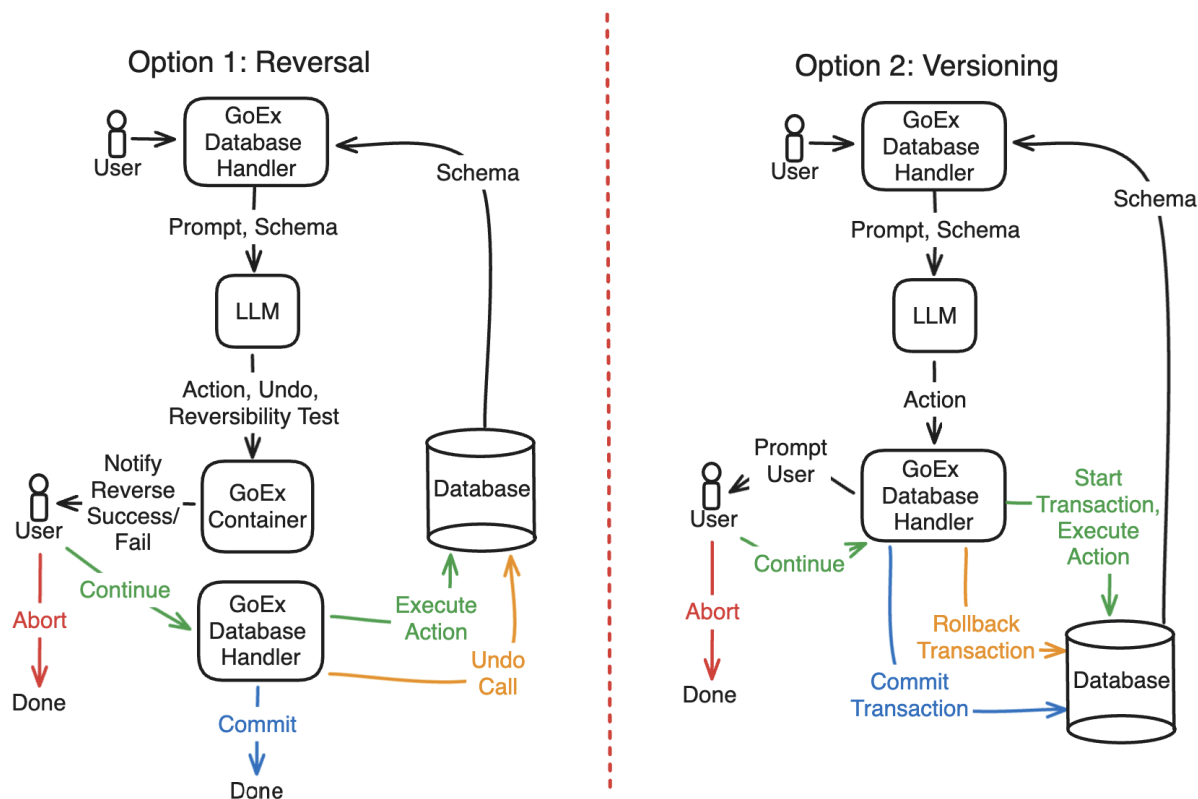


Figure 7.3: Runtime for executing actions on a database. We present two techniques to determine if a proposed action can be undone. On the left, for non-transactional databases like MongoDB, and for flexibility, we prompt the LLM to generate (Action, Undo-Action, test-bed) tuples, which we then evaluate in a isolated container to catch any false (Action, Undo-Action) pairs. On the right, we can provide a deterministic undo with guarantees by employing the transaction semantics of databases.

Policy

DBManager implements reversibility in two ways. The user chooses which one to use when they execute a prompt in GoEx.

leftmargin=* **Option 1 (Reversal)**. Makes use of a reverse database operation to perform the *undo*. It is done by prompting the LLM with the original operation (action call) along with the schema to generate the reversal operation (undo call). Committing would require no action, and undoing would just be performing the undo call after the action call is done. This option scales better as additional users can continue perform database actions without needing to wait for the previous user to finish their transaction at the cost

of relying on the LLM to come up with an undo call, which may or may not have unexpected behaviors.

Option 2 (Versioning). Makes use of the traditional ACID transaction guarantees of the database and holds off on completing a transaction until the user specifies to do so, or rolls back to the previous state. Committing would involve committing the transaction, and undoing is synonymous to a rollback transaction. This branch is able to provide reversal guarantees that branch 1 cannot, at the expense of higher performance overhead.

Reversibility testing. Within Option 1, GoEx also performs a reversibility test to verify that the generated reversal operation indeed reverses the original operation. This requires a containerized environment to be separate from the original database to maintain the original database state. Since copying over the database into the container is very expensive, the approach is to ask the LLM to generate a bare-bones version of the database for reversibility testing, given the action, undo calls, and the database schema. The outcome of the test is sent back to the user for final confirmation before committing or undoing the operation. This method allows for efficient testing by decoupling the testing runtime from being scaled by the number of entries in the database.

File Systems

GoEx tries to present expressive abstractions to let LLM-powered systems to interact with file-systems using Git version control. To track the directory tree, on every GoEx filesystem-type execution, GoEx does an exhaustive, recursive walk of the directory and its subdirectories and stores the directory structure as a formatted string.

Abstractions

Filesystems operation support in GoEx uses abstractions similar to what is used to support database operations. **FSManager**, is a filesystems manager that tracks (1) the directory tree structure with all filenames, and (2) the directory path that the user wishes to execute the filesystem's operations in. The tree structure, which is updated with executions, enables the LLM to generate operations that reflect the actual state of the user's filesystem.

Utilizing the relevant abstractions presented by journaling and log-structured filesystem for undo-semantics is left as future work, as the current GoEx system aims for compatibility.

Policy

The options are similar to the database case, where Option 1 is for reversals and Option 2 is for versioning. The largest differences are how **FSManager** carries out reversibility testing and that versioning is accomplished using Git.

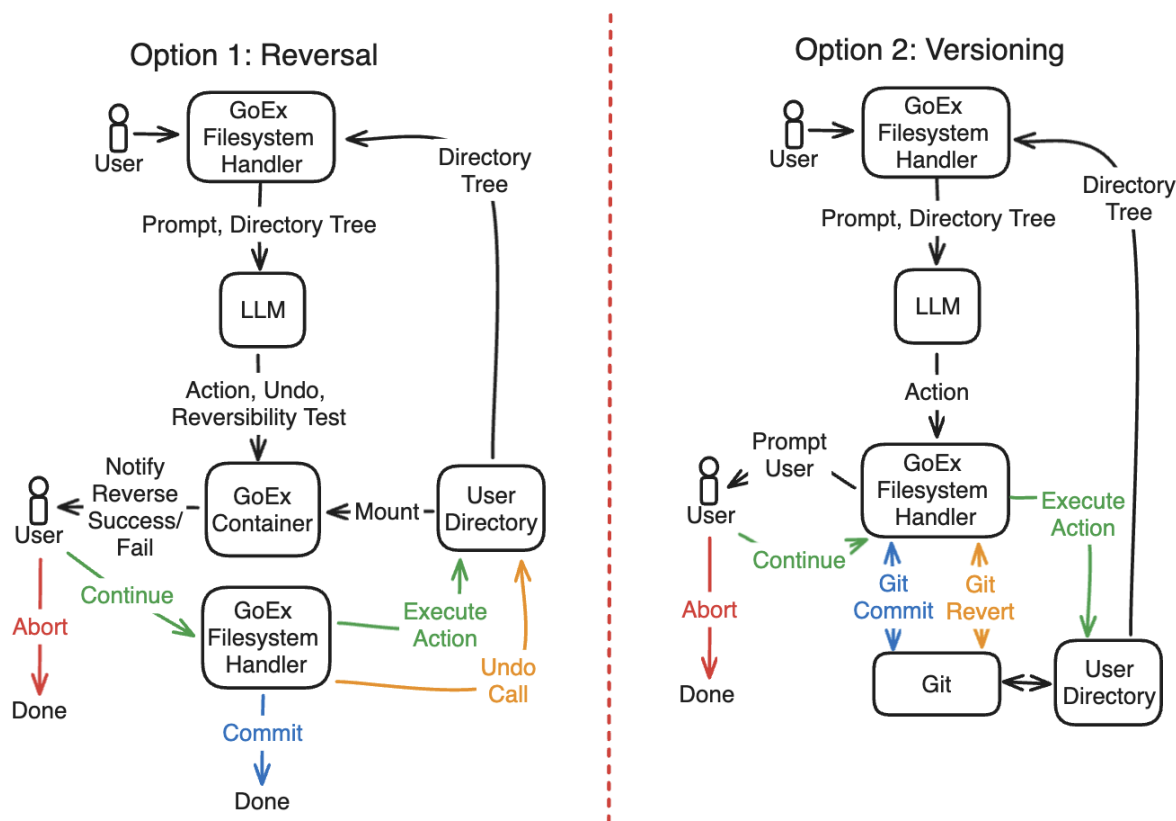


Figure 7.4: Runtime for executing actions on a filesystem. GoEx presents two abstractions. On the left, the LLM is prompted to come up with an (Action, Undo-Action, test-bed) which GoEx evaluates in a isolated container to catch any false (Action, Undo-Action) pairs. On the right presents deterministic guarantees by using versioning control system like Git or Git LFS.

Git. GoEx uses Git to perform versioning. Since Git is already a version-control system for files, it is a straightforward solution to use, but has several limitations. Git does not have the ability to version track outside of the directory that it was initialized in. GoEx limits the user execution scope to the specified path in `FSManager`—which is always inside of a Git repository—and its subdirectories in accordance to our blast-radius confinement abstraction to prevent the LLM from performing arbitrary actions in undesired parts of the user’s system. With larger directories, Git versioning can be expensive space-wise. GoEx leverages Git LFS for larger directories as an optimization. A threshold is defined for directory size that GoEx would then check whether or not to initialize Git LFS (200 MB by default).

Reversibility testing. Similar to supporting databases operations, the LLM generates the testing code using the action and undo calls, along with the directory tree. Inside

the container, the specified path is mounted in read-only mode to again do blast radius containment. GoEx begins by duplicating the directory contents in the container, then run the action and undo calls on the copied directory, and finally compare contents. Depending on the original operation, the content comparison can just be a check of filenames or an exhaustive file content comparison of all the files. We rely on the LLM to come up with the test-case. Unsurprisingly, here GoEx allows you to trade off guarantees for performance.

7.6 Discussion

Is Post-Facto LLM Validation Always Preferable?

It is not lost on us that while post-facto LLM validation has many benefits, our advocacy of it is also somewhat philosophical. For example, if one were to bake a cake it's probably better to taste the cake than check the recipe. But on the the contrary, if one were to produce an audit report, it might be preferable to check the process. We acknowledge both options—post-facto LLM validation, and pre-facto LLM validation—as two techniques to evaluate the LLM's actions, however, this paper focuses on verifying results as this is, perhaps, more appropriate in most scenarios we consider, which are complex microservice settings.

Designing LLM-friendly APIs

The conversation around LLM-powered systems design is predominantly centered around designing systems to conform with the API semantics of existing applications and services. However, an equally interesting question is what API design in an LLM-centric world would look like.

LLMs introduce a paradigm where applications and services can anticipate and adapt to the intricacies of LLM interactions. A notable feature that embodies this adaptability is the implementation of “dry-run” semantics, akin to the functionality commonly visible in infrastructure products such as [11], [80], and where API calls can be tested to predict their success without executing any real changes. This concept can be extended beyond mere prediction, serving as a bridge between LLMs' proposed actions and user consent. By repurposing “dry-run” operations, service providers can offer a preview of the uncommitted state resulting from an LLM's actions, allowing users to evaluate and approve these actions before they are finalized. This process adds an essential layer of user oversight, ensuring that actions align with user expectations and intentions.

Chaining-aware. Applications and Services should by-default expect their APIs to be chained with each other when used by agents. To support such a scenario, there needs to be a way to express which APIs can be commutative, associative or distributive with a given set of APIs.

Tracking LLM agents

The introduction of a nonce mechanism (i.e., a session identifier) would enable LLMs to present their identity and facilitate smoother interactions with API providers. This could serve various purposes, such as identifying a session initiated by an LLM or providing a context for transactions. A transaction ID, for example, can enable a system to identify and potentially rollback actions based on this ID. This not only improves the traceability of interactions but also contributes to the overall robustness and reliability of the system by providing an auditable framework for LLM-powered systems.

7.7 Related Work

Isolation. Prior work on enabling automated LLM-powered systems [215] draws on concepts from existing computer systems [36, 151, 212, 95] and emphasizes isolation between LLM-powered applications in order to secure the overall system. Isolation is one facet of safely executing LLM-powered systems, as it alone cannot ensure that the final execution outcome aligns with the user’s intended action.

Trustworthiness in LLMs. There is a rich body of work benchmarking LLMs on their robustness [25]. Recently, trustworthiness [197] has been introduced as a multifaceted benchmark encapsulating robustness, stereotype bias, toxicity, privacy, ethics, and fairness. [197] found that more advanced LLMs (e.g., GPT-4) exhibit higher, albeit still imperfect, trustworthiness.

Attacks on LLMs and defenses. LLMs are also susceptible to prompt hacking attacks, of which include prompt injection [138, 51, 229, 164, 107] and jailbreaking [74, 5]. Such attacks can lead to unpredictable and malicious decisions made by the LLM. There is also an active line of work on defending against such attacks on LLMs [139, 28, 181, 190, 226]. These attacks and defenses will continue to evolve, and consequently the potential of LLMs being susceptible to having their trustworthiness undermined necessitates a runtime that can provide execution of LLM-decided actions while limiting risk.

7.8 Conclusion

The evolution of LLMs from chatbots to deeply embedding them in applications and services for autonomous operation among themselves and other agents presents an exciting future. In this paper, we introduce the concept of “post-facto LLM validation,” as opposed to pre-facto LLM validation, to enable users to verify and roll back the effects caused by executing LLM generated actions (e.g., code, API invocations, and tool use). We propose GoEx, a runtime for LLMs with an intuitive undo and damage confinement abstractions, enabling the safer deployment of LLM agents in practice. We hope our attempt to formalize our

vision and present open research questions towards realizing the goal of autonomous LLM-powered systems in the future, is a step towards a world where LLM-powered systems can independently, with minimal human verification, interact with other tools and services.

Chapter 8

Conclusion

The combined advancements of R2E, LEAP, TEMPERA, Gorilla, RAFT, and GoEx represent significant progress toward practical training and deployment of Large Language Models (LLMs) for interactive and autonomous tasks.

R2E establishes a gym-like environment in which a general-purpose LLM can interact, offering a realistic platform for honing agent capabilities through real-world scenarios. LEAP highlights the value of self-reflection, enabling an LLM agent to learn from its previous actions and improve its future performance. By integrating systematic self-analysis into the training process, LEAP paves the way for more adaptive and resilient agents. TEMPERA introduces a novel reinforcement learning approach that equips LLMs with the ability to dynamically alter their actions at test time. Not only does this facilitate generalization to out-of-domain tasks, but it also allows for more flexible and responsive decision-making. Gorilla addresses a key bottleneck by improving LLM competence in understanding and generating API calls. This advancement extends LLMs beyond passive knowledge repositories, enabling them to perform intricate, tool-assisted tasks autonomously and effectively. RAFT streamlines function calling and environmental comprehension without requiring custom fine-tuning. By granting LLMs seamless access to real-time, contextually relevant information, RAFT ensures outputs are increasingly accurate, grounded, and free from irrelevant distractions. Finally, GoEx introduces a robust runtime system that safeguards the execution of LLM-generated API calls in user-centric applications. Incorporating well-established security principles—such as least-privilege access control, undo operations, and damage confinement—GoEx ensures that LLMs can be deployed safely as decision-making agents in complex, real-world settings.

Together, these innovations underscore a new era in LLM-based agent development, one that emphasizes safety, adaptability, reliability, and practical utility. Looking ahead, training and deploying the AI “Agents” in a complex system environment still has a long way to go. We now try to identify some key challenges here.

Training an LLM that can understand environment dynamics. Training a Large Language Model (LLM) to comprehend the dynamics of a given environment involves navigating multiple layers of complexity. First, the model must be exposed to diverse datasets

that approximate a range of environmental conditions and outcomes. These include not only structured data—such as sensor readings and system logs—but also unstructured data like descriptive text and user feedback. Reinforcement learning techniques can play a vital role: the model can interact with simulated or real environments, receiving feedback and refining its behavior over time. Yet, ensuring robust generalization across varied and unpredictable scenarios remains a central challenge. Incorporating domain-specific knowledge into the LLM may further improve its understanding of complex system dynamics. Achieving this goal likely requires a holistic training approach that blends supervised and unsupervised methods with expert input, enabling the model to adapt more seamlessly to real-world conditions.

Enabling an LLM to perform closed-loop decisions. Realizing effective closed-loop decision-making in an LLM requires continuous monitoring, assessment, and adaptation of its actions in response to real-time feedback. This capability hinges on the model’s ability to process incoming data streams efficiently, quickly identify what information is most salient, and update its decisions accordingly. Achieving such responsiveness may involve advanced reinforcement learning strategies, computational optimizations, and algorithms that prioritize and filter incoming signals. The model must learn not only when to revise its actions, but also how to evaluate the potential consequences of different choices. Additionally, incorporating real-time learning and adaptation mechanisms ensures that the LLM remains effective as conditions evolve. Addressing data noise, anomalies, and system latency is also crucial, as these factors can significantly impact the model’s performance and reliability in closed-loop settings.

Designing systems to guarantee LLM execution safety. Safely deploying LLMs in complex, real-world environments demands careful architectural design, rigorous validation, and ongoing oversight. Robust testing protocols should evaluate how the model behaves under diverse conditions, including rare edge cases and potential failure modes. These evaluations can inform the development of fail-safes and fallback mechanisms that halt or adjust the model’s actions if they deviate from acceptable norms. Continuous monitoring and auditing further enhance reliability, allowing for proactive risk mitigation and timely intervention when issues arise. Establishing clear operational guidelines, ethical standards, and transparent decision-making processes is equally essential. Regular consultations with domain experts and stakeholders help ensure that the LLM’s safety measures remain current and aligned with evolving best practices. Together, these strategies help create secure, reliable systems that maintain the highest standards of performance and integrity.

Bibliography

- [1] Michael Ahn et al. “Do as i can, not as i say: Grounding language in robotic affordances”. In: *arXiv preprint arXiv:2204.01691* (2022).
- [2] Tyler Akidau et al. “The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing”. In: *VLDB* (2015).
- [3] Jean-Baptiste Alayrac et al. “Flamingo: a visual language model for few-shot learning”. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 23716–23736.
- [4] Daniel Andor et al. “Giving BERT a calculator: Finding operations and arguments with reading comprehension”. In: *arXiv preprint arXiv:1909.00109* (2019).
- [5] Cem Anil et al. “Many-shot Jailbreaking”. In: (2024).
- [6] Rohan Anil et al. “Palm 2 technical report”. In: *arXiv preprint arXiv:2305.10403* (2023).
- [7] Anthropic. “Prompt engineering for Claude’s long context window”. In: 2023.
- [8] <https://www.anthropic.com/index/introducing-claude> Anthropic. *Claude*. 2022. URL: <https://www.anthropic.com/index/introducing-claude>.
- [9] Akari Asai et al. “Self-RAG: Learning to Retrieve, Generate, and Critique through Self-Reflection”. In: *arXiv preprint arXiv:2310.11511* (2023).
- [10] Jacob Austin et al. “Program synthesis with large language models”. In: *arXiv preprint arXiv:2108.07732* (2021).
- [11] AWS. *Testing your AWS KMS API calls*. <https://docs.aws.amazon.com/kms/latest/developerguide/programming-dryrun.html>.
- [12] Stephen H. Bach et al. *PromptSource: An Integrated Development Environment and Repository for Natural Language Prompts*. 2022. arXiv: [2202.01279 \[cs.LG\]](https://arxiv.org/abs/2202.01279).
- [13] Ramakrishna Bairi et al. “CodePlan: Repository-level Coding using LLMs and Planning”. In: *Neural Information Processing Systems Workshop on Foundation Models for Decision Making (FMDM-NeurIPS)*. Nov. 2023. URL: <https://www.microsoft.com/en-us/research/publication/codeplan-repository-level-coding-using-llms-and-planning/>.

- [14] Rohan Bavishi et al. “AutoPandas: neural-backed generators for program synthesis”. In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019), pp. 1–27.
- [15] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit.* O’Reilly Media, Inc., 2009.
- [16] Express Computer Blog. “Quantifying GitHub Copilot’s impact on code quality”. In: <https://www.expresscomputer.in/news/quantifying-github-copilots-impact-on-code-quality-ai/104480/> ().
- [17] Sebastian Borgeaud et al. “Improving language models by retrieving from trillions of tokens”. In: *International conference on machine learning*. PMLR. 2022, pp. 2206–2240.
- [18] Tom Brown et al. “Language models are few-shot learners”. In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.
- [19] Sébastien Bubeck et al. “Sparks of artificial general intelligence: Early experiments with gpt-4”. In: *arXiv preprint arXiv:2303.12712* (2023).
- [20] Collin Burns et al. “Discovering latent knowledge in language models without supervision”. In: *arXiv preprint arXiv:2212.03827* (2022).
- [21] Paris Carbone et al. “Apache Flink: Stream and batch processing in a single engine”. In: *The Bulletin of the Technical Committee on Data Engineering* (2015).
- [22] Nicholas Carlini et al. “Extracting training data from large language models”. In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 2633–2650.
- [23] Nicholas Carlini et al. “Quantifying Memorization Across Neural Language Models”. In: *The Eleventh International Conference on Learning Representations*. 2022.
- [24] Nicholas Carlini et al. “The secret sharer: Evaluating and testing unintended memorization in neural networks”. In: *28th USENIX Security Symposium (USENIX Security 19)*. 2019, pp. 267–284.
- [25] Yupeng Chang et al. “A survey on evaluation of large language models”. In: *ACM TIST* (2023).
- [26] Bei Chen et al. “Codet: Code generation with generated tests”. In: *arXiv preprint arXiv:2207.10397* (2022).
- [27] Mark Chen et al. “Evaluating large language models trained on code”. In: *arXiv preprint arXiv:2107.03374* (2021).
- [28] Sizhe Chen et al. “StruQ: Defending Against Prompt Injection with Structured Queries”. In: *arXiv preprint arXiv:2402.06363* (2024).
- [29] Xinyun Chen et al. “Teaching large language models to self-debug”. In: *arXiv preprint arXiv:2304.05128* (2023).
- [30] Yew Ken Chia et al. “Contrastive Chain-of-Thought Prompting”. In: *arXiv preprint arXiv:2311.09277* (2023).

