# HotGPT: How to Make Software Documentation More Useful with a Large Language Model?

Yiming Su, Chengcheng Wan, Utsav Sethi, Shan Lu, Madan Musuvathi, Suman Nath
{yimingsu,cwan,usethi,shanlu}@uchicago.edu,{madanm,Suman.Nath}@microsoft.com

## ABSTRACT

It is well known that valuable information is contained in the natural language components of software systems, like comments and manual, and such information can be used to improve system performance and reliability. Past research has attempted to extract such information through task-specific machine learning models and tool chains. Here, we investigate a general, one-model-fit-all solution through a state-of-the-art large language model (e.g., the GPT series). Our investigation covers three representative tasks: extracting locking rules from comments, synthesizing exception predicates from comments, and identifying performance-related configurations; it reveals challenges and opportunities in applying large language models to system maintenance tasks.

## CCS CONCEPTS

• **Software and its engineering → Software development techniques**; • **Computing methodologies → Natural language processing**.

## KEYWORDS

Software documentation, large language model

**ACM Reference Format:**
Yiming Su, Chengcheng Wan, Utsav Sethi, Shan Lu, Madan Musuvathi, Suman Nath. 2023. HotGPT: How to Make Software Documentation More Useful with a Large Language Model?. In *Workshop on Hot Topics in Operating Systems (HOTOS '23), June 22–24, 2023, Providence, RI, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3593856.3595910


*HOTOS '23, June 22–24, 2023, Providence, RI, USA*

ACM ISBN 979-8-4007-0195-5/23/06…$15.00
https://doi.org/10.1145/3593856.3595910

```java
/* @param n the {@code long} to divide by
 * @return a {@link BigFraction} instance with the
     resulting values
 * @throws MathArithmeticException if the fraction
     to divide by is zero */
public BigFraction divide(final long n) {
    return divide(BigInteger.valueOf(n));
}
```

**Figure 1: Comments on parameters and exception-throwing conditions** (Apache Commons Math 3.6.1)

## 1 INTRODUCTION

Modern software systems have vast amounts of natural language components, such as code comments (e.g., Figure 1) and software manuals, which contain valuable information about system usage and behavior. Extracting such information can be used for system understanding, bug finding, failure diagnosis, configuration tuning, and many more tasks.

Unfortunately, it is difficult to automatically extract such information. Many techniques have been explored in the past, including replacing natural languages with domain-specific languages in writing comments [23], building task-specific machine learning models [2, 17, 24] and customized Natural Language Processing pipelines to process comments [19, 20], complementing documentation understanding with source code analysis [9, 12, 13], etc. These techniques are effective in specific tasks, but all fall short as a general solution that can be easily used to process a variety of natural language artifacts for a variety of purposes.

In this paper, we explore whether the recent advancement of large language models (LLM), such as the GPT series [3, 16, 18], can be leveraged to produce an easy-to-use and one-model-fit-all solution for processing existing natural language components of software systems. These language models have achieved great success on many natural language processing tasks, including translation, text completion, keyword extraction, and question answering, and have shown potential in providing coding assistance [8].

Specifically, we identified three representative tasks and investigate how (well) we can use a large language model, GPT-3 [3], to replace customized solutions originally designed for each task—an approach we refer to as HotGPT. These tasks process different natural language components

| Input | Output | Previous techniques |
|---|---|---|
| Javadoc comments | Exception conditions in Java | Language&task-specific ML models [2, 17, 24] |
| Free-text comments | Lock usage rules in pre-defined templates | A pipeline of NLP tools [19, 20] |
| Manual of configurations | Whether a configuration is performance-related | Static program analysis [13] |

**Table 1: Tasks explored in this work**

of software systems, produce different types of output, support different types of system jobs (e.g., bug finding, performance tuning), and were previously solved by different solutions, as summarized in Table 1.

