UNDERSTANDING AND MITIGATING HALLUCINATION OF LANGUAGE MODELS IN SOFTWARE ENGINEERING

by

Bonan Kou

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

Department of Computer Science

West Lafayette, Indiana

August 2026

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Tianyi Zhang, Chair

Department of Computer Science

Dr. Dan Goldwasser

Department of Computer Science

Dr. Lin Tan

Department of Computer Science

Dr. Ming Yin

Department of Computer Science

Approved by:

Dr. Voicu S. Popescu

To everyone who has helped me along the way.

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest gratitude to my advisor, Dr. Tianyi Zhang, without whom this dissertation would simply not exist. Over the years of my PhD study, he has shaped every aspect of how I do research: how to identify problems that matter rather than problems that are merely convenient, how to design studies rigorous enough to survive scrutiny, how to interpret results honestly, and how to communicate ideas so that others can build on them. He read countless drafts of my papers—often late at night and always with remarkable care—and his detailed, constructive feedback taught me more about scientific writing than any course ever could. Beyond research skills, he showed me by example what it means to be a responsible and generous scholar: he was always patient when experiments failed, always encouraging when reviews were harsh, and always willing to spend hours discussing a half-formed idea until it became a real one. Whenever I doubted myself, he believed in me first and let me catch up later. I could not have asked for a better mentor, and the standards of rigor and integrity he set will stay with me in whatever I do next.

I am also deeply grateful to my committee members, Dr. Dan Goldwasser, Dr. Lin Tan, and Dr. Ming Yin, for their insightful questions and constructive feedback at my preliminary and final examinations, which substantially sharpened the framing and the rigor of this dissertation. Their perspectives from natural language processing, software engineering, and human-computer interaction pushed me to think about my work from angles I would have otherwise missed.

I owe a great deal to Purdue University and the Department of Computer Science for providing an environment in which a young researcher can thrive: world-class faculty who keep their doors open, well-maintained computing infrastructure that made the large-scale experiments in this dissertation possible, and a departmental staff whose quiet, reliable work spared me from countless administrative burdens. I am grateful for the teaching and research assistantships that supported my study, and for the many seminars, reading groups, and hallway conversations at Purdue that repeatedly reminded me why I chose to pursue

research in the first place. West Lafayette has been a wonderful place to grow, and I will always be proud to call myself a Boilermaker.

I would like to thank my collaborators, including Dr. Muhao Chen, Dr. Lei Ma, Zhijie Wang, Shengmai Chen, and Zijie Zhou, for their invaluable feedback and contributions to the works presented in this dissertation. Working with them taught me the value of collaboration across institutions and research areas. I also thank my labmates in the Human-Centered Software Systems Lab at Purdue for countless discussions, paper reviews, and practice talks, and for making the lab a supportive and enjoyable place to work.

Finally, I thank my family and friends for their love and encouragement throughout this journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Large Language Models (LLMs) have shown remarkable capabilities in software engineering tasks such as summarizing developer discussions, generating API documentation, and synthesizing code from natural language descriptions. However, a fundamental obstacle to deploying LLMs in practice is *hallucination*—models confidently produce content that is unfaithful to their input or factually incorrect. Hallucination is particularly harmful in software engineering, where misinformation about APIs, program behavior, or code semantics can silently propagate into production systems.

This dissertation argues that *hallucination can be understood and mitigated by external guidance, self-reflection, and implicit signals*, and supports this statement with three works. First, ASSORT summarizes Stack Overflow posts extractively and mitigates unfaithful content with *external guidance*: a pre-trained natural language inference model re-anchors machine-generated summaries to faithful sentences of the source post, requiring no labeled training data. Second, AUTODOC generates API documentation from crowdsourced knowledge, grounding generation in evidence retrieved by a fine-tuned dense passage retrieval model and countering hallucination through *self-reflection*: an LLM-based validation component that checks extracted knowledge before it enters the final document. Third, to understand why code LLMs hallucinate, we conduct the first large-scale empirical study of attention alignment between six LLMs and human programmers during code generation. Analyzing *implicit signals*—model attention computed by twelve methods—reveals a consistent misalignment between model and human attention and shows that a considerable portion of code generation errors, including hallucinated code, can be explained by five recurring attention patterns.

Together, these works demonstrate that external guidance and self-reflection are effective levers for mitigating hallucination, while implicit signals provide a promising foundation for understanding and anticipating it. We conclude with takeaways from building these systems and future directions toward trustworthy, hallucination-aware AI assistants for software engineering.

1. INTRODUCTION

Modern software development is knowledge-intensive. To solve a programming problem, developers have traditionally searched Stack Overflow for solutions and consulted API documentation to learn unfamiliar libraries [1, 2]—manual processes that require locating, navigating, and reading through large volumes of scattered information. Recently, this workflow has begun to shift toward machine intelligence: developers now delegate entire coding tasks to Large Language Models (LLMs) such as GPT-4 [3] and Claude [4]. These models can summarize lengthy developer discussions, draft API documentation, and synthesize executable code directly from natural language descriptions [5–7]. The promise is enormous: instead of manually searching, navigating, and reading crowdsourced information, developers could rely on LLMs to leverage decades of accumulated software knowledge and deliver the right information to them far more efficiently.

However, a fundamental obstacle stands between this promise and practice: *hallucination*. In their widely adopted survey, Ji et al. define hallucination as generated content that is nonsensical or unfaithful to the provided source content, further distinguishing *intrinsic* hallucination, which contradicts the source, from *extrinsic* hallucination, which cannot be verified from the source [8]. For the LLM era, Huang et al. refine this definition into *faithfulness* hallucination—output that deviates from the user's instruction or the provided context—and *factuality* hallucination—output that conflicts with verifiable real-world facts [9, 10]. Both kinds arise in software engineering, and this dissertation confronts both: a summary that distorts what the original Stack Overflow post said and generated code that ignores an essential condition in the task description are faithfulness hallucinations, while an API document that describes behavior the API does not have is a factuality hallucination (Section 2.4 discusses this taxonomy and its code-specific refinements [11, 12] in detail). Hallucination is particularly harmful in software engineering. Unlike casual conversation, software artifacts are consumed by developers who act on them: a hallucinated API caveat can propagate into production code, and a subtly wrong summary can misdirect hours of debugging. Prior studies have repeatedly shown that developers place significant trust in automatically surfaced information [5, 6, 13–15], which makes silent misinformation especially dangerous.

At the beginning of this research, the natural question was how to *prevent* generative techniques from producing unfaithful content in developer-facing tasks. This dissertation approaches the question from three complementary angles. First, generation can be *grounded*: instead of allowing a model to freely generate text, we can constrain it with external guidance—domain knowledge, retrieved evidence, and supervision signals from models trained on better-resourced domains. Second, generation can be *checked*: a model can be asked to reflect on and validate its own output before that output reaches the user. Third, and more fundamentally, hallucination can be *understood*: by inspecting the implicit signals inside a model, such as its attention distribution, we can diagnose when and why a model deviates from the intent of its input. These three angles form the central claim of this dissertation.

1.1 Thesis Statement

Hallucination of large language models in software engineering can be understood and mitigated by external guidance, self-reflection, and implicit signals.

The three key terms in this statement deserve elaboration, since each names a distinct family of levers that this dissertation develops and evaluates.

External guidance refers to constraints and signals that originate *outside* the generative model itself and are used to steer or restrict what the model can produce. A generative model left alone draws on its parametric memory, i.e., the patterns it absorbed during training. This memory is precisely where hallucination originates: when it is incomplete or misaligned with the task domain, the model fills the gap with plausible fabrications. External guidance counteracts this by anchoring generation to things that are verifiably real: the sentences of a source document, domain-specific features engineered from the structure of the task, evidence retrieved from a trusted corpus, or supervision signals borrowed from models trained in better-resourced domains. In this dissertation, external guidance takes the form of a pre-trained natural language inference model that re-anchors a machine-generated summary to

faithful sentences of the source post (Chapter 3), and dense retrieval that grounds every generated knowledge snippet in a concrete Stack Overflow post (Chapter 4).

Self-reflection refers to using the model itself—or a second instance of it—to examine, verify, and repair its own output before that output reaches the user. The premise is an asymmetry that recurs throughout this dissertation: *checking* whether a statement is supported by given evidence is an easier task than *generating* a correct statement outright, so even an imperfect model can profitably audit its own generations. In this dissertation, self-reflection takes the form of an LLM-based knowledge validation component that checks each extracted knowledge snippet against its source post and discards unsupported content (Chapter 4).

Implicit signals refer to the byproducts of a model's computation—attention distributions, token probabilities, and other internal states—that are not part of the output itself but carry information about *how* the model processed its input. If hallucination is a predictable consequence of a model attending to the wrong parts of an instruction, then these signals offer a window for understanding, and ultimately anticipating, when generation will go wrong. In this dissertation, implicit signals take the form of model attention, computed by twelve different methods and compared at scale against the attention of human programmers (Chapter 5).

To support the thesis statement, this dissertation presents three works. The first two build practical systems that mitigate hallucination when generating software engineering artifacts: ASSORT mitigates unfaithful summarization with *external guidance* (Chapter 3), and AUTODOC mitigates hallucinated API knowledge with retrieval grounding and *self-reflection* (Chapter 4). Building these systems exposed a deeper problem: we could suppress hallucination empirically, but we lacked a principled understanding of *why* models hallucinate in the first place. This motivated the third work, an empirical study that uses *implicit signals*—the attention of code generation models—to understand how LLMs process natural language instructions and why they sometimes generate code that deviates from them (Chapter 5).

1.2 Mitigating Hallucination with External Guidance

The first work targets a scenario where faithfulness is paramount: summarizing Stack Overflow (SO) posts. SO has become an integral part of the modern programming workflow, yet locating essential information in it remains time-consuming: for any programming question there are often multiple relevant posts to consider, and individual posts can be lengthy. A survey with 72 professional developers confirmed that sifting through many online posts is cognitively demanding, and that developers wish for tool support to navigate posts quickly [1]. Automatic summarization promises exactly this relief. However, a summary is only useful if developers can trust it: a developer who acts on a summary that distorts what the original post said may adopt the wrong solution and lose hours to misdirected debugging.

Prior attempts at SO post summarization relied on information retrieval methods and hand-crafted heuristics, which a study by Nadi and Treude found insufficient for handling the ambiguity and sophistication of natural language [16]. Deep learning summarizers from the NLP community are far more capable, but they introduce the very risk this dissertation targets. Abstractive models fine-tuned on general-domain corpora such as news articles are prone to producing inaccurate terminology and narratives when applied to the software domain—a form of hallucination induced by domain shift. Worse, the software domain lacks the massive labeled corpora these models are trained on: the largest available dataset of SO summaries, SOSum [17], is orders of magnitude smaller than general-domain corpora like CNN/DailyMail, so simply fine-tuning a general summarizer on SO data is not enough.

We present Assort, a deep-learning framework for SO post summarization that keeps generation grounded by design. Assort is *extractive*: it selects summative sentences from the original post rather than freely generating new text, so every sentence in a summary is guaranteed to come from the source. At the heart of Assort is an indirectly supervised method, Assort$_{IS}$, which requires no labeled SO data at all. Assort$_{IS}$ first lets a summarizer pre-trained in a better-resourced domain draft a summary; since that draft is unreliable under domain shift, a pre-trained natural language inference model then acts as an external guide, tracing the draft—including whatever it may have hallucinated—back to the faithful sentences of the original post that support it. To calibrate this zero-label design, Assort

also includes a supervised counterpart, ASSORT$_S$, which combines BERT embeddings with domain-specific features, trains separate models for different question types (*how-to*, *conceptual*, and *debug-corrective*), and ensembles their predictions with a question classifier; it serves as a strong supervised reference point in our evaluation.

On a benchmark of manually curated SO summaries, ASSORT$_{IS}$ outperforms six existing techniques by at least 7% in F1 score without seeing a single labeled SO summary, while the supervised reference point ASSORT$_S$ leads by at least 13%. Crucially, when less than 20% of the labeled training data is available, ASSORT$_{IS}$ surpasses ASSORT$_S$ itself, demonstrating that externally guided indirect supervision is the more practical recipe in low-resource domains. In a user study with 12 participants, 76–85% preferred summaries generated by ASSORT$_{IS}$ or ASSORT$_S$ over the best baseline in terms of usefulness, comprehensiveness, and conciseness, with no significant preference between the two—the zero-label method reads just as well as the fully supervised one. With respect to the thesis statement, ASSORT establishes the first pillar: anchoring a model's output to the source document through an external verifier yields summaries that developers measurably trust.

1.3 Mitigating Hallucination with Retrieval and Self-Reflection

The second work scales from summarizing a single post to generating a complete software artifact: API documentation. API documents are critical for developers to write and debug code, yet previous studies have repeatedly shown that modern libraries and frameworks are plagued with obsolete and incomplete documentation [2]. Meanwhile, a wealth of up-to-date API knowledge—undocumented caveats, error conditions, alternative usages—is buried in crowdsourced discussions on SO, scattered across many lengthy threads. Existing approaches for extracting this knowledge are primarily heuristic-based or rely on supervised models, so they either lack the flexibility to handle sophisticated natural language or require significant labeled data.

LLMs are attractive for consolidating crowdsourced knowledge into documentation, but they raise two problems that simple prompting cannot solve. The first is hallucination: when asked directly about an API, an LLM freely mixes genuine knowledge with fabricated behav-

ior, and an API document that misstates parameter semantics is worse than no document at all. The problem is amplified for less popular APIs, which are underrepresented in training data, leaving the model to rely on its parametric memory precisely where that memory is weakest. The second is redundancy: the same knowledge appears in many posts, and naive extraction floods the document with duplicates.

We present AutoDoc, an LLM-based pipeline that generates API documents covering seven types of API knowledge defined in the literature [18]. AutoDoc combines both mitigation strategies advocated by this dissertation. For external guidance, it grounds generation in retrieved evidence: a dense passage retrieval model [19], fine-tuned on SO data to account for the unique linguistic patterns of developer discussions, locates the answer posts most likely to contain each type of API knowledge, and the LLM extracts knowledge only from these posts. For self-reflection, AutoDoc performs LLM-based *knowledge validation*: after extracting a knowledge snippet, the LLM is prompted to check the snippet against its source post and discard content that cannot be supported. A knowledge summarization step then removes redundancy in the validated knowledge.

We evaluated AutoDoc on 48 APIs of three popularity levels from four ecosystems (Java, Android, Kotlin, and TensorFlow) against five baselines, including two prior automated documentation approaches and three GPT-4o prompting strategies. AutoDoc generates documents that are up to 77.7% more accurate and 9.5% less redundant, and on average 34.4% of the knowledge in its documents is not covered by official documentation. An ablation study confirms that each safeguard earns its place: removing knowledge validation lowers accuracy by 3.7%, and removing summarization leaves 31.7% more duplicated knowledge. Notably, with the full pipeline in place, even a 7B-parameter open-source model (Mistral-7B) generates documents approaching GPT-4o quality, suggesting that engineered guidance and reflection can substitute for raw model scale. In a user study with 12 participants, 80% preferred AutoDoc's documents over both baselines in all five quality dimensions. With respect to the thesis statement, AutoDoc establishes the second pillar: retrieval grounding plus self-reflection makes free-form LLM generation trustworthy enough for a high-stakes artifact.

1.4 Understanding Hallucination through Implicit Signals

After building ASSORT and AUTODOC, we had effective recipes for suppressing hallucination, but the recipes were empirical. We still could not answer a basic question: when a model reads a natural language instruction, does it actually attend to the information a human expert considers essential? If it does not, then hallucination is not an occasional glitch but a predictable consequence of misplaced attention. We realized that we lacked a fundamental understanding of hallucination, and that this understanding had to come from inside the model.

Attention analysis is a natural instrument for this question, because it directly operationalizes our hypothesis about hallucination: if a model under-attends to a keyword that a human considers essential—a boundary condition, a type constraint, an exception to handle—then the generated code is likely to omit or fabricate exactly that semantics. Attention analysis is also a proven methodology for understanding model behavior in computer vision and autonomous driving, where recent studies further show that aligning model attention with human attention improves both model performance and user trust [20, 21]. Yet no prior work had asked whether LLMs attend to a programming task description the way programmers do, let alone whether misplaced attention accounts for hallucinated code. Answering these questions required overcoming two obstacles.

First, no code generation benchmark records *programmer* attention—which words and phrases a human considers essential to solving a task. Second, there is no consensus on how to compute *model* attention for LLMs: self-attention scores can be aggregated across layers and heads in many ways, and gradient-based and perturbation-based alternatives measure different things.