- [31] Dante R Chialvo and Per Bak. “Learning from mistakes”. In: *Neuroscience* 90.4 (1999), pp. 1137–1148.
- [32] Wei-Lin Chiang et al. *Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality*. Mar. 2023. URL: <https://lmsys.org/blog/2023-03-30-vicuna/>.
- [33] Aakanksha Chowdhery et al. “Palm: Scaling language modeling with pathways”. In: *arXiv preprint arXiv:2204.02311* (2022).
- [34] Hyung Won Chung et al. “Scaling instruction-finetuned language models”. In: *arXiv preprint arXiv:2210.11416* (2022).
- [35] Karl Cobbe et al. “Training verifiers to solve math word problems”. In: *arXiv preprint arXiv:2110.14168* (2021).
- [36] Ellis Cohen and David Jefferson. “Protection in the Hydra operating system”. In: *SOSP* (1975).
- [37] Mingkai Deng et al. “RLPrompt: Optimizing Discrete Text Prompts With Reinforcement Learning”. In: (2022). DOI: [10.48550/ARXIV.2205.12548](https://arxiv.org/abs/2205.12548). URL: <https://arxiv.org/abs/2205.12548>.
- [38] Franck Dernoncourt and Ji Young Lee. “Pubmed 200k rct: a dataset for sequential sentence classification in medical abstracts”. In: *arXiv preprint arXiv:1710.06071* (2017).
- [39] Jacob Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: [10.18653/v1/N19-1423](https://aclanthology.org/N19-1423). URL: <https://aclanthology.org/N19-1423>.
- [40] Jacob Devlin et al. “Robustfill: Neural program learning under noisy i/o”. In: *International conference on machine learning*. PMLR. 2017, pp. 990–998.
- [41] Yangruibo Ding et al. “Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion”. In: *arXiv preprint arXiv:2310.11248* (2023).
- [42] Xueying Du et al. *ClassEval: A Manually-Crafted Benchmark for Evaluating LLMs on Class-level Code Generation*. 2023.
- [43] Dheeru Dua et al. “DROP: A Reading Comprehension Benchmark Requiring Discrete Reasoning Over Paragraphs”. In: *Proc. of NAACL*. 2019.
- [44] Dheeru Dua et al. “DROP: A Reading Comprehension Benchmark Requiring Discrete Reasoning Over Paragraphs”. In: *North American Chapter of the Association for Computational Linguistics*. 2019. URL: <https://api.semanticscholar.org/CorpusID:67855846>.

- [45] Amy Edmondson. “Psychological safety and learning behavior in work teams”. In: *Administrative science quarterly* 44.2 (1999), pp. 350–383.
- [46] Amy C Edmondson. “Learning from mistakes is easier said than done: Group and organizational influences on the detection and correction of human error”. In: *The Journal of Applied Behavioral Science* 32.1 (1996), pp. 5–28.
- [47] William Fedus, Barret Zoph, and Noam Shazeer. *Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity*. 2021.
- [48] Vitaly Feldman. “Does learning require memorization? a short tale about a long tail”. In: *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*. 2020, pp. 954–959.
- [49] Tianyu Gao, Adam Fisch, and Danqi Chen. *Making Pre-trained Language Models Better Few-shot Learners*. 2020. DOI: [10.48550/ARXIV.2012.15723](https://doi.org/10.48550/ARXIV.2012.15723), URL: <https://arxiv.org/abs/2012.15723>.
- [50] Tianyu Gao, Adam Fisch, and Danqi Chen. “Making pre-trained language models better few-shot learners”. In: *arXiv preprint arXiv:2012.15723* (2020).
- [51] Kai Greshake et al. “Not what you’ve signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection”. In: *ACM AISEc*. 2023.
- [52] Alex Gu et al. “CRUXEval: A Benchmark for Code Reasoning, Understanding and Execution”. In: *arXiv preprint arXiv:2401.03065* (2024).
- [53] Qingyan Guo et al. “Connecting large language models with evolutionary algorithms yields powerful prompt optimizers”. In: *arXiv preprint arXiv:2309.08532* (2023).
- [54] Kelvin Guu et al. “Retrieval augmented language model pre-training”. In: *International conference on machine learning*. PMLR. 2020, pp. 3929–3938.
- [55] Theo Haerder and Andreas Reuter. “Principles of transaction-oriented database recovery”. In: *ACM CSUR* (1983).
- [56] Dan Hendrycks et al. “Measuring coding challenge competence with apps”. In: *arXiv preprint arXiv:2105.09938* (2021).
- [57] Neil Houlsby et al. “Parameter-efficient transfer learning for NLP”. In: *International Conference on Machine Learning*. PMLR. 2019, pp. 2790–2799.
- [58] Edward J Hu et al. “Lora: Low-rank adaptation of large language models”. In: *arXiv preprint arXiv:2106.09685* (2021).
- [59] Jiaxin Huang et al. “Large Language Models Can Self-Improve”. In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Ed. by Houda Bouamor, Juan Pino, and Kalika Bali. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 1051–1068. DOI: [10.18653/v1/2023.emnlp-main.67](https://doi.org/10.18653/v1/2023.emnlp-main.67), URL: <https://aclanthology.org/2023.emnlp-main.67>.

- [60] Jie Huang et al. “Large language models cannot self-correct reasoning yet”. In: *arXiv preprint arXiv:2310.01798* (2023).
- [61] Marko Ivanković et al. “Code coverage at Google”. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019, pp. 955–963.
- [62] Srinivasan Iyer et al. “OPT-IML: Scaling Language Model Instruction Meta Learning through the Lens of Generalization”. In: *arXiv preprint arXiv:2212.12017* (2022).
- [63] Gautier Izacard et al. “Atlas: Few-shot Learning with Retrieval Augmented Language Models”. In: *Journal of Machine Learning Research* 24.251 (2023), pp. 1–43. URL: <http://jmlr.org/papers/v24/23-0037.html>.
- [64] Naman Jain et al. “Jigsaw: Large language models meet program synthesis”. In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, pp. 1219–1231.
- [65] Naman Jain et al. “LLM-Assisted Code Cleaning For Training Accurate Code Generators”. In: *arXiv preprint arXiv:2311.14904* (2023).
- [66] Charlie Cheng-Jie Ji et al. “Gorilla OpenFunctions v2”. In: 2024.
- [67] Yuezihan Jiang et al. “Instance-wise Prompt Tuning for Pretrained Language Models”. In: *arXiv preprint arXiv:2206.01958* (2022).
- [68] Zhengbao Jiang et al. “How Can We Know What Language Models Know?” In: *Transactions of the Association for Computational Linguistics* 8 (2020), pp. 423–438. DOI: [10.1162/tac1_a_00324](https://doi.org/10.1162/tac1_a_00324). URL: <https://aclanthology.org/2020.tac1-1.28>.
- [69] Carlos E Jimenez et al. “SWE-bench: Can Language Models Resolve Real-World GitHub Issues?” In: *arXiv preprint arXiv:2310.06770* (2023).
- [70] Qiao Jin et al. “Pubmedqa: A dataset for biomedical research question answering”. In: *arXiv preprint arXiv:1909.06146* (2019).
- [71] Mandar Joshi et al. “Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension”. In: *arXiv preprint arXiv:1705.03551* (2017).
- [72] Jean Kaddour et al. “Challenges and applications of large language models”. In: *arXiv preprint arXiv:2307.10169* (2023).
- [73] Nikhil Kandpal, Eric Wallace, and Colin Raffel. “Deduplicating training data mitigates privacy risks in language models”. In: *International Conference on Machine Learning*. PMLR. 2022, pp. 10697–10707.
- [74] Daniel Kang et al. “Exploiting programmatic behavior of LLMs: Dual-use through standard security attacks”. In: *arXiv preprint arXiv:2302.05733* (2023).
- [75] Darren Key, Wen-Ding Li, and Kevin Ellis. “I speak, you verify: Toward trustworthy neural program synthesis”. In: *arXiv preprint arXiv:2210.00848* (2022).

- [76] Urvashi Khandelwal et al. “Generalization through memorization: Nearest neighbor language models”. In: *arXiv preprint arXiv:1911.00172* (2019).
- [77] Geunwoo Kim, Pierre Baldi, and Stephen McAleer. “Language models can solve computer tasks”. In: *arXiv preprint arXiv:2303.17491* (2023).
- [78] Takeshi Kojima et al. “Large language models are zero-shot reasoners”. In: *arXiv preprint arXiv:2205.11916* (2022).
- [79] Mojtaba Komeili, Kurt Shuster, and Jason Weston. “Internet-augmented dialogue generation”. In: *arXiv preprint arXiv:2107.07566* (2021).
- [80] Kubernetes. *kubectl Usage Conventions*. <https://kubernetes.io/docs/reference/kubectl/conventions/>.
- [81] Tom Kwiatkowski et al. “Natural questions: a benchmark for question answering research”. In: *Transactions of the Association for Computational Linguistics* 7 (2019), pp. 453–466.
- [82] Marie-Anne Lachaux et al. “Unsupervised translation of programming languages”. In: *arXiv preprint arXiv:2006.03511* (2020).
- [83] Yuhang Lai et al. “DS-1000: A natural and reliable benchmark for data science code generation”. In: *International Conference on Machine Learning*. PMLR, 2023, pp. 18319–18345.
- [84] Angeliki Lazaridou et al. “Internet-augmented language models through few-shot prompting for open-domain question answering”. In: *arXiv preprint arXiv:2203.05115* (2022).
- [85] Hung Le et al. “Coderl: Mastering code generation through pretrained models and deep reinforcement learning”. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 21314–21328.
- [86] Brian Lester, Rami Al-Rfou, and Noah Constant. “The Power of Scale for Parameter-Efficient Prompt Tuning”. In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 3045–3059. DOI: [10.18653/v1/2021.emnlp-main.243](https://doi.org/10.18653/v1/2021.emnlp-main.243). URL: <https://aclanthology.org/2021.emnlp-main.243>.
- [87] Brian Lester, Rami Al-Rfou, and Noah Constant. “The power of scale for parameter-efficient prompt tuning”. In: *arXiv preprint arXiv:2104.08691* (2021).
- [88] Patrick Lewis et al. “Retrieval-augmented generation for knowledge-intensive nlp tasks”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 9459–9474.
- [89] Raymond Li et al. “StarCoder: may the source be with you!” In: *arXiv preprint arXiv:2305.06161* (2023).

- [90] Xiang Lisa Li and Percy Liang. “Prefix-tuning: Optimizing continuous prompts for generation”. In: *arXiv preprint arXiv:2101.00190* (2021).
- [91] Yujia Li et al. “Competition-level code generation with alphacode”. In: *Science* 378.6624 (2022), pp. 1092–1097.
- [92] Yaobo Liang et al. “Taskmatrix. ai: Completing tasks by connecting foundation models with millions of apis”. In: *arXiv preprint arXiv:2303.16434* (2023).
- [93] Xi Victoria Lin et al. “RA-DIT: Retrieval-Augmented Dual Instruction Tuning”. In: *arXiv preprint arXiv:2310.01352* (2023).
- [94] Xi Victoria Lin et al. “Ra-dit: Retrieval-augmented dual instruction tuning”. In: *arXiv preprint arXiv:2310.01352* (2023).
- [95] Theodore A Linden. “Operating system structures to support security and reliable software”. In: *CSUR* (1976).
- [96] Chenxiao Liu et al. “Code Execution with Pre-trained Language Models”. In: *arXiv preprint arXiv:2305.05383* (2023).
- [97] Hao Liu, Carmelo Sferrazza, and Pieter Abbeel. “Chain of hindsight aligns language models with feedback”. In: *arXiv preprint arXiv:2302.02676* 3 (2023).
- [98] Jiachang Liu et al. “What Makes Good In-Context Examples for GPT-3?” In: *Workshop on Knowledge Extraction and Integration for Deep Learning Architectures; Deep Learning Inside Out*. 2021. URL: <https://api.semanticscholar.org/CorpusID:231632658>.
- [99] Jiachang Liu et al. “What Makes Good In-Context Examples for GPT-3?” In: *Proceedings of Deep Learning Inside Out (DeeLIO 2022): The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures*. Dublin, Ireland and Online: Association for Computational Linguistics, May 2022, pp. 100–114. DOI: [10.18653/v1/2022.deelio-1.10](https://doi.org/10.18653/v1/2022.deelio-1.10). URL: <https://aclanthology.org/2022.deelio-1.10>.
- [100] Jiawei Liu et al. “Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation”. In: *arXiv preprint arXiv:2305.01210* (2023).
- [101] Nelson F Liu et al. “Lost in the middle: How language models use long contexts”. In: *arXiv preprint arXiv:2307.03172* (2023).
- [102] Pengfei Liu et al. “Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing”. In: *ACM Computing Surveys* 55.9 (2023), pp. 1–35.
- [103] Tianyang Liu, Canwen Xu, and Julian McAuley. “RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems”. In: *arXiv preprint arXiv:2306.03091* (2023).
- [104] Xiao Liu et al. “GPT understands, too”. In: *arXiv preprint arXiv:2103.10385* (2021).

- [105] Xiao Liu et al. “P-tuning v2: Prompt tuning can be comparable to fine-tuning universally across scales and tasks”. In: *arXiv preprint arXiv:2110.07602* (2021).
- [106] Xiao Liu et al. “P-tuning: Prompt tuning can be comparable to fine-tuning across scales and tasks”. In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. 2022, pp. 61–68.
- [107] Yi Liu et al. “Prompt Injection attack against LLM-integrated Applications”. In: *arXiv preprint arXiv:2306.05499* (2023).
- [108] Yinhan Liu et al. “Roberta: A robustly optimized bert pretraining approach”. In: *arXiv preprint arXiv:1907.11692* (2019).
- [109] Zihan Liu et al. “ChatQA: Building GPT-4 Level Conversational QA Models”. In: *arXiv preprint arXiv:2401.10225* (2024).
- [110] Ziming Liu et al. “Towards understanding grokking: An effective theory of representation learning”. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 34651–34663.
- [111] Sheng Lu et al. “Are Emergent Abilities in Large Language Models just In-Context Learning?” In: *arXiv preprint arXiv:2309.01809* (2023).
- [112] Yao Lu et al. “Fantastically Ordered Prompts and Where to Find Them: Overcoming Few-Shot Prompt Order Sensitivity”. In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 8086–8098. DOI: [10.18653/v1/2022.acl-long.556](https://doi.org/10.18653/v1/2022.acl-long.556), URL: <https://aclanthology.org/2022.acl-long.556>.
- [113] Yao Lu et al. “Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity”. In: *arXiv preprint arXiv:2104.08786* (2021).
- [114] Aman Madaan et al. “Learning performance-improving code edits”. In: *arXiv preprint arXiv:2302.07867* (2023).
- [115] Aman Madaan et al. “Memory-assisted prompt editing to improve GPT-3 after deployment”. In: *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. 2022, pp. 2833–2861.
- [116] Aman Madaan et al. “Self-refine: Iterative refinement with self-feedback”. In: *arXiv preprint arXiv:2303.17651* (2023).
- [117] Aditya Menon et al. “A machine learning framework for programming by example”. In: *International Conference on Machine Learning*. PMLR. 2013, pp. 187–195.
- [118] Microsoft. *Presidio - Data Protection and De-identification SDK*. <https://github.com/microsoft/presidio>.
- [119] Sewon Min et al. “MetaICL: Learning to Learn In Context”. In: *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2022, pp. 2791–2809.

- [120] Sewon Min et al. “Rethinking the Role of Demonstrations: What Makes In-Context Learning Work?” In: *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. 2022, pp. 11048–11064.
- [121] Marvin Minsky and Seymour Papert. “An introduction to computational geometry”. In: *Cambridge tiass., HIT* 479.480 (1969), p. 104.
- [122] Swaroop Mishra et al. “Cross-task generalization via natural language crowdsourcing instructions”. In: *arXiv preprint arXiv:2104.08773* (2021).
- [123] Niklas Muennighoff et al. “Crosslingual Generalization through Multitask Finetuning”. In: *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki. Toronto, Canada: Association for Computational Linguistics, July 2023, pp. 15991–16111. DOI: [10.18653/v1/2023.acl-long.891](https://doi.org/10.18653/v1/2023.acl-long.891). URL: <https://aclanthology.org/2023.acl-long.891>.
- [124] Reiichiro Nakano et al. “Webgpt: Browser-assisted question-answering with human feedback”. In: *arXiv preprint arXiv:2112.09332* (2021).
- [125] Ansong Ni et al. “Lever: Learning to verify language-to-code generation with execution”. In: *International Conference on Machine Learning*. PMLR. 2023, pp. 26106–26128.
- [126] Erik Nijkamp et al. “CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis”. In: *The Eleventh International Conference on Learning Representations*. 2022.
- [127] Erik Nijkamp et al. “CodeGen2: Lessons for Training LLMs on Programming and Natural Languages”. In: *arXiv preprint arXiv:2305.02309* (2023).
- [128] Theo X Olausson et al. “Demystifying GPT Self-Repair for Code Generation”. In: *arXiv preprint arXiv:2306.09896* (2023).
- [129] OpenAI. *ChatGPT plugins*. <https://openai.com/blog/chatgpt-plugins>. 2023.
- [130] OpenAI. *GPT-4 Technical Report*. 2023. arXiv: [2303.08774 \[cs.CL\]](https://arxiv.org/abs/2303.08774).
- [131] OpenAI and <https://openai.com/blog/chatgpt>. *ChatGPT*. 2022. URL: <https://openai.com/blog/chatgpt>.
- [132] Long Ouyang et al. “Training language models to follow instructions with human feedback”. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 27730–27744.
- [133] Xudong Pan et al. “Privacy risks of general-purpose language models”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 1314–1331.
- [134] Aaron Parisi, Yao Zhao, and Noah Fiedel. *TALM: Tool Augmented Language Models*. 2022.

- [135] Joon Sung Park et al. “Generative agents: Interactive simulacra of human behavior”. In: *arXiv preprint arXiv:2304.03442* (2023).
- [136] Shishir G Patil et al. “Gorilla: Large language model connected with massive apis”. In: *arXiv preprint arXiv:2305.15334* (2023).
- [137] Baolin Peng et al. “Check your facts and try again: Improving large language models with external knowledge and automated feedback”. In: *arXiv preprint arXiv:2302.12813* (2023).
- [138] Fábio Perez and Ian Ribeiro. “Ignore previous prompt: Attack techniques for language models”. In: *arXiv preprint arXiv:2211.09527* (2022).
- [139] Julien Piet et al. “Jatmo: Prompt injection defense by task-specific finetuning”. In: *arXiv preprint arXiv:2312.17673* (2023).
- [140] Alethea Power et al. “Grokking: Generalization beyond overfitting on small algorithmic datasets”. In: *arXiv preprint arXiv:2201.02177* (2022).
- [141] Archiki Prasad et al. “Grips: Gradient-free, edit-based instruction search for prompting large language models”. In: *arXiv preprint arXiv:2203.07281* (2022).
- [142] Shuofei Qiao et al. “Reasoning with Language Model Prompting: A Survey”. In: *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki. Toronto, Canada: Association for Computational Linguistics, July 2023, pp. 5368–5393. DOI: [10.18653/v1/2023.acl-long.294](https://doi.org/10.18653/v1/2023.acl-long.294). URL: <https://aclanthology.org/2023.acl-long.294>
- [143] Yujia Qin et al. *ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs*. 2023.
- [144] Morgan Quigley et al. “ROS: an open-source Robot Operating System”. In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5.
- [145] Alec Radford et al. “Language models are unsupervised multitask learners”. In: *OpenAI blog* 1.8 (2019), p. 9.
- [146] Rafael Rafailov et al. “Direct preference optimization: Your language model is secretly a reward model”. In: *arXiv preprint arXiv:2305.18290* (2023).
- [147] Rafael Rafailov et al. “Direct preference optimization: Your language model is secretly a reward model”. In: *NeurIPS* (2024).
- [148] Colin Raffel et al. “Exploring the limits of transfer learning with a unified text-to-text transformer.” In: *J. Mach. Learn. Res.* 21.140 (2020), pp. 1–67.
- [149] Ori Ram et al. “In-context retrieval-augmented language models”. In: *arXiv preprint arXiv:2302.00083* (2023).
- [150] Vipula Rawte, Amit Sheth, and Amitava Das. “A survey of hallucination in large foundation models”. In: *arXiv preprint arXiv:2309.05922* (2023).

- [151] Charles Reis, Alexander Moshchuk, and Nasko Oskov. “Site isolation: Process separation for web sites within the browser”. In: *USENIX Security*. 2019.
- [152] Tal Ridnik, Dedy Kredo, and Itamar Friedman. “Code Generation with AlphaCodium: From Prompt Engineering to Flow Engineering”. In: *arXiv preprint arXiv:2401.08500* (2024).
- [153] Bernardino Romera-Paredes et al. “Mathematical discoveries from program search with large language models”. In: *Nature* (2023), pp. 1–3.
- [154] Frank Rosenblatt. *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory, 1957.
- [155] Baptiste Roziere et al. “Code llama: Open foundation models for code”. In: *arXiv preprint arXiv:2308.12950* (2023).
- [156] Ohad Rubin, Jonathan Herzig, and Jonathan Berant. “Learning To Retrieve Prompts for In-Context Learning”. In: *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Seattle, United States: Association for Computational Linguistics, July 2022, pp. 2655–2671. DOI: [10.18653/v1/2022.naacl-main.191](https://doi.org/10.18653/v1/2022.naacl-main.191). URL: <https://aclanthology.org/2022.naacl-main.191>.
- [157] Barbara G Ryder. “Constructing the call graph of a program”. In: *IEEE Transactions on Software Engineering* 3 (1979), pp. 216–226.
- [158] Victor Sanh et al. “Multitask prompted training enables zero-shot task generalization”. In: *arXiv preprint arXiv:2110.08207* (2021).
- [159] Teven Le Scao et al. “Bloom: A 176b-parameter open-access multilingual language model”. In: *arXiv preprint arXiv:2211.05100* (2022).
- [160] Timo Schick, Helmut Schmid, and Hinrich Schütze. “Automatically Identifying Words That Can Serve as Labels for Few-Shot Text Classification”. In: *Proceedings of the 28th International Conference on Computational Linguistics*. Barcelona, Spain (Online): International Committee on Computational Linguistics, Dec. 2020, pp. 5569–5578. DOI: [10.18653/v1/2020.coling-main.488](https://doi.org/10.18653/v1/2020.coling-main.488). URL: <https://aclanthology.org/2020.coling-main.488>.
- [161] Timo Schick and Hinrich Schütze. “Exploiting cloze questions for few shot text classification and natural language inference”. In: *arXiv preprint arXiv:2001.07676* (2020).
- [162] Timo Schick and Hinrich Schütze. “Exploiting Cloze-Questions for Few-Shot Text Classification and Natural Language Inference”. In: *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*. Online: Association for Computational Linguistics, Apr. 2021, pp. 255–269. DOI: [10.18653/v1/2021.eacl-main.20](https://doi.org/10.18653/v1/2021.eacl-main.20). URL: <https://aclanthology.org/2021.eacl-main.20>.