Our exploration shows the great potential of using LLMs to help system tasks, achieving similar or higher accuracy than previous task-specific techniques. However, several pitfalls and challenges remain, which we describe in this paper. We posit that building reliable tools that harness the capabilities of these models is an exciting but open research problem.

# 2 FROM COMMENTS TO PREDICATES

## 2.1 Task Overview & Design

Javadoc [15] is a widely used tool that generates HTML documentation from comments written in a format called *doc comment* or *doc string*. A typical line of Javadoc comment consists of a keyword headed by "@", called a block tag, and a natural language description in the topic defined by the block tag. An example is shown in Figure 1.

In the past, researchers have designed special ML models to transform the `@throws` part of Javadoc into exception-throw conditions in Java, which can then be used for automated run-time checking [2, 24]. These techniques involve task-specific NLP analysis and customized pattern-matching rules revolving around the grammar of the comment. Here, HotGPT aims to accomplish the same task using a language-agnostic large language model, Codex [4], a variant of GPT-3.

**Prompt Design.** A prompt is the text input to the language model. After many attempts, we settled down on a design that consists of three parts as shown in Figure 2:

1) An instruction text enclosed in `/* */`;

2) An example for Codex to learn from (Line 3–5 in Figure 2). For each software project to be processed, we randomly choose a function with `@throws` comment from it, and manually compose such an example for processing all other `@throws` comments.

3) The synthesis task for Codex, which includes the "Comment", the function "Signature", and an empty "output" line waiting to be filled. The Comment line and the Signature line are automatically extracted from program source code.

```
1  /* Summarize the comment in Java code using
       signature provided. */
2
3  Comment: if the queue or transformer is null
4  Signature: transformQueue(java.util.Queue queue,
       commons.collections4.Transformer transformer)
5  output:queue==null || transformer==null
6
7  Comment: if the fraction to divide by is zero
8  Signature: divide (final long n)
9  output:
10 n==0
11
12 Comment: if the fraction to divide by is zero
13 ...
```

**Figure 2: An Example Prompt and Codex output ( in green ) For Method `divide()`**

Although our prompt follows the generic structure recommended by GPT-3, an instruction, some optional examples, and the question, it took us many tries in the design.

The result of Codex was very sensitive to the instruction sentence. Some semantically similar instructions like "Convert this sentence to code" and "Extract specification from code" produced low-quality output, with at least 15% precision reduction. Some other instructions produced meaningful output, but required much effort in post-processing, which we will explain later.

We also tried not using the function signature but got poor results — knowing the type and parameter names helped Codex in synthesizing exception predicates; we tried having no example or multiple examples (e.g., up to 5), which unfortunately was both detrimental.

**Codex output post-processing.** Ideally, we want Codex to output exactly a predicate in Java that reflects the condition under which an exception is to be thrown. However, in practice, the output of Codex could be messy. When we used "Summarize the comment in Java code by signature" as the instruction, Codex tends to generate multiple lines of code, with the exception predicate embedded in an exception-throwing code structure made up by Codex (Figure 3). We had to write a parser to extract the exception predicate. With

```
...
output:
if (n == 0) {
 throw new IllegalArgumentException("Division by zero");
}
...
```

**Figure 3: Codex output ( in green ) under an alternative prompt for Figure 2 example.** Extra code parsing is needed to extract the exception predicate `n==0`.

| | Jdoctor | C2S | HotGPT |
|---|---|---|---|
| Precision | 0.97 | 0.98 | 0.96 |
| Recall | 0.79 | 0.91 | 1.00 |

**Table 2: Specification Translation Precision and Recall**

our final prompt, Codex tends to output the expected condition predicate first (e.g., `n==0` on Line 10 of Figure 2), and then part of the prompt after an empty line (e.g., Line 12 in Figure 2). Thus, we simply truncate the raw output and take the predicate line before the empty line.