The third work therefore turns from building systems to studying models. We construct the first programmer attention dataset, covering all 1,111 programming tasks from the HumanEval and MBPP benchmarks, labeled by two experienced programmers and validated by a third. We then conduct the first large-scale study of *attention alignment* between LLMs and human programmers in code generation, comparing the programmer labels against model attention computed by twelve methods from three families—self-attention-based, gradient-

based, and perturbation-based—across six LLMs of different sizes, including GPT-4. The study yields two findings that bear directly on hallucination. First, there is a consistent *misalignment* between model and human attention, in every model and under every attention computation method: models systematically overlook parts of the instruction that programmers consider essential, which is precisely the precondition under which a model must fall back on its parametric memory and fabricate the missing semantics. Second, the misalignment indeed manifests as hallucination: an in-depth analysis of 211 incorrect code generations from the two best models shows that 27% of the errors can be explained by five recurring attention patterns, including cases where the model ignores a stated condition and invents behavior that was never asked for—hallucinated code whose origin is traceable to where the model looked. With respect to the thesis statement, this study establishes the third pillar: implicit signals are a viable lens for understanding and, ultimately, anticipating hallucination.

1.5 Contributions

This dissertation makes the following contributions:

- **Assort (Chapter 3): mitigating hallucination with external guidance.** A framework for Stack Overflow post summarization that keeps generation faithful by construction: it is extractive, so every summary sentence provably comes from the source post, and its indirectly supervised method uses a pre-trained natural language inference model as an external guide that re-anchors the hallucination-prone draft of an out-of-domain summarizer to faithful sentences of the original post—without any labeled SO data. It outperforms six existing techniques by at least 7% in F1 score (13% for its supervised counterpart), showing that external guidance can restore faithfulness where domain shift would otherwise induce hallucination.

- **AutoDoc (Chapter 4): mitigating hallucination with retrieval grounding and self-reflection.** An LLM-based pipeline for generating API documentation from crowdsourced knowledge that attacks hallucination on two fronts: a fine-tuned dense retrieval model grounds every knowledge snippet in a concrete SO post rather than

in parametric memory, and an LLM-based validation component self-checks each extracted snippet against its source and discards fabricated API behavior. Its documents are up to 77.7% more accurate than five baselines, and ablations confirm that removing the self-checking safeguard measurably increases hallucinated content.

- **Attention alignment study (Chapter 5): understanding hallucination through implicit signals.** The first large-scale study of attention alignment between six LLMs and human programmers on 1,111 code generation tasks, together with the first programmer attention dataset. The study reveals a consistent attention misalignment between models and programmers and traces hallucination to its origin: 27% of analyzed code generation errors—including code semantics fabricated against the stated task requirements—are explained by five recurring attention patterns, establishing implicit signals as a diagnostic lens for understanding and anticipating hallucination.

1.6 Outline

The rest of this dissertation is organized as follows. Chapter 2 reviews related work on text summarization for software artifacts, automated API documentation, code generation with LLMs, and hallucination understanding and mitigation. Chapter 3 presents Assort, our framework for faithful Stack Overflow post summarization. Chapter 4 presents AutoDoc, our LLM-based API documentation pipeline with retrieval grounding and self-validation. Chapter 5 presents our empirical study of attention alignment between code generation models and human programmers. Chapter 6 concludes the dissertation, revisits the thesis statement in light of the three works, distills broader insights, and discusses future research directions.

2. RELATED WORK

This chapter situates the dissertation in four bodies of literature. Section 2.1 reviews techniques that summarize software engineering documents, especially Stack Overflow posts, which is the problem setting of ASSORT (Chapter 3). Section 2.2 reviews automated API documentation and knowledge extraction from software engineering documents, the problem setting of AUTODOC (Chapter 4). Section 2.3 reviews large language models for code generation and attention analysis of neural models, which form the background of our attention alignment study (Chapter 5). Finally, Section 2.4 reviews approaches for understanding and mitigating hallucination in large language models, the phenomenon that unifies the three works of this dissertation.

2.1 Summarization of Software Engineering Documents

2.1.1 Stack Overflow Post Summarization

Several approaches have been presented to summarize SO posts to concise texts to facilitate SO post navigation [1, 16, 22, 23]. AnswerBot extracts summative paragraphs from SO posts based on features such as information entropy and paragraph position [1]. CraSolver uses a similar multi-factor ranking mechanism to summarize bug solutions [23]. Both AnswerBot and CraSolver generate summaries at the paragraph level. By contrast, our approach generates more fine-grained sentence-level summaries of SO posts. Nadi and Treude experimented with four different IR-based approaches to select essential sentences from SO posts [16]. They conducted a survey with 43 developers and found that while participants would like to get navigation support for browsing SO, the IR-based approaches failed to provide such support. Motivated by these findings, we propose a novel DL-based framework that can more accurately identify summative sentences in SO posts in this work. Several approaches have been proposed to generate question titles for SO threads based on code snippets [24, 25]. However, since question title generation aims to summarize a question in a one-liner, these approaches cannot be applied to SO post summarization, which has a very different problem setting.

2.1.2 Text Summarization for Bug Reports

There are several summarization techniques for other types of SE documents such as bug reports [26–30]. Liu et al. designed a new metric called *believability score*, which measures the degree to which a sentence is supported or refuted by other comments. Based on the idea of *believability score*, they further developed BugSum, which selects summative sentences in a bug report [27]. Rastkar et al. created a small dataset with human-annotated summaries for 36 bug reports and used it to train a Logistic Regression classifier with 24 explicit features to identify summative sentences in a bug report [28]. Mani et al. experimented with four unsupervised models— Centroid, MMR, Grasshopper, and Diverse Rank—on the task of bug report summarization [29]. Unlike these techniques, we proposed two learning paradigms for building DL models for SO post summarization.

2.1.3 General Text Summarization

Many neural-based text summarization techniques have been proposed recently [31–36]. For example, Liu et al. presented a BERT-based model and trained it on CNN/Daily-Mails [31]. Narayan et al. presented a CNN-based model that summarizes a single news article into a one-liner and trained it with XSum [32]. In another work, Narayan et al. framed the task of text summarization as a ranking problem and proposed a global optimization method for training [33]. Dong et al. framed text summarization as a contextual bandit problem, in which each document is considered as a context and each combination of selected sentences is an action to take [34]. While these models perform well on news articles, reusing these models in another domain, such as SO posts, is not an easy task due to domain shift. In this work, we propose ASSORT$_S$, a supervised model that accounts for unique characteristics of SO posts and demonstrate that it significantly outperforms a fine-tuned version of *BertSum* (Section 3.7.1).

2.1.4 Other Tool Support for Information Seeking in Stack Overflow

In addition to text summarization, many other kinds of tool support have been proposed to facilitate information seeking in SO [37–51]. For example, SeaHawk [37] is an Eclipse plugin that integrates SO into an IDE. Chatbot4QR [52] expands a user query with tags of similar SO questions in order to improve search results. Ye et al. [39] propose a re-ranking mechanism for SO search results of a user-defined query based on different search focus users may have. They identify linguistic patterns in different types of queries and train separate models based on query types to re-rank search results. SISE uses linguistic patterns to identify sentences that mention an API in SO posts and augments API documents with those sentences [44].

2.2 Automated API Documentation and Knowledge Extraction

2.2.1 Automated API Documentation

Several approaches have been proposed for automated API documentation. SISE [44] is a framework that augments official API documents with insightful sentences from SO posts. Unlike AutoDoc, SISE relies on pattern-based methods that lack flexibility and generate coarse-grained API document by extracting original sentences without paraphrasing. Uddin et al. [53] also proposed to automatically produce API documentation from API code examples and reviews from SO posts. Their generated API documents consist of a code example, a textual description, and reviews with positive and negative sentiments. Stylos et al. [54] introduced Jadeite, which displays commonly used classes more prominently and automatically identifies the most common ways to construct an instance of any given class. Dekel and Herbsleb [55] improves API documentation by decorating method invocations whose targets have associated usage directives, such as rules or caveats. Following a similar method, Chen and Zhang also proposed to incorporate insights from crowdsourced FAQs into API documentation [56]. They connected API documents and informal documentation by capturing the developers' Web browsing behaviors.

2.2.2 Knowledge Extraction from SE Documents

There are several approaches that aim to extract API knowledge from API documents and SO discussions [44, 45, 57–61]. Some relied on rule-based pattern matching [45, 60]. For example, Li et al. [45] developed a set of linguistic patterns to extract 10 types of API usage caveats from SO posts. Recently, neural networks improved the flexibility of some knowledge extraction methods in terms of processing SE documents. For example, Liu et al. [57] trained a feed-forward neural network to classify descriptive sentences of APIs from the API documentation. With a similar idea, DeepTip [58] uses a CNN model to extract sentence-level API usage tips from SO posts with a trained CNN model. Compared with previous work, our approach performs knowledge extraction in a finer-grained way because of the pre-trained language models' ability to understand the deep semantics of SO posts. Furthermore, previous neural approaches need to acquire a large labeled dataset first and train a model from scratch. However, our approach makes use of pre-trained models and thus requires no labeled data.

2.3 Large Language Models for Code Generation

2.3.1 Code Generation from Natural Language

Since CodeBERT [62], there has been a large body of literature where Large Language Models (LLMs) are used in code generation [7, 63–74]. Both Guo et al. [63] and Zeng et al. [65] used a pre-trained BERT [75] model to encode NL questions and database schemas for text-to-SQL generation. CodeBERT adopts the same model architecture as BERT but is trained with a hybrid objective on code and text data from GitHub repositories [62]. Codex, CodeGPT, and GraphCodeBERT improve CodeBERT by leveraging data flow in the pre-training stage [76]. Recently, modern LLMs such as GPT-4 and Google Bard excel in code generation tasks on various benchmarks [77, 78]. The highly accurate and contextually rich code snippets generated by these models enable the automation of various programming tasks.

In addition to developing new LLMs for code, prior work has also presented new methods to enhance LLMs for more accurate and robust code generation [66, 70, 79–83]. Instead of directly generating code from text, REDCODER [70] first retrieves similar code snippets from a corpus and then passes them along with the text description to an LLM for code generation. Shen et al. [66] proposed to leverage domain knowledge from documentation and code comments to assist code generation. Chakraborty et al. proposed a new pre-training task called *naturalizing of source code* (i.e., translate an artificially created code to a human-written form) to help an LLM learn how to generate natural code [81]. Chen et al. proposed to use an LLM to automatically generate test cases to examine the code generated by the same LLM [80]. Zan et al. proposed to first decompose a LLM to two LLMs, one for generating program sketches and the other for filling the sketches [82]. SkCoder [83] is designed to mimic developers' code reuse behavior by first retrieving a code example related to a given task, extracting a sketch from it, and editing the sketch based on the task description.

2.3.2 Empirical Studies on Code Generation

Recently, many studies have evaluated LLM-based code generation models in different aspects, including performance [7, 84–87], robustness [88, 89], security [90–92], code smells [93], usability [5, 73, 94, 95], and licensing [96]. The most related to us are those that investigate the explainability of LLMs for code [97–100]. Karmakar et al. studied what pre-trained BERT-based models such as CodeBERT and GraphCodeBERT have learned about code using probing tasks [97]. A probing task is essentially a prediction task on a specific code property such as code length and cyclomatic complexity to test whether the model is sensitive to the property. Compared with our work, they did not analyze the code generation process, e.g., why and how certain code is generated based on a text description. Liguori et al. proposed a perturbation-based method to evaluate NMT models for code generation [98]. In addition, Zhang et al. investigated the code generation process by analyzing the self-attention layers in CodeBERT and GraphCodeBERT [99]. Wan et al. did a similar self-attention analysis and also designed a new probe task on code structures to analyze whether LLMs have learned information about code structures [100]. Compared with these studies,

our work differs by analyzing the consistency between LLMs' attention and programmers' attention on the NL description of a programming task.

2.3.3 Model Attention Analysis in Other Domains

Many attention calculation methods have been developed to explain models in other domains. For example, in CV domain, Selvaraju et al. [101] proposed to use the gradients of a convolutional neural network (CNN) to indicate the importance of each pixel in an image for a specific prediction. Similarly, Zhou et al. [102] used the global average pooling mechanism to calculate the importance of each region for CNN to classify an image correctly. Autonomous driving is another domain where the need for explainability is strong. For example, Zeiler et al. [103] use deconvolution layers to understand how autonomous vehicles capture real-time image segments using CNNs. In another work, Chen et al. [104] proposed a data-efficient policy learning approach called Semantic Predictive Control (SPC) that explains how perceived environmental states are mapped to actions.

2.4 Understanding and Mitigating Hallucination in Large Language Models

Hallucination is the phenomenon that unifies the three works of this dissertation. This section first reviews how hallucination is defined in natural language generation (Section 2.4.1), then surveys the emerging literature that characterizes hallucination specifically in code generation and clarifies which types of hallucination this dissertation considers (Section 2.4.2), and finally reviews approaches for mitigating hallucination (Section 2.4.3) and for understanding it through model internals (Section 2.4.4).

2.4.1 Hallucination in Natural Language Generation

Hallucination was first characterized in task-specific natural language generation models, such as abstractive summarizers and neural machine translation systems. In their widely adopted survey, Ji et al. define hallucination as generated content that is nonsensical or unfaithful to the provided source content, and further distinguish *intrinsic* hallucination (output that contradicts the source) from *extrinsic* hallucination (output that cannot be

verified from the source) [8]. With the rise of general-purpose LLMs, Huang et al. refine this taxonomy for the LLM era into *factuality* hallucination (output that conflicts with verifiable real-world facts) and *faithfulness* hallucination (output that deviates from the user's instruction or the provided context) [9]. These two dimensions—faithfulness to the input and factuality with respect to the world—recur throughout the software engineering manifestations of hallucination studied in this dissertation.

2.4.2 Hallucination in Code Generation

A rapidly growing body of work characterizes hallucination in the code domain, where the notion of "factually incorrect" takes on domain-specific forms such as fabricated APIs and code that silently deviates from the task requirement. Liu et al. conducted the first systematic study of hallucinations in LLM-generated code and established a taxonomy in which generated code may conflict with the user's *intent*, with its own *context* (internal inconsistency), or with real-world *knowledge* (e.g., referring to nonexistent APIs or misusing library facts) [11]. Tian et al. define code hallucination from an execution-based perspective: generated code that is syntactically valid and even semantically plausible, yet fails to execute correctly or to satisfy the specified requirements; their CodeHalu benchmark organizes such failures into four categories—mapping, naming, resource, and logic hallucinations—verified by executing the generated programs [12]. Zhang et al. extend the characterization from standalone benchmarks to practical, repository-level code generation, deriving a taxonomy of hallucination phenomena from six mainstream LLMs, studying their root causes (e.g., training data noise and lack of project context), and showing that retrieval-augmented generation alleviates them [105]. Hallucination in code also creates concrete security risks: Spracklen et al. analyze *package hallucination*, where models recommend installing packages that do not exist; across 576,000 generated code samples, 19.7% of recommended packages were hallucinated, exposing users to package-confusion supply chain attacks [106]. Finally, at a finer granularity, Jiang et al. study code hallucination at the token level—localizing the incorrect token at which an LLM first deviates from a correct implementation—and release

Collu-Bench, a benchmark of 13K hallucination instances with per-token log probabilities and execution feedback for learning to predict hallucinations [107].

Scope of this dissertation. Following the faithfulness/factuality framing of Huang et al. [9], this dissertation considers three manifestations of hallucination in software engineering. Chapter 3 targets *faithfulness* hallucination in summarization: summaries that distort or fabricate content absent from the source Stack Overflow post, especially under domain shift. Chapter 4 targets *factuality* (knowledge-conflicting) hallucination in API documentation: generated knowledge snippets that misstate API behavior or cannot be supported by any crowdsourced evidence, closely related to the knowledge-conflicting category of Liu et al. [11]. Chapter 5 concerns *intent-conflicting* hallucination in code generation—code whose semantics deviate from the natural language task description, corresponding to the requirement-violating and logic categories of prior taxonomies [11, 12]—and studies its attention-level causes. Execution-level failures such as resource errors [12] and security consequences such as package confusion [106] are outside the scope of this dissertation.

2.4.3 Mitigating Hallucination

Existing mitigation approaches can be broadly organized into two families that mirror the first two pillars of this dissertation.

Grounding generation with external evidence. A prominent line of work reduces hallucination by conditioning generation on retrieved evidence rather than on parametric memory alone. Retrieval-augmented generation couples a dense retriever [19] with a generator so that outputs can be traced back to source documents [108], and similar retrieve-then-generate designs have been adopted for code generation, for example REDCODER [70]. In the code domain, De-Hallucinator iteratively grounds model predictions with project-specific API references retrieved from the code base [109], and Zhang et al. report that retrieval-augmented generation is among the most effective mitigations for repository-level code hallucination [105]. In the same spirit, extractive formulations sidestep free-form generation altogether by restricting outputs to spans of the source document. The works in this dissertation instantiate this family in the software engineering domain: Assort (Chapter 3)

constrains summaries to sentences of the original post and uses natural language inference to re-anchor abstractive drafts to the source, and AutoDoc (Chapter 4) grounds API knowledge extraction in SO posts retrieved by a fine-tuned dense passage retrieval model.