- [163] Timo Schick et al. “Toolformer: Language models can teach themselves to use tools”. In: *arXiv preprint arXiv:2302.04761* (2023).
- [164] Sander Schulhoff et al. “Ignore This Title and HackAPrompt: Exposing Systemic Vulnerabilities of LLMs through a Global Scale Prompt Hacking Competition”. In: *arXiv preprint arXiv:2311.16119* (2023).
- [165] John Schulman et al. “Proximal policy optimization algorithms”. In: *arXiv preprint arXiv:1707.06347* (2017).
- [166] Yongliang Shen et al. “Hugginggpt: Solving ai tasks with chatgpt and its friends in huggingface”. In: *arXiv preprint arXiv:2303.17580* (2023).
- [167] Freda Shi et al. “Large language models can be easily distracted by irrelevant context”. In: *International Conference on Machine Learning*. PMLR. 2023, pp. 31210–31227.
- [168] Weijia Shi et al. “Detecting Pretraining Data from Large Language Models”. In: *arXiv preprint arXiv:2310.16789* (2023).
- [169] Weijia Shi et al. “In-Context Pretraining: Language Modeling Beyond Document Boundaries”. In: *arXiv preprint arXiv:2310.10638* (2023).
- [170] Weijia Shi et al. “Replug: Retrieval-augmented black-box language models”. In: *arXiv preprint arXiv:2301.12652* (2023).
- [171] Taylor Shin et al. “AutoPrompt: Eliciting Knowledge from Language Models with Automatically Generated Prompts”. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2020, pp. 4222–4235.
- [172] Taylor Shin et al. “AutoPrompt: Eliciting Knowledge from Language Models with Automatically Generated Prompts”. In: *Empirical Methods in Natural Language Processing (EMNLP)*. 2020.
- [173] Noah Shinn, Beck Labash, and Ashwin Gopinath. “Reflexion: an autonomous agent with dynamic memory and self-reflection”. In: *arXiv preprint arXiv:2303.11366* (2023).
- [174] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. “Repository-level prompt generation for large language models of code”. In: *International Conference on Machine Learning*. PMLR. 2023, pp. 31693–31715.
- [175] Disha Shrivastava et al. “RepoFusion: Training Code Models to Understand Your Repository”. In: *arXiv preprint arXiv:2306.10998* (2023).
- [176] Kurt Shuster et al. “Blenderbot 3: a deployed conversational agent that continually learns to responsibly engage”. In: *arXiv preprint arXiv:2208.03188* (2022).
- [177] Avi Singh et al. “Beyond human data: Scaling self-training for problem-solving with language models”. In: *arXiv preprint arXiv:2312.06585* (2023).
- [178] Sanjay Subramanian et al. “Modular Visual Question Answering via Code Generation”. In: *arXiv preprint arXiv:2306.05392* (2023).

- [179] Tianxiang Sun et al. “Black-box tuning for language-model-as-a-service”. In: *arXiv preprint arXiv:2201.03514* (2022).
- [180] Zhiqing Sun et al. “Principle-driven self-alignment of language models from scratch with minimal human supervision”. In: *arXiv preprint arXiv:2305.03047* (2023).
- [181] Xuchen Suo. “Signed-Prompt: A New Approach to Prevent Prompt Injection Attacks Against LLM-Integrated Applications”. In: *arXiv preprint arXiv:2401.07612* (2024).
- [182] Dídac Surís, Sachit Menon, and Carl Vondrick. “Vipergpt: Visual inference via python execution for reasoning”. In: *arXiv preprint arXiv:2303.08128* (2023).
- [183] Mirac Suzgun et al. “Challenging big-bench tasks and whether chain-of-thought can solve them”. In: *arXiv preprint arXiv:2210.09261* (2022).
- [184] Michael Tánzer, Sebastian Ruder, and Marek Rei. “Memorisation versus Generalisation in Pre-trained Language Models”. In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2022, pp. 7564–7578.
- [185] Rohan Taori et al. *Stanford Alpaca: An Instruction-following LLaMA model*. https://github.com/tatsu-lab/stanford_alpaca. 2023.
- [186] Gemini Team et al. “Gemini: a family of highly capable multimodal models”. In: *arXiv preprint arXiv:2312.11805* (2023).
- [187] Romal Thoppilan et al. “Lamda: Language models for dialog applications”. In: *arXiv preprint arXiv:2201.08239* (2022).
- [188] Hugo Touvron et al. “Llama 2: Open foundation and fine-tuned chat models”. In: *arXiv preprint arXiv:2307.09288* (2023).
- [189] Hugo Touvron et al. “Llama: Open and efficient foundation language models”. In: *arXiv preprint arXiv:2302.13971* (2023).
- [190] Sam Toyer et al. “Tensor trust: Interpretable prompt injection attacks from an online game”. In: *arXiv preprint arXiv:2311.01011* (2023).
- [191] Michele Tufano et al. “Methods2Test: A dataset of focal methods mapped to test cases”. In: *Proceedings of the 19th International Conference on Mining Software Repositories*. 2022, pp. 299–303.
- [192] Karthik Valmeekam, Matthew Marquez, and Subbarao Kambhampati. “Investigating the Effectiveness of Self-critiquing in LLMs solving Planning Tasks”. In: *NeurIPS 2023 Foundation Models for Decision Making Workshop*. 2023. URL: <https://openreview.net/forum?id=gGQfkyb0KL>.
- [193] Sai Vemprala et al. “Chatgpt for robotics: Design principles and model abilities”. In: *2023* (2023).
- [194] Tu Vu et al. “Freshllms: Refreshing large language models with search engine augmentation”. In: *arXiv preprint arXiv:2310.03214* (2023).

- [195] Alex Wang et al. “GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding”. In: *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*. Brussels, Belgium: Association for Computational Linguistics, Nov. 2018, pp. 353–355. DOI: [10.18653/v1/W18-5446](https://doi.org/10.18653/v1/W18-5446). URL: <https://aclanthology.org/W18-5446>.
- [196] Alex Wang et al. “SuperGLUE: A Stickier Benchmark for General-Purpose Language Understanding Systems”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019. URL: <https://proceedings.neurips.cc/paper/2019/file/4496bf24afe7fab6f046bf4923da8de6-Paper.pdf>.
- [197] Boxin Wang et al. “DecodingTrust: A comprehensive assessment of trustworthiness in GPT models”. In: *arXiv preprint arXiv:2306.11698* (2023).
- [198] Boxin Wang et al. “Instructretro: Instruction tuning post retrieval-augmented pre-training”. In: *arXiv preprint arXiv:2310.07713* (2023).
- [199] Guanzhi Wang et al. “Voyager: An open-ended embodied agent with large language models”. In: *arXiv preprint arXiv:2305.16291* (2023).
- [200] Xuezhi Wang et al. “Rationale-Augmented Ensembles in Language Models”. In: *arXiv preprints arXiv:2207.00747* (2022).
- [201] Yizhong Wang et al. “Benchmarking Generalization via In-Context Instructions on 1,600+ Language Tasks”. In: *arXiv* (2022).
- [202] Yizhong Wang et al. “Self-instruct: Aligning language models with self-generated instructions”. In: *arXiv preprint arXiv:2212.10560* (2022).
- [203] Yizhong Wang et al. “Super-naturalinstructions: Generalization via declarative instructions on 1600+ nlp tasks”. In: *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. 2022, pp. 5085–5109.
- [204] Zhiruo Wang et al. “Execution-based evaluation for open-domain code generation”. In: *arXiv preprint arXiv:2212.10481* (2022).
- [205] Cody Watson et al. “On learning meaningful assert statements for unit test cases”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 2020, pp. 1398–1409.
- [206] Jason Wei et al. “Chain-of-thought prompting elicits reasoning in large language models”. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 24824–24837.
- [207] Jason Wei et al. “Emergent Abilities of Large Language Models”. In: *Transactions on Machine Learning Research* (2022). Survey Certification. ISSN: 2835-8856. URL: <https://openreview.net/forum?id=yzkSU5zdwd>.
- [208] Jason Wei et al. “Finetuned language models are zero-shot learners”. In: *ICLR*. 2022.

- [209] Jerry Wei et al. “Larger language models do in-context learning differently”. In: *arXiv preprint arXiv:2303.03846* (2023).
- [210] Jason Weston and Sainbayar Sukhbaatar. “System 2 Attention (is something you might need too)”. In: *arXiv preprint arXiv:2311.11829* (2023).
- [211] Norbert Wiener. “Cybernetics; or control and communication in the animal and the machine.” In: (1948).
- [212] Maurice Vincent Wilkes and Roger Michael Needham. “The Cambridge CAP computer and its operating system”. In: (1979).
- [213] BigScience Workshop et al. “Bloom: A 176b-parameter open-access multilingual language model”. In: *arXiv preprint arXiv:2211.05100* (2022).
- [214] Qingyun Wu et al. “Autogen: Enabling next-gen LLM applications via multi-agent conversation framework”. In: *arXiv preprint arXiv:2308.08155* (2023).
- [215] Yuhao Wu et al. “SecGPT: An Execution Isolation Architecture for LLM-Based Systems”. In: *arXiv preprint arXiv:2403.04960* (2024).
- [216] Zhuofeng Wu et al. “IDPG: An Instance-Dependent Prompt Generation Method”. In: *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Seattle, United States: Association for Computational Linguistics, July 2022, pp. 5507–5521. DOI: [10.18653/v1/2022.naacl-main.403](https://doi.org/10.18653/v1/2022.naacl-main.403). URL: <https://aclanthology.org/2022.naacl-main.403>.
- [217] Zhiheng Xi et al. “The rise and potential of large language model based agents: A survey”. In: *arXiv preprint arXiv:2309.07864* (2023).
- [218] Wenhan Xiong et al. “Effective long-context scaling of foundation models”. In: *arXiv preprint arXiv:2309.16039* (2023).
- [219] Frank F Xu et al. “A systematic evaluation of large language models of code”. In: *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 2022, pp. 1–10.
- [220] Peng Xu et al. “Retrieval meets Long Context Large Language Models”. In: *arXiv preprint arXiv:2310.03025* (2023).
- [221] Chengrun Yang et al. “Large language models as optimizers”. In: *arXiv preprint arXiv:2309.03409* (2023).
- [222] Zhilin Yang et al. “HotpotQA: A Dataset for Diverse, Explainable Multi-hop Question Answering”. In: *Conference on Empirical Methods in Natural Language Processing*. 2018. URL: <https://api.semanticscholar.org/CorpusID:52822214>.
- [223] Zhilin Yang et al. “HotpotQA: A dataset for diverse, explainable multi-hop question answering”. In: *arXiv preprint arXiv:1809.09600* (2018).

- [224] Shunyu Yao et al. “ReAct: Synergizing Reasoning and Acting in Language Models”. In: *arXiv preprint arXiv:2210.03629* (2022).
- [225] Shunyu Yao et al. “Tree of thoughts: Deliberate problem solving with large language models”. In: *arXiv preprint arXiv:2305.10601* (2023).
- [226] Jingwei Yi et al. “Benchmarking and defending against indirect prompt injection attacks on large language models”. In: *arXiv preprint arXiv:2312.14197* (2023).
- [227] Pengcheng Yin et al. “Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow”. In: *International Conference on Mining Software Repositories*. MSR. ACM, 2018, pp. 476–486. DOI: <https://doi.org/10.1145/3196398.3196408>.
- [228] Pengcheng Yin et al. “Natural language to code generation in interactive data science notebooks”. In: *arXiv preprint arXiv:2212.09248* (2022).
- [229] Jiahao Yu et al. “Assessing prompt injection risks in 200+ custom gpts”. In: *arXiv preprint arXiv:2311.11538* (2023).
- [230] Eric Zelikman et al. “Parsel: A (de-) compositional framework for algorithmic reasoning with language models”. In: *arXiv preprint arXiv:2212.10561* (2023).
- [231] Eric Zelikman et al. “STaR: Bootstrapping Reasoning With Reasoning”. In: *Advances in Neural Information Processing Systems*. Ed. by Alice H. Oh et al. 2022. URL: https://openreview.net/forum?id=_3ELRdg2sgI.
- [232] Aohan Zeng et al. “Glm-130b: An open bilingual pre-trained model”. In: *arXiv preprint arXiv:2210.02414* (2022).
- [233] Fengji Zhang et al. “Repocoder: Repository-level code completion through iterative retrieval and generation”. In: *arXiv preprint arXiv:2303.12570* (2023).
- [234] Kechi Zhang et al. “Self-Edit: Fault-Aware Code Editor for Code Generation”. In: *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Toronto, Canada: Association for Computational Linguistics, July 2023, pp. 769–787.
- [235] Kechi Zhang et al. “ToolCoder: Teach Code Generation Models to use APIs with search tools”. In: *arXiv preprint arXiv:2305.04032* (2023).
- [236] Susan Zhang et al. “Opt: Open pre-trained transformer language models”. In: *arXiv preprint arXiv:2205.01068* (2022).
- [237] Tianjun Zhang et al. “The Wisdom of Hindsight Makes Language Models Better Instruction Followers”. In: *arXiv preprint arXiv:2302.05206* (2023).
- [238] Tianyi Zhang et al. “Coder reviewer reranking for code generation”. In: *International Conference on Machine Learning*. PMLR. 2023, pp. 41832–41846.
- [239] Yue Zhang et al. “Siren’s song in the AI ocean: a survey on hallucination in large language models”. In: *arXiv preprint arXiv:2309.01219* (2023).

- [240] Zhuosheng Zhang et al. “Igniting Language Intelligence: The Hitchhiker’s Guide From Chain-of-Thought Reasoning to Language Agents”. In: *arXiv preprint arXiv:2311.11797* (2023).
- [241] Tony Z. Zhao et al. *Calibrate Before Use: Improving Few-Shot Performance of Language Models*. 2021. DOI: [10.48550/ARXIV.2102.09690](https://doi.org/10.48550/ARXIV.2102.09690) URL: <https://arxiv.org/abs/2102.09690>.
- [242] Wayne Xin Zhao et al. “A survey of large language models”. In: *arXiv preprint arXiv:2303.18223* (2023).
- [243] Zexuan Zhong, Dan Friedman, and Danqi Chen. “Factual Probing Is [MASK]: Learning vs. Learning to Recall”. In: *North American Association for Computational Linguistics (NAACL)*. 2021.
- [244] Denny Zhou et al. “Least-to-Most Prompting Enables Complex Reasoning in Large Language Models”. In: *arXiv preprint arXiv:2205.10625* (2022).
- [245] Shuyan Zhou et al. “Docprompting: Generating code by retrieving the docs”. In: *The Eleventh International Conference on Learning Representations*. 2022.
- [246] Daniel M Ziegler et al. “Fine-tuning language models from human preferences”. In: *arXiv preprint arXiv:1909.08593* (2019).

Appendix A

R2E

Callgraphs

A callgraph [157] is a directed graph where nodes represent subroutines (functions, methods, constructors, etc.), and edges denote the *calling* relationships between them. It is a directed graph $G = (V, E)$ where V is a set of subroutines (functions, methods, constructors, etc.) and E is a set of edges representing the *calling* relationship between these subroutines. If a subroutine u invokes (i.e., calls) a subroutine v , then there is a directed edge from u to v in the callgraph. For instance, in Chapter [2.2], the function `indexer` calls the function `_parse_sigmf_capture`. This repository abstraction enables analyzing several properties of repository code.

For instance, we use it to extract the dependencies that a function relies on for its execution—a valuable property for test generation. In this work, we use the `pycg` tool to generate callgraphs for python repositories.

Dependency Slicing

While callgraphs abstract direct interactions between functions, a python function can interact with parts of the repository through global variables, too—in the same file and imported from other files. We can summarize these interactions in a **dependency slice** D_f for a function f , as the set of all functions F' that f calls, and all global variables G' that f accesses, both directly and indirectly.

For a function f , we define a mapping called `depends` which identifies all functions F' that f calls, and all global variables G' that f accesses.

Then a **dependency slice** D_f is the transitive closure of all functions and global variables that f depends on, directly or indirectly.

$$D_f = \bigcup_{(F', G') \in \text{depends}^*(f)} (F' \cup G') \quad (\text{A.1})$$

Computing this slice is generally an undecidable problem, but we make a few simplifying assumptions to make it tractable. We begin by utilizing the callgraph to identify the functions that f calls. We then use bytecode analysis to identify the set of global variables that f accesses. We add these functions and global variables to the slice and recursively repeat the process for each function in the slice. We utilize this context for test generation in Chapter [2.3](#).

Below, we list the prompt used for test harness generation.

You are a python programming expert who was hired to write tests for Python functions.

You will be given a python function in a python file and you will write a complete test that covers the function and all the different corner cases.

You can assume a compiled reference implementation of the function is available, and hence do not need to predict the expected output of the function.

That is, the test you write will use the reference implementation to generate the expected output.

Also, assume the function provided is correct and hence the test should focus on the behavior that is defined by the function ONLY.

Ensure that the tests align with the function's expected input types, avoiding scenarios that the function is not designed to handle.

Completely avoid testing with invalid input types or values, testing for error handling, and checking 'assertRaises'.

Set a fixed random seed in tests involving randomness to ensure consistent and reproducible results when necessary.

Avoid mocking calls to APIs or functions (e.g., builtins.open) when actual implementations are simple, accessible, and their use does not compromise the test's isolation or determinism.

Particularly, avoid mocking calls to any file I/O APIs, and instead try to create temporary files and directories for testing purposes.

You will return the test for that function and NOT return anything except for the test.

Put your fixed test program within code delimiters, for example:

```
"""python
# YOUR CODE HERE
"""
```

Write a test using the 'unittest' library for the function 'function_name'. Assume the reference implementation is 'reference_function_name'. Both the function and the reference are in the module 'fut_module'. Only return the test code and do NOT return anything else. Enclose your code within code delimiters, for example:

```
"""python
# YOUR CODE HERE
"""
```

Analysis of Equivalence Test Generation

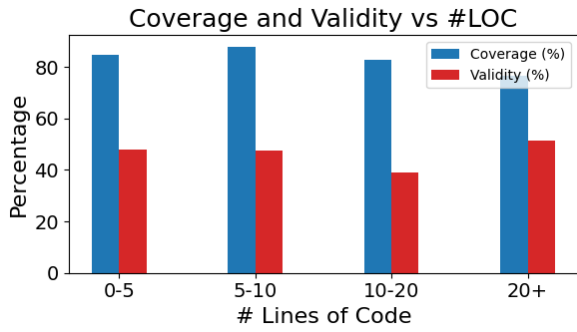


Figure A.1: Varying number of lines

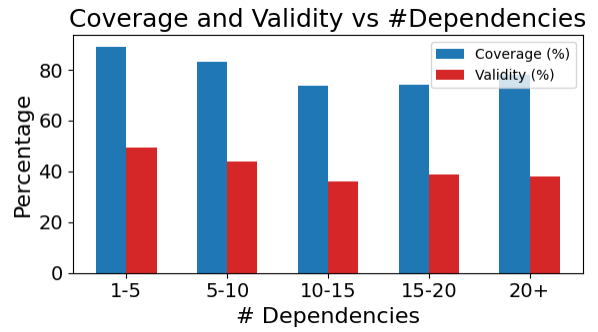


Figure A.2: Varying number of dependencies

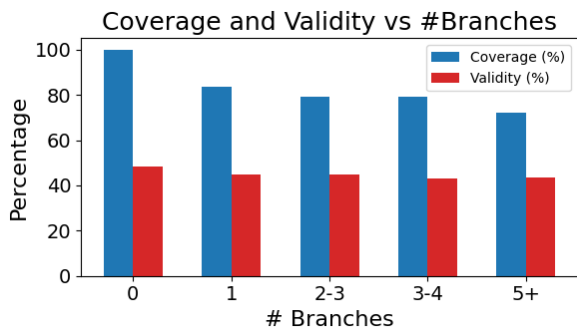


Figure A.3: Varying number of branches

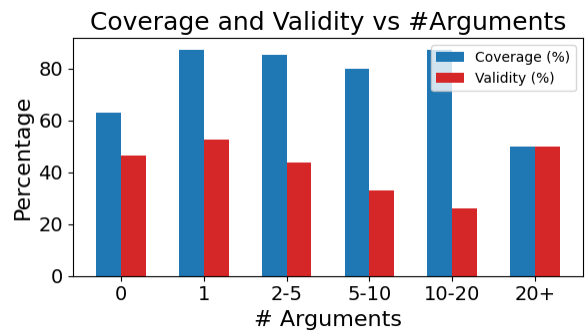


Figure A.4: Varying number of arguments

You are a python programming expert who is refining docstrings in existing programs. You will be given a python function in a python file with an existing (possibly under-specified) docstring with corresponding unit tests for the function and optionally some input-output examples extracted from the unittest in a serialized format. Your goal is to refine the associated docstring by making it more informative, precise and complete without adding verbosity or detailed programming logic to the docstring. The docstring should particularly describe the format and types of the expected inputs and output as well as the behavior of the function. You will return the function definition, docstring enclosed in markdown code delimiters. The docstrings must be formatted in the google docstring format and examples should be added if they clarify the function and look helpful without being very long. Do not guess outputs for functions but only copy the expected outputs as provided. Finally, do not throw away existing details from the docstrings and only insert content you are sure about. Do NOT have repeated content in the docstring and ONLY describe the high-level function behavior without going into implementation details

```

### Code Snippet:
{original_code_snippet}

### Unit Tests:
"""python
{test_code}
"""

### Argument Types: {argument_types}
### Output Types: {output_type}
### Examples: {examples_substring}

```

Refine the docstring for the function `function_name`. Return only the updated function with docstring enclosed in markdown and ignore the remaining code. Remember to make the docstring precise and informative regarding global function behavior (input-output properties) without being too verbose. Do not specify detailed function logic or very domain-specific content in the docstring (unless already described in the docstring).

Spec-Refinement User Study

We study the quality of the refined docstrings, specifically on their effectiveness as problem descriptors for code completion. We analyze two high-level aspects of docstrings:

- **Interface Understandability:** Does the refined docstring clarify the function's input-

output interface? Here, we look at how the parameter descriptions and the I/O examples aid in understanding the interface.

- **Problem Intent:** Does the refined docstring articulate the problem the function is intended to solve? Here, we look at how the docstring text and the I/O examples illustrate how the function solves the problem and if it can lead to alternate solutions that do not satisfy the original implementation.

Each criterion was evaluated using a 3-point scale, ranging from ‘Poor’ (1) to ‘Excellent’ (3). The lead authors applied this rubric to study a sample of 62 problems. Results in the following tables.

Param Desc	I/O Examples	Overall
2.9	2.8	2.9

Table A.1: Interface Understandability Scores

Text Desc	I/O Examples	Overall
2.5	2.1	2.5

Table A.2: Problem Intent Scores

The list of unique input and output data types is provided below. This highlights that problems in our benchmark are interesting.

We list down the list of models considered for code generation experiments here.

Model ID	Link
codellama/CodeLlama-34b-Python-hf	CodeLlama-34b-Python-hf
codellama/CodeLlama-13b-Python-hf	CodeLlama-13b-Python-hf
codellama/CodeLlama-7b-Python-hf	CodeLlama-7b-Python-hf
gpt-3.5-turbo-1106-16k	OpenAI
gpt-4-1106	OpenAI

Table A.3: List of models

Code Generation

To compute Pass1, we generate 5 completions for each problem instance using each model. We use nucleus sampling with $p = 0.95$ and $T = 0.2$. Below we list the prompts used (inspired from [128])

You are a Python programming expert who is going to generate a Python function in a file using the function docstring. You will use the existing context of relevant files provided for implementation and ONLY return the completed function. Enclose the completed function in markdown code delimiters and do NOT return anything else.

```
### Code Snippet  
{code_snippet}
```

Complete the function {function_name}. Only return the completed function enclosed in markdown code delimiters

Self Repair

We use GPT-4 and GPT-4-Turbo models for the self-repair task. We find problems from R2E where models fail to any generate correct completion and we can extract the failing scenario (since some of the tests are dynamic, it is not always possible to extract the failing scenario). Additionally, since our GPT-4 has a context length of 8k, we additionally filter very long problems from the repair dataset We then use the failing scenario as the prompt for the self-repair task.

You are a Python programming expert who is going to generate a Python function in a file using the function docstring. You will use the existing context of relevant files provided for implementation and ONLY return the completed function. Enclose the completed function in markdown code delimiters and do NOT return anything else.

Code Snippet:

{code_snippet}

Inputs:

{captured_inputs}

Expected Output:

{captured_output}

Error Trace:

{output}

Instruction

You will first reason using a concise (at most 2-3 sentences) textual explanation of what is wrong with the function. After you have pointed out what is wrong with the code, you will then generate a fixed version of the program. You will ONLY return the completed function. Follow the following format which presents the reason for the failure followed by the repaired program enclosed in backticks.