## 2.2 Evaluation

**Methodology.** We evaluate HotGPT on 6 well-maintained Java libraries, which were also used in the evaluation of Jdoctor [2] and C2S [24]—prior techniques that turn `@throws` Javadoc into predicates. These libraries were used by previous work partly because developers already provided corresponding code expressions for 60% of their 778 `@throws` comments in the Javadoc, which offers perfect ground truth for evaluation. So, like previous work, we focus on these `@throws` comments that have ground truth. We will measure *precision*, defined as the proportion of total translation that is correct: $\frac{C}{C+W}$, with $C$ being the number of correctly translated predicates and $W$ being the number of incorrectly translated predicates. Since the precision metric only penalizes wrong-output but not no-output, previous work [2, 24] also measured *recall*, computed as $\frac{C}{C+M}$, with $M$ being the number of cases where the tool fails to output any predicate for a comment. We will use the same definition below.

**Results.** As shown in Table 2, HotGPT is effective at translating throw comments into code specifications. Comparing with Jdoctor and C2S, whose results are from their papers on exactly the same dataset, we achieve similar precision and better recall. It is worth noticing that both Jdoctor and C2S rely on detailed analysis of the format of comments, both syntactically and semantically, to achieve high precision. Instead, we leverage the capability of Codex to conduct the translation, without conducting the task-specific analyses.

We also checked whether the results are sensitive to language models' hyper-parameters. The answer was not so much—much less than that under different prompt designs.

# 3 FROM COMMENTS TO LOCKING RULES

## 3.1 Task Overview & Design

Published in HotOS 2005, HotComments [20] pioneered extracting system rules from code comments using NLP techniques. Due to the limitation of NLP techniques at that time, HotComments took many steps: it manually identifies popular words that refer to locks (e.g., spinlock, rwlock) and replaces them with the word "lock"; it then breaks all comments to sentences, and uses a word splitter [21] to break a sentence into words; it then uses Part-of-Speech (POS) tagging and Semantic Role Labeling [21] to tell whether a word in a sentence is a verb or a noun, to distinguish main clauses from sub clauses, and to tell subjects from objects; it then determines whether a sentence contains a locking rule based on how the word "lock" is used in the sentence (e.g., used as a verb or a noun, appearing in the main clause or not, used as a subject or not, etc.). Finally, locking rules that fall into 4 carefully designed templates are extracted. In this task, we attempt to use GPT-3 to replace this long chain of NLP tools.

**Prompt Design.** We explored different designs in two directions: (1) a generic prompt that covers all four types of locking rules targeted by HotComments; (2) a set of dedicated prompts that each targets one type of rule.

Our final design of the generic prompt is shown in Figure 4. It informally introduces the concept of locks (we omitted this paragraph in the figure for space constraints); provides a list of locking rules we are targeting, which come from HotComments [20]; puts the comment to be analyzed after "Read this sentence:"; and asks a series of questions.

This design came after several tries. Our initial design did not contain a background paragraph about locks. As a result, GPT-3 treated many irrelevant comments as related to locks, like "Hold reference count during initialization."—GPT-3 outputs that a lock named "reference count" should be held. Adding the background paragraph largely solved this problem. We initially did not include the four locking-rule templates from HotComments. As a result, GPT-3 identified many comments that are related to locks and yet difficult to use in correctness checking, like "For dynamic locks, a static lock_class_key variable is passed in through the mutex_init()".

In addition to the generic prompt, we also designed a collection of dedicated prompts. Each dedicated prompt is very simple, only including one question specifically designed for one HotComments locking rule, like "Does the following text explicitly specify that a lock or semaphore must, or must

```
<... a background paragraph about locks ...>
Here are some templates about patterns of locks/
    semaphores. 1: Lock must, or must not, be held
     before entering function. 2: Lock must, or
    must not, be held before leaving function. 3:
    Lock A must, or must not, be held before Lock
    B. 4: Lock must, or must not, be held here.

Read this sentence: is_cpuset_subset(p, q) - Is
    cpuset p a subset of cpuset q? One cpuset is a
     subset of another if all its allowed CPUs and
     Memory Nodes are a subset of the other, and
    its exclusive flags are only set if the other'
    s are set. Call holding manage_mutex.

Does the sentence describe constraint(s) about
    locks or semaphores using the above templates?
     If so, output: (1) the name of the lock/
    semaphore, (2) to hold or not to hold the lock
    /semaphore,(3) the condition(s) for holding or
     not holding, and(4) the template number that
    the condition belong to.

Name of lock/semaphore: manage_mutex
To hold or not to hold: Hold
Condition(s): Before entering is_cpuset_subset
Template number: 1
```

