

PromptSet: A Programmer’s Prompting Dataset

Kaiser Pister*

kaiser@cs.wisc.edu

University of Wisconsin-Madison

Madison, USA

Patrick Brophy

University of Wisconsin-Madison

Madison, USA

Dhruba Jyoti Paul

University of Wisconsin-Madison

Madison, USA

Ishan Joshi

University of Wisconsin-Madison

Madison, USA

ABSTRACT

The rise of capabilities expressed by large language models has been quickly followed by the integration of the same complex systems into application level logic. Algorithms, programs, systems, and companies are built around structured prompting to black box models where the majority of the design and implementation lies in capturing and quantifying the ‘agent mode’. The standard way to shape a closed language model is to prime it for a specific task with a tailored prompt, often initially handwritten by a human. The textual prompts co-evolve with the codebase, taking shape over the course of project life as artifacts which must be reviewed and maintained, just as the traditional code files might be. Unlike traditional code, we find that prompts do not receive effective static testing and linting to prevent runtime issues. In this work, we present a novel dataset called PromptSet, with more than 61,000 unique developer prompts used in open source Python programs. We perform analysis on this dataset and introduce the notion of a static linter for prompts. Released with this publication is a HuggingFace dataset and a Github repository to recreate collection and processing efforts, both under the name pisterlabs/promptset.

CCS CONCEPTS

- Computing methodologies → Natural language generation

KEYWORDS

Prompt Management, Large Language Models, Dataset, Information systems, Ethnography, Taxonomy

ACM Reference Format:

Kaiser Pister, Dhruba Jyoti Paul, Patrick Brophy, and Ishan Joshi. 2024. PromptSet: A Programmer’s Prompting Dataset. In *2024 International Workshop on Large Language Models for Code (LLM4Code ’24)*, April 20, 2024, Lisbon, Portugal. ACM, New York, NY, USA, 8 pages. <https://doi.org/10.1145/3643795.3648395>

*Also with Pister Labs.

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. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

LLM4Code ’24, April 20, 2024, Lisbon, Portugal

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0579-3/24/04

<https://doi.org/10.1145/3643795.3648395>

1 INTRODUCTION

As large language models (LLMs) become more effective, more software is written to control and capture their potential. For any given task, a developer will have three common options to improve LLM performance: improve the foundational model, finetune a foundational model towards a specific task, or use a task tailored prompt. Major improvements across many benchmarks can be achieved by retraining the base model from scratch, however outside of large research labs, this is an intractable task [14, 21, 28]. Finetuning can function as a cheaper alternative to retraining from scratch, but can require a modest sized dataset to achieve meaningful results. Additionally, it leads to catastrophic forgetting and complex model management when working with multiple tasks [11, 27]. Prompting can serve as a cheap, efficient way to control an LLM without significant investment in infrastructure or data [4, 36]. Since the introduction of few-shot learning, prompting has become a necessary feature of working with LLMs, and despite the proliferation of research into prompt engineering, in-context-learning and similar fields, there has been relatively little exploration of prompt management [24, 38]. What’s more, there exists no standard methodology for working with prompts, leading to incoherent collections of files, folders, JSON objects, and code strings inhibiting readability, reusability, and maintainability [15].

With LLMs moving from research into production, it is ever more important to treat the prompts which control business logic as first class citizens in the CI/CD pipelines of traditional software engineering. There are many works focusing on assessing and optimizing the effectiveness of a prompt, once it is in your system (either automatically or manually), but these rely on runtime computation against an LLM backbone and do not provide any static guarantees [18, 19, 35, 37].

In this work, we propose static analysis passes to add to an existing CI/CD pipeline to preemptively detect non-traditional errors in prompts, such as misuse of variable formatting, typo detection, and input sanitization. In order to motivate these passes, we introduce a suite of extraction techniques to parse prompts from code files and a novel dataset, PromptSet, of more than 61,000 unique developer prompts used in open source Python programs. Finally, we are the first to our knowledge to discuss unit testing in the context of prompts as code.

2 BACKGROUND

We abstract the language model to its functional purpose of decoding tokens based on an input sequence of tokens [10]. The internal

mechanisms are independent of the work presented here. For an in-depth understanding of the aforementioned architecture, we direct readers toward the existing literature on this subject. [5, 22, 31].