Reasoning

{function_name} is failing because of ...

Repaired Function

"""python

def function_name(...): ...

"""

```
# nobuco/locate/link.py
import inspect

def get_link(file=None, line=None):
    """ Print a link in PyCharm to a line in file.
        Defaults to line where this function was called. """
    if file is None:
        file = inspect.stack()[1].filename
    if line is None:
        line = inspect.stack()[1].lineno
    string = f'File "{file}", line {max(line, 1)}'.replace("\\", "/")
    return string

def get_link_to_obj(obj):    ## CodeLLaMa-34B-Python completion

    obj_type = type(obj)
    if obj_type is property:
        obj = obj.fget
    if not hasattr(obj, '__module__') or not hasattr(obj, '__qualname__'):
        return None
    module = inspect.getmodule(obj)
    if module is None:
        return None
    if not hasattr(module, '__file__'):
        return None
    if obj_type is type:
        obj_source = obj.__qualname__
    elif obj_type is types.ModuleType:
        obj_source = obj.__name__
    else:
        obj_source = f'{obj.__module__}.{obj.__qualname__}'
    return f'File "{module.__file__}", line {inspect.getsourcelines(obj)
[1]}'.replace('\\', '/')



---


def standardize_and_get_inchi(mol: Mol, options: str='', log_level=None,
warnings_as_errors=False) -> str:
    """Return InChI after standardising molecule and inferring stereo from
coordinates."""
    mol = deepcopy(mol)
    mol = assert_sanity(mol)
    mol = remove_isotopic_info(mol)
    has_pose = mol.GetNumConformers() > 0
    if has_pose:
        RemoveStereochemistry(mol)
    mol = RemoveHs(mol)
    try:
        mol = neutralize_atoms(mol)
```

```

    except AtomValenceException:
        logger.warning('Failed to neutralize molecule. Using uncharger.
InChI check might fail.')
        mol = Uncharger().uncharge(mol)
    mol = add_stereo_hydrogens(mol)
    if has_pose:
        AssignStereochemistryFrom3D(mol, replaceExistingTags=True)
    with CaptureLogger():
        inchi = MolToInchi(mol, options=options, logLevel=log_level,
treatWarningAsError=warnings_as_errors)
    return inchi

...

def check_identity(mol_pred: Mol, mol_true: Mol, inchi_options: str='') ->
dict[str, Any]:
    """Check if two molecules are identical in terms of their InChI
representation.

    This function compares the InChI strings of two molecules after
standardizing them and
optionally applying InChI generation options. ...
    Args:
        ...
    Returns:
        ...
    Example:
        >>> ..."""
...

## CodeLLaMa-13B-Python completion

mol_pred = deepcopy(mol_pred)
mol_true = deepcopy(mol_true)
mol_pred = assert_sanity(mol_pred)
mol_true = assert_sanity(mol_true)
mol_pred = remove_isotopic_info(mol_pred)
mol_true = remove_isotopic_info(mol_true)
has_pose_pred = mol_pred.GetNumConformers() > 0
has_pose_true = mol_true.GetNumConformers() > 0
if has_pose_pred:
    RemoveStereochemistry(mol_pred)
if has_pose_true:
    RemoveStereochemistry(mol_true)
mol_pred = RemoveHs(mol_pred)
mol_true = RemoveHs(mol_true)
try:
    mol_pred = neutralize_atoms(mol_pred)
except AtomValenceException:
    logger.warning('Failed to neutralize molecule. Using uncharger.
InChI check might fail.')

```

```

        mol_pred = Uncharger().uncharge(mol_pred)
    try:
        mol_true = neutralize_atoms(mol_true)
    except AtomValenceException:
        logger.warning('Failed to neutralize molecule. Using uncharger.
InChI check might fail.')
        mol_true = Uncharger().uncharge(mol_true)
    mol_pred = add_stereo_hydrogens(mol_pred)
    mol_true = add_stereo_hydrogens(mol_true)
    if has_pose_pred:
        AssignStereochemistryFrom3D(mol_pred, replaceExistingTags=True)
    if has_pose_true:
        AssignStereochemistryFrom3D(mol_true, replaceExistingTags=True)
    inchi_pred = standardize_and_get_inchi(mol_pred, options=inchi_options
)
    inchi_true = standardize_and_get_inchi(mol_true, options=inchi_options
)
    results = _compare_inchis(inchi_true, inchi_pred)
    results['inchi_crystal_valid'] = _is_valid_inchi(inchi_true)
    results['inchi_docked_valid'] = _is_valid_inchi(inchi_pred)
    results['inchi_crystal'] = inchi_true
    results['inchi_docked'] = inchi_pred
    return {'results': results}

```

```
import torch
```

```

def get_2d_sincos_pos_embed_from_grid(embed_dim: int, grid: torch.Tensor)
-> torch.Tensor:
    """Get 2D sine-cosine positional embedding from grid.
    Args:
        embed_dim: embedding dimension.
        grid: positions
    Returns:
        (torch.Tensor): [grid_size*grid_size, embed_dim] or [1+grid_size*
grid_size, embed_dim]"""
    assert embed_dim % 2 == 0
    emb_h = get_1d_sincos_pos_embed_from_grid(embed_dim // 2, grid[0])
    emb_w = get_1d_sincos_pos_embed_from_grid(embed_dim // 2, grid[1])
    emb = torch.cat([emb_h, emb_w], dim=1)
    return emb

def get_1d_sincos_pos_embed_from_grid(embed_dim: int, pos: torch.Tensor)
-> torch.Tensor:
    """Get 1D sine-cosine positional embedding.
    Args:
        embed_dim: output dimension for each position
        pos: a list of positions to be encoded: size (M,)
    Returns:
        (torch.Tensor): tensor of shape (M, D)"""
    assert embed_dim % 2 == 0

```

```

    omega = torch.arange(embed_dim // 2, dtype=torch.float)
    omega /= embed_dim / 2.0
    omega = 1.0 / 10000 ** omega
    pos = pos.reshape(-1)
    out = torch.einsum('m,d->md', pos, omega)
    emb_sin = torch.sin(out)
    emb_cos = torch.cos(out)
    emb = torch.cat([emb_sin, emb_cos], dim=1)
    return emb

def get_2d_sincos_pos_embed(embed_dim: int, grid_size: int, cls_token:
bool=False) -> torch.Tensor:
    """Generates a 2D sine-cosine positional embedding tensor.

    This function creates a positional embedding for a 2D grid using sine
    and cosine functions.
    The embedding can optionally include a leading zero vector to
    represent a classification (CLS) token.

    Args:
        embed_dim (int): The dimensionality of the embedding for each
        position.
        grid_size (int): The height and width of the square grid for which
        embeddings are generated.
        cls_token (bool): If True, the output tensor will include an
        additional first row with zeros
        to represent a CLS token. Defaults to False.

    Returns:
        torch.Tensor: A tensor of shape (grid_size * grid_size, embed_dim)
        without a CLS token, or
        (1 + grid_size * grid_size, embed_dim) with a CLS
        token. The tensor contains
        the positional embeddings for the grid and is of
        type `torch.float32`. """
    grid = torch.stack(torch.meshgrid(torch.arange(grid_size), torch.
    arange(grid_size)), dim=-1)
    grid = grid.reshape(-1, 2).float()
    emb = get_2d_sincos_pos_embed_from_grid(embed_dim, grid)
    if cls_token:
        cls_emb = torch.zeros(1, embed_dim)
        emb = torch.cat([cls_emb, emb], dim=0)
    return emb

```

```

from typing import Tuple
import torch

def get_3d_sincos_pos_embed(embed_dim: int, tube_shape: Tuple[int, int,
int], stride, offset, kernel_size, cls_token: bool=False) -> torch.
Tensor:
    """Get 3D sine-cosine positional embedding.

```

```

    Args:
        tube_shape: (t_size,
        .... """
    ...
    pos_embed_spatial = get_2d_sincos_pos_embed_from_grid(
embed_dim_spatial, grid)
    ...

def get_2d_sincos_pos_embed(embed_dim: int, grid_size: int, cls_token:
bool=False) -> torch.Tensor:
    """Get 2D sine-cosine positional embedding.
    Args:
        grid_size: int of the grid height and width
        cls_token: bool, whether to contain CLS token
    Returns:
        (torch.Tensor): [grid_size*grid_size, embed_dim] or [1+grid_size*
grid_size, embed_dim]"""
    ...
    pos_embed = ...

def get_1d_sincos_pos_embed_from_grid(embed_dim: int, pos: torch.Tensor)
-> torch.Tensor:
    """Get 1D sine-cosine positional embedding.
    Args:
        embed_dim: output dimension for each position
        pos: a list of positions to be encoded: size (M,)
    Returns:
        (torch.Tensor): tensor of shape (M, D)
    """
    assert embed_dim % 2 == 0
    omega = torch.arange(embed_dim // 2, dtype=torch.float)
    ...
    return emb

def get_2d_sincos_pos_embed(embed_dim: int, grid_size: int, cls_token:
bool=False) -> torch.Tensor:
    """Generates a 2D sine-cosine positional embedding tensor.

    This function creates a positional embedding for a 2D grid using sine
and cosine functions.
    The embedding can optionally include a leading zero vector to
represent a classification (CLS) token.

    Args:
        embed_dim (int): The dimensionality of the embedding for each
position.
        grid_size (int): The height and width of the square grid for which
embeddings are generated.
        cls_token (bool): If True, the output tensor will include an
additional first row with zeros

```

to represent a CLS token. Defaults to False.

Returns:

torch.Tensor: A tensor of shape (grid_size * grid_size, embed_dim) without a CLS token, or (1 + grid_size * grid_size, embed_dim) with a CLS token. The tensor contains the positional embeddings for the grid and is of type `torch.float32`.

```
"""
assert embed_dim % 2 == 0
grid = torch.arange(grid_size, dtype=torch.float)
grid = torch.meshgrid(grid, grid, indexing='ij')
grid = torch.stack(grid, dim=0)
pos_embed = get_2d_sincos_pos_embed_from_grid(embed_dim, grid)
return pos_embed
```

```
class Image(BaseImage):
```

```
    def qvec2rotmat(self):
        return qvec2rotmat(self.qvec)
```

```
def read_next_bytes(fid, num_bytes, format_char_sequence, endian_character
    = '<'):
    data = fid.read(num_bytes)
    return struct.unpack(endian_character + format_char_sequence, data)
```

```
def read_images_binary(path_to_model_file):
    """Read camera images and their properties from a binary file. This
    function parses a binary file containing the properties of images used
    in 3D reconstruction
    and returns a dictionary of Image namedtuples, each containing the
    properties of an image.
```

Args:

path_to_model_file (str): The path to the binary model file to be read.

Returns:

dict: A dictionary where keys are image IDs (int) and values are Image namedtuples.

Each Image namedtuple has the following fields:

- id (int): The unique identifier of the image.
- qvec (numpy.ndarray): A 4-element array representing the quaternion vector.
- tvec (numpy.ndarray): A 3-element array representing the translation vector.
- camera_id (int): The identifier of the camera used to capture the image.
- name (str): The name of the image file.


```

        - xys (numpy.ndarray): An Nx2 array of [x, y] coordinates of
        2D points in the image.
        - point3D_ids (numpy.ndarray): An N-element array of
        identifiers of corresponding 3D points"""

```

```
## GPT-4 Completion
```

```

with open(path_to_model_file, 'rb') as fid:
    num_reg_images = read_next_bytes(fid, 4, 'I')[0]
    images = {}
    for _ in range(num_reg_images):
        image_id = read_next_bytes(fid, 4, 'I')[0]
        qvec = np.array(read_next_bytes(fid, 4 * 4, 'dddd'))
        tvec = np.array(read_next_bytes(fid, 3 * 4, 'fff'))
        camera_id = read_next_bytes(fid, 4, 'I')[0]
        image_name = read_next_bytes(fid, 1, 'c')[0]
        num_points2D = read_next_bytes(fid, 4, 'I')[0]
        xys = np.array(read_next_bytes(fid, num_points2D * 2 * 4, 'ff'
* num_points2D))
        point3D_ids = np.array(read_next_bytes(fid, num_points2D * 4,
'I' * num_points2D))
        images[image_id] = Image(id=image_id, qvec=qvec, tvec=tvec,
camera_id=camera_id, name=image_name, xys=xys, point3D_ids=point3D_ids)
    return images

```

```
# xrloc/map/read_write_model.py
```

```
def read_next_bytes(fid, num_bytes, format_char_sequence, endian_character
='<'):

```

```
    ...
```

```
def write_next_bytes(fid, data, format_char_sequence, endian_character
='<'):

```

```
    ...
```

```
def write_images_binary(images, path_to_model_file):

```

```
    """

```

```
    see: src/base/map.cc

```

```
        void Reconstruction::ReadImagesBinary(const std::string& path)

```

```
        void Reconstruction::WriteImagesBinary(const std::string& path)

```

```
    """

```

```
with open(path_to_model_file, 'wb') as fid:

```

```
    write_next_bytes(fid, len(images), 'Q')

```

```
    for _, img in images.items():

```

```
        write_next_bytes(fid, img.id, 'i')

```

```
        write_next_bytes(fid, img.qvec.tolist(), 'dddd')

```

```
        write_next_bytes(fid, img.tvec.tolist(), 'ddd')

```

```
        write_next_bytes(fid, img.camera_id, 'i')

```

```
        for char in img.name:

```

```
            write_next_bytes(fid, char.encode('utf-8'), 'c')

```

```
        write_next_bytes(fid, b'\\x00', 'c')
```

```

        write_next_bytes(fid, len(img.point3D_ids), 'Q')
    for xy, p3d_id in zip(img.xys, img.point3D_ids):
        write_next_bytes(fid, [*xy, p3d_id], 'ddq')

def read_points3d_binary(path_to_model_file):
    ...

def write_points3d_binary(points3D, path_to_model_file):
    """
    see: src/base/map.cc
    void Reconstruction::ReadPoints3DBinary(const std::string& path)
    void Reconstruction::WritePoints3DBinary(const std::string& path)
    """
    with open(path_to_model_file, 'wb') as fid:
        write_next_bytes(fid, len(points3D), 'Q')
        for _, pt in points3D.items():
            write_next_bytes(fid, pt.id, 'Q')
            write_next_bytes(fid, pt.xyz.tolist(), 'ddd')
            write_next_bytes(fid, pt.rgb.tolist(), 'BBB')
            write_next_bytes(fid, pt.error, 'd')
            track_length = pt.image_ids.shape[0]
            write_next_bytes(fid, track_length, 'Q')
            for image_id, point2D_id in zip(pt.image_ids, pt.point2D_idxs)
:
                write_next_bytes(fid, [image_id, point2D_id], 'ii')

def read_point3d_feature_binary(path_to_feature_file):
    """Read 3D point features.

    Args:
        path_to_feature_file (str): Path to point feature file

    Returns:
        dict{int:np.array(dim)}: 3D point feature dict
    """
    point3d_features = {}
    with open(path_to_feature_file, 'rb') as file:
        num_points3d = struct.unpack('<Q', file.read(8))[0]
        dim_feature = struct.unpack('<Q', file.read(8))[0]
        for i in range(num_points3d):
            point3d_id = struct.unpack('<Q', file.read(8))[0]
            feature_num = struct.unpack('<I', file.read(4))[0]
            fmat = '<{0}f'.format(dim_feature)
            descriptor = np.array(struct.unpack(fmat, file.read(4 *
dim_feature)))
            point3d_features[point3d_id] = descriptor
    return point3d_features

def read_images_binary(path_to_model_file):

```

```

"""
Read camera images and their properties from a binary file.

This function parses a binary file containing the properties of images
used in 3D reconstruction
and returns a dictionary of Image namedtuples, each containing the
properties of an image.

Args:
    path_to_model_file (str): The path to the binary model file to be
read.

Returns:
    dict: A dictionary where keys are image IDs (int) and values are
Image namedtuples.
        Each Image namedtuple has the following fields:
        - id (int): The unique identifier of the image.
        - qvec (numpy.ndarray): A 4-element array representing the
quaternion vector.
        - tvec (numpy.ndarray): A 3-element array representing the
translation vector.
        - camera_id (int): The identifier of the camera used to
capture the image.
        - name (str): The name of the image file.
        - xys (numpy.ndarray): An Nx2 array of [x, y] coordinates of
2D points in the image.
        - point3D_ids (numpy.ndarray): An N-element array of
identifiers of 3D points corresponding to the 2D points.
"""

## GPT-4 Completion

images = {}
with open(path_to_model_file, 'rb') as fid:
    num_reg_images = read_next_bytes(fid, 8, 'Q')[0]
    for _ in range(num_reg_images):
        reg_image_id = read_next_bytes(fid, 4, 'i')[0]
        qvec = np.array(read_next_bytes(fid, 8 * 4, 'dddd'))
        tvec = np.array(read_next_bytes(fid, 8 * 3, 'ddd'))
        camera_id = read_next_bytes(fid, 4, 'i')[0]
        image_name = ''
        while True:
            char = read_next_bytes(fid, 1, 'c')[0]
            if char == b'\x00':
                break
            image_name += char.decode('utf-8')
        num_points2D = read_next_bytes(fid, 8, 'Q')[0]
        xys = np.zeros((num_points2D, 2), dtype=np.float32)
        point3D_ids = np.zeros(num_points2D, dtype=np.int64)
        for i in range(num_points2D):

```

```

        xys[i] = read_next_bytes(fid, 8 * 2, 'dd')
        point3D_ids[i] = read_next_bytes(fid, 8, 'Q')[0]
        images[reg_image_id] = Image(id=reg_image_id, qvec=qvec, tvec=
tvec, camera_id=camera_id, name=image_name, xys=xys, point3D_ids=
point3D_ids)
    return images

```

```
import torch
```

```

def _product_attr(vision, text, alter):
    """
    Args:
        vision: N x D
        text: M x D
        alter: N x M, to replace results in some cases, see details in
Returns
Returns: N x M.
    For (n, m) element, set J_m = {j : text[m, j] == 1}.
    - if |J_m| > 0, it equals to (prod_{j in J_m} vision[n, j])** (1/|
J_m|)
    - if |J_m| == 0, it equals to alter[n, m]
    """
    vision = vision.unsqueeze(1)
    text = text.unsqueeze(0)
    num_attr = text.sum(-1)
    queried_attr = vision * text
    queried_attr = queried_attr.masked_fill(text == 0, 1)
    queried_attr = torch.float_power(queried_attr.prod(dim=2), 1 / torch.
max(num_attr, torch.ones_like(num_attr))).float()
    no_attr_queries = num_attr.squeeze(0) == 0
    queried_attr[:, no_attr_queries] = alter[:, no_attr_queries]
    return queried_attr

def obj_with_attributes(input_embeddings, query_embeddings, n_obj, n_part,
n_attr):
    """Compute the similarity between object embeddings and query
embeddings based on attributes.

    This function calculates the similarity score between each pair of
object and query embeddings.
    The score is computed as the square root of the product of the object
score and the geometric
    mean of the queried attributes, if any attributes are queried. If no
attributes are queried,
    the object score is returned as is.
    ...
    """
    vision = input_embeddings[:, :n_obj]
    text = query_embeddings[:, n_obj:n_obj + n_attr]
    alter = input_embeddings[:, n_obj + n_attr:]

```

```

    queried_attr = _product_attr(vision, text, alter)
    obj_score = (input_embeddings[:, :n_obj] * query_embeddings[:, :n_obj]
    ).sum(dim=1, keepdim=True)
    scores = torch.sqrt(obj_score * queried_attr)
    return scores

```

Error

Traceback (most recent call last):

File "<string>", line 17, in test_obj_with_attributes

File "/capture_args.py", line 107, in wrapper

```

    output = func(*args, **kwargs)
    ~~~~~

```

File "/tmp/tmptgi66m5s/paco_query_utils.py", line 62, in

obj_with_attributes

```

    queried_attr = _product_attr(vision, text, alter)
    ~~~~~

```

File "/tmp/tmptgi66m5s/paco_query_utils.py", line 22, in _product_attr

```

    queried_attr = vision * text
    ~~~~~

```

RuntimeError: The size of tensor a (5) must match the size of tensor b (2) at non-singleton dimension 2

Listing A.1: GPT-4 failing to understand the `_product_attr` helper function used in its completion of `obj_with_attributes`.

Dependency vs Usage Tradeoff

The first example demonstrates dependency context vs full context for `get_2d_sincos_pos_embed` function. In the dependency context, only two functions which the oracle ground truth functions calls are provided and the task is to stitch them together. This becomes simpler in the full context case where how to call those functions is also provided (via other functions).

Length of retrieval We compare how the performance of the models is impacted by the length (# tokens) of the retrieval context. Since we perform dependency-only-context retrieval, we only have the context required to understand the necessary functions for solving the problem instance. We find that the performance is not strongly correlated with the length of the retrieval context (Figure [A.6](#)). This suggests that the choice of the retrieved context is a bigger factor than the length.

CoT on R2E We use 0-shot and 2-shot CoT to evaluate more enhanced code generation approaches. The following table describes performance.

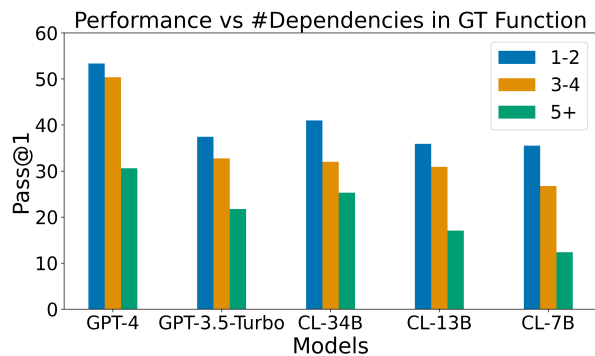


Figure A.5: Varying number of dependencies

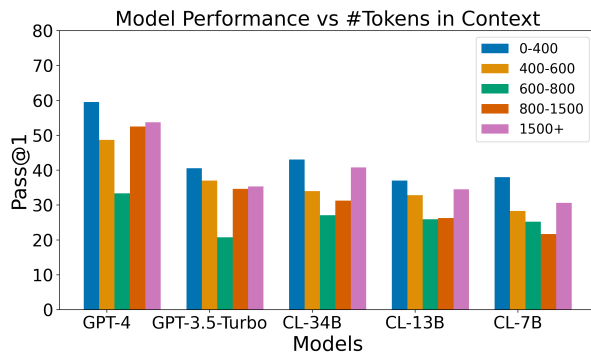


Figure A.6: Varying number of context tokens

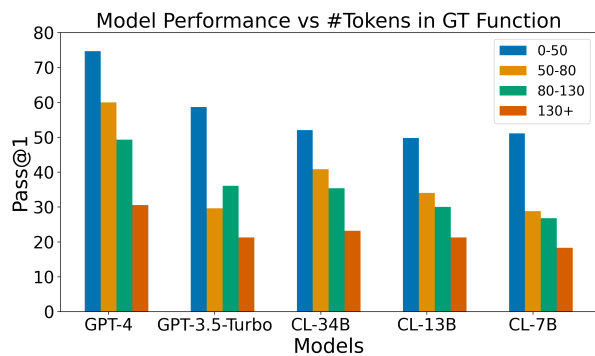


Figure A.7: Varying number of ground truth tokens

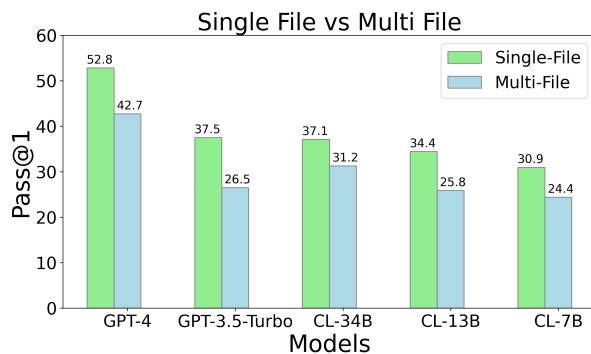


Figure A.8: Varying File usage

	Base	CoT-0-shot	CoT-2-shot
GPT-4-Turbo	48.9	45.8	-
GPT-4	33.2	33	28.8

Table A.4: Effect of CoT on code generation on a subset of our R2E benchmark

Appendix B

In Context Principle Learning

B.1 Additional Results on Text Reasoning

Chapter [B.3](#) shows additional Text Reasoning results. The upper part of Chapter [B.3](#) is identical to Chapter [3.1](#); the lower part of Chapter [B.3](#) contains zero-shot results, with principles that were learned from the given few-shot examples; this setup is similar to related work such as [221](#).