Self-reflection and self-consistency. A second family asks the model itself to detect or repair its mistakes. Self-refinement methods iteratively critique and revise model outputs [110]; chain-of-verification prompts the model to generate and answer verification questions about its own draft before finalizing an answer [111]; SelfCheckGPT detects hallucinations by measuring the consistency among multiple stochastic samples of the same model [112]; and self-consistency decoding aggregates multiple reasoning paths to reduce spurious outputs [113]. Beyond a single model instance, multi-agent debate lets several LLM instances propose and critique answers over multiple rounds, improving factuality through mutual correction [114]. In the code domain, CodeT uses an LLM to generate test cases that check the code generated by the same LLM [80]. The knowledge validation component of AutoDoc (Chapter 4) belongs to this family: it prompts the LLM to verify each extracted knowledge snippet against its source post and filters out unsupported content, which improves the accuracy of the generated documents.

Intervening on internal signals at decoding time. A third family reduces hallucination without retraining by exploiting the model's internal signals during inference. Building on the contrastive decoding framework [115], DoLa contrasts the next-token distributions of later and earlier transformer layers, amplifying the factual knowledge that emerges in higher layers [116]; SLED refines this by tracking how logits evolve across layers [117]; and ActLCD casts decoding as a sequential decision problem, learning a reinforcement learning policy that decides *when* to apply layer contrasting [118]. Inference-time intervention instead shifts activations along truthfulness-associated directions in a small set of attention heads [119]. These approaches are complementary to the systems in this dissertation, and their reliance on internal signals foreshadows the diagnostic perspective of Chapter 5.

Training and fine-tuning for faithfulness. A fourth family bakes faithfulness into the model itself. FactGen constructs entailment-consistent pre-training targets with a natural language inference model and rewards faithful candidates through contrastive ranking fine-tuning [120]. For general-purpose LLMs, Tian et al. fine-tune models with preference pairs

labeled by automatic factuality judgments [121], FLAME makes the alignment pipeline itself factuality-aware [122], and R-Tuning teaches models to refuse questions beyond their parametric knowledge instead of guessing [123]. These approaches require access to model weights and considerable training cost; the works in this dissertation instead treat the generative model as a black box and wrap it with external guidance and self-reflection, a design that applies equally to closed-source LLMs.

2.4.4 Understanding Hallucination through Model Internals

Orthogonal to the mitigation families above, a further line of work does not suppress hallucination but seeks to *understand* and *predict* it from signals inside the model. An early finding is that models carry calibrated self-knowledge: Kadavath et al. show that LLMs can largely predict whether they know the answer to a question [124], and Burns et al. recover latent truth directions from activations without any supervision [125]. Subsequent work probes hidden states directly to assess the truthfulness of statements [126] and to detect hallucination as it happens: Duan et al. find that an LLM's hidden states react differently when it processes a hallucinated versus a correct answer, suggesting the model is implicitly "aware" of its own hallucination [127]; CH-Wang et al. train probes over internal states to flag hallucinated spans in both sampled and induced generations [128]; and Orgad et al. show that the internal representations of LLMs encode considerably more information about truthfulness than their outputs reveal—information concentrated in specific answer tokens that can be exploited to predict error *types*, though the encoding does not transfer universally across tasks [129]. Uncertainty offers another intrinsic signal: Farquhar et al. detect confabulations by estimating *semantic entropy*—uncertainty over the meanings, rather than the surface forms, of sampled generations [130]. These intrinsic signals also power the decoding-time mitigations reviewed in Section 2.4.3 [116, 118, 119]. In the code domain, prior work analyzes self-attention to characterize what pre-trained code models have learned [99, 100], and Collu-Bench shows that implicit signals such as per-token log probabilities and generated token types carry predictive information about where a model will hallucinate [107].

Compared with these studies, our attention alignment study (Chapter 5) is the first to compare LLM attention against *human programmer* attention at scale in code generation, and to show that recurring attention patterns explain a substantial portion of generation errors, including hallucinated code semantics. This positions implicit signals not merely as a post-hoc explanation device but as a diagnostic lens on why hallucination happens.

Overall, prior work established grounding, self-checking, decoding-time intervention, faithfulness-oriented training, and internal-signal analysis largely in general NLP settings, and the code-hallucination literature has so far focused on characterizing and benchmarking the phenomenon. This dissertation adapts and extends these ideas to software engineering artifacts—crowdsourced posts, API documents, and code—where faithfulness requirements are strict and domain shift is severe, and demonstrates their effectiveness end to end.

3. MITIGATING HALLUCINATION WITH EXTERNAL GUIDANCE: AUTOMATED SUMMARIZATION OF STACK OVERFLOW POSTS

This chapter presents ASSORT, our framework for automated summarization of Stack Overflow posts, and the first pillar of the thesis statement: mitigating hallucination with *external guidance*. Summarization is a setting where unfaithful generation is both likely and costly. Powerful pre-trained summarizers exist, but they are trained on general-domain corpora such as news articles; applied to software discussions, they hallucinate terminology and narratives, and a developer misled by a distorted summary may adopt the wrong solution. Training a faithful summarizer from scratch is not a way out either, since labeled SO summaries are scarce.

The centerpiece of this chapter is ASSORT$_{IS}$, an indirectly supervised method that resolves this dilemma with external guidance. Instead of trusting the output of an out-of-domain summarizer, ASSORT$_{IS}$ treats that output only as a hint: a pre-trained natural language inference model—the external guide—checks the generated draft against the original post and traces it back to the source sentences that support it. The final summary is thereby re-anchored to, and guaranteed to consist of, sentences of the original post, and the whole method requires no labeled SO data at all. To calibrate how far this zero-label design can go, we also build ASSORT$_{S}$, a supervised extractive model trained on labeled SO data with domain-specific features and question-type-aware ensembling, which serves as a strong supervised reference point alongside six existing techniques in our evaluation.

3.1 Introduction

Online Q&A forums such as Stack Overflow (SO) have become an integral part of modern programming workflow [131–134]. However, locating essential information in online posts can be time-consuming since there are often multiple relevant posts to consider, and some posts can be lengthy. This is confirmed by a recent survey with 72 professional software de-

velopers [1]. Participants complained that sifting through many online posts was cognitively demanding and wished to get tool support for quickly navigating online posts.

Generating concise summaries of SO posts is a promising way to facilitate the navigation of SO posts [1, 16, 22, 135]. Nadi and Treude experimented with four approaches to capture the gist of a SO post by extracting essential sentences from it [16]. They conducted a user study with 43 developers and confirmed that seeing essential sentences in a SO post indeed increased developers' confidence when assessing the relevance and quality of the post. However, they also found that selecting sentences only based on heuristics or information retrieval (IR) methods was not sufficient. Indeed, these approaches are inherently limited since they lack the flexibility of handling ambiguous or sophisticated narratives in natural language.

The Natural Language Processing (NLP) community has made significant progress in text summarization using deep learning (DL) [33, 34, 136–138]. Those DL-based approaches require to be trained with massive parallel corpora of text documents and summaries. For example, the CNN/DailyMails dataset [139] contains 286K pairs of news articles and human-written summaries. However, no such large datasets exist for SO posts, except for a recent dataset named SOSum [17]. While SOSum contains manually curated summaries for 2,278 SO answer posts, it is still much smaller compared with general-domain corpora such as CNN/DailyMails. Furthermore, no experiment has been done to prove that this dataset is sufficient to train a DL model for summarizing SO posts.

To bridge the gap, we propose a framework for SO post summarization, ASSORT (Automated Summarization of Stack OveRflow posT). ASSORT provides a novel supervised model ASSORT$_S$ to account for the unique characteristics of SO posts. First, ASSORT$_S$ uses a combination of BERT embeddings and domain-specific features to characterize sentences in a SO post. Second, since answers to different types of questions have different linguistic norms, we train a question classifier and separate models with the same BERT-based architecture for different types of questions. Third, to account for the uncertainty of the question classifier, the outputs of these models are then ensembled based on the probability distribution of the question classifier, as shown in Figure 3.2.

Furthermore, since acquiring labeled training data is costly, ASSORT also includes an indirect supervision method called ASSORT$_{IS}$, which does not need to be trained with any labeled SO data. ASSORT$_{IS}$ makes use of supervision signals from pre-trained models in another domain, such as news articles, for SO post summarization. To handle domain shift issues, ASSORT$_{IS}$ uses a pre-trained Natural Language Inference (NLI) model to trace back to key sentences in the original SO post based on the initially generated summary. Compared with the initially generated summary, which may contain inaccurate terminologies and narratives due to domain shift, extracting aligned sentences from the original post can more accurately capture the gist of the post.

We evaluate both learning methods in ASSORT with the comparison to a BERT-based text summarization model [31], which is fine-tuned with the same SO training data as ASSORT$_S$. We also select three heuristics-based methods and one unsupervised learning method from prior work [16, 140, 141] as baselines. We find that both ASSORT$_S$ and ASSORT$_{IS}$ outperform all baselines by at least 13% and 7% in terms of the F1 score. This implies that only finetuning a general model with a relatively small dataset such as SOSum [17] is not sufficient. To improve training efficiency, it is necessary to account for the unique characteristics of SO posts. Furthermore, since ASSORT$_{IS}$ is not trained on any SO data, it achieves lower accuracy than ASSORT$_S$. However, when less than 20% of the original training data are available, ASSORT$_{IS}$ achieves better summarization accuracy than ASSORT$_S$. This implies that indirect supervision remains a promising yet cheap alternative for building generalizable models for specific domains when there is a lack of labeled training data.

We conduct a qualitative user study with 12 participants to evaluate the quality of summaries generated by different techniques. Compared with the best baseline model [31], the majority of participants (76%-85%) preferred summaries generated by ASSORT$_S$ or ASSORT$_{IS}$ in terms of usefulness, comprehensiveness, and conciseness. Participants found the summaries generated by ASSORT$_S$ more concise and useful in practice, while they found the summaries generated by ASSORT$_{IS}$ more comprehensive and provided a better overview. Overall, neither ASSORT$_S$ or ASSORT$_{IS}$ really triumph over each other according to user feedback.

In summary, this chapter makes the following contributions:

- We design a new supervised model that generates concise and self-contained summaries of SO posts. This model accounts for the uniqueness of SO posts and achieves state-of-the-art accuracy in SO post summarization.
- We propose a new indirect supervision method to further tackle the challenge of lacking labeled training data in SO post summarization. We demonstrate the feasibility of developing models with acceptable accuracy but with no cost of data labeling.
- We conduct a comprehensive evaluation of the proposed learning methods with six comparison baselines, an ablation study, and a qualitative user study.

3.2 Motivating Example

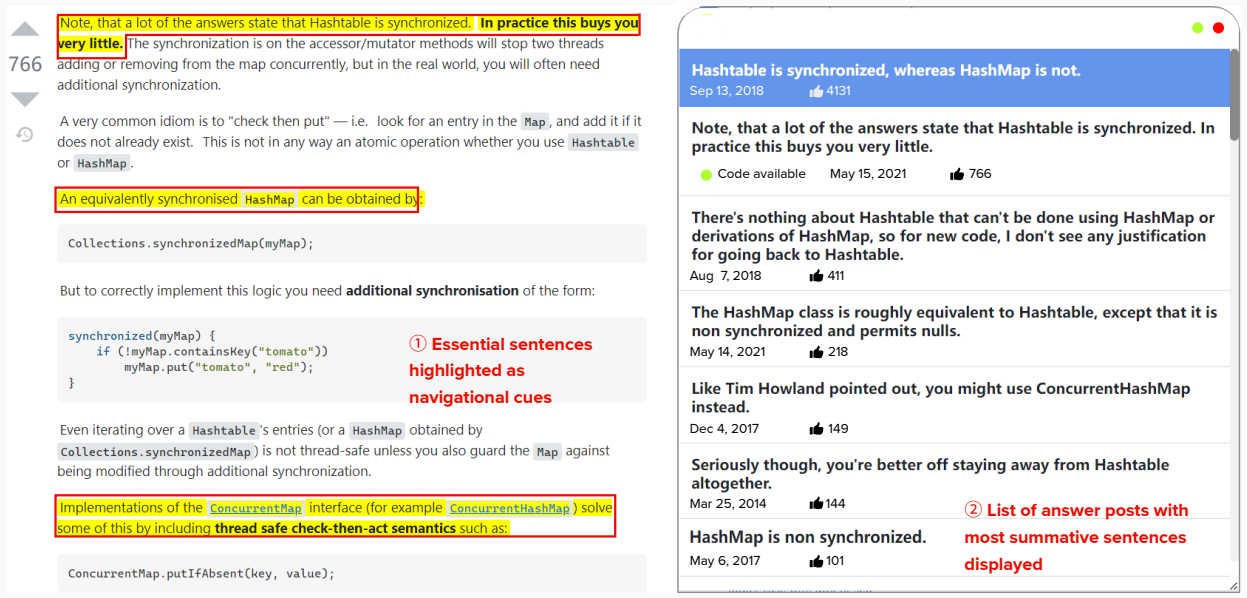

Figure 3.1. A Chrome extension built upon ASSORT. On the left side, summative sentences are highlighted in yellow. On the right side, a navigation panel provides a bird's-eye view of the thread by showing the first sentence in the summary of each answer.

This section illustrates how summarizing SO posts helps developers quickly navigate SO posts and get an overview of various answers given by other developers. Suppose Alex is an Android developer and she wants to transfer key-value pairs between two layers of a Spring

MVC framework. She knows both `HashMap` and `HashTable` can serve as the data structure for this task. Yet she is not sure about the pros and cons of these two APIs. So she searches online.

The first search result from Google is a Stack Overflow question—*"What are the differences between a HashMap and a HashTable in Java?"*[1] There are 35 answer posts to this question. Alex finds it time-consuming to read all of them. So she decides to first read the accepted answer. The accepted answer (Post 40878) points out three major differences between `HashMap` and `HashTable`: (1) `HashTable` is synchronized, whereas `HashMap` is not; (2) `HashTable` does not allow `null` keys or values; (3) `HashMap` has the flexibility to be replaced with `LinkedHashMap`. Alex finds this answer helpful, but she is not sure how comprehensive this answer is.

Alex starts reading other highly voted answers to check if they include any important information not covered by the accepted answer. However, she finds herself submerged by the abundant information in those posts. Since many posts are lengthy with details that she does not care about, Alex spends most of her time locating helpful information in those posts. For example, Figure 3.1 shows the second answer (Post 41454) in this thread. It is a long post with 4 code snippets. Yet the gist of it is a simple message—*the synchronization in `HashTable` is not sufficient, and synchronized `HashMap` and `ConcurrentMap` are better choices*. Without any tool support, Alex has to read the entire post linearly to get this key message, which can take quite a few minutes.

Like many other developers, Alex only reads a handful of answers and returns to her own code due to her limited time budget [132, 142, 143]. This inevitably makes her overlook answers that are not highly ranked but contain useful information that she is unaware of. For example, the 20th answer (Post 37031553) mentions that `HashMap` has $O(log(n))$ complexity and is faster than `HashTable`. None of the top five answers mentions this. If performance is a top concern to Alex, reading this post can help Alex make a more informed decision. However, without any tool support, Alex is unlikely to reach this deep in the thread practically.

[1] ↑https://stackoverflow.com/questions/40471

ASSORT helps Alex by automatically summarizing each answer post into concise text, so Alex can get a quick overview of many posts and prioritize which posts to spend more time on. In this way, she can make a more informed decision on which API to use. Specially, we build a Chrome extension for Stack Overflow, which highlights summative sentences from each post and renders a list of post summaries, as shown in Figure 3.1. By looking at the navigation panel, Alex quickly realizes that three answers mention `HashTable` is synchronized. By contrast, the other four answers suggest staying away from `HashTable`. Alex is interested in the second answer since its author expressed a strong opinion against `HashTable`. After clicking on it, Alex jumps to the corresponding answer, where three summative sentences have been highlighted as navigation cues. The first sentence points out that the synchronization in `HashTable` is not sufficient, while the second and third sentences propose synchronized `HashMap` and `ConcurrentMap` as alternatives. Alex feels that these highlighted sentences summarize the gist of the answer well, so she returns to the navigation panel to go deeper in the thread. As she scrolls down the panel, she notices the summary of the 20th answer—$O(log(n))$ *for* ***`HashMap`*** *vs.* $O(n)$ *in* ***`HashTable`***. This indicates that `HashMap` has lower time complexity than `HashTable`. Alex dives into this answer for details, as performance is always a top concern for her. Without ASSORT, this information would have been buried deep in the thread and unlikely to be discovered. Being aware of this information, Alex now feels more confident in making an informed decision between these two APIs.

3.3 Task Definition

We introduce the definition of *Stack Overflow Post Summarization* as follows: Given a SO answer post consisting of N sentences, the goal is to select a small set of sentences to form a succinct and self-contained summary. Essentially, this can be viewed as a contextualized sentence classification task where a sentence can either be in or not in the summary.

This task is also known as *Extractive Summarization* (ES) in NLP. It is in contrast to another type of summarization called *Abstractive Summarization* (AS). Instead of selection summative sentences from the original document, AS generates new text to summarize the

document [144]. Compared with ES, a unique challenge in AS is that distorted or fabricated text is likely to be introduced during text generation [145, 146]. Several studies have shown that factual inconsistency occurs in up to 30% of abstractive summaries [147–150]. Furthermore, since most abstractive summarization models are pre-trained on news articles or general-domain corpora, such errors may become more prevalent when reusing those models on SO posts due to domain shift.