Following the discovery of zero-shot and few-shot abilities of LLMs, a cottage industry of observation tools has developed in the market, allowing programmers to adequately monitor and experiment with open access language models [1, 29]. These tools give users the ability to detect anomalies in their agent systems, affording reliability in a design space that is notoriously unreliable. In parallel, quality is afforded by a large array of tools for optimizing the performance of individual prompts. Online courses for prompt engineering as a discipline have cropped up adjacent to Github repositories of "awesome-prompts" to improve a developer's manual prompting ability [6, 33]. Although the topic is often criticised as a pseudo-science, common tricks and minor variations have demonstrably large effects on the quality of results [2]. Automatic optimization tools leverage an LLM as a self-optimizer across an established dataset to improve prompt quality [35].

In traditional software engineering reliability and quality are checked regularly through the use of continuous integration and delivery (CI/CD) systems. CI/CD pipelines can manage many aspects of a code base, such as formatting, linting, and unit testing. Regardless of the implementation, these systems are responsible for reducing the number of issues deployed to production systems. Formatting keeps code consistent, simplifying reviews between team members. Linting uses static analysis to keep a code base clean from common problems. Unit testing guarantees run-time assumptions at a granular level. As development teams scale to non-trivial sizes, CI/CD becomes one of the key factors in maintaining feature velocity [13].

Primarily due to the infancy of the field of prompting, we find the CI/CD ecosystem for prompting wanting. Formatting for prompts resides at the level of the developer's editor and is agnostic to the purpose of the prompt. Often a prompt exists alongside the rest of the code as a simple string. There does not exist any form of linting for prompts, even bugs as simple as a mismatched variable interpolation won't be detected until run-time.

Functionality (integration) testing of prompts is quite popular, and there are many tools that support this behavior [25]. We distinguish between the effectiveness of a prompt (its evaluation results on a dataset) and the characteristics of the prompt. For example, a unit test of a prompt might be an assertion that the prompt does not attempt to interpolate a string into an integer formatted slot (e.g. "Num: {:02d}".format("x")). A bug of this class could be detected by a static analysis pass, similar to any traditional type error.

2.1 Taxonomies of Prompts

Other works have looked to build a prescriptive taxonomy of effective prompting techniques [9, 33]. We leverage these works to categorize and classify our prompt dataset, but find that a large majority of our prompts do not fit neatly into one or more of these categories. We take an unsupervised, descriptive approach with our classifications, and leave further category refinement to future work.

2.2 Other Prompting Datasets

At Mining Software Repositories '24, there was a code mining challenge addressing a similar task of understanding the purpose of prompts sent to DevGPT, a chat bot built for talking to code repositories on Github [34]. In contrast to the prompts we find in our dataset, DevGPT presents conversational text, similar to a user talking on ChatGPT. PromptSet represents programmatic prompts that often dictate application level logic. These prompts are not designed for chatting with a user, except occasionally by way of a programmed bot.

3 METHODOLOGY

Towards the goal of developing meaningful static analysis passes, we first perform an ethnographic survey of how developers build applications with open LLM SDKs. The code survey is intended to validate our hypotheses and motivate the development of targeted linting rules or unit tests. We make PromptSet openly available to facilitate future static analysis from the community.

Our prompt dataset originates from open source code hosted on Github. We scrape Python code files which fulfill a simple criteria of using a language model library; see appendix A for the specific query string. We search for usage of the openai, anthropic, cohere, and langchain libraries, as these are regularly cited as the most popular tools in online resources. We ignore forked repositories to reduce duplication. Notably, we exclude the transformers library which would include text prompts, but we find it to have too many false positives due to its other use cases. Figure 1 shows an example code file which has been flagged for processing.

3.1 Extracting Prompts

We utilize Tree-sitter for building abstract syntax tree (AST) representations of each scraped file and then query against the tree with specific patterns based on API design specifications and commonly observed behavior [30].

We expect our dataset to be a subset of the entire set of prompts in these files as discussed in section 5.1.

(1) OpenAI & Anthropic API calls:

Both OpenAI (beta, v1) and Anthropic have similar API design specifications. The SDKs expose .create functions on the .completions endpoint. OpenAI additionally exposes a .chat variant, which is captured with the same pattern. From the method calls, we extract specific argument values for positionally passed arguments, as well as the keyword arguments prompt and messages. Common additional arguments are tracked and displayed in Table 3.