Table B.1: **Text Reasoning Results:** Accuracy in textual reasoning datasets. The best approach for each base LLM in each dataset is in **bold**; the second-best approach is underlined.

		GPT-3.5-turbo	GPT-4	Gemini Pro
Few-shot results, identical to Chapter 3.2:				
HotpotQA	Few-shot CoT	29.10	36.35	28.25
	LEAP _{LOW-LEVEL}	32.60	<u>39.30</u>	23.70
	LEAP _{HIGH-LEVEL}	<u>30.35</u>	39.65	<u>25.50</u>
DROP	Few-shot CoT	<u>63.20</u>	72.05	64.60
	LEAP _{LOW-LEVEL}	63.35	79.55	<u>67.15</u>
	LEAP _{HIGH-LEVEL}	63.00	<u>78.60</u>	67.60
		GPT-3.5-turbo	GPT-4	Gemini Pro
Zero-shot results, using principles learned from few-shot examples:				
HotpotQA	Zero-shot CoT	<u>12.55</u>	27.80	19.75
	LEAP _{LOW-LEVEL}	6.65	31.20	<u>3.30</u>
	LEAP _{HIGH-LEVEL}	13.10	<u>30.90</u>	2.45
DROP	Zero-shot CoT	59.25	<u>82.20</u>	62.75
	LEAP _{LOW-LEVEL}	55.20	84.10	65.30
	LEAP _{HIGH-LEVEL}	<u>55.25</u>	81.80	<u>62.80</u>

B.2 Additional Results on Mathematical Reasoning

Table B.2: **Math Reasoning Results**: Accuracy in MATH and GSM8K. The best approach for each LLM and base task is in **bold**; the second-best approach is underlined.

		GPT-3.5-turbo	GPT-4	Claude-2	Gemini Pro
Few-shot results, identical to Chapter 3.2:					
GSM8K	Few-shot CoT	76.4	93.6	84.3	
	LEAP _{LOW-LEVEL}	77.4	94.1	82.7	
	LEAP _{HIGH-LEVEL}	<u>76.6</u>	<u>93.8</u>	<u>83.8</u>	
MATH	Few-shot CoT	55.6	63.5	<u>43.2</u>	
	LEAP _{LOW-LEVEL}	<u>56.1</u>	64.5	42.7	
	LEAP _{HIGH-LEVEL}	56.5	<u>64.0</u>	43.4	
		GPT-3.5-turbo	GPT-4	Claude-2	Gemini Pro
Zero-shot results, using principles learned from few-shot examples:					
GSM8K	Zero-shot CoT	76.9	93.2	75.4	
	LEAP _{LOW-LEVEL}	<u>74.4</u>	94.2	<u>76.7</u>	
	LEAP _{HIGH-LEVEL}	73.8	<u>94.1</u>	76.9	
MATH	Zero-shot CoT	54.2	63.5	40.2	
	LEAP _{LOW-LEVEL}	<u>52.0</u>	<u>63.2</u>	<u>40.5</u>	
	LEAP _{HIGH-LEVEL}	50.0	61.5	41.8	

Chapter B.4 shows additional Mathematical Reasoning results. The upper part of Chapter B.4 is identical to Chapter 3.2; the lower part of Chapter B.4 contains zero-shot results, with principles that were learned from the given few-shot examples; this setup is similar to related work such as 221.

B.3 Additional Results on Text Reasoning

Chapter B.3 shows additional Text Reasoning results. The upper part of Chapter B.3 is identical to Chapter 3.1; the lower part of Chapter B.3 contains zero-shot results, with principles that were learned from the given few-shot examples; this setup is similar to related work such as 221.

Table B.3: **Text Reasoning Results:** Accuracy in textual reasoning datasets. The best approach for each base LLM in each dataset is in **bold**; the second-best approach is underlined.

		GPT-3.5-turbo	GPT-4	Gemini Pro
Few-shot results, identical to Chapter 3.2:				
HotpotQA	Few-shot CoT	29.10	36.35	28.25
	LEAP _{LOW-LEVEL}	32.60	<u>39.30</u>	23.70
	LEAP _{HIGH-LEVEL}	<u>30.35</u>	39.65	<u>25.50</u>
DROP	Few-shot CoT	<u>63.20</u>	72.05	64.60
	LEAP _{LOW-LEVEL}	63.35	79.55	<u>67.15</u>
	LEAP _{HIGH-LEVEL}	63.00	<u>78.60</u>	67.60
		GPT-3.5-turbo	GPT-4	Gemini Pro
Zero-shot results, using principles learned from few-shot examples:				
HotpotQA	Zero-shot CoT	<u>12.55</u>	27.80	19.75
	LEAP _{LOW-LEVEL}	6.65	31.20	<u>3.30</u>
	LEAP _{HIGH-LEVEL}	13.10	<u>30.90</u>	2.45
DROP	Zero-shot CoT	59.25	<u>82.20</u>	62.75
	LEAP _{LOW-LEVEL}	55.20	84.10	65.30
	LEAP _{HIGH-LEVEL}	<u>55.25</u>	81.80	<u>62.80</u>

B.4 Additional Results on Mathematical Reasoning

Table B.4: **Math Reasoning Results**: Accuracy in MATH and GSM8K. The best approach for each LLM and base task is in **bold**; the second-best approach is underlined.

		GPT-3.5-turbo	GPT-4	Claude-2	Gemini Pro
Few-shot results, identical to Chapter 3.2:					
GSM8K	Few-shot CoT	76.4	93.6	84.3	
	LEAP _{LOW-LEVEL}	77.4	94.1	82.7	
	LEAP _{HIGH-LEVEL}	<u>76.6</u>	<u>93.8</u>	<u>83.8</u>	
MATH	Few-shot CoT	55.6	63.5	<u>43.2</u>	
	LEAP _{LOW-LEVEL}	<u>56.1</u>	64.5	42.7	
	LEAP _{HIGH-LEVEL}	56.5	<u>64.0</u>	43.4	
		GPT-3.5-turbo	GPT-4	Claude-2	Gemini Pro
Zero-shot results, using principles learned from few-shot examples:					
GSM8K	Zero-shot CoT	76.9	93.2	75.4	
	LEAP _{LOW-LEVEL}	<u>74.4</u>	94.2	<u>76.7</u>	
	LEAP _{HIGH-LEVEL}	73.8	<u>94.1</u>	76.9	
MATH	Zero-shot CoT	54.2	63.5	40.2	
	LEAP _{LOW-LEVEL}	<u>52.0</u>	<u>63.2</u>	<u>40.5</u>	
	LEAP _{HIGH-LEVEL}	50.0	61.5	41.8	

Chapter B.4 shows additional Mathematical Reasoning results. The upper part of Chapter B.4 is identical to Chapter 3.2; the lower part of Chapter B.4 contains zero-shot results, with principles that were learned from the given few-shot examples; this setup is similar to related work such as 221.

B.5 Additional Results on BBH

Chapter B.6 shows results on additional BBH tasks that we could not fit into Chapter B.5. We selected tasks to Chapter B.5 in the main paper where the difference was more meaningful. As shown in Chapter B.6, there were some tasks such as `temporal_sequences`, `web_of_lies` and `tracking_shuffled_objects_five_objects` where GPT-4 achieved 100% accuracy with *any* of the approaches, including both LEAP and the few-shot baseline.

B.6 Additional Qualitative Analysis

Object Counting

Question: I have a flute, a piano, a trombone, four stoves, a violin, an accordion, a clarinet, a drum, two lamps, and a trumpet. How many musical instruments do I have?

Principles: Focus on relevance: The response should be directly related to the user's query or statement. Irrelevant information can confuse the user and detract from the overall quality of the response. ...

CoT Response: Let's think step by step. We first identify the musical instruments on the list and include their quantity in parentheses: - flute (1) - piano (1) - trombone (1) - stoves (4) - violin (1) - accordion (1) - clarinet (1) - drum (1) - lamps (2) - trumpet (1) Now, let's add the numbers in parentheses: $1 + 1 + 1 + 4 + 1 + 1 + 1 + 1 + 2 + 1 = 14$. So the answer is $\boxed{14}$. ✗

LEAP_{low-level} Response: Let's think step by step. We first identify the musical instruments on the list and include their quantity in parentheses: - flute (1) - piano (1) - trombone (1) - violin (1) - accordion (1) - clarinet (1) - drum (1) - trumpet (1) Now, let's add the numbers in parentheses: $1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 8$. So the answer is $\boxed{8}$. ✓

Salient Translation Error Detection

Question: The following translations from German to English contain a particular error. That error will be one of the following types: Named Entities, Numerical Values, Modifiers or Adjectives, Negation or Antonyms, Facts, Dropped Content. Please identify that error. Source: Neu Fahrland ist ein Ortsteil von Potsdam. Translation: Neu Fahrland is a district of Pots. The translation contains an error pertaining to Options: (A) Modifiers or Adjectives (B) Numerical Values (C) Negation or Antonyms (D) Named Entities (E) Dropped Content (F) Facts

Principles: When analyzing translations, it is important to carefully consider the translation of named entities. Any changes or inaccuracies in the translation of names, places, or other entities should be identified as named entity errors.

CoT Response: Let's think step by step. We solve this question by first translating the source sentence to English and then by comparing our translation with the provided translation. According to Google Translate, the correct translation of the source sentence from German to English is "Neu Fahrland is a district of Potsdam." On the other hand, the provided translation is "Neu Fahrland is a district of Pots." Note that the provided translation is missing the word "dam" at the end of Pots. Because a significant part of the translation is dropped, the translation contains an error pertaining to Dropped Content. So the answer is (E) . ✘

LEAP_{low-level} Response: Let's think step by step. We solve this question by first translating the source sentence to English and then by comparing our translation with the provided translation. According to Google Translate, the correct translation of the source sentence from German to English is "Neu Fahrland is a district of Potsdam." On the other hand, the provided translation is "Neu Fahrland is a district of Pots." Note that the provided translation omits the word "dam" from Potsdam. Because a significant part of the entity name is removed in the translation, the translation contains an error pertaining to Named Entities. So the answer is (D) . ✔

Boolean Expressions

Question: not not (False and not False) is

Principles: ... 3. Double negation, as in 'not not', cancels out and returns the original value. ...

CoT Response: Let's think step by step. Remember that (i) expressions inside brackets are always evaluated first and that (ii) the order of operations from highest priority to lowest priority is "not", "and", "or", respectively. We first simplify this expression "Z" as follows: "Z = not not (False and not False) = not not (A)" where "A = False and not False". Let's evaluate A: A = False and not False = False and True = False. Plugging in A, we get: Z = not not (A) = not not (False) = not not True = not False = True. So the answer is True. ✘

LEAP_{low-level} Response: Let's think step by step. Remember that (i) expressions inside brackets are always evaluated first and that (ii) the order of operations from highest priority to lowest priority is "not", "and", "or", respectively. We first simplify this expression "Z" as follows: "Z = not not (False and not False) = not not (A)" where "A = False and not False". Let's evaluate A: A = False and not False = False and True = False. Plugging in A, we get: Z = not not (A) = not not (False) = not not False = False. So the answer is False. ✔

Snarks

Question: Which statement is sarcastic? Options: (A) It's not our job to fund education! It should be the government paying the maintenance to keep schools open (B) It's not our job to fund education! It should be the kids paying the maintenance to keep schools open

Principles: To improve future responses, it is important to carefully consider the context and tone of the statements when identifying sarcasm. Sarcasm often involves saying the opposite of what is meant . . . Paying attention to the incongruity between the literal meaning of the words and the intended meaning can help in accurately identifying sarcasm.

CoT Response: Let's think step by step. If we look at (A), it suggests that funding education is not the responsibility of the speaker or the people, but rather the government. This statement is often used sarcastically to criticize individuals or groups who do not prioritize or support funding for education. If we look at (B), it suggests that funding education is not the responsibility of the speaker or the people, but rather the kids. This statement is not typically used sarcastically and can be seen as a genuine belief or opinion. Above the above, the sarcastic option is (A). So the answer is (A).

X

LEAP_{low-level} Response: Let's think step by step. If we look at (A), it suggests that it is not the responsibility of the speaker or their group to fund education, and instead, it should be the government's responsibility. This statement aligns with the commonly accepted understanding that education funding is a shared responsibility between individuals, communities, and the government. Therefore, (A) does not appear to contain sarcasm. If we look at (B), it suggests that it is not the responsibility of the speaker or their group to fund education, and instead, it should be the kids' responsibility. This statement contradicts the generally accepted understanding that children should not be burdened with the financial responsibility of maintaining schools. The presence of such a contradiction in (B) suggests that it contains a taste of irony and sarcasm. Above the above, the sarcastic option is (B). So the answer is (B). ✓

Date Understanding

Question: May 6, 1992 is like yesterday to Jane, but that is actually ten years ago. What is the date one week ago from today in MM/DD/YYYY? Options: (A) 04/29/2002 (B) 04/24/2002 (C) 04/19/2002 (D) 04/28/2002 (E) 02/13/2002 (F) 05/20/2002

Principles: . . . When subtracting days from a given date, it is important to accurately calculate the number of days to subtract and perform the subtraction correctly. . . .

CoT Response: Let's think step by step. If May 6, 1992 is like yesterday to Jane, but that is actually ten years ago, then today's date is May 16, 2002. One week ago from today is May 9, 2002. So the answer is (D) . ✗

LEAP_{low-level} Response: Let's think step by step. If May 6, 1992 is ten years ago, then today's date is May 6, 2002. One week ago from today is April 29, 2002, that is 04/29/2002. So the answer is (A) . ✓

Multistep Arithmetic Two

Question: $((5 - 6 - 7 + 6) - (-1 - 1 * -7 + 9)) =$

Principles: It is crucial to pay attention to the signs and operations when simplifying expressions inside parentheses. Any mistake in this step can lead to incorrect results.

CoT Response: Let's think step by step. Let's recall that the order of operations in mathematics is as follows: (1) Parentheses, (2) exponents, (3) multiplication and division (from left to right), (4) addition and multiplication (from left to right). So, remember to always compute the expressions inside parentheses or brackets first. This equation can be written as "A - B", where $A = (5 - 6 - 7 + 6)$ and $B = (-1 - 1 * -7 + 9)$. Let's calculate $A = (5 - 6 - 7 + 6) = (5 - (6 - 7) + 6) = (5 - (-1) + 6) = (5 + 1 + 6) = 12$. Let's calculate $B = (-1 - 1 * -7 + 9) = (-1 - (-7) + 9) = (-1 + 7 + 9) = 15$. Then, the final equation is $A - B = 12 - 15 = -3$. So the answer is (-3) . ✗

LEAP_{low-level} Response: A: Let's think step by step. Let's recall that the order of operations in mathematics is as follows: (1) Parentheses, (2) exponents, (3) multiplication and division (from left to right), (4) addition and multiplication (from left to right). So, remember to always compute the expressions inside parentheses or brackets first. This equation can be written as "A - B", where $A = (5 - 6 - 7 + 6)$ and $B = (-1 - 1 * -7 + 9)$. Let's calculate $A = (5 - 6 - 7 + 6) = (5 - 6 - 7 + 6) = (-1 - 7 + 6) = (-8 + 6) = -2$. Let's calculate $B = (-1 - 1 * -7 + 9) = (-1 - (-7) + 9) = (-1 + 7 + 9) = (6 + 9) = 15$. Then, the final equation is $A - B = -2 - 15 = -17$. So the answer is (-17) . ✓

Disambiguation QA

Question: In the following sentences, explain the antecedent of the pronoun (which thing the pronoun refers to), or state that it is ambiguous. Sentence: The undergraduate applied to work with the scientist because he needed a lab assistant. Options: (A) The undergraduate needed an assistant (B) The scientist needed an assistant (C) Ambiguous

Principles: Pay attention to the causality and context of the sentence to identify the most likely referent for the pronoun. **It is not enough to simply assume that the pronoun could refer to any possible noun mentioned in the sentence.**

CoT Response: Let's think step by step. Here we need to determine who the pronoun "he" might be referring to. There are two possible referents for "he", namely the undergraduate and the scientist. The verb "applied" might be able to help us determine which one is more likely (if either). Let X be the undergraduate and Y the scientist. The sentence is then of the form "X applied to work with Y because (X or Y) needed a lab assistant." Let's consider Y first: "X applied to work with Y because Y needed a lab assistant." This case makes sense, because X applied to work with Y because Y had a need for a lab assistant. Now, consider X: "X applied to work with Y because X needed a lab assistant." This case also makes sense, **because X applied to work with Y because X had a need for a lab assistant.** Because both X and Y are possible at the same time, we conclude that the antecedent of the pronoun is ambiguous. So the answer is

(C). **LEAP_{low-level} Response:** Let's think step by step. Here we need to determine who the pronoun "he" might be referring to. There are two possible referents for "he", namely the undergraduate and the scientist. The verb "applied" might be able to help us determine which one is more likely (if either). Let X be the undergraduate and Y the scientist. The sentence is then of the form "X applied to work with Y because (X or Y) needed a lab assistant." Let's consider Y first: "X applied to work with Y because Y needed a lab assistant." This case makes sense, because X wanted to work with Y because Y needed assistance in the lab. Now, consider X: "X applied to work with Y because X needed a lab assistant." This case doesn't make much sense, because X, as an undergraduate, would typically be the one seeking an opportunity to work with a scientist, not the other way around. **Because in this context, X is the undergraduate and Y is the scientist, the answer should be the scientist.** So the answer is **(B)**.

B.7 Learned Principles

Tables **B.7-B.37** show the high-level principles learned by LEAP. Tables **B.38-B.41** show low-level principles learned by LEAP.

Table B.5: **BBH Results:** Accuracy in BBH tasks, across GPT-3.5-turbo, GPT-4, and Gemini Pro. The best approach for each base LLM in each tasks is in **bold**; the second-best approach is underlined. Each number represents the average across 3 identical runs with a temperature of zero.

Task	Approach	GPT-3.5-turbo	GPT-4	Gemini Pro
boolean_expressions	Few-shot CoT	94.13	96.80	91.20
	LEAP _{LOW-LEVEL}	<u>94.93</u>	98.00	93.60
	LEAP _{HIGH-LEVEL}	95.07	<u>97.60</u>	<u>92.40</u>
disambiguation_qa	Few-shot CoT	65.73	<u>85.60</u>	<u>66.00</u>
	LEAP _{LOW-LEVEL}	<u>67.20</u>	86.00	66.00
	LEAP _{HIGH-LEVEL}	68.00	85.20	70.40
formal_fallacies	Few-shot CoT	56.00	<u>81.60</u>	<u>58.80</u>
	LEAP _{LOW-LEVEL}	56.93	84.00	61.20
	LEAP _{HIGH-LEVEL}	<u>56.80</u>	<u>81.60</u>	57.60
hyperbaton	Few-shot CoT	<u>83.07</u>	98.00	<u>88.40</u>
	LEAP _{LOW-LEVEL}	81.33	99.20	91.20
	LEAP _{HIGH-LEVEL}	84.80	<u>98.80</u>	87.60
logical_deduction_five_objects	Few-shot CoT	60.67	<u>85.20</u>	60.40
	LEAP _{LOW-LEVEL}	<u>59.47</u>	85.60	62.80
	LEAP _{HIGH-LEVEL}	<u>57.73</u>	84.40	<u>60.80</u>
logical_deduction_seven_objects	Few-shot CoT	38.67	56.80	<u>49.20</u>
	LEAP _{LOW-LEVEL}	40.00	<u>58.40</u>	50.00
	LEAP _{HIGH-LEVEL}	<u>39.20</u>	59.60	48.40
movie_recommendation	Few-shot CoT	81.47	90.40	83.20
	LEAP _{LOW-LEVEL}	<u>82.53</u>	<u>91.60</u>	84.80
	LEAP _{HIGH-LEVEL}	84.00	92.00	<u>83.60</u>
multistep_arithmetic_two	Few-shot CoT	81.73	<u>93.20</u>	<u>79.60</u>
	LEAP _{LOW-LEVEL}	82.00	94.00	78.00
	LEAP _{HIGH-LEVEL}	82.00	92.40	81.20
navigate	Few-shot CoT	94.27	98.00	95.20
	LEAP _{LOW-LEVEL}	95.20	98.40	95.20
	LEAP _{HIGH-LEVEL}	<u>94.93</u>	98.40	94.80
object_counting	Few-shot CoT	96.80	98.80	94.80
	LEAP _{LOW-LEVEL}	97.87	99.20	<u>93.20</u>
	LEAP _{HIGH-LEVEL}	<u>97.60</u>	99.20	91.60
ruin_names	Few-shot CoT	<u>60.13</u>	<u>87.60</u>	<u>72.80</u>
	LEAP _{LOW-LEVEL}	53.47	89.60	<u>72.80</u>
	LEAP _{HIGH-LEVEL}	68.13	<u>87.60</u>	74.40
snarks	Few-shot CoT	<u>67.42</u>	<u>89.89</u>	82.58
	LEAP _{LOW-LEVEL}	70.60	89.33	79.21
	LEAP _{HIGH-LEVEL}	66.48	91.01	<u>80.90</u>
sports_understanding	Few-shot CoT	<u>93.87</u>	<u>94.40</u>	98.40
	LEAP _{LOW-LEVEL}	94.80	95.20	98.40
	LEAP _{HIGH-LEVEL}	88.00	<u>94.40</u>	98.00
word_sorting	Few-shot CoT	<u>53.07</u>	<u>94.40</u>	82.00
	LEAP _{LOW-LEVEL}	52.00	94.00	<u>83.20</u>
	LEAP _{HIGH-LEVEL}	53.60	95.60	84.80

Table B.6: Additional results on Big-Bench Hard tasks. The best approach for each base LLM in each tasks is in **bold**; the second-best approach is underlined. Each number represents the average across 3 identical runs with a temperature of zero.