In this chapter, we focus on extractive summarization as the first step towards SO post summarization while leaving the more challenging abstractive summarization task as future work. In Section 3.4, we first propose a supervised learning method with a novel model architecture tailored for SO posts. Then in Section 3.5, we propose an indirect supervision method that reuses pre-trained models from another domain while addressing domain shift issues via natural language inference.

3.4 Supervised Post Summarization

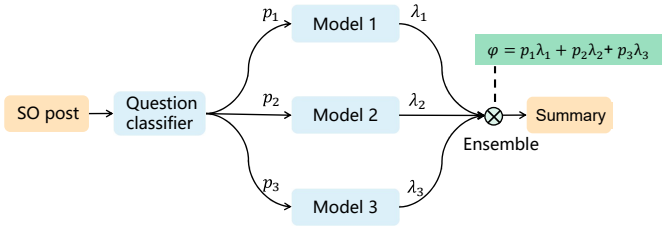

Figure 3.2. An overview of AssortS

Assort includes a supervised learning method named AssortS. As shown in Figure 3.2, AssortS takes three phases to summarize a SO answer post. Since answers to different types of questions follow different linguistic patterns, AssortS first predicts the type of the SO question (Phase I). To account for the uncertainty of the question classifier, the answer post is fed into three sentence classification models separately, each of which is trained for one type of SO questions (Phase II). Finally, AssortS ensembles the predictions of these models based on the likelihood of the question type to generate the final summary (Phase III).

3.4.1 Question Classification

While manually inspecting SO answer posts, we observed that answers to different kinds of questions have different linguistic forms. For example, answers to a how-to question often contain a step-by-step solution, while answers to a conceptual question often contain a definition or description of the concept. Based on this insight, we decide to first categorize SO posts based on their question types and then train separate post summarization models for different types of questions. We follow the SO question taxonomy from prior work [17, 151–154] and consider three representative types of SO questions—*how-to questions*, *conceptual questions*, and *bug fixing questions*. How-to questions ask for instructions for achieving a task, e.g., *"how do I undo the most recent local commits in Git?"*. Conceptual questions ask for clarifications on a concept, e.g., *"what are metaclasses in Python?"*. Bug fixing questions ask for solutions to fix some issues, e.g. *"git is not working after macOS Update"*.

We develop a Support Vector Machine (SVM) classifier to categorize SO posts based on their question titles. We train it with a combination of 506 classified SO questions from SOSum [17] and 365 new questions. In total, our dataset includes 301 how-to questions, 305 conceptual questions, and 265 bug-fixing questions. We choose these three types of questions, since they are the most common question types, covering 77% of SO questions based on prior work [152]. We discard questions that did not belong to the three types of questions when curating the dataset.

To select and label the additional 365 questions, the first author first ranked all SO questions by view count in descending order. Then, he inspected these questions and manually classified them based on the types until he found 365 questions belonging to one of the three types of questions under investigation. Then, the first author and another undergraduate student independently labeled the summative sentences in the answer posts of these questions following the labeling heuristics described in the SOSum paper [17]. In total, they labeled 785 answer posts under these 365 questions. The Cohen's Kappa score before the discussion is 0.72, which implies a substantial agreement [155]. The two labelers met to discuss their labels and resolved all disagreements. This labeling process took about 73 man-hours.

Despite the simplicity of SVM, this classifier achieves reasonable accuracy (78%) with 8:1:1 train/dev/test data split and 10-fold validation. To further account for the potential misclassification of the SVM classier, ASSORT$_S$ adopts an ensemble mechanism that incorporates the probability distribution of different types of questions predicted by the SVM classifier (Section 3.4.3). An ablation study confirms the benefit of incorporating question classification into ASSORT$_S$, improving its F1-score by 9% (Section 3.7.2).

3.4.2 Summative Sentence Identification

We train separate models to identify summative sentences for the three representative types of SO questions. These models share the same model architecture but are trained on answer posts to different types of questions to capture unique linguistic norms and writing styles of each question category. We explain the model architecture below.

Figure 3.3 gives an overview of the model architecture. ASSORT$_S$ performs hybrid learning by combining the semantic representation from a BERT model and a multifaceted set of domain-specific features. Each sentence in the given post is encoded into a vector embedding, which is then fed into a feedforward neural network (FNN) to decide whether the sentence is a summative sentence.

In this chapter, we use a BERT model that is fine-tuned with 152 million sentences from Stack Overflow [156]. Given a sentence from a SO post, ASSORT$_S$ computes its embedding from BERT by averaging the embedding vectors of all tokens in the sentence. The incorporation of deep contextualized sentence embeddings helps ASSORT$_S$ capture the semantics of a given sentence. This is confirmed by the ablation study in Section 3.7.2, where the inclusion of BERT increases the F1 score of our model by 17%.

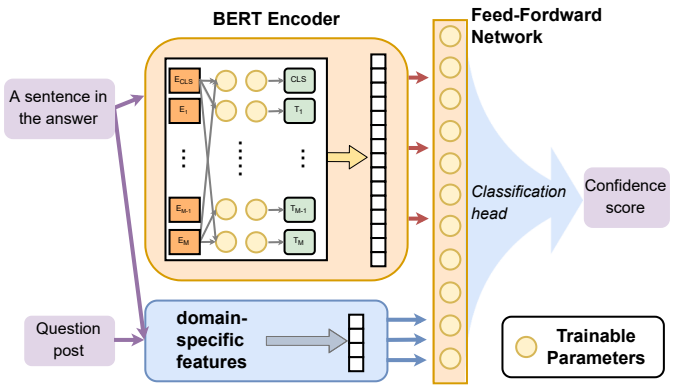

Figure 3.3. The architecture of the sentence classifier

While the contextualized embeddings from BERT are effective in capturing sentence semantics, they are not sufficient to capture specific features, such as bold text, which human readers often rely on to identify summative sentences. Therefore, we further design a set of domain-specific features to capture the semantic, structural, and stylistic information cues in SO posts. We elaborate on these features below.

First, we design five **semantic features** to select summative sentences based on their content.

- *Entity Overlap.* If a sentence mentions a software entity that is also mentioned in the question title or in a SO tag of the question, this sentence may provide some directly relevant information for the question. In this chapter, we use SEthesaurus [157] to identify software entities in a sentence. Let E_q be the combination of software entities in the question title and the SO tags in the question. Let E_s be the set of software entities mentioned in a sentence in the answer post. The entity overlap is computed as $|E_q| \cap |E_s|/|E_q|$.

- *Comparative Adjective.* This feature captures whether a sentence contains a comparative adjective. Sentences containing a comparative adjective often contain information that helps readers compare two APIs or two solutions, e.g., *"the stack is faster because all free memory is always contiguous."*

- *Superlative Adjective.* This feature captures whether a sentence contains a superlative adjective. Similar to comparative adjectives, sentences with superlative adjectives often indicate answerers' strong inclination for or against an approach, API, or bug fix, e.g., *"there is no doubt that application/json is the best MIME type for a JSON response."*

- *Imperative Sentence.* This feature captures whether a sentence is an imperative sentence. Imperative sentences often contain instructions to accomplish a programming task or to fix a bug, e.g., *"use git revert commit-id."*

- *Linguistic Patterns.* As we label SO posts for training ASSORT_S, we summarize 19 phrases that may imply important information in an answer (Table 3.1). Specifically, the first author randomly sampled 100 posts from the labeled dataset and manually analyzed the summative sentences of these posts. He identified an initial set of common phrases that were shared across multiple sentences. He then applied these patterns back to those sentences to measure the coverage and iteratively refined the patterns. Similar procedures have been adopted in prior work to identify linguistic patterns [1, 43, 140]. Each pattern corresponds to a dimension in the sentence embedding. If a linguistic pattern is matched, then the corresponding dimension is set to 1, otherwise 0.

Second, we design two **structural features** to select summative sentences based on their relations with other sentences and codes in the post.

- *Sentence Position.* This feature captures the position of a sentence in a given post. This feature is designed based on our observation that the leading sentences in many SO posts can serve as a good summary of the post.

- *Code Adjacency.* This feature indicates whether a sentence is around a code snippet. It is designed based on our observation that sentences around a code snippet enclosed by a `pre` tag often contain information cues for a programming task solution or a bug fix.

Third, we design three **stylistic features** to capture formatting styles that are used to highlight important information in a SO post.

- *Inline Code.* This feature indicates whether a sentence contains an inlined code fragment. This feature is designed based on our observation that, for how-to questions and bug-fixing questions, answerers sometimes suggest an alternative API, pinpoint a problematic piece of code, or provide a short code fragment as a solution.

- *Bold Text.* This feature indicates whether a sentence contains bold text. In SO posts, answerers often highlight important terms or statements in bold to draw attention from readers.

- *Step in a List.* This feature indicates whether a sentence is the first sentence of an item in a bulleted or numbered list. This feature is designed based on our observation that for how-to questions, many answerers typically provide a list of steps to accomplish a task.

The ablation study (Section 3.7.2) confirms the usefulness of these domain-specific features. Specifically, including these features leads to an increase of 6% in the F1 score. We also conducted an experiment to measure the contribution of each feature by removing each of them and evaluating the accuracy degradation. The results are included in the Supplementary Material.

Table 3.1. Linguistic patterns for sentences that may contain important information

No.	Phrase	No.	Phrase
1	However, ...	11	In practice, ...
2	First, ...	12	In fact, ...
3	In short, ...	13	Otherwise, ...
4	In this case, ...	14	If you care, ...
5	In general, ...	15	In contrast, ...
6	Finally, ...	16	On the other hand, ...
7	Then, ...	17	Below is ...
8	Alternatively, ...	18	Additionally, ...
9	In other words, ...	19	Furthermore, ...
10	In addition, ...		

3.4.3 Ensemble Inference

To account for the uncertainty of question classification in Phase I, we design an ensemble mechanism that merges the predictions from the three sentence classifiers to make the final decision about whether a sentence should be included in the summary.

The ensemble mechanism takes two input: (1) the softmax probabilities of k categories ($k = 3$ in this case) predicted by the question classifier, $p_1, ..., p_k$; (2) the probabilities of a sentence being a summative sentence from each sentence classifier, $\lambda_1, ..., \lambda_k$. The final score ϕ of a sentence is defined as: $\phi = \sum_{i=1}^{k} p_i \lambda_i$. If a sentence has a final score greater than a threshold θ, it is selected as a summative sentence. We experimentally determine θ to be 0.5 on a validation set of 303 posts (10% of the dataset) with the objective of maximizing the F-1 score. After every sentence in an answer post is classified, ASSORT collects all the selected sentences and outputs them as an extractive summary for the answer post.

3.5 Post Summarization via Indirect Supervision

While supervised learning can achieve superior performance, obtaining a large amount of labeled data is often costly, especially in specific domains such as software engineering. Therefore, we propose an indirect supervision approach, ASSORT$_{IS}$, to overcome this limit. Instead of acquiring labeled data for direct supervision, ASSORT$_{IS}$ uses supervision signals from pre-trained models in another domain, such as news article summarization. To address the challenge of data shift in cross-domain transfer, we use a pre-trained Natural Language Inference (NLI) model to select summative sentences in the original post based on the summary generated by the pre-trained text summarization model. Figure 3.4 provides an overview of our approach.

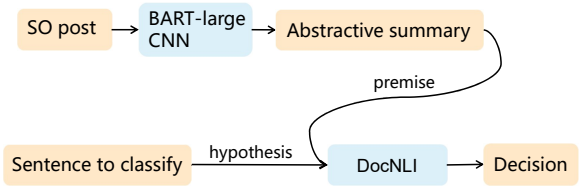

Figure 3.4. An overview of ASSORT$_{IS}$

In this chapter, we use BART-large-CNN [158] as the pre-trained text summarization model. Unlike our supervised model in Section 3.4, BART-large-CNN is an abstractive summarization model. In other words, BART-large-CNN generates a summary in its own words via an autoregressive decoding process, rather than selecting summative sentences in the input document. Specifically, BART-large-CNN is based on BART [137], a denoised auto-encoder that encodes an input document to a high-dimensional embedding for text reconstruction. In BART-large-CNN, the initial BART model is fine-tuned to generate text summaries using the CNN/DailyMail corpus [139], which contains 300K news articles and their human-written summaries. In this chapter, we select BART-large-CNN, since it is a representative and strong summarization model that outperforms other transformer-based models such as [138] by up to 6 points on CNN/DailyMail in ROUGE-based metrics [137].

Table 3.2. An abstractive summary generated by BART-large-CNN and the summary generated by our approach

Original post [Post 438095]
There is a potential problem here with regard to workflow persistence. If you create long running workflows that are persisted in a database, these workflows are not reloaded into memory until there is some external event that reloads them. As there they are responsible for triggering the event themselves but cannot until they are reloaded. And we have a catch 22 :-(. The proper way to do this is using an external service. And while this might feel like dividing the code into two places it really isn't. The reason is that the workflow is responsible for the big picture, IE what should be done. And the runtime service is responsible for the actual implementation or how it should be done. That way you can change the how without changing the why and when part. [**Source of the ineloquent sentence in BART-large-CNN's summary**]
Abstractive summary generated by BART-large-CNN
There is a potential problem with regard to workflow persistence. If you create long running workflows that are persisted in a database they will not be able to restart until there is an external event reloading them. The proper way of doing this is using an external service which can change the how part without changing what it should be done or when it's better than having two separate processes for different tasks as each one does their own thing but doesn't agree on exactly who must carry out its function at any given time. [**broken English**]
Summary generated by our indirect supervision approach
There is a potential problem here with regard to workflow persistence. The proper way to do this is using an external service. That way you can change the how without changing the why and when part.

While BART-large-CNN is demonstrated effective on news article summarization, it suffers from domain-shifting issues when applied to a different domain, such as SO post summarization. Furthermore, recent studies have shown that abstractive summarization models may introduce factual inconsistencies in generated text compared with original text [145, 146]. Table 3.2 shows an abstractive summary generated by BART-large-CNN. The last sentence in it (highlighted in red) is exceptionally long and contains broken English with some hallucinated information.

Despite the broken English, this sentence is not fully made up by BART-large-CNN. It is based on two sentences in the original post (highlighted in green). In fact, these two sentences provide a good summary of the last paragraph in the original post. Though BART-large-CNN recognizes them, it fails to generate an eloquent summary due to domain shift.

To address this issue, we propose associating abstractive summaries to the original post's corresponding sentences using a pre-trained Natural Language Inference (NLI) model. The task of NLI [159] studies to what extent a premise entails, contradicts, or remains neutral with a hypothesis. For example, suppose we have a hypothesis sentence, "*a kid ate a fruit*", and a premise sentence, "*a boy ate an apple*". An NLI model will predict that the premise entails the hypothesis. However, if the hypothesis is changed to "*a kid ate a banana*", the prediction will becomes to neutral or contradiction.

We use a pre-trained NLI model to decide which sentence in the original SO post is entailed by the abstractive summary and thus should be included in the final summary. Specifically, we use a RoBERTa model that is pre-trained on DocNLI [160], a document-level NLI dataset. DocNLI covers multiple text genres, such as news, fiction, and conversations, with multi-sentence (i.e., document) premises and single-sentence hypotheses. Given a summary generated by BART-large-CNN, our approach checks the logical relationship between the summary (i.e., the premise) and every sentence in the original post (i.e., the hypotheses). The DocNLI model will then produce a probability distribution over the three relationships—*entail*, *contradict*, and *neutral*. Our approach selects a sentence as part of the final summary if the entailment probability is the higher than the other two probabilities.

3.6 Evaluation

We conduct both quantitative experiments and user studies to answer the following four research questions:

- RQ1: How effective is our supervised model, ASSORT$_S$, in SO post summarization?
- RQ2: To what extent does each component in ASSORT$_S$ contribute to its effectiveness?
- RQ3: How does our indirectly supervised model, ASSORT$_{IS}$, compare to our surprised model?
- RQ4: How do real programmers perceive the summaries given by our models?

3.6.1 Experiment Setup

In addition to the 2,278 labeled posts from SOSum [17], we further manually labeled 785 answer posts following the same labeling procedure as described in [17]. The labeling process is described in Section 3.4. We use these 3,063 posts for training and evaluation. These posts are from 785 SO questions, including 254 *how-to* questions, 322 *conceptual* questions, and 209 *bug-fixing* questions.

We empirically decided the model structure and hyperparameters for the classification head of ASSORT$_S$. Specifically, we experimented with different kinds of models, as shown in Table 3.3. We chose feedforward neural network (FNN) since it performed the best. We also experimented with different numbers of hidden layers in the FNN and eventually chose 1 hidden layer, as shown in Table 3.4.