**Figure 4: Prompt for identifying locking rules (GPT-3 output in green).** (For space constraints, we omitted the first paragraph of our prompt that introduces lock background)

not, be held before function call or on entry (yes/no)?". The other three prompts ask "Does the following text explicitly specify that a lock or semaphore must, or must not, be held before exiting a function (yes/no)?", "Does the following text specify that a lock or semaphore must, or must not, be held before another lock (yes/no)?", and "Excluding on entry/call and exit, does the following text specify that a lock must, or must not, otherwise be held here (yes/no)?", respectively.

Although this design of dedicated prompts requires us to invoke GPT-3 multiple times upon each comment, we envision that it may provide some accuracy advantages over the generic prompt, as we have observed that GPT-3 is very consistent in answering yes/no questions and the answer, the literal "yes" or "no", is also easy to parse. Furthermore, the decomposition made our prompt design easier.

## 3.2 Evaluation

**Methodology.** We evaluated HotGPT on the same five Linux modules used in HotComments: `arch`, `drivers`, `fs`, `kernel`, `mm`. Not knowing the exact Linux version used by HotComments, we chose the one released right before the deadline for HotOS 2007 (v2.6.19). We extracted all the multi-line comments from those five modules using Python library comment_parser [1], and applied GPT-3 to every comment.

To measure the accuracy, we randomly sampled 50 positive comments and 50 negative comments from each module and manually checked the answers of GPT-3. Here, we refer to positive (or negative) comments as those considered by the generic GPT-3 prompt as containing locking rules (or not).

|  | arch | drivers | fs | kernel | mm |
|---|---|---|---|---|---|
| Positive | 0.60 | 0.78 | 0.70 | 0.54 | 0.90 |
| Negative | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |

**Table 3: Accuracy in identifying comments that contain locking rules.** Positive (Negative) are the comments considered to (not) contain locking rules by the generic prompt.

**Results.** In total, the generic prompt identified 1461 locking rules (93 in `arch`, 778 in `drivers`, 469 in `fs`, 73 in `kernel`, and 48 in `mm`). In comparison, HotComments identified 538 locking rules (50 in `arch`, 263 in `drivers`, 180 in `fs`, 29 in `kernel`, and 16 in `mm`). Since HotGPT and HotComments may not have used the same version of Linux and HotComments paper did not mention the accuracy of their rule identification (the 1461 rules identified by GPT-3 contain false positives), it is unrealistic to expect them to identify the same number of rules. We found it encouraging that (1) the number of rules is of roughly the same magnitude; (2) the proportional distribution of rules across modules is similar.

As shown in Table 3, the generic GPT-3 prompt has 100% accuracy in judging a comment to not contain one of those locking rules (the "Negative" row); it has 54–90% accuracy in judging a comment to contain one of those four locking rules. This accuracy imbalance is likely due to the majority of comments not containing a locking rule.

Once GPT-3 correctly identifies a rule-containing comment, its accuracy in identifying the lock name and differentiating locking from unlocking is very high, close to 100%, while its accuracy in pin-pointing the rule template ranges from 54% to 77% across the five kernel modules.

When we apply the dedicated GPT-3 prompts on those sampled positive comments, the average accuracy across the four dedicated prompts in each module ranges from 70% to 86%, an improvement from the generic prompt.