Task	Approach	GPT-3.5-turbo	GPT-4	Gemini-Pro
geometric_shapes	Few-shot	67.33	54.40	<u>50.00</u>
	+ LEAP _{LOW-LEVEL}	59.60	<u>57.20</u>	<u>50.00</u>
	+ LEAP _{HIGH-LEVEL}	<u>60.13</u>	58.80	50.40
causal_judgement	Few-shot	63.64	73.26	62.57
	+ LEAP _{LOW-LEVEL}	63.64	<u>72.73</u>	62.57
	+ LEAP _{HIGH-LEVEL}	63.10	<u>72.73</u>	62.57
date_understanding	Few-shot	86.67	<u>90.00</u>	87.60
	+ LEAP _{LOW-LEVEL}	83.73	91.20	87.60
	+ LEAP _{HIGH-LEVEL}	<u>85.60</u>	<u>90.00</u>	86.40
dyck_languages	Few-shot	35.73	56.80	0
	+ LEAP _{LOW-LEVEL}	33.07	56.80	0
	+ LEAP _{HIGH-LEVEL}	35.73	54.00	0
logical_deduction _three_objects	Few-shot	88.80	99.20	89.20
	+ LEAP _{LOW-LEVEL}	86.80	<u>98.80</u>	90.00
	+ LEAP _{HIGH-LEVEL}	85.73	<u>98.80</u>	90.80
penguins_in_a_table	Few-shot	76.94	97.26	81.51
	+ LEAP _{LOW-LEVEL}	73.74	<u>96.58</u>	<u>78.77</u>
	+ LEAP _{HIGH-LEVEL}	<u>73.97</u>	<u>96.58</u>	76.71
reasoning_about _colored_objects	Few-shot	82.13	95.20	83.20
	+ LEAP _{LOW-LEVEL}	<u>77.87</u>	91.20	84.00
	+ LEAP _{HIGH-LEVEL}	75.87	<u>94.00</u>	82.40
salient_translation _error_detection	Few-shot	55.73	68.80	56.00
	+ LEAP _{LOW-LEVEL}	<u>55.20</u>	<u>67.20</u>	52.80
	+ LEAP _{HIGH-LEVEL}	54.27	<u>67.20</u>	<u>55.20</u>
temporal_sequences	Few-shot	60.93	100.0	99.60
	+ LEAP _{LOW-LEVEL}	65.20	100.0	98.00
	+ LEAP _{HIGH-LEVEL}	<u>62.80</u>	100.0	<u>99.20</u>
tracking_shuffled_objects _five_objects	Few-shot	72.40	100.0	<u>64.00</u>
	+ LEAP _{LOW-LEVEL}	70.93	100.0	64.40
	+ LEAP _{HIGH-LEVEL}	<u>71.33</u>	100.0	<u>64.00</u>
tracking_shuffled_objects _seven_objects	Few-shot	65.07	100.0	56.8
	+ LEAP _{LOW-LEVEL}	<u>62.53</u>	100.0	<u>60.0</u>
	+ LEAP _{HIGH-LEVEL}	61.73	98.8	60.8
tracking_shuffled_objects _three_objects	Few-shot	<u>83.60</u>	100.0	69.60
	+ LEAP _{LOW-LEVEL}	82.80	100.0	<u>69.20</u>
	+ LEAP _{HIGH-LEVEL}	85.87	100.0	68.40
web_of_lies	Few-shot	98.67	100.0	100.0
	+ LEAP _{LOW-LEVEL}	99.20	100.0	100.0
	+ LEAP _{HIGH-LEVEL}	99.20	100.0	100.0

High-level principles ($\text{LEAP}_{\text{HIGH-LEVEL}}$) learned by LEAP for *boolean expressions* (BBH) using GPT-4-0613

1. Ensure clarity and conciseness: Responses should be clear and concise to avoid confusion and maintain the user’s interest.
 2. Avoid redundancies: Redundant information should be eliminated to maintain the user’s attention and improve the efficiency of communication.
 3. Maintain uniqueness: Each response should be unique to keep the user engaged and provide a personalized experience.
 4. Incorporate feedback: User feedback should be analyzed and incorporated into future responses to improve the quality of interaction.
 5. Focus on relevance: Responses should be directly related to the user’s query or statement to maintain relevance and provide accurate information.
 6. Prioritize logical reasoning: Logical reasoning should be the foundation of all responses to ensure accuracy and credibility.
 7. Preserve specific details: Specific details provided by the user should be preserved in the response to show understanding and maintain context.
 8. Continual improvement: Regular introspection and analysis should be conducted to continually improve the quality of responses.
-

Table B.7: High-level Principles 1

High-level principles (LEAP_{HIGH-LEVEL}) learned by LEAP for *causal judgement* (BBH) using GPT-4-0613

1. Consider all relevant details in the scenario: Ensure that all key details are taken into account when forming a conclusion. Do not overemphasize certain details at the expense of others.
 2. Understand the concept of intentionality: It's not enough to initiate an action; the action must be carried out as planned for it to be considered intentional.
 3. Consider all factors in a scenario: This includes the actor's control over the situation and the predictability of the outcome when determining intentionality.
 4. Distinguish between the intent to perform an action and the intent to cause a specific outcome: In questions of causation, it's important to understand these nuances to provide accurate analysis.
 5. Understand the context of causation: Intentionality involves not just the initial action but also the control and predictability of the outcome.
 6. Avoid overemphasis on initial decisions: The initial decision to perform an action should not be overemphasized at the expense of other factors, such as the accidental nature of the outcome.
 7. Improve understanding and interpretation of intent: This includes understanding the difference between the intent to perform an action and the intent to cause a specific outcome.
 8. Ensure adequate consideration of crucial factors: In any scenario, ensure that all aspects, especially those that are crucial to the outcome, are adequately considered before arriving at a conclusion.
-

Table B.8: High-level Principles 2

High-level principles ($\text{LEAP}_{\text{HIGH-LEVEL}}$) learned by LEAP for *date understanding* (BBH) using GPT-4-0613

1. Ensure clarity and precision: Responses should be clear and concise, avoiding any ambiguity or unnecessary complexity.
 2. Maintain relevance: The responses should be directly related to the query or topic at hand, avoiding any irrelevant information.
 3. Avoid redundancy: Each response should provide unique information, avoiding repetition of previously stated facts or ideas.
 4. Prioritize understanding: The responses should be designed to enhance the user's understanding of the topic, rather than simply providing information.
 5. Foster engagement: The responses should be engaging and interactive, encouraging further dialogue and exploration of the topic.
 6. Promote logical reasoning: The responses should be logically sound and well-reasoned, demonstrating a clear thought process.
 7. Respect user's perspective: The responses should respect the user's perspective and knowledge level, avoiding any condescension or oversimplification.
 8. Incorporate feedback: The responses should incorporate feedback from previous interactions, continuously improving in quality and relevance.
-

Table B.9: High-level Principles 3

High-level principles ($\text{LEAP}_{\text{HIGH-LEVEL}}$) learned by LEAP for *disambiguation qa* (BBH) using GPT-4-0613

1. The system should consider all possible interpretations of a sentence when determining the antecedent of a pronoun.
 2. In ambiguous cases, the system should recognize the ambiguity and avoid making assumptions.
 3. The system should consider the context and logical flow of information in the sentence to accurately determine the antecedent of a pronoun.
 4. The system should improve its understanding of how information is typically exchanged in conversations or narratives.
 5. The system should understand the dynamics of a conversation and the flow of information to interpret pronouns correctly.
 6. The system should consider the likelihood of each potential antecedent based on the structure and semantics of the sentence.
 7. The system should pay closer attention to the context and the logical sequence of events in the sentence.
 8. The system should consider the roles and actions of the subjects in the sentence to accurately determine the antecedent of the pronoun.
-

Table B.10: High-level Principles 4

High-level principles ($\text{LEAP}_{\text{HIGH-LEVEL}}$) learned by LEAP for *dyck languages* (BBH) using GPT-4-0613

1. Improve understanding of the rule that parentheses must be closed in the reverse order they were opened, which is fundamental in programming and mathematics.
 2. Enhance the method of analyzing sequences of parentheses by processing each input one by one and keeping track of the stack configuration.
 3. Train the model to recognize that different types of parentheses (e.g., square brackets, curly brackets, round brackets) must be matched with their corresponding closing parentheses.
 4. Improve the model’s ability to track the opening and closing of parentheses in a sequence and identify the ones that are still open at the end of the sequence.
 5. Develop a more detailed understanding of the rules of parentheses and the ability to apply these rules to a given sequence.
 6. Train the model to understand and apply the Last In, First Out (LIFO) principle when dealing with problems related to sequences, particularly those involving brackets or parentheses.
 7. Ensure the model understands that the sequence ends when all brackets are properly closed, not when all opening brackets have been matched with a closing bracket.
 8. Enhance the model’s ability to correctly balance sequences of parentheses.
-

Table B.11: High-level Principles 5

High-level principles ($\text{LEAP}_{\text{HIGH-LEVEL}}$) learned by LEAP for *formal fallacies* (BBH) using GPT-4-0613

1. Ensure clarity and precision: Responses should be clear and concise, avoiding any ambiguity or unnecessary complexity.
 2. Maintain relevance: The responses should be directly related to the query or topic at hand, avoiding any irrelevant information.
 3. Avoid redundancy: Each response should provide unique information, avoiding repetition of previously stated facts or ideas.
 4. Prioritize understanding: The responses should be designed to enhance the user's understanding of the topic, rather than simply providing information.
 5. Foster engagement: The responses should be engaging and interactive, encouraging further discussion or exploration of the topic.
 6. Uphold accuracy: The information provided in the responses should be accurate and reliable, based on verified sources or logical reasoning.
 7. Adapt to context: The responses should be tailored to the specific context of the conversation, taking into account the user's knowledge level, interests, and potential biases.
 8. Promote introspection: The responses should encourage the user to think critically and reflect on the information provided, fostering a deeper understanding and appreciation of the topic.
-

Table B.12: High-level Principles 6

High-level principles ($\text{LEAP}_{\text{HIGH-LEVEL}}$) learned by LEAP for *geometric shapes* (BBH) using GPT-4-0613

1. Improve the system’s understanding of SVG path data and how it represents shapes.
 2. Recognize that the number of "L" commands in an SVG path corresponds to the number of sides in the shape.
 3. Understand that the "M" command in SVG path data moves the current point to a specified location without creating a line.
 4. Count the number of "L" commands to determine the number of sides in the shape.
 5. Recognize that the final "L" command that connects back to the initial "M" command completes the shape, and should not be counted as creating an additional side.
 6. Improve the system’s ability to interpret when an SVG path starts and ends at the same point.
 7. Ensure that points that are visited more than once in the path, such as the starting point, are not double-counted when determining the number of sides in the polygon.
 8. Develop a clear understanding of geometric shapes and their properties to accurately analyze and interpret SVG paths.
-

Table B.13: High-level Principles 7

High-level principles (LEAP_{HIGH-LEVEL}) learned by LEAP for *hyperbaton* (BBH) using GPT-4-0613

1. The system should have a comprehensive understanding of English grammar rules, including the correct order of adjectives.
 2. The system should be able to correctly apply the identified grammatical rules to sentence analysis.
 3. The system should be programmed to handle cases where not all categories of adjectives are present, recognizing that the order of the remaining adjectives is still important.
 4. The system should be able to cross-verify its understanding of rules with the given options to ensure accuracy.
 5. The system should be able to correctly identify the category each adjective falls into and ensure they are in the correct sequence.
 6. The system should be able to correctly compare and evaluate multiple options against these rules to identify the correct answer.
 7. The system should be improved to recognize and apply the correct adjective order of Opinion, Size, Origin, Purpose.
 8. The system should be able to recognize when an adjective category (like Age or Color) is missing, and still maintain the correct order for the remaining adjectives.
-

Table B.14: High-level Principles 8

High-level principles ($\text{LEAP}_{\text{HIGH-LEVEL}}$) learned by LEAP for *logical deduction five objects* (BBH) using GPT-4-0613

1. Enhance Interpretation Skills: The system should improve its ability to interpret and integrate multiple pieces of information accurately.
 2. Improve Spatial Understanding: The system should be able to visualize or map out the spatial arrangement of objects based on given descriptions.
 3. Understand Spatial Prepositions: The system should have a clear understanding of spatial prepositions and their implications in a given context.
 4. Enhance Inference Capabilities: The system should be able to infer information that is not explicitly stated but can be logically deduced from the given information.
 5. Eliminate Redundancies: The system should focus on capturing the essence of the feedback while eliminating any redundancies.
 6. Maintain Clarity and Conciseness: Each point made by the system should be clear, concise, and directly derived from the introspection results.
 7. Retain Specific Details: The system should retain specific details in its responses to ensure accuracy and completeness.
 8. Limit Principles: The system should limit its principles to a maximum of eight to maintain focus and effectiveness.
-

Table B.15: High-level Principles 9

High-level principles ($\text{LEAP}_{\text{HIGH-LEVEL}}$) learned by LEAP for *logical deduction seven objects* (BBH) using GPT-4-0613

1. Ensure clarity and precision: Responses should be clear and concise, avoiding any ambiguity or unnecessary complexity.
 2. Maintain relevance: Responses should directly address the query or topic at hand, avoiding any unrelated or tangential information.
 3. Prioritize uniqueness: Strive to provide unique insights or perspectives in responses, avoiding repetition or common knowledge.
 4. Emphasize logical reasoning: Responses should be logically sound and well-reasoned, with each point building on the last to form a coherent argument or explanation.
 5. Eliminate redundancies: Avoid repeating the same information or points in a response, as this can dilute the message and make the response less engaging.
 6. Incorporate specific details: Where relevant, include specific details in responses to add depth and richness to the information provided.
 7. Limit response length: Keep responses to a reasonable length to ensure they are digestible and maintain the reader's interest.
 8. Continually improve: Regularly review and analyze responses to identify areas for improvement and implement changes as necessary.
-

Table B.16: High-level Principles 10

High-level principles ($\text{LEAP}_{\text{HIGH-LEVEL}}$) learned by LEAP for *logical deduction three objects* (BBH) using GPT-4-0613

1. Ensure clarity and precision: Responses should be clear and concise, avoiding any ambiguity or unnecessary complexity.
 2. Maintain relevance: Responses should directly address the query or topic at hand, avoiding any unrelated or tangential information.
 3. Prioritize uniqueness: Strive to provide unique insights or perspectives in responses, avoiding repetition or common knowledge.
 4. Foster engagement: Responses should be engaging and interesting, aiming to stimulate further discussion or thought.
 5. Uphold accuracy: Ensure all information provided is accurate and up-to-date, avoiding any misinformation or outdated facts.
 6. Promote comprehensibility: Use language and terminology that is easily understood by the intended audience, avoiding jargon or overly complex language.
 7. Encourage brevity: Keep responses succinct and to the point, avoiding unnecessary length or verbosity.
 8. Respect context: Consider the context of the query or discussion when formulating responses, ensuring they are appropriate and relevant.
-

Table B.17: High-level Principles 11

High-level principles ($\text{LEAP}_{\text{HIGH-LEVEL}}$) learned by LEAP for *movie recommendation* (BBH) using GPT-4-0613

1. Ensure Consistency in Labeling: Always cross-check the labels used in the reasoning process with those in the question to avoid discrepancies that could lead to incorrect conclusions.
 2. Comprehensive Analysis: Consider all aspects of the subject matter, such as time period, cultural significance, and popularity, not just the most obvious or immediate ones.
 3. Contextual Understanding: Always consider the broader context of the question to avoid focusing too narrowly on one aspect. This will help in generating more accurate and relevant answers.
 4. Avoid Redundancies: Strive to eliminate any redundancies in the reasoning process to maintain clarity and precision.
 5. Emphasize Uniqueness: Ensure that each principle is unique and offers a different perspective or approach to improve the reasoning process.
 6. Clarity and Conciseness: Make sure each principle is clear, concise, and directly derived from the introspection results.
 7. Detail Orientation: Do not omit specific details that could be crucial to the reasoning process.
 8. Continuous Improvement: Regularly review and update the principles based on new insights or feedback to ensure continuous improvement in the reasoning process.
-

Table B.18: High-level Principles 12

High-level principles ($\text{LEAP}_{\text{HIGH-LEVEL}}$) learned by LEAP for *multistep arithmetic two* (BBH) using GPT-4-0613

1. Strictly adhere to the order of operations (PEMDAS/BODMAS) when performing calculations.
 2. Correctly apply the rules of arithmetic, including multiplication and subtraction operations.
 3. Handle negative numbers accurately, understanding that the multiplication of two negative numbers results in a positive number.
 4. Interpret mathematical notation correctly, especially when dealing with negative numbers and subtraction operations.
 5. Understand that subtraction of a negative number is equivalent to addition of the absolute value of that number.
 6. Double-check intermediate results to ensure accuracy and avoid simple arithmetic errors.
 7. Ensure that the system is designed to correctly perform operations within parentheses.
 8. Improve handling of unique cases, such as double negatives, to avoid misinterpretation.
-

Table B.19: High-level Principles 13

High-level principles ($\text{LEAP}_{\text{HIGH-LEVEL}}$) learned by LEAP for *navigate* (BBH) using GPT-4-0613

1. Ensure clarity and precision: Responses should be clear, concise, and directly answer the question or statement at hand. Avoid ambiguity or overly complex language.
 2. Maintain relevance: Stay on topic and ensure that the response is directly related to the query or statement. Avoid going off on tangents or introducing unrelated information.
 3. Avoid redundancy: Each response should provide new information or a unique perspective. Avoid repeating the same points or ideas.
 4. Be insightful: Responses should provide meaningful insights or perspectives. They should not merely restate the obvious or provide generic answers.
 5. Use specific details: When appropriate, include specific details in responses to enhance understanding and provide context. Avoid being too vague or general.
 6. Be logical: Responses should follow a logical structure and reasoning. Avoid inconsistencies or contradictions in the response.
 7. Adapt to the context: The tone, language, and content of the response should be appropriate for the context. Avoid using inappropriate or irrelevant language or content.
 8. Be respectful: Always maintain a respectful and professional tone in responses. Avoid offensive or disrespectful language or content.
-

Table B.20: High-level Principles 14

High-level principles ($\text{LEAP}_{\text{HIGH-LEVEL}}$) learned by LEAP for *object counting* (BBH) using GPT-4-0613

1. Ensure clarity and precision: Responses should be clear, concise, and directly derived from the information provided. Avoid ambiguity and ensure the response is easily understood by the user.
 2. Eliminate redundancies: Avoid repeating the same information in different ways. Each response should provide new, unique insights.
 3. Focus on relevance: The response should be directly related to the user's query or statement. Irrelevant information can confuse the user and detract from the overall quality of the response.
 4. Maintain logical consistency: The response should follow a logical flow based on the user's input. Avoid contradicting previous statements or information.
 5. Prioritize user's needs: The response should be tailored to the user's needs and preferences. Consider the context and purpose of the user's query when formulating the response.
 6. Incorporate feedback: Regularly analyze user feedback to identify areas for improvement. Use this feedback to refine and enhance future responses.
 7. Promote engagement: The response should encourage further interaction from the user. This can be achieved by asking follow-up questions or providing additional relevant information.
 8. Uphold ethical standards: Ensure that the response respects the user's privacy and adheres to ethical guidelines. Avoid providing information that could potentially harm the user or others.
-

Table B.21: High-level Principles 15

High-level principles (LEAP_{HIGH-LEVEL}) learned by LEAP for *penguins in a table* (BBH) using GPT-4-0613

1. Comprehensive Data Analysis: The system should thoroughly analyze all data points to ensure no relevant information is missed.
 2. Accurate Identification: The system should correctly identify all instances that meet the given criteria.
 3. Counting Accuracy: The system should accurately count all instances that meet the given criteria.
 4. Data Interpretation: The system should improve its ability to interpret data correctly to ensure accurate results.
 5. Error Correction: The system should have mechanisms in place to correct errors in data analysis and interpretation.
 6. Continuous Improvement: The system should continuously strive to improve its performance based on feedback and introspection results.
 7. Redundancy Elimination: The system should focus on capturing the essence of the data, eliminating redundancies where possible.
 8. Clarity and Conciseness: The system should ensure that its responses are clear, concise, and directly derived from the data.
-

Table B.22: High-level Principles 16

High-level principles ($\text{LEAP}_{\text{HIGH-LEVEL}}$) learned by LEAP for *reasoning about colored objects* (BBH) using GPT-4-0613

1. Ensure clarity and precision: Responses should be clear and concise, avoiding any ambiguity or unnecessary complexity.
 2. Maintain relevance: The responses should be directly related to the query or topic at hand, avoiding any irrelevant information.
 3. Avoid redundancy: Each response should provide unique information, avoiding repetition of previously stated facts or ideas.
 4. Prioritize understanding: The responses should be designed to enhance the user's understanding of the topic, rather than simply providing information.
 5. Foster engagement: The responses should be engaging and interactive, encouraging further dialogue and exploration of the topic.
 6. Uphold accuracy: The information provided in the responses should be accurate and reliable, based on verified sources.
 7. Adapt to user's needs: The responses should be tailored to the user's level of knowledge and interest, providing more detailed information when necessary and simplifying complex concepts when appropriate.
 8. Promote introspection: The responses should encourage the user to think critically about the topic, promoting deeper understanding and personal growth.
-

Table B.23: High-level Principles 17

High-level principles ($\text{LEAP}_{\text{HIGH-LEVEL}}$) learned by LEAP for *ruin names* (BBH) using GPT-4-0613

1. Ensure clarity and conciseness: Responses should be clear and concise to avoid any confusion or misunderstanding. Avoid using complex language or jargon that may not be understood by all users.
 2. Eliminate redundancies: Avoid repeating the same information in different ways. This can make responses longer than necessary and can be confusing for the user.
 3. Directly derive from introspection results: Responses should be directly based on the analysis or introspection results. This ensures that the responses are relevant and accurate.
 4. Capture the essence of the feedback: The main points or key messages of the feedback should be captured in the responses. This ensures that the responses are meaningful and valuable to the user.
 5. Limit the number of principles: Too many principles can be overwhelming and difficult to remember. Limit the number of principles to a maximum of 8.
 6. Leave specific details in place: While it's important to be concise, it's also important not to remove specific details that are necessary for understanding the response. These details can provide context and depth to the response.
 7. Ensure uniqueness: Each principle should be unique and not overlap with other principles. This ensures that each principle provides a distinct value.
 8. Focus on improvement: The principles should focus on ways to improve future responses. This ensures that the principles are forward-looking and proactive.
-

Table B.24: High-level Principles 18

High-level principles ($\text{LEAP}_{\text{HIGH-LEVEL}}$) learned by LEAP for *salient translation error detection* (BBH) using GPT-4-0613

1. The system should enhance its focus on the overall meaning and context of the sentence, not just the presence or absence of certain words or phrases.
 2. The system should improve its ability to identify and classify errors accurately, particularly those related to dropped content.
 3. The system should pay more attention to Named Entities errors, ensuring that the entity in the translation matches the entity in the source.
 4. The system should improve its precision in identifying the number of errors present in a translation, avoiding overestimation or underestimation.
 5. The system should enhance its understanding of the context and specific details within the text to better identify and differentiate between different types of errors.
 6. The system should improve its ability to accurately translate and compare the original and translated texts to identify any discrepancies.
 7. The system should be able to recognize when a specific name, place, or location is altered in the translation, which constitutes a named entity error.
 8. The system should focus on improving its ability to identify errors related to named entities, rather than misclassifying them as other types of errors.
-

Table B.25: High-level Principles 19

High-level principles ($\text{LEAP}_{\text{HIGH-LEVEL}}$) learned by LEAP for *snarks* (BBH) using GPT-4-0613