Table 3.3. Model Accuracy with Different Types of Classification Heads

	F1	**Δ**
Feedforward NN	**0.71**	**-**
Random forest	0.56	-0.15
Decision tree	0.54	-0.17
Linear regression	0.65	-0.06
Logistic regression	0.63	-0.08
Ada boost	0.59	-0.12
Naive Bayes Classifier	0.62	-0.09

Table 3.4. Model Accuracy with Different Numbers of Hidden Layers in the FNN

# hidden layers	**Precision**	**Recall**	**F1**
1	**0.73**	**0.69**	**0.71**
2	0.69	0.71	0.70
3	0.73	0.71	0.72
4	0.73	0.70	0.71

To train ASSORT$_S$, for each type of questions, we randomly split the corresponding answer posts into training, development, and test sets with a split ratio of 8:1:1. Then, we train a summative sentence identification model for answers to each type of question following the model architecture in Figure 3.3. During training and testing, an answer post to be summarized is broken down into individual sentences. In total, we have 14,165 sentences for the 2,424 answer posts in the training set. Each model was trained with a batch size of 512 sentences in 150 epochs. In each batch, each sentence is fed to ASSORT$_S$ one by one with the corresponding question title. We use the Adam optimizer [161] for training and the learning rate is set to $1e^{-5}$. Training ASSORT$_S$ took 3 hours on a single GPU (NVIDIA GeForce GT 1030).

To design an ablation study to answer RQ2, we create four variants of ASSORT$_S$ with one key component removed in each variant. The four key components are: (1) the BERT embeddings, (2) the domain-specific features, (3) the question classifier, (4) the ensemble

mechanism. Specifically, when the question classifier is ablated, ASSORT$_S$ uses a sentence classifier trained on all types of questions to predict the final score of a sentence. When the ensemble mechanism is ablated, ASSORT$_S$ first predicts the question type of a given post and then only uses the sentence classifier for the predicted question type to identify summative sentences, rather than using all three classifiers.

To answer RQ3, we first compare ASSORT$_{IS}$ against ASSORT$_S$ on *a full training setting* where we train ASSORT$_S$ with the entire training set. Then, we compare them in different *low-resource settings* where only a subset of the original training data is available. In those low-resource settings, we randomly select SO posts from the original dataset to make subsets of training data. Since ASSORT$_{IS}$ does not require any training data, the low-resource baselines are created to make a fair comparison between ASSORT$_S$ and ASSORT$_{IS}$ in conditions where training data are scarce.

3.6.2 Compared Baselines

We select a state-of-the-art extractive summarization model on the general domain [31] and also fine-tune it on the same training set of 2,424 SO posts as ASSORT$_S$. Furthermore, we select three heuristics-based methods and one unsupervised method that can perform sentence-level summarization from prior work [16, 140, 141]. These four methods have also been experimented in [16].

(1) *BertSum* [31] is an extractive summarization model that first uses BERT to encode sentences and then uses a transformer to select summative sentences [31]. It outperforms several previous techniques [33, 162–164] on two popular text summarization datasets—NYT [165] and CNN/DailyMail [139]. In this experiment, we use the checkpoint of *BertSum* that has the best performance on CNN/DailyMail.

(2) *BertSum (fine-tuned)* [31] is a fine-tuned version of *BertSum*. It is fine-tuned with the training data of ASSORT$_S$, including 2,424 SO posts and their summaries.

(3) *wordpattern* identifies essential sentences in a SO post using a set of 360 word patterns. These patterns are initially designed by Robillard and Chhetri [140] to identify sentences containing indispensable knowledge in API documentations.

(4) *simpleif* is a technique proposed by Nadi and Treude [16]. It is designed based on the insight that essential sentences may contain contextual information expressed in the form of conditions. Thus, *simpleif* identifies all sentences that have the word "if" in them as essential sentences.

(5) *contextif* is another technique proposed by Nadi and Treude [16]. It uses a set of heuristics to identify essential sentences that carry technical context and are useful.

(6) *lexrank* is a commonly used unsupervised text summarization approach [141]. It uses a stochastic graph-based method to compute the relative importance of sentences in a document and generates an extractive summary by selecting the top k sentences. We use the default k value, 5, in an open-source implementation of *lexrank* [166].

We do not compare with paragraph-level summarization techniques such as AnswerBot [1] and CraSolver [23], since it is not a head-to-head comparison. Take AnswerBot as an example. First, the problem setting is different. AnswerBot summarizes multiple SO threads, while ASSORT summarizes key points in a single post. Second, even if we adapt AnswerBot to only summarize a single post, AnswerBot can only produce a summary by selecting paragraphs. By contrast, ASSORT produces a more fine-grained summary by selecting sentences. Thus, AnswerBot always generates longer and more coarse-grained summarizations than ASSORT, leading to low precision on the benchmark.

3.6.3 Evaluation Metrics

We use three metrics—precision, recall, and F1 to measure the effectiveness of our methods and the comparison baselines in SO post summarization. Each metric is calculated at the sentence level. Given a set of summative sentences in a set of SO posts G, let M be the set of summative sentences selected by an extractive summarization model. The precision of the model is calculated as $|G \cap M|/|M|$. And the recall of the model is defined as $|G \cap M|/|G|$.

Furthermore, we measure the F1 score, which combines the precision and recall of a model into a single metric by taking their harmonic mean.

We make sure both ASSORT$_S$ and ASSORT$_{IS}$ and all the baselines are evaluated with the same test set (i.e., 304 posts and their summaries) to make the comparison fair. A 10-fold validation is performed when calculating the metrics for our approaches and all the baselines.

3.6.4 User Study Design

To answer RQ4, we conduct a within-subjects user study to evaluate the summary quality. We recruit 12 graduate students through the department mailing list from an R1 university. Participants have an average of four years of programming experience. We randomly selected 40 answer posts from our dataset, including 13 *how-to questions*, 13 *conceptual questions*, and 14 *bug-fixing questions*. These tasks were randomly sampled from the test set while ensuring a balanced number of answers in each category. In each user study, we select 10 out of the 40 answer posts and ask the participant to review their summaries. We counterbalance the post assignment so that each post is evaluated by three different participants.

For each answer post, participants first report their expertise of the concepts in the question on a 7-point scale. 1 means "Haven't even heard of it before" and 7 means "I am an expert". Then, they will be provided with the question post, the answer post, and the summaries generated by ASSORT$_S$, ASSORT$_{IS}$, and *BertSum (fine-tuned)*. Specifically, we select *BertSum (fine-tuned)* as our baseline in the user study since it performs the best among all six baselines in the quantitative experiment. The participants first read the question to understand the context and then read the answer post to understand the content to be summarized. The participants are then asked to read all three summaries and evaluate the quality of these summaries by answering the following five multiple-choice questions.

(1) *Which summary provides the most helpful information?*

(2) *Which summary provides the best overview of the post?*

(3) *Which summary provides the most comprehensive information?*

(4) *Which summary is the most concise without being incomplete?*

(5) *Which summary do you prefer to see in practice?*

To mitigate bias, the order of summaries is randomized to mitigate bias and we also do not reveal which model generated which summary. At the end of the user study, participants answer several open-ended questions about whether they wish to see SO post summaries when browsing SO and what kinds of characteristics an ideal SO summary should possess.

3.7 Results

3.7.1 RQ1: SO Post Summarization Accuracy

Table 3.5 shows the summarization accuracy of ASSORT$_S$ and the baselines after 10-fold validation is performed. ASSORT$_S$ achieves the best results in all three metrics. Furthermore, the domain shift from general text corpora to the SO post corpus is non-trivial. The original *BertSum* model, which is trained on news articles and their summaries from CNNDailymails, only achieves an F1 score of 53%. While fine-tuning *BertSum* with 2,424 labeled SO posts increases the F1 score from 53% to 58%, it is still 13% below ASSORT$_S$. Given that *BertSum* also uses BERT for sentence encoding, this result implies that directly reusing a pre-trained model, even with finetuning, is not an optimal solution. Incorporating domain-specific features and ensembling based on question types are necessary to improve the effectiveness of SO post summarization.

Table 3.5. Comparison of ASSORT$_S$ and baselines

	Precision	**Recall**	**F1**
Assort$_S$ ★	**0.73**	**0.69**	**0.71**
BertSum ★	0.47	0.60	0.53
BertSum (fine-tuned) ★	0.51	0.67	0.58
wordpattern ◇	0.40	0.03	0.06
simpleif ◇	0.39	0.15	0.21
contextif ◇	0.39	0.15	0.22
lexrank ☆	0.61	0.45	0.52

★: DL-based, ◇: heuristics-based, ☆: unsupervised

3.7.2 RQ2: Ablation Study

Table 3.6 shows the ablation study results. On the one hand, ablating the BERT embedding leads to the largest accuracy degradation, 17% in the F1 score. This indicates that incorporating deep contextualized embeddings from a language model is critical for a domain-specific task such as SO post summarization. On the other hand, only including BERT is also not sufficient. Removing each of the other three components, which are specifically designed to account for the unique characteristics of SO posts, leads to a non-trivial decrease in the F1 score. Specifically, the removal of domain-specific features decreases the F1 score by 6%. This indicates that deep contextualized embeddings alone cannot cover the unique structural and linguistic patterns that distinguish SO posts from general text data. Removing the question classifier decreases the F1 score by 9%. This indicates that accounting for different linguistic norms in answers to different types of questions indeed helps. Removing the ensemble mechanism leads to a decrease of 4% in the F1 score. This indicates that the ensemble mechanism can help to alleviate the influence of question classification errors.

Table 3.6. Contribution of each component in ASSORT$_S$

	Precision	**Recall**	**F1**
Assort$_S$	**0.73**	**0.69**	**0.71**
—w/o BERT	0.61	0.48	0.54
—w/o Domain-specific features	0.70	0.61	0.65
—w/o Ensemble	0.68	0.66	0.67
—w/o Question classifier	0.61	0.63	0.62

3.7.3 RQ3: Supervision vs. Indirect Supervision

Figure 3.5 compares the accuracy of ASSORT$_S$ and ASSORT$_{IS}$ when various amounts of training data are available. Given that ASSORT$_{IS}$ only uses pre-trained models and does not require any training data, the accuracy of ASSORT$_{IS}$ is constant (65%) in these settings.

When all training data (i.e., 2,424 posts and their summaries) is available, ASSORT$_S$ outperforms ASSORT$_{IS}$ by 6%. This indicates that directly training a supervised model is a better choice when there are sufficient training data. However, when 20% or less of the original training data is available, ASSORT$_{IS}$ outperforms ASSORT$_S$. Therefore, when the training data is small, indirect supervision can be a better option than directly training a model with small training data. Furthermore, with an F1 score of 65%, ASSORT$_{IS}$ outperforms all six comparison baselines by 7% to 59%. This result is significant since the best comparison baseline is fine-tuned on all training data. This implies that indirect supervision can be a promising yet low-cost alternative compared with unsupervised approaches and model fine-tuning.

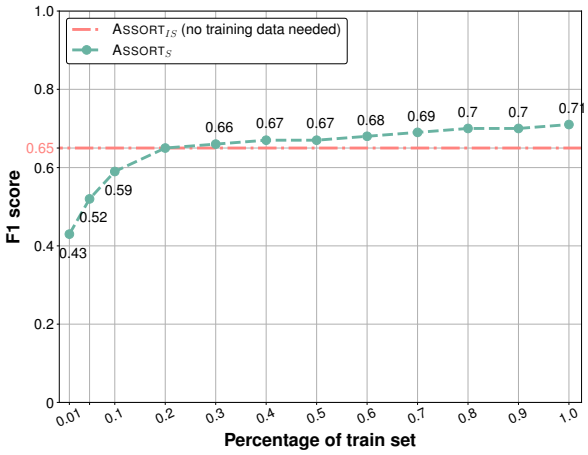

Figure 3.5. The F1 score of the direct vs. indirect supervision methods when various amounts of training data are available.

3.7.4 RQ4: Human Evaluation

Figure 3.6 shows the choice of participants over the summaries generated by ASSORT$_S$, ASSORT$_{IS}$, and *BertSum (fine-tuned)* in five aspects. Overall, participants strongly preferred summaries generated by ASSORT$_S$ or ASSORT$_{IS}$ over *BertSum (fine-tuned)*. Between ASSORT$_S$ and ASSORT$_{IS}$, more participants found summaries generated by ASSORT$_S$ more

helpful, concise, and practical. Yet more participants found summaries generated by ASSORT$_{IS}$ more comprehensive and providing a better overview of the post.

Since each answer post and its summaries were reviewed by three participants, we further analyzed the consistency among those participants when they answered each multiple-choice question. In 60% of the cases, the three participants assigned to the same post consistently chose summaries generated by ASSORT over *BertSum (fine-tuned)* in a multiple-choice question. In 91% of cases, at least two out of three participants consistently chose either ASSORT$_S$ or ASSORT$_{IS}$ over *BertSum (fine-tuned)*. However, the choices between ASSORT$_S$ and ASSORT$_{IS}$ were not very consistent among participants assigned to the same post. In only 40% of cases, all three participants consistently chose ASSORT$_S$ or ASSORT$_{IS}$. This implies the participants chose between ASSORT$_S$ and ASSORT$_{IS}$ largely based on their personal preference. Neither ASSORT$_S$ or ASSORT$_{IS}$ really triumph over each other.

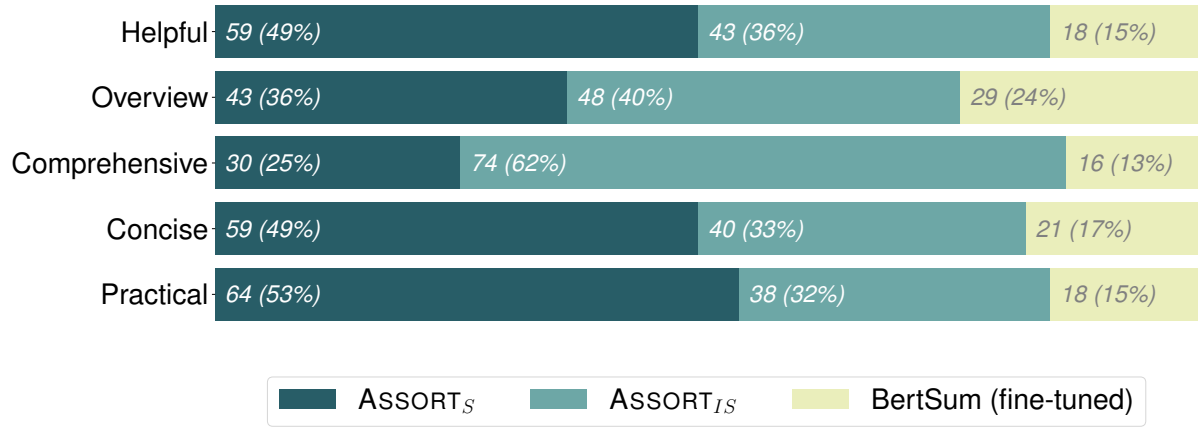

Figure 3.6. Participants' choices over different types of summaries in five aspects

We also investigated how participants' expertise on concepts in a SO question affects their preferences over post summaries. As described in Section 3.6.4, participants first reported their expertise on a 7-point scale. We categorized their expertise as "novice" if their rating is 1-2, "regular" if their rating is 3-5, and "expert" if their rating is 6-7. Figure 3.7 shows the choices of participants with different levels of expertise. We observed an tendency of favoring ASSORT$_S$ among novices while an tendency of favoring ASSORT$_{IS}$ among experts. One plausible reason for this is that novices prefer to see short summaries and feel overwhelmed

if a summary contains too much information, while experts prefer to see more comprehensive information and feel less overwhelmed. Pearson's Chi-square test of independence shows that the choice difference among participants with different levels of expertise for post summaries is statistically significant ($p = 0.0006$).

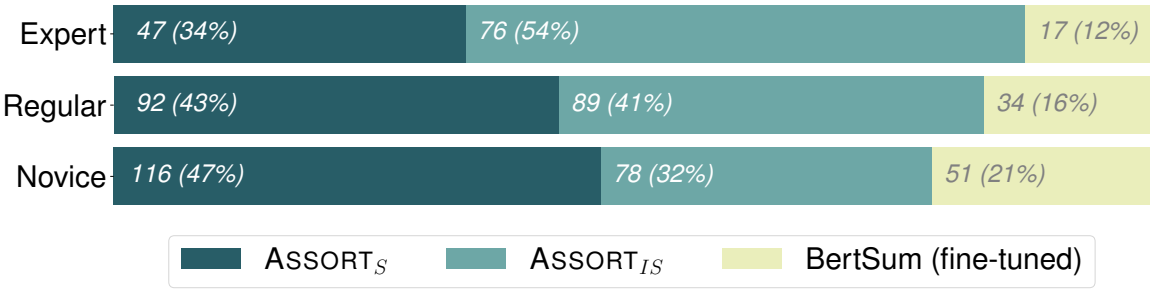

Figure 3.7. Choices of participants with different levels of expertise

In the final survey, 84% of participants confirmed they would like to see post summaries when browsing SO posts. For example, P4 elaborates, "*seeing a concise yet informative summary can help me quickly decide whether a post is worth reading or not.*" Another participant (P10) said, "*most information in a long post is useless to me, what I want is the solution and solution only*".