(2) Cohere API calls:

The Cohere SDK exposes multiple methods for handling text input. The two main entry points are .chat and .summarize. We pattern match on these calls, and follow the same methodology as above for extracting the arguments.

(3) LangChain PromptTemplate and Message classes:

LangChain introduces multiple ways to create prompts, in particular a PromptTemplate class and a Message (HumanMessage, AIMessage) class, which support completion and chat endpoints respectively. These are base classes that developers can extend with their own functionality. We

```

from openai import OpenAI
import pandas as pd, json

# OPEN API KEY
KEY = "<API-KEY>"
SYSTEM_PROMPT = """
Extract context-well-explaining keywords from Korean text given, up to 15 in amount in json format.
Extracted Keywords that don't fit into them should be avoided.
Print nothing except json (Format : [ "Keyword", "Keyword", .. "Keyword" ])
Korean Text: \"\"\""
"""

client = OpenAI(api_key=KEY)

def extractKeywords(text: str) -> list[str]:
    print(f"OpenAPI Request : {text}")
    response = client.chat.completions.create(
        model="gpt-3.5-turbo-1106",
        messages=[{"role": "system", "content": f"{SYSTEM_PROMPT.format(text)}"}],
        temperature=0.5,
        max_tokens=1000,
    )
    print(response.choices[0].message.content)
    result = json.loads(response.choices[0].message.content)
    return result

```

The diagram shows the code with several annotations:

- A red box highlights the first two lines: `from openai import OpenAI` and `import pandas as pd, json`. An arrow points from this box to the text "Flagged as a potential code file for investigation".
- A red box highlights the `SYSTEM_PROMPT` variable definition. An arrow points from this box to the text "Likely site for using a prompt".
- A red box highlights the `response = client.chat.completions.create()` call. An arrow points from this box to the text "Likely prompt variable".

Figure 1: Example code file

search for constructor calls of classes which contain `*Prompt` or `*Message` in their name, and extract the arguments passed to the initialize function.

(4) LangChain Tools:

LangChain exposes a Python decorator for automatically converting a function into an LLM tool using the function's docstring and type hints as the tool's description. Similarly, they allow for extending a `BaseTool` class. We match on `@tool` decorators and `BaseTool` super classes to detect these use cases.

(5) `PromptTemplate.from_file`

LangChain allows users to save prompts to standard text files, and load them at run time for processing. The `from_file` function accomplishes this task, and we match directly on any use of `*Template.from_file`.

(6) Prompt and Template variables

From our observations, we see that the vast majority of variables used as prompts are named with the key phrase `prompt` or `template`. We flag any variable declarations matching this

pattern, but note that this could be a noisy heuristic. In practice it proves to match the quality of the other extraction techniques.¹

(7) "content" in dictionaries

The basic structure for many chat messaging templates is using a Python dictionary with the keys `role` and `content`. The `role` entry typically contains "user", "system" or "assistant". The `content` entry contains the message that will be sent to LLM. We directly search for dictionaries with the specific key matching "content" to extract these prompts.

(8) DevGPT Conversation Prompts

Finally, we add the prompts from the recent DevGPT MSR challenge to compare and contrast against PromptSet. These prompts are different in nature, and we keep them separate through our evaluation.

3.2 Post-Processing

After extracting the key regions of the AST from each file, we use the black formatter to consistently format the inputs for better

¹We found the message name was overloaded too often to be of much use.

readability and deduplication [8]. Finally, we use tree-sitter again to extract strings, identifiers, and interpolations per prompt.

3.3 Increasing Yield

After performing a first iteration through these heuristics, we reviewed the extracted prompts as well as the files which failed to find any prompts and refined the extraction process. In particular, we added a naive β -substitution preprocessing step to replace constant expressions with their respective values throughout the file. As we process a file in search of prompt patterns, we track constant variable declarations by looking for static single assignment to variable names in the same file. Furthermore, we construct a simple string-evaluation language for processing string operations such as concatenation and interpolation across the static variable set. See listing 1 for an example.

We note that this string interpreter is sound but not complete, and cannot process all Python string manipulations. Further exploration of partial execution models could be leveraged to increase the yield of prompts, but we leave this to future work.

```

1 import cohere # file is flagged for processing
2
3 co = cohere.Client()
4 pre = "You are an agent working at the check-in desk."
5 query = pre + " User said: {history}"
6 co.generate(query) # flag `query`
7
8 # compute: query [pre:= "You are...", history: <free>]
9 # query := "You are an ... User said: PLACEHOLDER"

```

Listing 1: naive β -substitution

3.4 Discarded Heuristics

We consider a multi-line string heuristic but upon review found too many false positives given the correlation between the streamlit library, docstrings, and LLM development. We consider a sequence classifier to detect prompts but discard it for similar reasons.