In summary, it is promising to replace the long chain of NLP tools used 15 years ago with a large language model (LLM). However, an LLM does not solve all the problems: those four rule templates designed by HotComments cannot

| | Precision | Recall | Accuracy |
|---|---|---|---|
| HotGPT | 83% | 76% | 81% |
| LearnConf* | 87% | 46% | 76% |

**Table 4: Performance configuration identification** (*: We use a broader definition of "performance", which caused LearnConf to have a lower recall.)

be replaced yet. Furthermore, the accuracy of HotGPT is sufficient for bootstrapping rule extraction but is not sufficient yet to completely take humans out of the loop.

# 4 FROM MANUAL TO PERF. CONFIG.

## 4.1 Task Overview & Design

An important task of system performance tuning is to understand which software configurations can affect performance. In the past, researchers have used expensive program analysis [9, 13] to automatically identify these performance-related configurations. Here, HotGPT tries to use GPT-3 to figure out whether a configuration can affect performance based on the configuration's documentation (manual).

The prompt design is much simpler comparing with the previous two tasks. We simply use the classic prompt structure for GPT-3 Q&A engine and ask about one configuration, together with its manual description, at a time. Figure 5 shows an example of our prompt and GPT-3's answer about configuration `mapreduce.map.output.compress`.

We tried prompt variations like, "Is the following parameter related to performance?", "Will this parameter affect the memory consumption or the execution speed?", and others. However, asking about "execution speed" gives us the most reasonable answers.

## 4.2 Evaluation

**Methodology.** Our evaluation mainly focuses on 287 user-facing configurations in 4 systems, MapReduce, HDFS, HBase, and Cassandra — the exact set of configurations used to evaluate LearnConf [13]. In addition, since the training data of GPT-3 may already include all the documentation of these four systems, we also study the configurations of a grocery ordering application, DDMC [7], the most popular (i.e., most stars) Java applications on GitHub that are created after the release of GPT-3 and contain user-facing configurations. To get the ground truth about which configurations are performance related, three authors have independently checked the configurations with related information and reached a consensus through discussion.

**Results.** As shown in Table 4, HotGPT performed well on the dataset of LearnConf, achieving 81% accuracy for those 287 configurations. Most of the mistakes by GPT-3 are caused by the manual not providing sufficient information.

```
Q: Based on following description, will this
    parameter affect execution speed? Why?
mapreduce.map.output.compress: Should the outputs
    of the maps be compressed before being sent
    across the network. Uses SequenceFile
    compression.
A: Yes, this parameter will affect the execution speed.
Compressing the map outputs before sending them across the
network reduces the amount of data that needs to be
transferred, which can improve the overall execution speed.
```

**Figure 5: The prompt and an informative answer ( in green ) from GPT-3 about `map.output.compress`**

GPT-3 is able to provide its reasoning process in response to the "why" prompt, which is quite logical as shown in Figure 5. Sometimes, GPT-3's answers even go beyond what the manual says. For example, in explaining why HBase' configuration io.storefile.bloom.block.size affects performance, GPT-3 says "a larger (bloom filter) block size will result in fewer disk seeks", which is not part of the manual.

In comparison, GPT-3 has similar accuracy as LearnConf. LearnConf has higher precision, but lower recall (i.e., more false negatives). The main reason is that LearnConf has a narrower definition of *performance configurations* — it only considers a configuration $C$ as performance related if $C$ can affect the execution of a time/memory-consuming operation from a pre-defined set. Of course, by analyzing source code LearnConf can figure out other information about *how* a configuration's setting can affect performance (monotonicity, slope, etc.), which is often not documented in the manual.

HotGPT also performed well for the new application, DDMC. Among the 10 configurations described in DDMC manual, HotGPT achieves 90% accuracy in identifying perf. configs.