1. Ensure clarity and precision: Responses should be clear and concise, avoiding any ambiguity or confusion.
 2. Eliminate redundancies: Avoid repeating the same information or ideas in different ways.
 3. Direct derivation: Each response should be directly derived from the information provided, without making unnecessary assumptions or inferences.
 4. Maintain uniqueness: Each response should provide unique insights, rather than reiterating common knowledge or previously stated information.
 5. Focus on relevance: Responses should be directly relevant to the question or topic at hand, avoiding any irrelevant or off-topic information.
 6. Prioritize insightful content: Responses should aim to provide new insights or perspectives, rather than simply restating the obvious.
 7. Respect the context: The context in which the question or topic is presented should be taken into account when formulating responses.
 8. Preserve specific details: While responses should be concise, they should not omit important details that are necessary for a full understanding of the topic.
-

Table B.26: High-level Principles 20

High-level principles ($\text{LEAP}_{\text{HIGH-LEVEL}}$) learned by LEAP for *sports understanding* (BBH) using GPT-4-0613

1. Understand the Context: The system should focus on understanding the context and the exact question being asked, differentiating between different types of questions and providing reasoning accordingly.
 2. Accurate Knowledge: The system should have accurate and up-to-date knowledge about the subject matter. In this case, understanding the rules of basketball and the correct terminology is crucial.
 3. Avoid Unstated Assumptions: The system should be careful not to make assumptions that are not explicitly stated in the sentence or question.
 4. Consider Structure: The system should consider the structure of the subject matter for accurate analysis. For example, understanding the structure of the NBA Finals, which includes teams from both conferences.
 5. Adapt Response Complexity: The system should adapt its responses based on the complexity and detail level of the question or task at hand. A simpler, more direct response may be more appropriate in some cases.
 6. Consider All Plausible Scenarios: The system should consider all plausible scenarios, not just the most common or current ones. For example, the possibility of player trades in the NBA.
 7. Update Information: The system should be updated with the most recent and accurate information about the subject matter to provide the most accurate analysis.
 8. Unique and Insightful Principles: The system should generate unique and insightful principles to improve future responses, focusing on capturing the essence of the feedback while eliminating redundancies.
-

Table B.27: High-level Principles 21

High-level principles ($\text{LEAP}_{\text{HIGH-LEVEL}}$) learned by LEAP for *temporal sequences* (BBH) using GPT-4-0613

1. Ensure clarity and precision: Responses should be clear and concise, avoiding any ambiguity or confusion.
 2. Eliminate redundancies: Avoid repeating the same information or ideas in different ways.
 3. Direct derivation: Each response should be directly derived from the information provided, without making unnecessary assumptions or inferences.
 4. Maintain uniqueness: Each response should provide unique insights, rather than reiterating common knowledge or previously stated information.
 5. Focus on relevance: Responses should be directly relevant to the question or topic at hand, avoiding any irrelevant or off-topic information.
 6. Prioritize insightful content: Responses should aim to provide new insights or perspectives, rather than simply restating the obvious.
 7. Respect the context: The context in which the question or topic is presented should be taken into account when formulating responses.
 8. Keep the audience in mind: The responses should be tailored to the needs and expectations of the intended audience.
-

Table B.28: High-level Principles 22

High-level principles ($\text{LEAP}_{\text{HIGH-LEVEL}}$) learned by LEAP for *tracking shuffled objects five objects* (BBH) using GPT-4-0613

1. Ensure clarity and precision: Responses should be clear and concise, avoiding any ambiguity or unnecessary complexity.
 2. Maintain relevance: The responses should be directly related to the query or topic at hand, avoiding any irrelevant information or digressions.
 3. Prioritize uniqueness: Each response should offer a unique perspective or insight, avoiding repetition or redundancy.
 4. Incorporate feedback: Future responses should take into account any feedback received, using it to improve the quality and relevance of the responses.
 5. Focus on logic: Responses should be logically sound, with each point or argument following logically from the previous one.
 6. Be insightful: Responses should aim to provide new insights or perspectives, rather than simply reiterating known information.
 7. Maintain consistency: The tone, style, and content of the responses should be consistent, ensuring a coherent and cohesive narrative.
 8. Respect context: The responses should take into account the context in which the query or topic is being discussed, ensuring that they are appropriate and relevant.
-

Table B.29: High-level Principles 23

High-level principles ($\text{LEAP}_{\text{HIGH-LEVEL}}$) learned by LEAP for *tracking shuffled objects seven objects* (BBH) using GPT-4-0613

1. Ensure clarity and precision: Responses should be clear and concise, avoiding any ambiguity or unnecessary complexity.
 2. Maintain relevance: The responses should be directly related to the query or topic at hand, avoiding any irrelevant information or digressions.
 3. Prioritize uniqueness: Strive to provide unique insights or perspectives in the responses, avoiding repetition or common knowledge.
 4. Foster engagement: Responses should be engaging and interesting, aiming to stimulate further discussion or thought.
 5. Uphold accuracy: Ensure that all information provided in the responses is accurate and up-to-date, avoiding any misinformation or outdated facts.
 6. Promote comprehensibility: Use simple and understandable language in the responses, ensuring they are accessible to a wide range of audiences.
 7. Encourage brevity: Keep responses succinct and to the point, avoiding unnecessary length or verbosity.
 8. Emphasize logical reasoning: Responses should be logically sound and well-reasoned, ensuring they are credible and persuasive.
-

Table B.30: High-level Principles 24

High-level principles ($\text{LEAP}_{\text{HIGH-LEVEL}}$) learned by LEAP for *tracking shuffled objects three objects* (BBH) using GPT-4-0613

1. Accurate Tracking: The system should accurately track changes in a sequence of events to ensure correct interpretation and response.
 2. Avoid Assumptions: The system should not make assumptions that are not supported by the given information. It should only rely on the facts presented.
 3. Strict Adherence: The system should strictly adhere to the described events and changes, without deviating from the provided information.
 4. Partner Swaps: In the context of a dance, the system should correctly follow partner swaps to ensure no dancer is left without a partner.
 5. Data Verification: The system should verify the data it receives to ensure it is accurate and reliable before making any decisions or predictions.
 6. Continuous Improvement: The system should continuously learn and improve its performance based on feedback and analysis of past responses.
 7. Contextual Understanding: The system should understand the context of the information provided to make accurate interpretations.
 8. Error Correction: The system should have mechanisms in place to correct errors and inaccuracies in its responses.
-

Table B.31: High-level Principles 25

High-level principles ($\text{LEAP}_{\text{HIGH-LEVEL}}$) learned by LEAP for *web of lies* (BBH) using GPT-4-0613

1. The AI model should always pay close attention to the format of the question.
 2. The model should provide the answer in the same format as the question to ensure consistency.
 3. If the question is a yes/no question, the answer should be given as "Yes" or "No" to directly address the question.
 4. The model should be trained to understand the format of the question and provide the answer in the same format.
 5. The model's responses should align with the expectations set by the question.
 6. The answer should be directly relevant to the question and easy for the user to understand.
 7. The model should match the style of the answer to the style of the question.
 8. In a conversational style question, the answer should also be given in a conversational style.
-

Table B.32: High-level Principles 26

High-level principles ($\text{LEAP}_{\text{HIGH-LEVEL}}$) learned by LEAP for *word sorting* (BBH) using GPT-4-0613

1. Maintain the application of the principle of alphabetical sorting in future tasks.
 2. Prepare the model to handle more complex cases, such as words with identical prefixes.
 3. Ensure the model can handle words that differ in case or punctuation.
 4. Continually assess the model’s understanding and application of alphabetical order.
 5. Maintain the model’s level of understanding and accuracy for similar tasks.
 6. Expose the model to a variety of tasks to ensure it can generalize the principle across different contexts.
 7. Ensure the model can handle sorting words with special characters or numbers.
 8. Regularly evaluate and improve the model’s performance based on feedback and introspection results.
-

Table B.33: High-level Principles 27

High-level principles ($\text{LEAP}_{\text{HIGH-LEVEL}}$) learned by LEAP for DROP using Gemini Pro

1. Maintain precision when dealing with percentages to avoid rounding errors.
 2. Perform calculations using the full precision available and only round as a final step, if necessary.
 3. Avoid rounding when answering questions that ask for exact values.
 4. Consider the level of rounding carefully to ensure that it does not introduce significant errors.
 5. Rounding can lead to incorrect conclusions and misinterpretations of data.
 6. Keep the original values and perform calculations using the full precision available.
 7. Rounding should be done only after all calculations are complete.
 8. Rounding errors can lead to incorrect results.
-

Table B.34: High-level Principles 28

High-level principles ($\text{LEAP}_{\text{HIGH-LEVEL}}$) learned by LEAP for DROP using GPT-4-0613

1. Improve context understanding: The system should be designed to better understand the context of the question, including specific details and requirements, to generate accurate answers.
 2. Prioritize direct information: The system should prioritize direct information given in the passage over inferred information to avoid unnecessary assumptions or calculations.
 3. Accurate interpretation: The system should carefully interpret the context of the question, recognizing specific details and changes in entities or events over time.
 4. Adhere to the required format: The system should ensure that the final answer adheres to the required format, such as providing a single numerical answer when required.
 5. Avoid unnecessary assumptions: The system should not make unnecessary assumptions when the required information is directly provided in the passage.
 6. Improve accuracy of analysis: The system should strive to improve the accuracy of its analysis by focusing on the specific details provided in the passage and the question.
 7. Recognize transitions and transformations: The system should be capable of recognizing when one entity transitions or transforms into another, as this can affect the interpretation of the question and the accuracy of the answer.
 8. Focus on direct answers: When the answer is directly stated in the passage, the system should focus on providing that answer rather than attempting to infer or calculate the answer.
-

Table B.35: High-level Principles 29

High-level principles ($\text{LEAP}_{\text{HIGH-LEVEL}}$) learned by LEAP for HotpotQA using Gemini Pro

1. Verify the accuracy of information before drawing conclusions, especially when comparing start dates of entities.
 2. Consider all relevant aspects when identifying commonalities between entities, not just the most obvious or prominent ones.
 3. Analyze the work of mathematicians or scientists within the broader context of their field of study, rather than focusing solely on specific subfields or topics.
 4. Consider all common professions when answering questions about the professions of multiple people, not just the most prominent one.
 5. Avoid making erroneous conclusions based on incorrect data.
 6. Ensure a comprehensive analysis of all shared characteristics to provide accurate and complete answers about commonalities.
 7. Provide a more accurate assessment of similarities and differences in the work of mathematicians or scientists by considering the broader field of study.
 8. Eliminate redundancies and capture the essence of the feedback to create clear, concise, and directly derived principles.
-

Table B.36: High-level Principles 30

High-level principles ($\text{LEAP}_{\text{HIGH-LEVEL}}$) learned by LEAP for HotpotQA using GPT-4-0613

1. Ensure comprehensive analysis of all relevant information, considering all aspects of the subject matter, not just the most prominent ones.
 2. Improve categorization of work or field of study, understanding the hierarchical and categorical relationships between different fields.
 3. Always conclude responses with a clear answer statement, such as "So the answer is answer_i ".
 4. Expand the range of accessible information to answer specific questions accurately.
 5. Improve understanding of the topic at hand, pulling from a wider range of data if necessary.
 6. Recognize when a question is asking for a specific piece of information and strive to provide that.
 7. Avoid defaulting to "unknown" when the answer isn't immediately apparent, instead, make an effort to find the required information.
 8. Improve the ability to recognize and understand the background and inspirations of characters or subjects in various fields.
-

Table B.37: High-level Principles 31

Low-level principles (LEAP_{LOW-LEVEL}) learned by LEAP for HotpotQA using GPT-4-0613

1. The principle that should be looked at carefully to improve the performance in the future is to ensure a comprehensive analysis of all the relevant information. In this case, all the professions of the individuals should have been considered, not just the most prominent one. This will help to provide a more accurate and complete answer.
 2. The system should be more careful when categorizing the type of work or field of study of individuals. Even if their specific areas of focus or contributions differ, they may still belong to the same broader field or type of work. In this case, both individuals are mathematicians, so they are known for the same type of work. The system should be able to recognize and understand the hierarchical and categorical relationships between different fields of study or types of work.
 3. Pay attention to the format! End your response with "So the answer is answer_i ".
 4. The system should be able to access a broader range of information to answer specific questions accurately. In this case, it would need to know about the background and inspirations of Matt Groening's characters in 'The Simpsons'. This suggests that the system needs to be able to pull from a wider range of data or have a more comprehensive understanding of the topic at hand. It's also important for the system to recognize when a question is asking for a specific piece of information and to strive to provide that, rather than defaulting to "unknown" when the answer isn't immediately apparent.
-

Table B.38: Low-level Principles 1

Low-level principles ($\text{LEAP}_{\text{LOW-LEVEL}}$) learned by LEAP for HotpotQA using Gemini Pro

1. When comparing the start dates of two entities, it is crucial to ensure the accuracy of the information used. Incorrect data can lead to erroneous conclusions. Always verify the accuracy of information before drawing conclusions.
 2. When answering questions about commonalities between two entities, it is important to consider all relevant aspects and not just the most obvious or prominent one. A comprehensive analysis of all shared characteristics is necessary to provide an accurate and complete answer.
 3. When analyzing the work of mathematicians or scientists, it is important to consider the broader field of study that their contributions belong to, rather than focusing solely on the specific subfields or topics that they worked on. This broader perspective allows for a more accurate assessment of the similarities and differences in their work and helps to avoid incorrect conclusions about the nature of their contributions.
 4. When answering questions about the professions of multiple people, it is important to consider all of the professions that they have in common, not just the most prominent one.
-

Table B.39: Low-level Principles 2

Low-level principles ($\text{LEAP}_{\text{LOW-LEVEL}}$) learned by LEAP for DROP using GPT-4-0613

1. The system should be designed to understand the context of the question better. In this case, it should have recognized that the question was asking for the duration of existence of the European Coal and Steel Community before it transitioned into the European Economic Community. Understanding the specific context and requirements of a question is crucial for generating accurate answers.
 2. The system should be designed to carefully interpret the context of the question. In this case, it should have recognized that the ECSC transformed into the EEC in 1958, and therefore, its existence as the ECSC ended in that year. Understanding the context and specific details of the question is crucial for generating accurate answers.
 3. The system should prioritize direct information given in the passage over inferred information. In this case, the direct information was the total number of touchdown passes thrown by Stafford, which was clearly stated in the passage. The system should not make unnecessary assumptions or calculations when the required information is directly provided. This will help to avoid errors and improve the accuracy of the analysis.
 4. Pay attention to the format! Make sure your final answer should be a single numerical number, in the form `boxedanswer`, at the end of your response.
-

Table B.40: Low-level Principles 3

Low-level principles ($\text{LEAP}_{\text{LOW-LEVEL}}$) learned by LEAP for DROP using Gemini Pro

1. When dealing with percentages, it is important to maintain precision and avoid rounding errors, especially when the difference between values is small. Rounding should only be done as a final step, if necessary, to ensure that the result is presented in a clear and concise manner.
 2. When dealing with percentages, it is important to maintain precision and avoid rounding errors. Rounding should only be done as a final step, after all calculations are complete.
 3. When dealing with percentages, it is important to maintain precision and avoid rounding errors. Rounding can lead to incorrect conclusions and misinterpretations of data. Always keep the original values and perform calculations using the full precision available. Rounding should only be done as a final step, if necessary, and the level of rounding should be carefully considered to ensure that it does not introduce significant errors.
 4. When answering questions that ask for exact values, it is important to not round the answer unless specifically instructed to do so. Rounding the answer can lead to incorrect results, as seen in this example.
-

Table B.41: Low-level Principles 4

Appendix C

Tempera

C.1 Training Detail

We provide the training details here. We use standard PPO algorithm to do online policy optimization with GAE. We provide all the hyperparameters here for a reference. We'll specify our neural network architecture in the following section. Note that we perform additional observation normalization (i.e., keeping a running mean and std) and reward normalization. We also adopt the same number of parallel environment as the few-shot setting (e.g., 32 in our few-shot experiments). We found a large size of parallel environment helps boost the performance.

Table C.1: Hyperparameters used for TEMPERA in all the tasks.

	Hyperparameter Value
Steps per training	8
Time limit	8
Number Parallel Processes	256
Learning rate	0.00005
Entropy Coefficient	0.005
Value loss Coefficient	0.5
Mini Batch Size	32
Gamma	0.99
GAE Lambda	0.95
Number of in-context Exemplars	4
Number of example pool	16
Positive lambda coefficient (λ_1)	2.0
Negative lambda coefficient (λ_2)	1.8

Table C.2: Few-shot classification results. We compare against different baselines in this setting. Results show that TEMPERA surpasses various baselines including finetuning, prompt tuning and discrete prompt search. The standard deviations are shown in brackets.

		RTE	QNLI	SNLI	MNLI	MRPC
Finetuning	Finetuning (few-shot)	58.6 (3.9)	60.2 (4.7)	54.64 (9.7)	47.8 (7.5)	77.4 (3.6)
Continuous Prompt	Soft Prompt Tuning	54.7 (10.9)	49.7 (0.2)	36.13 (14.6)	33.2 (0.0)	51.6 (0.9)
	Black-Box Tuning	52.6 (0.9)	48.8 (0.6)	46.58 (1.3)	42.9 (2.0)	61.6 (0.9)
Discrete Prompt	Manual Prompt	51.6	50.8	31.11	51.7	67.4
	In-Context Demo.	60.4 (0.7)	53.8 (0.4)	47.11 (1.4)	53.4 (1.5)	45.8 (0.8)
Discrete Prompt	TEMPERA (ours)	60.3 (2.2)	57.4 (1.5)	56.4 (3.2)	45.2 (2.0)	74.0 (1.0)

C.2 Network Architecture

We follow the GPT [18] architecture and use the encoder layer for our policy network. Note that our policy and baseline network shares the same attention-based encoder. The attention is flat over all the possible candidate examples. We use a 3-layer encoder block with 3 heads and 48 latent dimension. We build two different head with 2-layer MLP for each as the policy head and baseline head. We also don't use dropout for the policy learning part. We found this boost up the performance.

C.3 Additional Experiments

We perform additional experiments on some more tasks like RTE, QNLI, SNLI, MNLI and MRPC. Results show that we are consistently better than most of the discrete prompt optimization methods and continuous prompt tuning methods. On several tasks, we are also better than finetuning the entire model.

C.4 Natural Instructions and Promptsources

We provide all the instructions we used in our experiments from Natural Instructions. Here we just provide a few examples. Please refer to the github for all the instruction they provided. We also provide all the verbalizers we used in our experiments from Promptsources. Here we only provide a few examples. Please also refer to their github for the full verbalization.

C.5 Dataset Detail

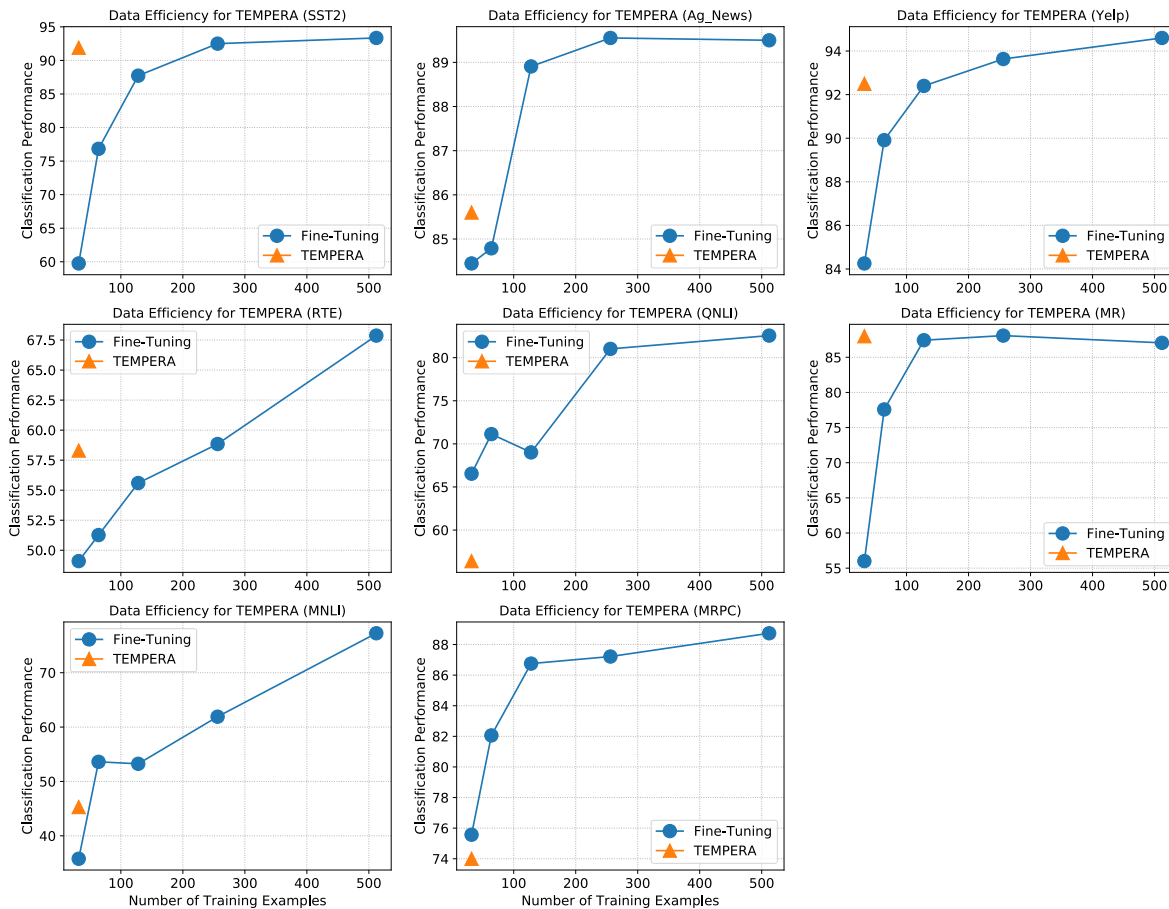


Figure C.1: **Data Efficiency for TEMPERA:** We plot all the finetuning performance for 8 tasks we tested. We see that TEMPERA often achieves the better few-shot performance except for MRPC and QNLI.

For the Finetuning, we use standard finetuning of the RoBERTa model from huggingface for 100 epochs, a learning rate of 0.0003 and the optimizer of Adam.

C.6 Comparison of Different Method

We compare the different property of different prompting methods in this section in order to give a better understanding of different algorithms.

	Frozen LM	Gradient-Free	Guided-Optimization	Interpretable	Query-Dependent
Fine-Tuning	✗	✗	✓	✓	✗
Manual Prompt	✓	✓	✗	✓	✗
Instructions	✓	✓	✗	✓	✗
In-Context Demonstration	✓	✓	✗	✓	✗
Soft Prompt Tuning	✓	✗	✓	✗	✗
Discrete Prompt Search	✓	✓	✗	✓	✗
AutoPrompt	✓		✓	✗	✗
RLPrompt	✓	✓	✓	✗	✗
Tempera (ours)	✓	✓	✓	✓	✓

Figure C.2: **Comparison of Different Prompting Methods:** We compare the different property of different algorithms. We can see that TEMPERA is gradient-free, the resulting prompt is interpretable and query-dependent.

Table C.3: Natural instructions used for TEMPERA in all the tasks.