4 RESULTS

We describe PromptSet in two parts. First we provide a surface overview of the dataset, then we provide specific findings from our analysis. The prompt scraping and extraction was last run on January 10, 2024. By manually searching Github with our queries, we approximate that there are 153,000 code files which match our search criteria. Due to rate limiting, we are able to download 93,142 of these files, so we conclude that our dataset represents 60.7% of the open-source API-based LLM-usage. The 93,142 code files come from 37,944 repositories.

4.1 Dataset Overview

Using the methodology described in section 3.1, we extract 118,862 total prompts from the scraped files, as seen in table 1. The extracted prompts come from 37,112 of the code files (20,598 repositories). The remaining 56,030 files do not contain any prompts matching our extraction criteria, in part due to over scraping from Github and in part due to strict pattern matching. We perform a manual review of 200 code files which report no prompts found and manually tag 36 as false negatives, i.e. these files did contain prompts, but our extraction methodology did not find them. The majority of

Table 1: Prompt Count per Source

Library	Source	Count
OpenAI/Anth.	completions.create	12,420
Cohere	.chat	260
LangChain	@tool	1,425
LangChain	Template/Message class	24,302
LangChain	from_file	21
All	Prompt/Template name	94,897
All	Content Key in dictionary	34,324
DevGPT	Conversations	13,748

Table 2: Unique Prompts

Set	Total Found	Unique	Length > 10	Repositories
PromptSet	118,862	61,448	57,981	20,598
DevGPT	13,748	13,236	13,053	-

Table 3: LLM Call Arguments

Parameter	Most Common ²	2nd	3rd
model	gpt-3.5-turbo	davinci-003	gpt-4
temperature	0	0.7	0.5
top_p	1	0.95	0.5
max_tokens	100	1024	1000

the true negatives are files that only use the semantic embedding functionality of these libraries or provide light wrappers around the library APIs. A quick check against the search terms on Github reveals that the ratio of extracted prompts matches roughly with the distribution between LangChain (most popular), OpenAI (popular), and Cohere (uncommon). Tool usage was introduced relatively recently, so the small count of tools is expected.

In order to perform per-prompt analysis, we join the prompts into a single set for testing. The results of deduplication are shown in table 2. We display length and language distributions of the dataset in figures 2 and 3. Language is detected on prompts of length greater than ten characters using the `ftlangdetect` package [16, 17]. The majority of prompts are written in English, accounting for 84.1% of the strings we extracted. The remaining 15.9% fall between Mandarin, Japanese, Spanish, French, German, and Korean (below 1% are not mentioned).

Briefly, we investigate the distributions of interpolations, and confirm that the most common variables are `chat`, `query`, `input` and similar placeholder input values for conversational AI. There was minimal representation of type-formatting statements in the dataset, (e.g. `"ratio:.2f"`), all of which were floating point formatting.

Finally, we perform a Zipf's law analysis on the input tokens using the `cl100k_base` tokenizer, as that supports the most common models used. From figure 4, we see that the mass is distributed

²Data reported from original November dataset

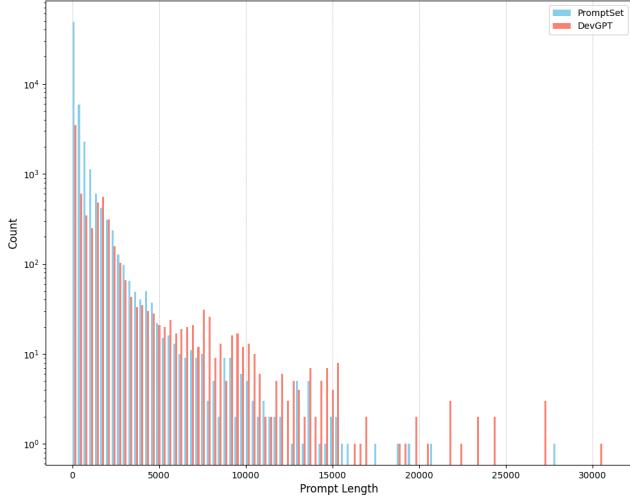


Figure 2: Distribution of prompt lengths in PromptSet.