# 5 EXPLORATION ON GPT-4

Two months after submitting this work, Open AI released GPT-4 [14], the latest version of the GPT series. To understand whether and how our system tasks can benefit from this state-of-the-art LLM, we re-run the experiments in Section 2-4 with GPT-4 and present the result highlight below.

**From comments to predicates.** Counterintuitively, HotGPT's precision slightly regressed from 96% down to 93% in translating Javadoc comments into predicates with GPT-4. On one hand, with GPT-4, HotGPT is able to correctly translate some comments with complicated conditions that it failed to translate before, such as "if a value of `array` is outside of `arange`". We believe this can be attributed to the

```
Comment: if the map is null
Signature: switchClosure(java.util.Map
    predicatesAndClosures)
```

**Figure 6: A comment that GPT-4 performs worse**

increased model capabilities of [14]. On the other hand, with GPT-4, HotGPT sometimes translates comments into predicates without considering the method signature provided, producing incorrect predicates and causing a regression in overall accuracy. For example, for the task shown in Figure 6, HotGPT's output was "`predicatesAndClosures==null`", which is correct. However, with GPT-4, the new output becomes "`map==null`", which is incorrect.

**From comments to locking rules.** We see an increase in the accuracy of extracting lock-related rules from comments. This increase ranges from 6% to 15% across the four Linux modules. GPT-4 appears to consider comments more comprehensively and precisely than GPT-3.

**From manual to performance configurations.** Utilizing GPT-4, HotGPT achieves a high recall, correctly identifying 94% of performance configurations. This is a big improvement in recall from both the original HotGPT (76%) and LearnConf (46%). However, with GPT-4, the precision of HotGPT drops from 83% to 70%. Checking and comparing the results, we believe GPT-4 has a stronger capability of parsing performance relationships from manuals and thus tends to over-estimate performance sensitivity.

## 6 RELATED WORK

Recent work has explored the use of LLMs for various software related tasks, particularly for coding assistance. Jigsaw [10] complements LLMs with testing and syntax checking to synthesize code for using Python Pandas APIs based on a natural language description of desired functionality. Other works similarly explore generating code in multiple languages [4, 6, 8, 22] using natural language or partial code as input. In contrast to these works, which focus on coding assistance, this paper explores the extraction of specifications, rules, and other knowledge from pre-existing software documentation, which can then be used for correctness checking, performance tuning, and other system tasks.

There is work in non-software-related domains that use LLMs to extract structured information from text [5], although these rely on LLM-fine-tuning using domain-specific information, with results not intended to be used programmatically. There is also work that explores the summarization of software code snippets using LLMs [11].

## 7 DISCUSSION & CONCLUSIONS

Our exploration shows that it is promising to use LLMs, like GPT-3 and GPT-4, as a generic tool for extracting useful information from software documentation, which can then be used to support various systems tasks. But several important challenges remain.

First, large language models' accuracy is promising but offers no guarantees. As shown by our evaluation, LLM's answer could be incorrect. But even when correct, the answer can be incomplete. For example, although the answer regarding configuration `mapreduce.map.output.compress` in Figure 5 is informative, it did not mention that compression itself could take time and hence affect performance from another aspect. Overall, LLM's answers provide a good starting point for follow-up manual or tool examination, and can serve as an initial screening.

Second, the performance of tools depends on prompt engineering or tuning, which remains a black art; whether background information is needed (e.g., the one about lock), whether a working example is needed, what kind of wording/phrasing would work the best all require trial and error.

Third, despite the phenomenal capabilities these models exhibit, LLMs have to be significantly more reliable before they can be used at scale in scenarios requiring no or little manual inspection. As our experiments with GPT-4 demonstrate, progress is not always guaranteed to be positive, requiring future systems to handle regressions with care.

Finally, domain expert knowledge is still needed: if we did not use the four locking-rule templates from HotComments, we could not make GPT-3 or GPT-4 to extract actionable items from comments.

Due to these challenges, we believe that harnessing these large language models to build automated and reliable tools for various system tasks is an exciting research area. We need rigorous techniques to convert the fuzzy natural language processing capabilities of these models into precise, deterministic, and correct tools that systems designers require.