3.8 Discussion

Our experiment shows that only fine-tuning a general text summarization model such as [31] is insufficient. Our supervised approach, ASSORT$_S$, achieves significantly higher summarization accuracy by incorporating specific designs for SO posts, such as question classification and domain-specific features. This may carry a more general implication—as we reuse models from NLP or ML, we should consider renovating their model architectures and adapting them to account for the unique characteristics of SE tasks.

Our indirect supervision approach, ASSORT$_{IS}$, is proposed to address the challenge of lacking labeled data in SO post summarization. ASSORT$_{IS}$ has achieved reasonable accuracy and is proven to be a better choice when labeled data is insufficient, e.g., less than 485 posts in our experiment setting. Furthermore, our user study shows that while summaries generated

by ASSORT$_{IS}$ and ASSORT$_{S}$ were deemed good in different aspects (e.g., *comprehensiveness* vs. *conciseness*), there was no significant preference difference in general. This implies that ASSORT$_{IS}$ could be an acceptable yet low-cost solution in practice.

Both ASSORT$_{S}$ and ASSORT$_{IS}$ can be generalized to summarize other types of SE documents, such as API documentation, tutorials, and bug reports. Since ASSORT$_{IS}$ only uses pre-trained models, it can be reused as-is. By contrast, ASSORT$_{S}$ would benefit from several extensions, e.g., re-training a topic classifier rather than a question classifier, re-designing some domain-specific features based on the linguistic norms in the target corpus. Furthermore, since ASSORT$_{IS}$ does not require any labeled training data, it can also be used as a starting point to explore the feasibility and opportunities of summarizing other types of SE documents.

Threats to validity. In terms of internal validity, labeling summative sentences in SO posts is a subjective task. Therefore, different labelers will not necessarily select the same set of summative sentences from the same post. In this chapter, two labelers expanded SOSum with 785 answer posts retrieved under 365 SO questions. We strictly follow the labeling procedure established in [17]. The Cohen's kappa score is 0.72 on the initial labeling results before discussion, which indicates moderate agreement. Most disagreements were resolved after discussion. In the final dataset, only those sentences that both labelers agreed on are labeled as summative sentences.

In terms of external validity, since our dataset is constructed by retrieving posts under the most popular questions, it inevitably favors popular tags, such as Java and Python. Our dataset contains 1,089 unique tags, while there are more than 50K tags on Stack Overflow. The accuracy of ASSORT$_{S}$ may drop a bit with unfamiliar topics due to unseen terminologies. Furthermore, our current model is only trained with SO posts to three common types of questions, since we aim to develop a proof of concept and demonstrate its feasibility for the scope of this chapter. To support other types of questions, one should enrich our dataset and re-train the model to obtain optimal accuracy. Finally, the accuracy of ASSORT$_{S}$ may also decrease when reused *as-is* for other types of SE documents. This is because ASSORT$_{S}$ includes unique designs for SO posts, which is not applicable to other types of SE documents.

In terms of construct validity, our question classifier does not always make the correct prediction. Sometimes, the boundary between different question categories can be blurred with questions like *"How to fix headers-already-sent error in PHP?"*. This question should be classified as *bug-fixing*, since it mentions an error. However, it is classified as *how-to* by our question classifier since the question title contains a phrase *"how to"*. To mitigate this threat, ASSORT$_S$ includes an ensemble mechanism that merges predictions from separate summative sentence prediction models. Another threat to construct validity lies in the user study design. In the current design, participants were only asked to evaluate the quality of the post summaries with regard to the SO question. However, the usefulness of these summaries when being used to solve real programming tasks remains to be determined.

Future work. Currently, ASSORT$_S$ only uses a feedforward neural network as the classification head on top of BERT. Prior work [31] has shown that the choice of classification heads has an influence on model accuracy. It would be worthwhile to further experiment with other types of classification heads, such as transformers and LSTMs. Furthermore, we have only considered single-document summarization in this chapter. It would be interesting to investigate how to perform multi-document summarization on a thread of SO posts. The challenge of multi-document summarization is not only to shorten the text, but also to organize information around the key aspects to represent diverse views. Compared with extractive summarization, abstractive summarization may be a more preferred solution, since it is capable of rephrasing sentences from different posts and generating gradual transitions. Diversity-based ranking algorithms such as Maximal Marginal Relevance (MMR) [167] can also be considered here.

3.9 Summary

In this chapter, we presented ASSORT, a framework for Stack Overflow post summarization built around an indirectly supervised method, ASSORT$_{IS}$. ASSORT$_{IS}$ shows that faithful summaries can be obtained without any labeled SO data: a pre-trained out-of-domain summarizer produces a draft that is unreliable under domain shift, and a pre-trained natural language inference model then serves as an external guide that traces the draft back to

sentences of the original post, turning an unfaithful abstraction into a grounded extractive summary. ASSORT$_{IS}$ outperforms six existing techniques by at least 7% in F1 score. Its supervised reference point, ASSORT$_{S}$, trained on labeled SO data with domain-specific features and question-type-aware ensembling, sets the ceiling at 13% above the baselines—yet when less than 20% of the labeled data is available, ASSORT$_{IS}$ overtakes it, confirming that externally guided indirect supervision is the more practical recipe in low-resource domains. A user study with 12 participants further confirmed that developers preferred summaries produced by both methods over the best baseline, with no significant preference between the two.

With respect to the thesis statement, this chapter demonstrates that external guidance is an effective lever against hallucination: rather than hoping a model internalizes faithfulness through training, an external verifier can re-anchor its output to the source, and the result is summaries that developers measurably trust. However, ASSORT operates on one post at a time and selects sentences rather than composing new text. Many software engineering tasks demand more: consolidating knowledge scattered across *many* documents into a coherent, newly written artifact. That setting gives a generative model far more freedom—and far more opportunity to hallucinate. The next chapter takes on this harder setting: generating complete API documentation from crowdsourced knowledge, where we complement external guidance with a second mitigation strategy, self-reflection.

4. MITIGATING HALLUCINATION WITH RETRIEVAL AND SELF-REFLECTION: AUTOMATING API DOCUMENTATION FROM CROWDSOURCED KNOWLEDGE

The previous chapter showed that external guidance can keep a summarizer faithful when condensing a single document. This chapter scales the challenge up: generating a complete API document by consolidating knowledge scattered across thousands of crowdsourced posts. Here, extraction alone no longer suffices—the output must be composed, deduplicated, and organized by an LLM, which reintroduces the very risk we set out to eliminate: the model may fabricate API behavior that no post ever described. We present AutoDoc, an LLM-based documentation pipeline that combines both mitigation strategies of the thesis statement. For *external guidance*, AutoDoc grounds generation in evidence retrieved by a dense passage retrieval model fine-tuned on Stack Overflow data, so that every knowledge snippet originates from a concrete post. For *self-reflection*, AutoDoc prompts the LLM to validate each extracted knowledge snippet against its source post and discards content that cannot be supported, before a final summarization step removes redundancy.

4.1 Introduction

API documents are critical for software developers to write and debug code [44, 60, 168, 169]. However, previous studies have shown that modern libraries and frameworks are plagued with incomplete and low-quality API documents [2, 18, 170–173]. Meanwhile, popular Q&A websites such as Stack Overflow (SO) have accumulated a wealth of information on API usages [60]. Compared with official API documentation, Q&A posts are more up-to-date and comprehensive, covering various undocumented issues developers have encountered in practice.

However, searching and sifting through SO posts for specific API knowledge is time-consuming, since such knowledge is often buried in many lengthy SO threads. To facilitate efficient knowledge retrieval from SO posts, several approaches have been proposed to extract API-related information from SO posts automatically [44, 53, 56, 174]. Existing approaches

are primarily heuristic-based or rely on supervised ML models. Such methods either lack the flexibility to extract knowledge from sophisticated natural language narratives or require a significant amount of labeled data. Curating such datasets requires significant human effort.

In this chapter, we propose a new approach called AutoDoc that leverages Large Language Models (LLMs) to extract API knowledge from SO posts. AutoDoc generates well-organized API documents with seven types of API knowledge defined in the literature [18, 60, 175]. Figure 4.1 shows the pipeline of AutoDoc. First, given an API name and SO posts that are tagged with the API's library filtered from the entire SO data dump, AutoDoc uses a Dense Passage Retrieval (DPR) [19] to identify relevant SO answer posts that may contain any of the seven types of API knowledge. Since SO posts have unique linguistic patterns compared with general text, we fine-tuned the original DRP model with SO data. Then, with the relevant posts retrieved, AutoDoc uses GPT-4o to extract API knowledge from these posts. Since LLMs may generate misinformation (i.e., *hallucination*), AutoDoc performs LLM-based self-checking to infer the correctness of the extracted knowledge. Finally, AutoDoc prompts GPT-4o again to remove duplicates from the generated API documents. This step is necessary because the same knowledge can be extracted from multiple posts, causing information redundancy.

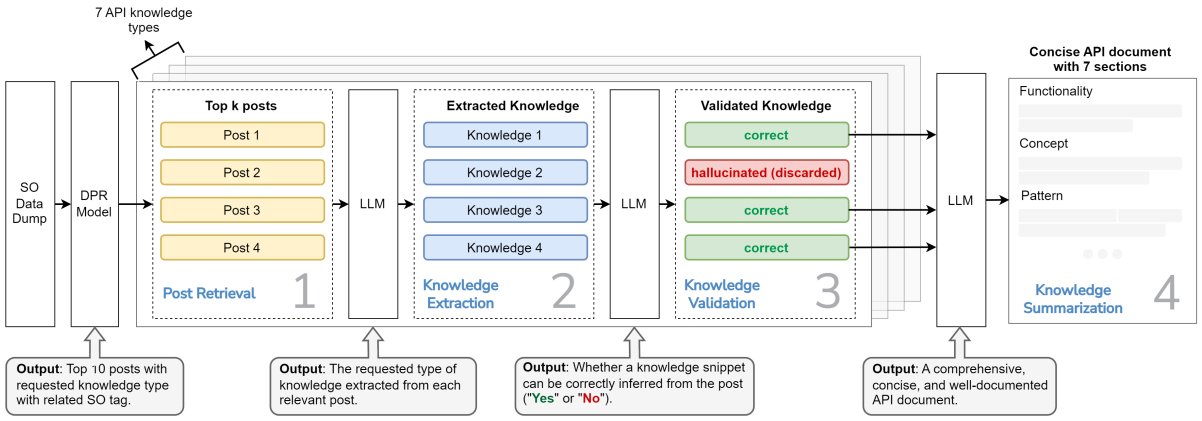

Figure 4.1. An Overview of the AutoDoc pipeline.

We evaluated AutoDoc against five baselines, including two existing approaches in automated API documentation [44, 176], and three prompting methods using GPT-4o [3]. We

measure the percentages of knowledge snippets that are correct, unique, and do not overlap with the official API documents in the generated documents. We conducted experiments on a benchmark of 48 APIs with different popularity levels from Java, Android, Kotlin, and TensorFlow. The results show that API documents generated by AutoDoc were up to 77.7% more correct and 9.5% more unique than API documents generated by the five baselines. Furthermore, on average, 34.4% knowledge snippets in API documents generated by AutoDoc are not documented in the official API documents.

We also conducted an ablation study to measure the contribution of individual components of AutoDoc. Our experiment results show that fine-tuning can improve DPR performance by 13%. Without the knowledge validation component, the accuracy of API documents generated by AutoDoc decreases by 3.7%. Without the knowledge summarization component, 31.7% more knowledge in documents generated by AutoDoc is duplicated. Furthermore, we replaced GPT-4o with three open source models (that is, Llama 3-70B-Instruct, Llama 2-70B-Chat, and Mistral-7B-v0.3) to assess the sensitivity of AutoDoc to the choice of LLM. We found that while larger LLMs produce higher-quality API documents, AutoDoc enables smaller open-source models to achieve comparable results. Specifically, the smallest LLM in our study, Mistral-7B-v0.3, can generate API documents that are 84.9% accurate, 81.6% unique, and contain 34.1% knowledge snippets that are not covered by official API documents.

Finally, we conducted a user study with 12 participants to evaluate the usefulness of the API documents generated by AutoDoc. Inspired by the user study design from prior work [177–180], we asked the participants to compare API documents generated by AutoDoc and two baseline approaches in terms of *helpfulness*, *comprehensiveness*, *conciseness*, *overview*, and *practice*. In the study, 80% participants preferred API documents generated by AutoDoc in all five dimensions.

In summary, we make the following contributions:

- We proposed a novel LLM-based pipeline for automated API documentation with a knowledge validation component that mitigates hallucination and a fine-tuned DPR model that effectively retrieves SO posts with API knowledge from SO data dump.

- We conducted a comprehensive evaluation, including a system-level comparison of AutoDoc against five baselines, a detailed assessment of each component in AutoDoc, and a user study with 12 participants.
- We have made the code and datasets publicly available to support reproduction and future research [181].

4.2 Motivating Example

This section illustrates how API documents generated by AutoDoc can provide a comprehensive view of different aspects of an API and reduce the information foraging effort during software development. Suppose a developer, Bob, is developing an Android application. Bob is implementing an image processing feature that extracts grayscale pixel values from RGB images using the `OpenCV` library for Android. However, after a quick search on Google, he realizes that `OpenCV` for Android is sparsely documented compared to its desktop counterpart and many APIs are not clearly explained. Bob decides to search on Stack Overflow, but he finds that there are many related posts. Sifting through these posts is time-consuming. After 15 minutes comparing different SO posts, Bob finally finds a post that describes how to extract grayscale pixels from a small region of an image using the `org.opencv.core.Mat` API.[1]

Another feature of Bob's application requires setting alarms that trigger specific actions. Bob decides to use the `AndroidAlarmManager` API in Flutter. After searching on Google, he realizes that, despite the abundant online information, he still needs to compare different posts to find the solution he desires manually. This is because many posts talk about irrelevant tasks like executing a periodic function with Android Alarm Manager. After comparing multiple SO threads for 10 minutes, Bob finds a solution on Stack Overflow.[2] The solution involves ensuring that the callback function for the alarm is static, a detail not mentioned in the official documentation.

[1] ↑https://stackoverflow.com/a/32058926
[2] ↑https://stackoverflow.com/a/71566535

Finally, Bob tries to support dual SIM capabilities using `MultiSimTelephonyManager` API. However, the official Android SDK documentation provides little information about this. After searching on Stack Overflow, Bob realizes many SO posts returned by the search query talk about irrelevant tasks such as changing data subscription (SIM card) in multi-sim Android phones or getting the SIM-ID of both SIMs in Android. Therefore, he needs to go through them one by one. After spending another 15 minutes reading multiple SO threads, he discovers a post highlighting the use of the `MultiSimTelephonyManager` API as a solution for accessing dual SIM information.[3] Using the MultiSim library, Bob can access crucial SIM data like IMEI and IMSI, overcome Android SDK constraints, and extend support to dual SIM users.

Bob notices that this whole process of information foraging—*searching online, and reading and comparing SO posts*—is very cumbersome and time-consuming. Therefore, he decides to give AutoDoc a try. He starts with generating an API document for `org.opencv.core.Mat` via AutoDoc. The document generated by AutoDoc for `org.opencv.core.Mat` has ten sections, one for each type of API knowledge with a series of knowledge snippets extracted from SO posts. Since Bob cares about how to use the API, he first goes to the *Functionality* section. He soon finds the desired knowledge on converting RGB images to grayscale. Then, Bob generates API documents for the `AndroidAlarmManager` API in Flutter and the `MultiSimTelephonyManager` API with AutoDoc. The documents generated by AutoDoc for these two APIs contain seven sections with the same seven section titles. Reading the section titles, Bob quickly found the desired knowledge in the respective documents. Due to the page limit, we put documents generated for `AndroidAlarmManager` and `MultiSimTelephonyManager` in the Supplementary Material.

4.3 Definition of API Knowledge Types

AutoDoc aggregates different types of API knowledge shared on Stack Overflow into a comprehensive, well-organized document. Thus, before designing AutoDoc, we first performed a literature review to identify the types of API knowledge that developers need

[3]↑https://stackoverflow.com/a/41544422

when learning and using APIs [18, 45, 55, 60, 135, 182–186]. We summarize seven types of knowledge below.