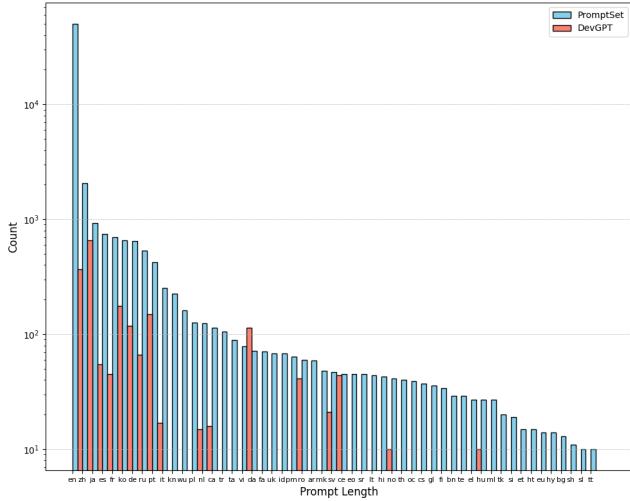


Figure 3: Distribution of languages in PromptSet.

above the ideal line, meaning there is a more even distribution across the token set than in traditional writing.

4.2 Categorization and Clustering

Using the dataset, we begin investigation in multiple directions to better understand the potential use cases of these prompts. To start, we follow the work of White et al. and categorize the prompts, using the six categories laid out in their work [33]. We craft a prompt based on their explanation of the categories and send 2,200 input prompts to the gpt-4-preview-1106 model for prediction across PromptSet and DevGPT (2,000 for PromptSet, 200 for DevGPT). The

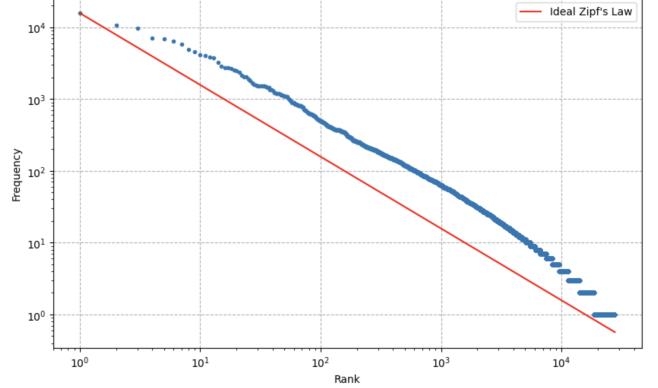


Figure 4: Zipf's law plotted on tokens from PromptSet.

results are shown in figure 5 and an enumeration of the categories with extracted examples is provided in table 4.³

These results show that there is some alignment between the prescribed "good prompting techniques" and the prompting techniques we find in the wild, however there are still large discrepancies. Many users are enacting the category-2, "Output Customization-Persona", beginning their prompts with "Act as a...", to elicit a specific type of response. The distribution of DevGPT prompts is more evenly spread than the distributions of PromptSet which favors categories 1 and 2 (Input and Output). We believe this to be the case because developers require strict control of the input and output to their systems. Only once those are under control can they leverage categories 3-6. On the other hand, in a conversation there are no such restrictions.

Many of the prompts fail to fall into any of the categories, and we suspect this is due to a few contributing factors. First, some of the prompts are partial prompts which might not have a clear category without more context. Second, since the prompts were labeled with an LLM with a meta-prompt that did not undergo any optimization, there is the possibility for error. Third, the taxonomy proposed by White et al. is now ten months old, and the study of prompting has progressed much since its release. It is possible that a new category could emerge, though we do not see a clear trend in the dataset at this point.

In order to perform our own classification, we create a clustering nearest neighbor plot with the semantic meaning of each prompt. To start, we embed each prompt using the all-MiniLM-L6-v2 model from the sentence-transformers library [23]. We fit a t-SNE with 10 clusters to the embedding outputs and show the results in figure 6. Manual inspection of the clusters shows many similarities, and we assign labeled names based on the most common patterns we see.