We are currently experiencing an accelerating pace of improvements in the capabilities of LLMs. It is quite possible that future versions of these models solve many of the shortcomings we document here. Nevertheless, our work is just a starting point in this area. We hope that future work explores techniques to incorporate LLMs into the workflow of many system tasks, such as testing, debugging, and maintenance.

## ACKNOWLEDGEMENT

We thank the reviewers for their insightful feedback. The authors' research is supported by NSF (CNS1764039, CNS1956180, CCF2119184), the CERES Center for Unstoppable Computing, the Marian and Stuart Rice Research Award, Microsoft research dissertation grant, University of Chicago College Research Fellow Grant, and research gifts from Facebook.

# REFERENCES


[1] Jean-Ralph Aviles. 2022. comment_parser: Parse comments from various source files. Online document https://pypi.org/project/comment-parser/.

[2] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 242–253.

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

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

[5] Alexander Dunn, John Dagdelen, Nicholas Walker, Sanghoon Lee, Andrew S Rosen, Gerbrand Ceder, Kristin Persson, and Anubhav Jain. 2022. Structured information extraction from complex scientific text with fine-tuned large language models. *arXiv preprint arXiv:2212.05238* (2022).

[6] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[7] GitHub. 2022. DDMC: a grocery ordering application. Online document https://github.com/LovelyWhite/ddmc.

[8] Github. 2023. Copilot: Your AI pair programmer. Online document https://github.com/features/copilot.

[9] Yigong Hu, Gongqi Huang, and Peng Huang. 2020. Automated reasoning and detection of specious configuration in large systems with symbolic execution. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 719–734.

[10] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large Language Models Meet Program Synthesis *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1219–1231. https://doi.org/10.1145/3510003.3510203

[11] Junaed Younus Khan and Gias Uddin. 2023. Automatic Code Documentation Generation Using GPT-3. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3551349.3559548

[12] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. 2006. From uncertainty to belief: Inferring the specification within. In *Proceedings of the 7th symposium on Operating systems design and implementation*. 161–176.

[13] Chi Li, Shu Wang, Henry Hoffmann, and Shan Lu. 2020. Statically inferring performance properties of software configurations. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[14] OpenAI. 2023. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774* (2023).

[15] Oracle. 2023. How to Write Doc Comments for the Javadoc Tool. Online document https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html.

[16] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *arXiv preprint arXiv:2203.02155* (2022).

[17] Hung Phan, Hoan Anh Nguyen, Tien N Nguyen, and Hridesh Rajan. 2017. Statistical learning for inference between implementations and documentation. In *2017 IEEE/ACM 39th International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track (ICSE-NIER)*. IEEE, 27–30.

[18] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[19] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /* iComment: Bugs or bad comments?*. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. 145–158.

[20] Lin Tan, Ding Yuan, and Yuanyuan Zhou. 2007. Hotcomments: how to make program comments more useful?. In *HotOS*, Vol. 7. 49–54.

[21] NLP tools. 2023. Online document http://l2r.cs.uiuc.edu/Ëœcogcomp/tools.php.

[22] Immanuel Trummer. 2022. CodexDB: Synthesizing code for query processing from natural language instructions using GPT-3 Codex. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2921–2928.

[23] Alvaro Veizaga, Mauricio Alferez, Damiano Torre, Mehrdad Sabetzadeh, and Lionel Briand. 2021. On systematically building a controlled natural language for functional requirements. *Empirical Software Engineering* 26, 4 (2021). https://doi.org/10.1007/s10664-021-09956-6

[24] Juan Zhai, Yu Shi, Minxue Pan, Guian Zhou, Yongxiang Liu, Chunrong Fang, Shiqing Ma, Lin Tan, and Xiangyu Zhang. 2020. C2S: translating natural language comments to formal program specifications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 25–37.