- *Functionality* describes the actions or operations that an API can perform. Several studies have identified it as the most desired knowledge on Stack Overflow [18, 60, 135].
- *Concept* refers to the abstract ideas and terminologies that an API aims to model. Previous studies have shown that knowing the concepts helps developers recognize relevant information and understand the purpose of the API [18, 60, 135, 182, 183].
- *Pattern* illustrates common use cases and code examples for using an API. Previous studies have shown that API usage patterns are essential for developers to learn new APIs [18, 60, 182, 186]. Specifically, in a user study with 54 participants, API usage patterns were found to significantly improve task completion rates in multiple API-related programming tasks [182].
- *Directive* provides guidelines on the proper use of an API, including best practices to follow and actions to avoid [18, 45, 60, 184, 185]. A previous study has identified directive knowledge as the third popular knowledge type in JDK documents [18].
- *Performance* refers to an API's time and memory efficiency. This type of knowledge is particularly useful in industrial settings with high standards for code performance [18, 55, 60]. For example, a previous study systematically surveyed several major APIs, including the JAVA standard library, Eclipse, and JMS, and found that performance knowledge for API usage is one of the most needed guidelines by developers [55].
- *Environment* specifies the necessary conditions, system requirements, or configurations under which an API can function correctly [18, 170, 183]. While developers often need this type of knowledge to perform their work [170, 187], it is barely covered by official API documents [18].
- *Alternative* refers to other APIs with similar functionality, which can be considered as replacements or complementary options [55, 60]. For example, Liu et al. found that 12.5% of developers asked about alternative APIs on Stack Overflow [60].

Table 4.1 illustrates each of the seven knowledge types with a concrete example for the TensorFlow API `tf.gather`, alongside the query template that AutoDoc later uses to retrieve posts likely to contain that type of knowledge (Section 4.4.1).

Table 4.1. Examples of different knowledge types and corresponding prompts to retrieve relevant posts

Knowledge Type	Examples	Dense Retrieval Query
Functionality	`tf.gather` is used to select tensor elements at specific indices.	*"How does [tf.gather] work? [API_description]"*
Concept	`Tensor` is essentially a high-dimensional array.	*"What are the concepts behind [tf.gather]? [API_description]"*
Performance	`tf.gather` has overhead when used on large tensors.	*"How is the performance of [tf.gather]? [API_description]"*
Directive	When using `tf.gather`, ensure indices are within the shape of the input tensor.	*"What should I ensure when using [tf.gather]? [API_description]"*
Pattern	`tf.gather` is commonly used in embedding lookup operations.	*"When is [tf.gather] commonly used? [API_description]"*
Environment	`tf.gather` requires TensorFlow installed and supports both CPU and GPU execution.	*"What are the environment requirements for [tf.gather]? [API_description]"*
Alternative	Alternatives to `tf.gather` include `tf.scatter_nd` and `tf.index_select`.	*"What are the alternatives to [tf.gather]? [API_description]"*

4.4 Approach

This section describes the four key components of AutoDoc: (1) *relevant post retrieval*, (2) *knowledge extraction*, (3) *knowledge validation*, and (4) *knowledge summarization*, as shown in Figure 4.1.

4.4.1 Relevant Post Retrieval

Table 4.2. Prompt to extract Functionality knowledge of Android `ActionBar` from Post 62606723 and the output from GPT-4o.

CONTEXT
You are an API expert who excels at extracting, validating, and summarizing API knowledge from Stack Overflow posts as instructed.
INSTRUCTION
Extract functionality knowledge of ActionBar from this post, including knowledge you can infer from this post. If the post does not contain such knowledge, reply *"No such knowledge"* and nothing more.
API DESCRIPTION
The ActionBar class in Android provides a dedicated space at the top of an activity for displaying the app's title, navigation controls, and other interactive elements like action buttons.
KNOWLEDGE TYPE DESCRIPTION
Functionality knowledge describes the actions or operations an API can perform.
EXAMPLE KNOWLEDGE
tf.gather is used to select tensor elements at specific indices.
POST
Use one of the ".NoActionBar" theme and set the windowActionBar attribute to false. Beginning with the API level 21 action bar may be represented with any toolbar widget, within the application layout. An application may choose to designate a Toolbar as the action bar for an Activity using the setSupportActionBar() method.
GPT-4o Response
1. Beginning with API level 21, the action bar can be represented by any Toolbar widget within the application's layout. 2. To designate a Toolbar as the action bar for an Activity, use the setSupportActionBar() method.

Since the SO data dump contains too much information to be passed into the LLM in one shot, AutoDoc first leverages Dense Passage Retrieval (DPR) [19] to identify SO answer

posts that may contain each type of knowledge of a given API. DPR is a neural retriever that leverages contrastive learning to distinguish relevant and irrelevant information to a user query. Compared with other token-based or embedding-based retrieval methods, DPR has demonstrated superior performance and scalability to massive text corpora [19].

The original DPR model is trained on five popular open-domain QA datasets [19].Given the unique linguistic characteristics of SO posts, we fine-tuned the DPR model on a curated SO dataset to address the domain shift issue. We constructed the fine-tuning dataset based on the insight that finding relevant posts with API knowledge is similar to finding answers to an SO question. Specifically, the dataset contains 144K SO question posts in the past five years and 175K corresponding answer posts with scores larger or equal to 5. All data were filtered from the official Stack Exchange Data Dump [188]. During the fine-tuning process, we used each question title as a query. We used the answer posts to each question as positive passages and the answer posts to other questions as negative passages for contrastive learning. We used the Haystack framework for fine-tuning. We fine-tuned the DPR model for one epoch with a batch size of 16 and a gradient accumulation step of 8. The fine-tuning finished in 2 hours with 8 NVIDIA A5500 GPUs.

For each API knowledge type of each API, AutoDoc uses the DPR model to retrieve the top 10 answer posts that may contain such knowledge from the SO data dump. We chose the number 10 to keep the API documents generated by AutoDoc informative but not overwhelming. Queries used to retrieve posts that contain each type of knowledge can be found in our GitHub repository [181].

4.4.2 Knowledge Extraction

Given the posts retrieved for each type of knowledge from the previous step, AutoDoc uses GPT-4o to extract the corresponding knowledge from each post. We followed prior work [189, 190] to design a prompt template for few-shot prompting. Table 4.2 illustrates the prompt used to extract the functionality knowledge of the `ActionBar` API in Android from a Stack Overflow post [191] and the response from GPT-4o. The prompt includes six fields: (1) *context information* that helps the model understand the domain we want it to

operate within, (2) *instruction* that asks the model to extract the required knowledge, (3) *API description* that describes the target API in one sentence, (4) *knowledge type description* that specifies which type of knowledge to extract, (5) *knowledge example* that provides an example of the expected type of knowledge, (6) *post* where knowledge should be extracted. These API descriptions were summarized by GPT-4o from the official API documents rather than manually written by the authors to avoid bias. The first author manually evaluated the correctness of these API descriptions and found no errors. We present all API descriptions, API type descriptions, and knowledge examples on our GitHub repository [181].

Specifically, in the *instruction* field, the prompt asks the LLM to first determine whether the given post contains API knowledge or not before extraction. If the post does not contain the required API knowledge, the knowledge extraction model will output *"No such knowledge"*. This design is inspired by the observation that the DPR model sometimes retrieves irrelevant SO posts. If the LLM responds *"No such knowledge"*, AutoDoc will consider this post irrelevant and discard it. Table 4.3 shows an example of an incorrectly retrieved post for functionality knowledge of the `SAXParser` API. This post only mentions the API name but does not contain any actual functionality knowledge.

Table 4.3. A post that is incorrectly retrieved by the DPR model and rejected by the knowledge extraction component.

Incorrectly retrieved post [Functionality, SAXParser]
I have resolved this issue, it was due to another bug in my own code. So it had nothing to do with the SAXParser. Thanks for all the help!
GPT-4o response
No such knowledge.

4.4.3 Knowledge Validation

Previous studies [9, 10, 192] have shown that hallucination remains a major hurdle for adopting LLMs in downstream tasks. Since developers rely on API documentation to build software, it is critical to mitigate LLM hallucinations and ensure the reliability of generated API documents. Recent work [193] shows that LLMs can detect its own hallucination via self-

reflection. Inspired by this finding, AutoDoc prompts the LLM to verify the correctness of the knowledge given in the post and discard the extracted knowledge that failed the verification. Table 4.4 illustrates the prompt used to validate a performance knowledge snippet of `ByteArray` extracted from a SO post [191] and the response from GPT-4o. The prompt includes five fields: (1) *context information* that helps the model understand the domain we want it to operate within, (2) *instruction* that asks the model to validate the given knowledge based on the given post, (3) *API description* that describes the target API in one sentence, (4) *knowledge* that needs to be validated, and (5) *post* based on which the knowledge should be validated.

Table 4.4. The prompt for validating a *Performance* knowledge of Kotlin `ByteArray` from Post 65199270.

CONTEXT
You are an API expert who excels at extracting, validating, and summarizing API knowledge from Stack Overflow posts as instructed.
INSTRUCTION
Can this knowledge of ByteArray be extracted from this post? If so, reply "Yes". If not, reply "No". Just reply "Yes" or "No" and nothing more.
API DESCRIPTION
The ByteArray class in Kotlin represents an array of bytes, offering a fixed-size collection of byte values that can be accessed and manipulated by their indices.
EXTRACTED KNOWLEDGE
Reading all bytes of an InputStream into a ByteArray using the "readBytes()" extension method can be memory-inefficient for large inputs, as it loads the entire byte array into memory.
POST
The easiest way to make a ByteArray in Kotlin in my opinion is to use byteArrayOf(). It works for an empty ByteArray, as well as one which you already know the contents of.
GPT-4o Response
No.

Table 4.5. The prompt for summarizing knowledge of the `Model` API in Tensorflow.

CONTEXT
You are an API expert who excels at extracting, validating, and summarizing API knowledge from Stack Overflow posts as instructed.
INSTRUCTION
Summarize these knowledge snippets regarding `Model` into a concise and accurate list. Focus on the seven types of knowledge as described below. For each type, create a section in the final API document.
API DESCRIPTION
A TensorFlow `Model` class defines a neural network's layers and forward pass for training and inference.
ALL KNOWLEDGE TYPE DESCRIPTIONS
KNOWLEDGE_TYPE_DESCRIPTIONS
KNOWLEDGE LIST
KNOWLEDGE_TO_BE_SUMMARIZED

4.4.4 Knowledge Summarization

Since knowledge extracted from relevant posts may contain duplication, the knowledge summarization component is needed to eliminate redundant information. This component uses GPT-4o to summarize knowledge snippets extracted from multiple relevant SO posts into a concise API document. This step is necessary because the relevant SO posts may contain duplicated information, and the same post can be retrieved for different knowledge types of the same API. Table 4.5 illustrates the prompt used to summarize knowledge snippets of `Model` extracted from all relevant SO posts. The prompt includes five fields: (1) *context information* that helps the model understand the domain we want it to operate within, (2) *instruction* that asks the model to summarize the given knowledge snippets, (3) *API description* that describes the target API in one sentence, (4) *knowledge type descriptions* that explain each knowledge type, and (5) *list of knowledge* that need to be summarized.

4.5 Quantitative Evaluation

We conduct experiments to answer seven research questions below:

- **RQ1**: How accurately and comprehensively can AutoDoc generate API document compared to existing approaches, and how much knowledge can be found in the official documents?
- **RQ2**: What leads to the incorrect information in generated documents?
- **RQ3**: To what extent do documents generated by different approaches complement each other?
- **RQ4**: What is the impact of API popularity on the effectiveness of AutoDoc?
- **RQ5**: To what extent does each component contribute to the effectiveness of AutoDoc?
- **RQ6**: How sensitive is AutoDoc to the choice of LLM?
- **RQ7**: What is the impact of temperature setting on the effectiveness of AutoDoc?

4.5.1 Comparison Baselines

We compared AutoDoc against five baselines, including API caveat [176], SISE [44], and three prompting methods using GPT-4o [3]. We elaborate on each of them below.

- *SISE* [44] is an API document augmentation approach that selects insightful sentences with API knowledge from SO posts. It uses an SVM model that identifies insightful sentences based on both syntactic and semantic features. Since the source code and training data of SISE are not publicly available, we re-implemented SISE based on the approach descriptions in the paper. We curated a training dataset with 2,296 manually labeled sentences from SO posts related to the APIs used in our evaluation, as well as a test set with 460 labeled sentences. Our implementation of SISE achieves a precision of 62.5%, which is comparable to the precision of the original SISE (60%). Our implementation of SISE can be found in our GitHub repository [181].

- *API caveat* [176] is a heuristic-based approach for knowledge extraction from SO posts. It uses 19 linguistic patterns to extract insightful sentences in SO posts. The authors released the regex patterns used to capture these linguistic patterns, so we directly used these patterns in our paper.

- *GPT-4o* [3]. Instead of using the sophisticated pipeline in AutoDoc for API documentation, an alternative way is to directly prompt GPT-4o to generate an API document. We experimented with three prompting methods, including zero-shot prompting (ZSL), few-shot prompting (FSP), and chain-of-thought (CoT). Furthermore, to make it a fair comparison between GPT-4o and AutoDoc, we asked GPT-4o to generate a document that includes the seven types of API knowledge defined in Section 4.3 and included a description of each type of knowledge in the prompt. We present the prompts for these three GPT-4o baselines in our GitHub repository [181].

4.5.2 Evaluation Benchmark

We constructed an API benchmark with 48 APIs with different popularity levels and applied AutoDoc and baselines to generate documents for these APIs. These APIs were selected from Java SDK, TensorFlow, Android SDK, and Kotlin SDK. These four libraries covered different topics in software engineering. For example, while Java and Kotlin are used for general purposes, Android SDK is used to develop mobile apps, and TensorFlow is used to develop machine learning projects. We performed stratified sampling to select APIs with different popularity levels. API popularity was determined by its percentile ranking in GitHub code search. The popular APIs are from the top 10 percentile, the less popular ones are from 10 to 90 percentile, and the least popular ones are from the last 10 percentile. We randomly select 12 APIs for each of the 4 libraries. Among the 12 APIs, 4 are high-popularity APIs, 4 are medium-popularity APIs, 4 are low-popularity APIs. While 48 APIs may not sound like a large number, it requires manual assessment of 48 APIs $\times$ 6 approaches = 288 API documents, which contains 3,840 knowledge snippets. Furthermore, the size of our benchmark is comparable to the evaluation benchmark of SISE [44], which includes 10 Java APIs. We present the complete list of the 48 APIs on our GitHub repository [181].

4.5.3 Evaluation Metrics and Data Labeling

Inspired by existing research on API documentation [44, 171], we use three metrics to measure the effectiveness of AutoDoc. First, we follow SISE [44] to manually label the ***correctness*** of each knowledge snippet. Second, Treude et al. [171] find that a good API document should cover sufficient information briefly without redundant or unnecessary text. Thus, we measure the ***uniqueness*** by counting the unique knowledge snippets in a document. Third, Boudin et al. [194] show that it is important to provide information that developers cannot learn from the official documents. Thus, we measure the ***overlap*** between an automatically generated document and the official document. All metrics reported in this chapter are calculated as the total number of *correct*/*unique*/*overlapping* knowledge snippets across all API documents in a given category (e.g., all documents generated by AutoDoc), divided by the total number of knowledge snippets in those documents.

The first two authors labeled the API documents generated by different approaches in two rounds.

First round of labeling.

In the first labeling round, the two labelers independently identified and labeled all knowledge snippets in 5 API documents to familiarize themselves with the labeling tasks. These API documents were randomly sampled from 288 API documents generated by AutoDoc and five baseline approaches. They started by highlighting all knowledge snippets in an API document. Specifically, each sentence in the document was labeled in four dimensions: (1) *whether it is part of a knowledge snippet or not*, (2) *whether it is correct or not*, (3) *whether it is unique in the document or not*, and (4) *whether it overlaps with the official API document or not*. We measured the agreement level of the two labelers using Cohen's Kappa. The Cohen's Kappa scores for the four dimensions are 0.60, 0.69, 0.84, and 0.58 respectively. The labelers discussed their labels to establish the labeling guidelines outlined below:

For ***Correctness***, the labelers compared knowledge snippets in the generated API documents with online resources such as SO posts, blogs, tutorials, and official documentation. Partially correct snippets were considered incorrect. For example, a knowledge snippet that

describes a usage of an API that will cause errors in specific versions of that API without mentioning this limitation will be labeled as incorrect. Table 4.6 shows an incorrect *Pattern* knowledge of `ActionBar` and the correct version of the same knowledge snippet.

Table 4.6. Different knowledge examples.

Correct and Incorrect Knowledge Snippets
1. Set titles in the ActionBar using `setTitle()` [**Correct**]
2. Set titles in the ActionBar using `getSupportActionBar()` [**Incorrect**]
Duplicated Knowledge Snippets
1. Later versions of `UUID` do not require system coordination.
2. `UUID V2` does not require system coordination.
Overlapping Knowledge Snippet (Overlap highlighted)
1. In that case, you should define your layers in `__init__()` and you should implement the model's forward pass in `call()` when using `Model` [**Official documentation of `Model`**]
2. Use forward pass in `call()` when using `Model` [**Knowledge extracted by AutoDoc**]

For ***Uniqueness***, the labelers identified all duplicated knowledge snippets and reported the number of unique knowledge snippets in the document. A knowledge snippet was labeled to be duplicated if it completely paraphrased another snippet without providing any novel insight. Table 4.6 shows two duplicated knowledge snippets where both imply `UUID V2` does not require system coordination.

For ***Overlap***, a knowledge snippet from an auto-generated document was labeled as an overlap if it can be found in or inferred from the official API documents. Table 4.6 shows an example of overlapping knowledge for the `Model` API.