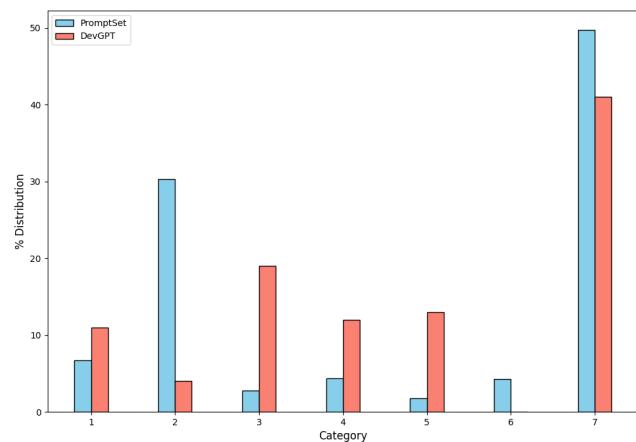
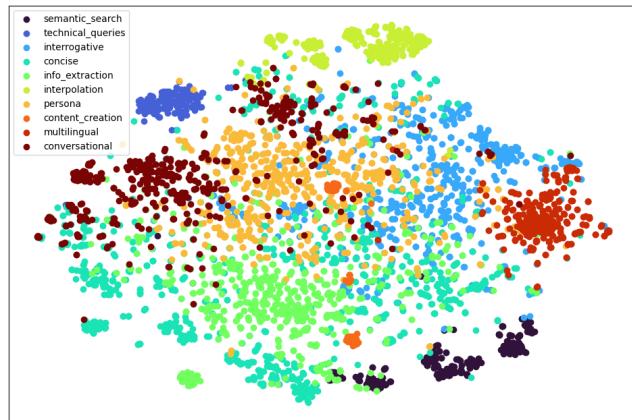
4.3 Technique Propagation

Next we investigate the propagation of research techniques into PromptSet in table 5. We use a few heuristics to derive usage of some of the most popular techniques, such as chain-of-thought, few shot

³Interestingly, only two of the prompts in the 2,000 PromptSet sampled prompts caused breaks in our prompt.

Table 4: Prompt Patterns Per White et al.

Pattern Name	Source	Example ⁴
Input Semantics	jxnl	You are an expert at outputting json. You always output valid JSON based on the pydantic schema given to you.
Output Customization	benczech212	You are a wizard shop owner named {ASSISTANT_NAME}. Only talk on the behalf of {ASSISTANT_NAME}. My name is {USER_NAME}
Error Identification		
Prompt Improvement	Kaastor	Given the following conversation and a follow up question, rephrase the follow up question to be a standalone question.
Interaction	offtian	You are playing the '20 Questions' game with another player. Your role is to answer 'Yes' or 'No' to questions based on a given concept or object.
Context Control	nachollorca	Your task is to answer a question given some context given here, delimited by triple backticks:

**Figure 5: Categorization of PromptSet.****Figure 6: t-SNE of PromptSet.**

prompting, and special tokens [3, 32]. For each of these techniques, we assign a few representative strings to filter on through each of the splits. Chain-of-thought, for example, matches with "step-by-step", "step by step", "let('s think", and "thought(s)". We expect

this to be an underestimate on usage. The doc column represents prompts mentioning "documents" which have a high prevalence in recent product developments.

4.4 Error Investigation

To perform an error investigation, we first look for typos in the prompts. Prompts tend to be natural language requests, typed by hand and as we have observed, can be riddled with typos. We use the cspell package on each prompt and find close to 28,000 spelling errors across the unique English prompts of PromptSet [7]. Many of the spelling errors stem from proper nouns and code related terminology, so we remove capitalized mistakes and words with underscores, reducing the error count to 16,989. To further improve the accuracy two authors independently manually tag 200 of these errors, and find a true positive rate between 33%-40%. If we extrapolate with this rate, there would be a typo in approximately 1 of 8 prompts.

These mistakes are not limited to junior developers. In writing this work, we found typos in our own prompts as well as the academic papers that we have reviewed [26].

Finally, we develop a simple white space detection lint pass, which checks for trailing and leading white space in prompts. The official documentation from OpenAI discusses that trailing white spaces in prompts can lead to poor tokenization which causes a degradation in performance of the model. While a simple `.strip()` can resolve the issue, there is no guarantee that this stripping happens on the server side. Thus white space detection is a perfect problem for a linter to solve. The results in table 6 show that a large portion of the prompts we find do indeed have trailing white spaces, such as newlines, tabs and spaces.