Task	Natural Instructions
SST-2	“In this task, you are given sentences from movie reviews. The task is to classify a sentence as “great” if the sentiment of the sentence is positive or as “terrible” if the sentiment of the sentence is negative.”
AG News	“Classify the news articles into the categories of World, Sports, Business, and Technology.”
CR	“In this task, you are given sentences from customer reviews. The task is to classify a sentence as “great” if the sentiment of the sentence is positive or as “terrible” if the sentiment of the sentence is negative.”
MR	“In this task, you are given sentences from movie reviews. The task is to classify a sentence as “great” if the sentiment of the sentence is positive or as “terrible” if the sentiment of the sentence is negative.”
Yelp	“In this task, you are given sentences from Yelp reviews. The task is to classify a sentence as “great” if the sentiment of the sentence is positive or as “terrible” if the sentiment of the sentence is negative.”
RTE	N/A
SNLI	“In this task, you’re given a pair of sentences, sentence 1 and sentence 2. Your job is to choose whether the two sentences clearly agree (entailment)/disagree (contradiction) with each other, or if this cannot be determined (neutral). Your answer must be in the form of the letters Yes, Maybe, and No respectively.”
QNLI	“You are given two sentences(Sentence1 and Sentence2). Answer “yes” if these sentences are a paraphrase of one another, otherwise answer “no”.”
MNLI	“In this task, you’re given a pair of sentences, sentence 1 and sentence 2. Your job is to choose whether the two sentences clearly agree (entailment)/disagree (contradiction) with each other, or if this cannot be determined (neutral). Your answer must be in the form of the letters Yes, Maybe, and No respectively.”

Table C.4: Verbalizers used for TEMPERA in all the tasks.

Task	Natural Instructions
SST-2	‘Someone just said to me “{{sentence}}”. Do you think they are {{“sad”}} or {{“happy”}}? {{ answer_choices[label]}}’
AG News	‘What label best describes this news article? {{text}} {{answer_choices[label]}}’
CR	‘Someone just said to me “{{sentence}}”. Do you think they are {{“sad”}} or {{“happy”}}? {{ answer_choices[label]}}’
MR	‘{{text}} Did the reviewer find this movie {{“good or bad”}}? {{ answer_choices[label] }}’
Yelp	‘{{ text }} Overall, the experience is {{ answer_choices[label] }}’
RTE	‘Does the claim “{{sentence2}}” follow from the fact that “{{sentence1}}”? Please answer either {{“yes”}} or {{“no”}}. {{answer_choices[label]}}’
SNLI	‘Suppose {{premise}} Can we infer that “{{hypothesis}}”? Yes, no, or maybe? {{ answer_choices[label] }}’
QNLI	‘{{sentence}} Does that sentence have all you need to answer the question “{{question}}”? {{answer_choices[label]}}’
MNLI	‘Suppose {{premise}} Can we infer that “{{hypothesis}}”? Yes, no, or maybe? {{ answer_choices[label] }}’
MRPC	‘Does the sentence {{sentence1}} paraphrase (that is, mean the same thing as) this sentence? {{sentence2}} {{ answer_choices[label] }}’

Table C.5: Scaling results for TEMPERA in 512 training data per class. Results show that TEMPERA also scales and achieves better results comparing to finetuning.

		SST2	MR	AG News	RTE
Finetuning	Finetuning (few-shot)	93.4	87.0	89.5	67.9
Discrete Prompt	TEMPERA (ours)	93.8	88.6	88.6	71.4

Table C.6: Details for the dataset including the type, size of training, evaluation and test. Note that here all the sizes are few-shot dataset.

Dataset	Type	C	Train = Dev	Test
SST2	Sentiment	2	32	1.8k
AG News	topic	4	64	7.6k
CR	Sentiment	2	32	2k
MR	Sentiment	2	32	2k
Yelp	Sentiment	2	32	38k
RTE	NLI	2	32	0.3k
SNLI	NLI	3	48	10k
QNLI	NLI	3	48	9.8k
MNLI	NLI	3	48	9.8k

Appendix D

Gorilla

D.1 Dataset Details

Our dataset is multi-faceted, comprising three distinct domains: Torch Hub, Tensor Hub, and HuggingFace. Each entry within this dataset is rich in detail, carrying critical pieces of information that further illuminate the nature of the data. Delving deeper into the specifics of each domain, Torch Hub provides 95 APIs. The second domain, Tensor Hub, is more expansive with a total of 696 APIs. Finally, the most extensive of them all, HuggingFace, comprises 925 APIs.

To enhance the value and utility of our dataset, we've undertaken an additional initiative. With each API, we have generated a set of 10 unique instructions. These instructions, carefully crafted and meticulously tailored, serve as a guide for both training and evaluation. This initiative ensures that every API is not just represented in our dataset, but is also comprehensively understood and effectively utilizable.

In essence, our dataset is more than just a collection of APIs across three domains. It is a comprehensive resource, carefully structured and enriched with added layers of guidance and evaluation parameters.

Domain Classification The unique domain names encompassed within our dataset are illustrated in Figure [D.1](#). The dataset consists of three sources with a diverse range of domains: Torch Hub houses 6 domains, Tensor Hub accommodates a much broader selection with 57 domains, while HuggingFace incorporates 37 domains. To exemplify the structure and nature of our dataset, we invite you to refer to the domain names represented in Figure [D.3](#).

API Call Task In this task, we test the model's capability to generate a single line of code, either in a zero-shot fashion or by leveraging an API reference. Primarily designed for evaluation purposes, this task effectively gauges the model's proficiency in identifying and utilizing the appropriate API call.

API Provider Component This facet relates to the provision of the programming language. In this context, the API provider plays a vital role as it serves as a foundation upon which APIs are built and executed.

Explanation Element This component offers valuable insights into the rationale behind the usage of a particular API, detailing how it aligns with the prescribed requirements. Furthermore, when certain constraints are imposed, this segment also incorporates those limitations. Thus, the explanation element serves a dual purpose, offering a deep understanding of API selection, as well as the constraints that might influence such a selection. This balanced approach ensures a comprehensive understanding of the API usage within the given context.

Code Example code for accomplishing the task. We de-prioritize this as we haven't tested the execution result of the code. We leave this for future works, but make this data available in-case others want to build on it.

D.2 Gorilla Details

We provide all the training details for Gorilla in this section. This includes how we divide up the training, evaluation dataset, training hyperparameters for Gorilla.

Data For HuggingFace, we devise the entire dataset into 90% training and 10% evaluation. For Torch Hub and Tensor Hub, we devise the data in to 80% training and 20% testing.

Training We train Gorilla for 5 epochs with the 2e-5 learning rate with cosine decay. The details are provide in Tab. [D.1](#). We finetune it on 8xA100 with 40G memory each.

Table D.1: Hyperparameters for training Gorilla

Hyperparameter Name	Value
learning rate	2e-5
batch size	64
epochs	5
warmup ratio	0.03
weight decay	0
max seq length	2048

Torch Hub domain names: Classification, Semantic Segmentation, Object Detection, Audio Separation, Video Classification, Text-to-Speech

Tensor Hub domain names: text-sequence-alignment, text-embedding, text-language-model, text-preprocessing, text-classification, text-generation, text-question-answering, text-retrieval-question-answering, text-segmentation, text-to-mel, image-classification, image-feature-vector, image-object-detection, image-segmentation, image-generator, image-pose-detection, image-rnn-agent, image-augmentation, image-classifier, image-style-transfer, image-aesthetic-quality, image-depth-estimation, image-super-resolution, image-deblurring, image-extrapolation, image-text-recognition, image-dehazing, image-deraining, image-enhancement, image-classification-logits, image-frame-interpolation, image-text-detection, image-denoising, image-others, video-classification, video-feature-extraction, video-generation, video-audio-text, video-text, audio-embedding, audio-event-classification, audio-command-detection, audio-paralinguists-classification, audio-speech-to-text, audio-speech-synthesis, audio-synthesis, audio-pitch-extraction

HuggingFace domain names: Multimodal Feature Extraction, Multimodal Text-to-Image, Multimodal Image-to-Text, Multimodal Text-to-Video, Multimodal Visual Question Answering, Multimodal Document Question Answer, Multimodal Graph Machine Learning, Computer Vision Depth Estimation, Computer Vision Image Classification, Computer Vision Object Detection, Computer Vision Image Segmentation, Computer Vision Image-to-Image, Computer Vision Unconditional Image Generation, Computer Vision Video Classification, Computer Vision Zero-Shot Image Classification, Natural Language Processing Text Classification, Natural Language Processing Token Classification, Natural Language Processing Table Question Answering, Natural Language Processing Question Answering, Natural Language Processing Zero-Shot Classification, Natural Language Processing Translation, Natural Language Processing Summarization, Natural Language Processing Conversational, Natural Language Processing Text Generation, Natural Language Processing Fill-Mask, Natural Language Processing Text2Text Generation, Natural Language Processing Sentence Similarity, Audio Text-to-Speech, Audio Automatic Speech Recognition, Audio Audio-to-Audio, Audio Audio Classification, Audio Voice Activity Detection, Tabular Tabular Classification, Tabular Tabular Regression, Reinforcement Learning Reinforcement Learning, Reinforcement Learning Robotics

Figure D.1: **Domain names:** Domain names with the three dataset. Tensor Hub is the smallest dataset while the other two hubs contain many more models.

```

### User: Our customer is a zoo and we want to help them detect movement of
different animals. Write a python program in 1 to 2 lines to call API in TensorFlowHub.
### Assistant:
domain: image-object-detection,
api_call: detector = hub.load('https://tfhub.dev/google/openimages_v4
/ssd_mobilenet_v2/1'), api_provider: TensorFlow Hub,
explanation: This API call loads the pre-trained Single Shot Detector (SSD) Mo-
bileNet V2 model from TensorFlow Hub, which has been trained on the Open Images
v4 dataset. This model is used for object detection in images. Once loaded, you can
pass an image (and an optional score threshold for detection confidence) to the detec-
tor to get back the detected objects along with their bounding boxes, class labels, and
associated confidence scores.
code: import tensorflow as tf; import tensorflow_hub as
hub; # Load the SSD MobileNet V2 model; detector =
hub.load('https://tfhub.dev/google/openimages_v4/ssd_mobilenet_v2/1'); # Use
the model to detect objects in an image (assuming 'image' is a TensorFlow tensor
representing your image); result = detector(image, score_threshold=0.5)

```

Figure D.2: **Example of the Dataset:** Two examples of the dataset, the above one is zero-shot (without information retrievers) and the bottom one is with information retriever.

D.3 Performance Comparison

We provide a full comparison of each model’s performance in this section. In Fig [D.5](#) and Fig. [D.6](#), the full set of comparisons is provided. We see that especially in zero-shot case, Gorilla surpasses the GPT-4 and GPT-3.5 by a large margin. The GPT-4 and GPT-3.5 gets around 40% accuracy in Torch Hub and Tensor Hub, which are two structured API calls. Compared to that, HuggingFace is a more flexible and diverse Hub, as a result, the performance on HuggingFace is not as competitive.

Evaluation

For ease of evaluation, we manually cleaned up the dataset to make sure each API call domain only contains the valid call in the form of:

```
API_name(API_arg1, API_arg2, ..., API_argk)
```

Our framework allows the user to define any combination of the arguments to check. For Torch Hub, we check for the API name `torch.hub.load` with arguments `repo_or_dir` and

User: I am an engineer at Uber and I need to find a API that can classify pedestrians, cars, etc. from an image of the scene. Write a python program in 1 to 2 lines to call API in TorchHub.

Reference API: domain: Object Detection, framework: PyTorch, functionality: Traffic Object Detection, Drivable Area Segmentation, Lane Detection, api_name: HybridNets, api_call: torch.hub.load(repo_or_dir='datvuthanh/hybridnets', model='hybridnets', pretrained=True), api_arguments: repo_or_dir, model, pretrained, python_environment_requirements: Python=3.7, PyTorch=1.10, example_code: 'import torch; model = torch.hub.load('datvuthanh/hybridnets', 'hybridnets', pretrained=True); img = torch.randn(1,3,640,384); features, regression, classification, anchors, segmentation = model(img), performance: dataset: [name: BDD100K, accuracy: Traffic Object Detection: Recall: 92.8, mAP0.5: 77.3, Drivable Area Segmentation: Drivable mIoU: 90.5, Lane Line Detection: Accuracy: 85.4, Lane Line IoU: 31.6], description: HybridNets is an end2end perception network for multi-tasks. Our work focused on traffic object detection, drivable area segmentation and lane detection. HybridNets can run real-time on embedded systems, and obtains SOTA Object Detection, Lane Detection on BDD100K Dataset.

Assistant:

domain: Object Detection, **api_call:** model = torch.hub.load('datvuthanh/hybridnets', 'hybridnets', pretrained=True),

api_provider: PyTorch,

explanation: Load the pretrained HybridNets model for object detection from PyTorch Hub to classify pedestrians, cars, etc. in an image of the scene,

code: import torch; model = torch.hub.load('datvuthanh/hybridnets', 'hybridnets', pretrained=True)

Figure D.3: **Example of the Dataset:** Two examples of the dataset, the above one is zero-shot (without information retrievers) and the bottom one is with information retriever.


```
generate_video = pipeline("text-to-video", model="your_model_name")
```

```
vqa = pipeline("visual-question-answering", model="microsoft/clip-vqa-base",
tokenizer="microsoft/clip-vqa-base")
```

```
depth_estimator = pipeline("depth-estimation", model="intel-isl/MiDaS",
tokenizer="intel-isl/MiDaS")
```

Figure D.4: **Hallucination Examples:** GPT-4 incurs serious hallucination errors in HuggingFace. We show a couple of examples in the figure.

`model`. For Tensor Hub, we check API name `hub.KerasLayer` and `hub.load` with argument `handle`. For HuggingFace, since there are many API function names, we don't list all of them here. One specific note is that we require the `pretrained_model_name_or_path` argument for all the calls except for `pipeline`. For `pipeline`, we don't require the `pretrained_model_name_or_path` argument since it automatically select a model for you once `task` is specified.

Hallucination

We found especially in HuggingFace, the GPT-4 model incurs serious hallucination problems. It would sometimes put a GitHub name that is not associated with the HuggingFace repository in to the domain of `pretrained_model_name_or_path`. Fig. [D.4](#) demonstrates some examples and we also observe that GPT-4 sometimes assumes the user have a local path to the model like `your_model_name`. This is greatly reduced by Gorilla as we see the hallucination error comparison in Tab. [5.1](#).

AST as a Hallucination Metric

We evaluated the generated results on 100 LLM generations (randomly chosen from our eval set). The accuracy using AST subtree matching is 78%, consistent with human evaluation with 78% accuracy in calling the right API. All the generations that AST flagged as incorrect, were the same ones that were manually also flagged as incorrect. Additionally, Gorilla generates supporting code to call the API which includes installing dependencies (e.g., `pip install transformers[sentencepiece]`), environment variables, etc. When we manually attempted to execute end-to-end code, 72% of all codes generated were executed successfully. It's worth noting that the 6% discrepancy were NOT semantic errors, but errors that arose due to factors external to the API in the supporting code - we have included an example to illustrate this further. Considering the significant time and effort required for manual

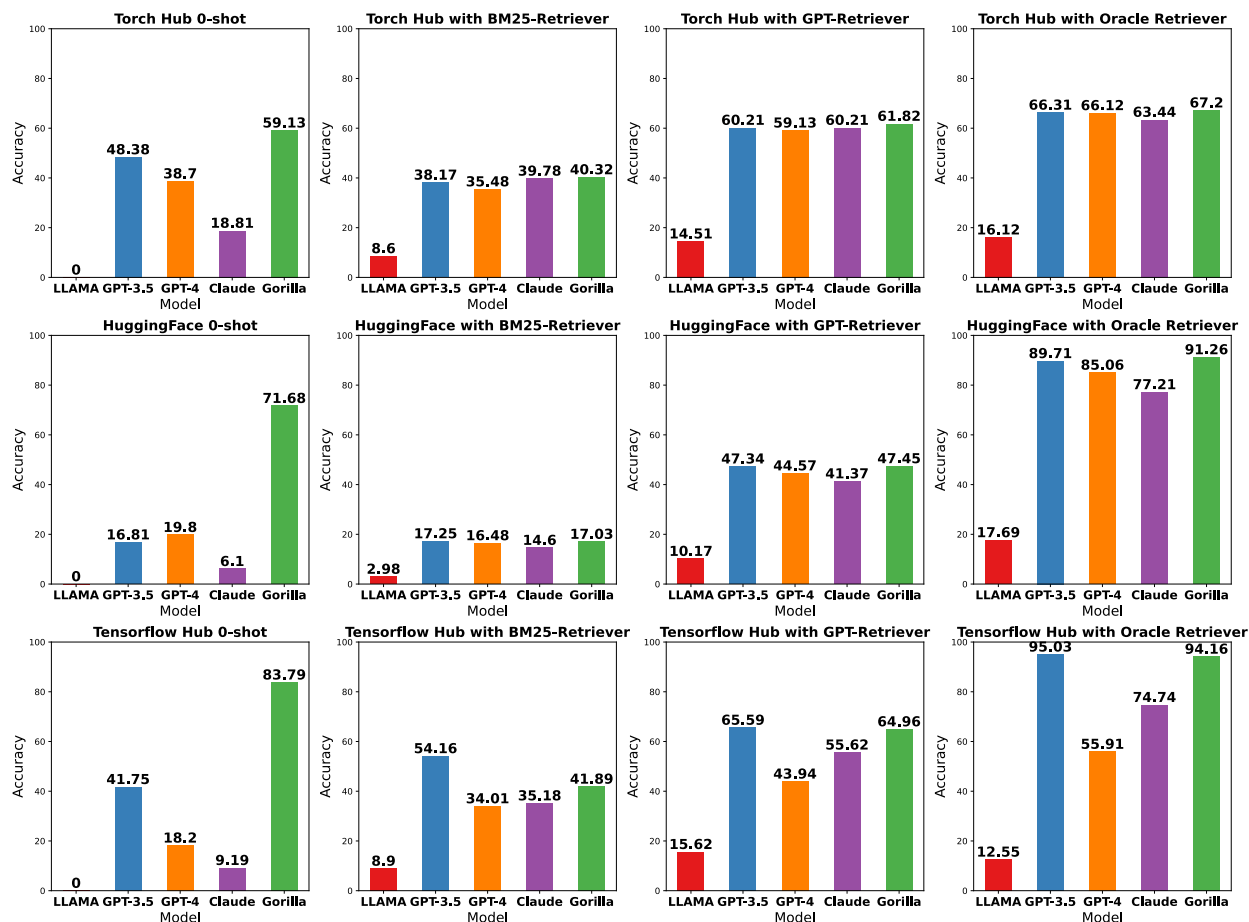


Figure D.5: **Performance:** We plot each model’s performance on different configurations. We see that Gorilla performs extremely well in the zero-shot setting. While even when the oracle answer is given, Gorilla is still the best.

validation of each generation, our data further reinforces our belief in the efficiency of using AST as a robust offline metric.

Here is a representative example, where we are able to load the correct model API. However, in the supporting code, after we have the output from the API, the `zip()` function tries to combine sentiments and scores together. However, since scores is a float, it’s not iterable. `zip()` expects both its arguments to be iterable, resulting in an `'float' object is not iterable` error.

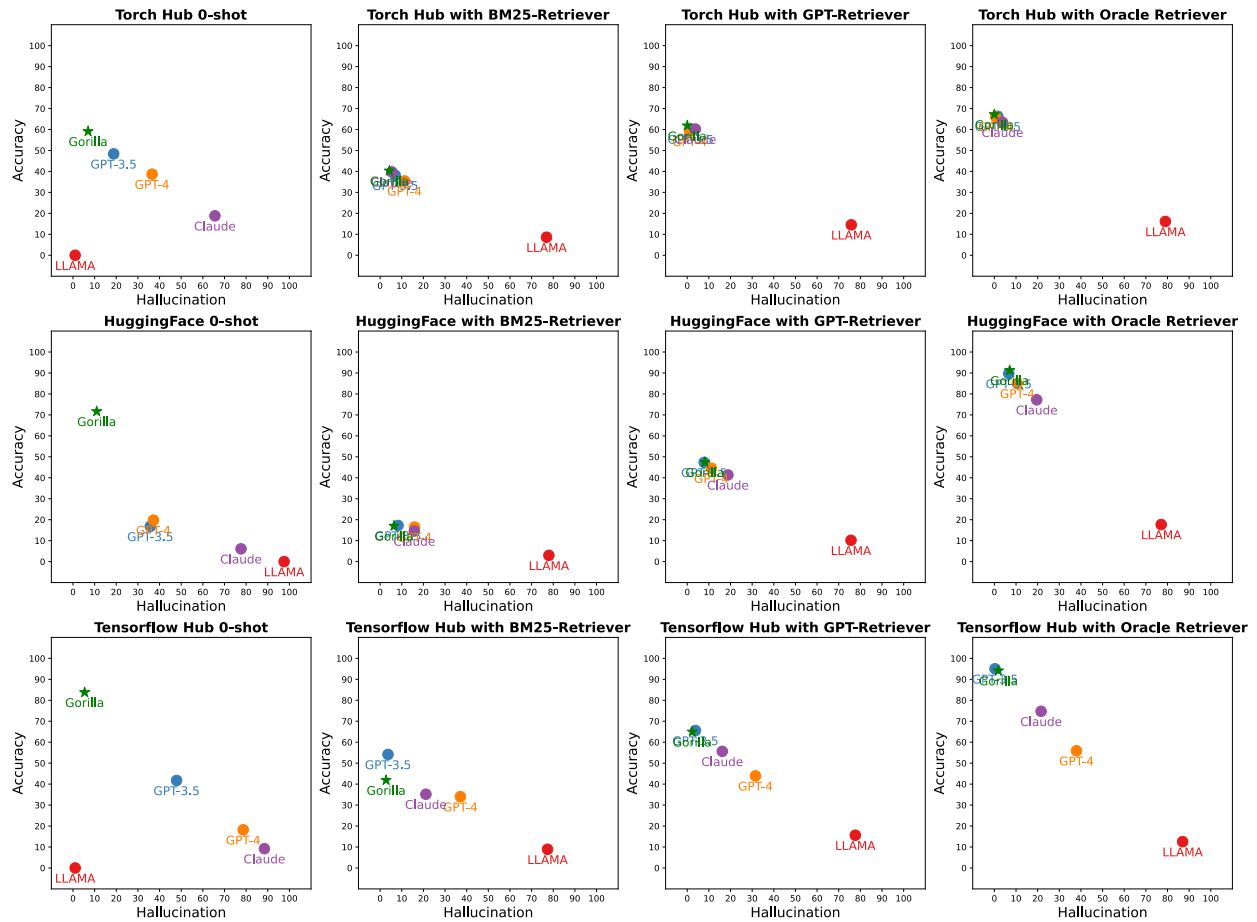


Figure D.6: **Accuracy vs Hallucination:** We plot each model’s performance on different configurations. We found that in the zero-shot setting, Groilla has the most accuracy gain while maintaining good factual capability. When prompting with different retrievers, Gorilla is still capable to avoid the hallucination errors.

```
from transformers import pipeline

def load_model():
    classifier = pipeline('sentiment-analysis',
        model='nlptown/bert-base-multilingual-uncased-sentiment')
    return classifier

def process_data(comments, classifier):
    response = classifier(comments)
    sentiments = response[0]['label'].split()
    scores = response[0]['score']
    result = [{'sentiment': sentiment, 'score': score}
        for sentiment, score in zip(sentiments, scores)]
    return result

comments = "These comments are about our news website."
# Load the model
classifier = load_model()
# Process the data
response = process_data(comments, classifier)
print(response)
```

Figure D.7: The API calls that the Gorilla model is trained on is able to call the right API, but the supporting `zip()` code has a bug.