5 DISCUSSION

In this work we introduce a novel dataset with the purpose of better understanding how developers are interacting with the newfound power of integrating LLMs into their applications. PromptSet indeed displays a diversity of ideas and we acknowledge that even this only represents a fraction of the programming prompt usage that exists.

⁴Prompts are abbreviated for space

Table 5: Research Technique Detection

Set	Total	concise	Few Shot	doc	CoT	Code Block	Instruction Block	Scratchpad	Tool use	Special Tokens
PromptSet	57953	176 (0.3)	1008 (1.7)	1939 (3.3)	1095 (1.9)	1696 (2.9)	927 (1.6)	170 (0.3)	168 (0.3)	178 (0.3)
DevSet	13053	2 (0.0)	88 (0.7)	1015 (7.8)	105 (0.8)	730 (5.6)	120 (0.9)	0 (0.0)	319 (2.4)	16 (0.1)

Table 6: Leading & Trailing Whitespace Detection

Set	Total	Trailing (%)	Leading (%)	All (%)
PromptSet	58,814	17,668 (30.0)	9,723 (16.5)	19,681 (33.5)
DevGPT	13,399	2,081 (15.5)	235 (1.8)	2,256 (16.8)

5.1 Limitations

As mentioned in the results, we expect to have approximately 60% public coverage of the libraries we mention, however there are other libraries we did not consider scraping and many closed source repositories that are obviously out of reach. Additionally, we restrict our set to Python, but there is an active JavaScript development community focused on LLM development as well. Beyond hitting API limits to fully search Github, the extraction techniques proposed do not find the full set of prompt strings used in the files mentioned, nor do they extract prompts from adjacent files in the same system (for example prompts that were imported from another file). This means that while PromptSet contains a large number of diverse prompts, it might not reflect the true distribution and characteristics of prompts. Consequently, our dataset is not exhaustive and may not include all relevant data points, a factor that must be considered when interpreting the findings, as it could lead to gaps that affect the overall results.

5.2 Future Work

Despite the potential benefit to readability, we cannot quantify that typos or prompt mistakes are bad for every possible task [20]. Instead we posit that in the majority of the cases, a developer would like to actively make a choice in handling the likely mistakes of typos.

A good unit test for a prompt might test that the prompt follows the project guidelines on appropriate wording, or asserts that the prompt does not allow for injection into a non-data section of the prompt [12]. Perhaps asserting that all prompts include the proper persona within a repository would be a helpful test for some developers. With unit testing, the power lies in the flexibility to tailor the test to each individual task and prompt.

The goal for PromptSet is to put forward a tool to parse prompts from files so that downstream applications can easily perform proper prompt management. We hope to establish a conversation about appropriate prompt hygiene so that the open source community can develop strong tooling for improving the CI/CD pipeline for prompts. In this work we propose a few surface level lint passes, such as typo detection, white space trimming and type annotation matching, however the possibilities go far beyond these simple tests.

REFERENCES

- [1] LangChain AI. 2023. LangServe. <https://github.com/langchain-ai/langserve>
- [2] Anthropic. 2023. Claude 2.1 Prompting. <https://www.anthropic.com/index/clause-2-1-prompting>
- [3] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Michal Podstawska, Hubert Niewiadomski, Piotr Nyczek, and Torsten Hoefler. 2023. Graph of Thoughts: Solving Elaborate Problems with Large Language Models. arXiv:2308.09687 [cs.CL]
- [4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901. https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfc4967418fb8ac142f64a-Paper.pdf
- [5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]
- [6] Miguel Corralm. 2023. Awesome Prompting. <https://github.com/corralm/awesome-prompting>
- [7] CSpell. 2023. CSpell. <https://www.npmjs.com/package/cspell>
- [8] Python Software Foundation. 2023. Black. <https://github.com/psf/black>
- [9] Thorsten Händler. 2023. Balancing Autonomy and Alignment: A Multi-Dimensional Taxonomy for Autonomous LLM-powered Multi-Agent Architectures. *ArXiv* abs/2310.03659 (2023). <https://api.semanticscholar.org/CorpusID:263671545>
- [10] Ari Holtzman, Peter West, and Luke Zettlemoyer. 2023. Generative Models as a Complex Systems Science: How can we make sense of large language model behavior? *preprint* (2023).
- [11] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *International Conference on Learning Representations*. <https://openreview.net/forum?id=nZeVkeeFYf9>
- [12] Yangsibo Huang, Samyak Gupta, Mengzhou Xia, Kai Li, and Danqi Chen. 2023. Catastrophic Jailbreak of Open-source LLMs via Exploiting Generation. *arXiv preprint arXiv:2310.06987* (2023).
- [13] Instagram. 2016. Continuous Deployment at Instagram. <https://instagram-engineering.com/continuous-deployment-at-instagram-1e18548f01d1>
- [14] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lamble, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Thibaut Le Scao, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7B. arXiv:2310.06825 [cs.CL]
- [15] Zhengbo Jiang, Frank F. Xu, Jun Araki, and Graham Neubig. 2020. How Can We Know What Language Models Know? arXiv:1911.12543 [cs.CL]
- [16] Armand Joulin, Edouard Grave, Piotr Bojanowski, Matthijs Douze, Hervé Jégou, and Tomas Mikolov. 2016. FastText.zip: Compressing text classification models. *arXiv preprint arXiv:1612.03651* (2016).
- [17] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. 2016. Bag of Tricks for Efficient Text Classification. *arXiv preprint arXiv:1607.01759* (2016).
- [18] Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. 2022. Fantastically Ordered Prompts and Where to Find Them: Overcoming Few-Shot Prompt Order Sensitivity. arXiv:2104.08786 [cs.CL]
- [19] Rajasekhar Reddy Mekala, Yasaman Razeghi, and Sameer Singh. 2023. EchoPrompt: Instructing the Model to Rephrase Queries for Improved In-context Learning. arXiv:2309.10687 [cs.CL]
- [20] Sewon Min, Xinxia Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2022. Rethinking the Role of Demonstrations:

- What Makes In-Context Learning Work? arXiv:2202.12837 [cs.CL]
- [21] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]
- [22] Mary Phuong and Marcus Hutter. 2022. Formal Algorithms for Transformers. arXiv:2207.09238 [cs.LG]
- [23] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. *CoRR* abs/1908.10084 (2019). arXiv:1908.10084 <http://arxiv.org/abs/1908.10084>
- [24] Laria Reynolds and Kyle McDonell. 2021. Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm. arXiv:2102.07350 [cs.CL]
- [25] SquidgyAI. 2023. Squidgy Testy. <https://github.com/squidgyai/squidgy-testy>
- [26] Robin Staab, Mark Vero, Mislav Balunović, and Martin Vechev. 2023. Beyond Memorization: Violating Privacy Via Inference with Large Language Models. arXiv:2310.07298 [cs.AI]
- [27] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford Alpaca: An Instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca
- [28] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenjin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghaf Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madijan Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL]
- [29] Traceloop. 2023. OpenTelemetry. <https://www.traceloop.com/blog/diy-observability-for-llm-with-opentelemetry>
- [30] tree sitter. [n. d.]. Tree-sitter. <https://tree-sitter.github.io/tree-sitter>
- [31] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fb0d053c1c4a845aa-Paper.pdf
- [32] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL]
- [33] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elashar, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. arXiv:2302.11382 [cs.SE]
- [34] Tao Xiao, Christoph Treude, Hideaki Hata, and Kenichi Matsumoto. 2024. DevGPT: Studying Developer-ChatGPT Conversations. In *Proceedings of the International Conference on Mining Software Repositories (MSR 2024)*.
- [35] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. 2023. Large Language Models as Optimizers. arXiv:2309.03409 [cs.LG]
- [36] Seonghyeon Ye, Hyeonbin Hwang, Sohee Yang, Hyeyoung Yun, Yireun Kim, and Minjoon Seo. 2023. In-Context Instruction Learning. arXiv:arXiv:2302.14691
- [37] Tony Z. Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate Before Use: Improving Few-Shot Performance of Language Models. arXiv:2102.09690 [cs.CL]
- [38] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silvia Pitis, Harris Chan, and Jimmy Ba. 2022. Large Language Models Are Human-Level Prompt Engineers. (2022). arXiv:2211.01910 [cs.LG]

A GITHUB SCRAPING CODE

```

1 for lib in ["openai",
2     "anthropic",
3     "cohere",
4     "langchain"]:
5     goto(f"https://github.com/search?" +
6         "q=%22from+{lib}%22+OR+" +
7         "%22import+{lib}%22+" +
8         "language%3Apython&type=code")

```

Received 25 January 2024