Second round of labeling

After they resolved the conflicts in their labels, the labelers started to label another five randomly sampled API documents following the established guidelines. At the end of the

second round, Cohen's Kappa scores for uniqueness, correctness, and overlap rose to 0.91, 0.95, and 0.88, indicating almost perfect agreement [155]. Therefore, the two labelers split and labeled the remaining 230 API documents separately. In total, the labeling process takes 102 person-hours.

4.6 Quantitative Results

4.6.1 RQ1: Correctness and Comprehensiveness

In Table 4.7, Column Correctness shows the percentage of correct knowledge snippets in a generated document on average. Column Uniqueness shows the percentage of unique knowledge among the correct knowledge snippets in each document. Column Overlap shows the percentage of correct knowledge snippets that have been covered by the official API documents. Column # of snippets shows the total number of knowledge snippets in each document, regardless of correctness. Column Avg. Length shows the average length of the generated documents in characters.

In terms of ***correctness***, AutoDoc generated the most accurate documents (96.2%), outperforming traditional, non-LLM-based approaches (i.e., SISE and API caveat) by a large margin. The reason for the low performance of SISE and API caveat is that they are based on relatively simple rules or features and lack the capability to reason about the deep semantics of the natural language discussions in SO posts. Even though the SVM model used in SISE achieved 62.5% accuracy in extracting insightful sentences on the original test set, its performance is not generalizable when applied to a broader scope of APIs and SO posts. This is a well-known generalizability issue of traditional, small models like SVM.

It is surprising to see that simply prompting GPT-4o still produces API documents with decent accuracy, even with zero-shot prompting (90.4%). It is known that GPT-4o is trained with a massive amount of Internet data, including official API documentation and Stack Overflow posts. Hence, this result implies that GPT-4o is very well-trained to memorize this data during pre-training. Yet supplementing external knowledge bases such as Stack Overflow posts still brings the benefit of generating more comprehensive API knowledge snippets. As shown in Column Overlap of Table 4.7, when extracting knowledge from SO

posts, AutoDoc generates API documents with significantly less overlap compared to relying on GPT-4o alone. Furthermore, among the three prompting methods, chain-of-thought prompting achieves the highest correctness rate. This indicates that guiding the LLM to perform explicit reasoning in the generation process helps reduce hallucinations.

In terms of ***uniqueness***, AutoDoc outperformed three prompting methods of GPT-4o by about 10%. This is because AutoDoc employs a knowledge summarization component that can effectively remove duplication in the document. Despite this, we noticed that API documents directly generated by GPT-4o contained more knowledge snippets compared to those generated by AutoDoc. To investigate why, we manually examined the API documents generated by GPT-4o and found that GPT-4o tended to generate many code examples to illustrate how to use the API, which belongs to the type of *pattern* knowledge. Thus, they were labeled as valid knowledge snippets and contributed to the higher number of knowledge snippets in the documents generated by GPT-4o.

In terms of ***overlap with official documents***, AutoDoc generates significantly fewer knowledge snippets that are covered by the official API documentation compared to the three prompting methods of GPT-4o. This is likely because GPT-4o relies on its internal knowledge to generate API documentation. Since GPT-4o is pre-trained on a massive amount of Internet data, including the official API documentation, GPT-4o has a tendency to spill out content it has seen during pre-training. By contrast, AutoDoc, SISE, and API caveat rely on Stack Overflow as an external knowledge base and extracted API knowledge from SO posts, which are less likely to overlap with the official API documentation.

In terms of ***document length***, documents generated by AutoDoc have an average length of 2,902 characters, which is about 34–47% shorter than the documents produced by the three GPT-4o prompting baselines (4,407 to 5,462 characters), while containing more correct and more unique knowledge. This confirms the effect of the knowledge summarization component: merging duplicated knowledge yields documents that are denser rather than merely shorter. The two non-LLM baselines produce documents of comparable length to AutoDoc, but with drastically lower correctness. We discuss opportunities for further improving conciseness in Section 4.8.

Table 4.7. Comparison of AutoDoc and baselines.

	Correctness	Uniqueness	Overlap	# of Snippets	Avg. Length (chars)
SISE	18.6%	90.1%	56.3%	25.9	2,473
API caveat	18.5%	91.3%	59.4%	24.7	3,276
GPT-4o (ZSP)	90.4%	85.9%	79.4%	32	5,419
GPT-4o (FSP)	92.4%	83.5%	87.8%	32.8	4,407
GPT-4o (COT)	93.2%	82.7%	82%	33.8	5,462
AutoDoc	96.2%	93.2%	65.6%	21	2,902

4.6.2 RQ2: Manual Error Analysis

While AutoDoc achieves 96.2% accuracy, 38 knowledge snippets still include incorrect information. To understand what leads to such incorrect information, the first authors manually analyzed the 38 incorrect knowledge snippets and labeled the potential reasons. Each snippet was examined by checking the initially retrieved SO posts and the intermediate outputs generated by AutoDoc at each step. Then, the first author discussed the analysis results with the other authors and categorized them into three types of errors.

SO post errors (13.2%)

These errors stem from incorrect or misleading information in the retrieved SO posts. GPT-4o reuses this incorrect information from the original SO post when generating the new document. For example, the post in Table 4.8 suggests: *"The point of UUIDs is that it doesn't matter who generates them..."*. This implies `UUID` does not require any system coordination, which is not true for `UUID` version 1 because it is generated using the MAC address and timestamp. GPT-4 reproduced this incorrect information when generating the document. 5 of the 38 incorrect knowledge snippets fall into this category.

Table 4.8. Error due to inaccurate SO post

An incorrect *Pattern* knowledge snippet for `UUID`
Their generation does not require system coordination, enhancing performance where distributed systems are involved.
Excerpt of SO post 73144845
The point of UUIDs is that it doesn't matter who generates them and there is no state you need to worry about.

Misinterpretation of SO posts (36.8%)

These errors occur when GPT-4o misrepresents accurate content from SO posts—either by overgeneralizing, misclassifying, or extracting misleading summaries. For example, as shown in Table 4.9, GPT-4o extracted a knowledge snippet claiming that `MediaPlayer` accepts input as long as it is encoded as an audio file. While this appears generally correct, it overgeneralizes the original intent of the SO post, which specifically discussed MP3 files. In fact, `MediaPlayer` only supports a limited set of audio formats recognized by Android (e.g., MP3 or MP4). 14 incorrect knowledge snippets fall into this category.

Table 4.9. Error due to misinterpretation of SO post

An incorrect *Concept* knowledge snippet for `MediaPlayer`
`MediaPlayer` accepts input as long as it is still encoded as an audio file.
Excerpt of SO post 70556830
Q: *Can I play MP3 files without a file extension in Android app?* **A:** `MediaPlayer` takes the input(the file) as long as it is still encoded as an audio file.

Hallucinations without SO support (50%)

These errors are fabricated information that cannot be traced back to any retrieved SO post or manually verified by any other information sources. Table 4.10 shows an example.

GPT-4o hallucinated the existence of a `delete()` method in the `LinkedList` API. In reality, the standard Java `LinkedList` class does not define any method named `delete()`, and no retrieved SO post mentioned such functionality. 19 of the 38 incorrect knowledge snippets fall into this category.

Table 4.10. Error due to hallucination without SO support

An incorrect *Functionality* knowledge snippet for `LinkedList`
`delete()` removes the last element from the linked list.
Excerpt of source post
No source post found.

4.6.3 RQ3: Complementarity of Baselines

To systematically evaluate the complementarity of AutoDoc with other baselines, we conduct a manual analysis on 10 randomly sampled APIs. For each API, the first author manually compared the knowledge snippets generated by AutoDoc with all five baselines.

Table 4.11. Knowledge from baselines uncovered by AutoDoc.

	# of Uncovered	# of Correct Snippets
SISE	2.7 (50.9%)	5.3
API caveat	4.8 (60%)	8
GPT-4o (COT)	23.4 (82.9%)	28.2
GPT-4o (FSP)	21.8 (68.3%)	30.9
GPT-4o (ZSL)	25 (78.6%)	31.8

Table 4.11 shows the number of knowledge snippets from baseline documents that are both correct and uncovered by AutoDoc. Compared with documents generated by GPT-4o, documents generated by API caveat and SISE contain fewer knowledge snippets that are not covered by documents generated by AutoDoc. This is because they extract API knowledge from the same set of SO posts like AutoDoc. Since GPT-4o is prompted to directly generate a document without referring to SO posts, GPT-4o mainly relies on its internal memory obtained from the pre-training data. As shown in Table 4.7, about 80%

of the knowledge snippets in GPT-4o's documents can be found in official documentation, whereas only 65.6% of the snippets in AutoDoc's documents align with official sources. This difference explains why GPT-4o introduces more knowledge snippets that are absent from AutoDoc's results.

Table 4.12. AutoDoc on APIs of different popularities.

	Correctness	Uniqueness	Overlap	# of Snippets
High	99.1%	92.6%	58.4%	22.9
Medium	96.3%	95%	65%	26.1
Low	91.1%	90.7%	70.6%	14

4.6.4 RQ4: APIs with Different Popularity Levels

Table 4.12 shows the performance of AutoDoc for APIs with different popularity. In terms of **correctness**, we observed that the accuracy of knowledge snippets generated by AutoDoc declines as the APIs become less popular. This is because less popular APIs are less frequently discussed on SO, providing fewer reference posts for AutoDoc to extract API knowledge from when generating documentation. As a result, AutoDoc relies more heavily on its pre-trained knowledge, which is prone to hallucination. In terms of **uniqueness**, the performance of AutoDoc is the worst for low-popularity APIs. This is because for APIs with low popularity, less relevant posts will be returned by the post retrieval component. In the case where one or more sections of API document corresponding to some knowledge types contain few or no knowledge snippets, the summarization component will fill these sections based on the model's internal memory. Table 4.10 shows an example where GPT-4o made up a knowledge snippet when the DPR model found no relevant posts. In terms of **overlap**, we observe that for APIs with low popularity, AutoDoc generated more knowledge snippets that can be found in official API documents. A possible reason is that less popular APIs were less frequently discussed on SO. Therefore, it is hard for the DPR model to retrieve posts that contain undocumented knowledge of these APIs. Without enough reference posts, LLMs like GPT-4o could elaborate on some knowledge snippets when asked to summarize them. In that case, the extra content elaborated by the LLM (e.g., explaining a parameter

used in an API) can only come from the LLM's pre-trained knowledge, which can overlap with official API documents.

4.6.5 RQ5: Contribution of Each Component

In this section, we report the contribution of each component of AutoDoc: (1) fine-tuned DPR model, (2) knowledge extraction component, (3) knowledge validation component, and (4) knowledge summarization component.

Fine-tuned DPR model

To evaluate the performance of the fine-tuned DPR model, we compared it against the original DPR model and two popular retrieval methods, BM25 [195] and E5 (an embedding model) [196]. The evaluation was performed on 10 APIs randomly sampled from our benchmark. For the retrieved posts for these 10 APIs, please check our GitHub repository [181].

The first two authors who labeled the generated API documents also labeled the retrieved posts. Specifically, they labeled 10 APIs $\times$ 7 knowledge types $\times$ 10 retrieved posts per type $\times$ 4 retrievers = 2,800 retrieved posts. They first independently labeled 100 posts randomly selected from posts retrieved by both models. Each post can either be relevant or irrelevant to the DPR query. In the first labeling round, the two labelers achieved a Cohen's Kappa score of 0.63. They discussed their labels to establish a labeling rule that a post should be labeled as relevant only if it contains the requested type of knowledge. With that rule in mind, a post containing only *functionality* knowledge of an API should be labeled as irrelevant if it was retrieved by the query of *concept* knowledge. After resolving all conflicts in their labels, the two labelers independently labeled another 100 randomly sampled posts, achieving a Cohen's Kappa score of 0.74. Since this score indicated a substantial level of agreement, they split and labeled the remaining posts separately. In total, the labeling process took 17.2 person-hours.

Table 4.13. Performance of different retrieval methods.

Knowledge	E5	BM25	DPR	DPR (fine-tuned)
Functionality	7%	35%	26.9%	41%
Concept	13%	47%	36%	61%
Performance	2%	45%	33%	42%
Directive	9%	34%	33%	38%
Pattern	16%	50%	45%	56%
Environment	9%	46%	36%	48%
Alternative	10%	35%	38%	53%
Overall	9.4%	41.7%	35.4%	48.4%

Table 4.13 shows the average accuracy of the fine-tuned DPR model and three baselines on retrieving the top 10 most relevant posts for the 10 APIs randomly selected from our benchmark by knowledge type. Overall, the fine-tuned DPR model outperforms all baselines by a large margin. Surprisingly, BM25 outperforms the original DPR model with no fine-tuning. This is because the original DPR model does not understand the technical language in SO posts. On the other hand, BM25 uses term frequency-based scoring, allowing it to effectively capture relevant API names. We notice that fine-tuning increased the retrieval accuracy of the DPR model on all knowledge types by 13%. This indicates fine-tuning can effectively improve the performance of the DPR model on domain-specific tasks. Specifically, the retrieval accuracy for *concept* (+15%) and *alternative* knowledge (+15%) benefited the most from the fine-tuning. A possible explanation is that retrieving *concept* and *alternative* knowledge involves recognizing how different API components interact, their intended use cases, and possible substitutions, which is too hard for the original DPR without finetuning.

Knowledge extraction component

To evaluate the accuracy of the knowledge extracted by AutoDoc from SO answer posts, we randomly sampled 321 knowledge snippets from 1,920 knowledge extracted from 3,360 SO answer posts (10 posts retrieved for each knowledge type/API combination). This sample size is statistically meaningful with a 95% confidence interval. For each API knowledge, the first two authors independently verified whether the knowledge was accurate by searching

online. Then, they compared their verification results and resolved any disagreement. The Cohen's Kappa score was 0.84. Overall, 301 of the 321 relations were verified to be correct, indicating a high accuracy (93.7%).

Table 4.14. An identified incorrect *Alternative* knowledge for `MediaPlayer` from Post 76461455.

POST 76461455
On Android, you can use the MediaPlayer class to play sound. The MediaPlayer class can play audio and video files, and it provides a rich set of features such as seeking, looping, and volume control. On iOS, you can use the AVFoundation framework to play sound. The AVFoundation framework provides classes for playing, recording, and editing audio and video. To play a sound on iOS, you can use the AVPlayer class.
EXTRACTED ALTERNATIVE KNOWLEDGE
Alternatives to MediaPlayer on Android include the AVPlayer class from the AVFoundation framework on iOS for playing audio and video.
VALIDATION RESULT
No.

Knowledge validation component

We conducted an ablation experiment where the knowledge validation component was removed from AutoDoc.This ablation experiment was conducted on 10 APIs randomly sampled from our benchmark. Table 4.15 shows the results of this ablation study. Our experiment showed that the accuracy of knowledge snippets in the generated API document, on average, dropped 3.7% after removing it. This result indicated that the knowledge validation component can effectively remove hallucinations from the generated API documents. For example, Table 4.14 shows an example where the knowledge validation component identified an incorrect alternative knowledge of `MediaPlayer` in Android extracted by GPT-4-o from Post 76461455. Although the post mentioned `AVPlayer` can be used to play sound as well, it cannot replace `MediaPlayer` because they were used on different operating systems

(Android v.s. IOS). Therefore, as the knowledge validation component correctly concluded, Post 76461455 did not contain alternative knowledge of `MediaPlayer`.

Furthermore, we noticed that when the validation component was removed, the generated API document also became more duplicated. For example, the API documents became 15.2% more duplicated on average. This is because the knowledge validation component helps to remove hallucinated content extracted from a relevant post. Most of the hallucinated content filtered by the validation component was not factually wrong but could not be found in the relevant post and, therefore, should not be extracted. Such hallucinated contents are usually simple, general knowledge of an API (i.e., *"Model in TensorFlow is used to define and create neural network models"*), which can easily overlap, introducing duplication.

Knowledge summarization component

We also conducted an ablation study where the knowledge summarization component was removed from the pipeline. This ablation study was conducted on the same 10 APIs used to evaluate the knowledge validation component. Our experiment results (Table 4.15) showed that the generated API documents, on average, became 31.7% more duplicated after removal. This indicates the effectiveness of the knowledge summarization component. For example, Table 4.16 shows three duplicated knowledge snippets extracted from different posts and the summary made by the summarization component. The same concept knowledge, *"ByteArray is an array of bytes in Kotlin"*, was repeatedly extracted from different posts, causing redundancy in the generated document. The knowledge summarization component successfully detected and removed this duplication.

Table 4.15. Effect of knowledge validation and summarization.

	Correctness	Uniqueness	Overlap	# of Snippets
AutoDoc	96%	87.6%	66.7%	17.2
- validation	92.3%	75.8%	72%	18.2
- summarization	94.1%	59.3%	74.5%	21.7

Table 4.16. A list of duplicated knowledge snippets of `ByteArray` and the summarized version.

EXTRACTED KNOWLEDGE SNIPPETS
1. ByteArray is an array of bytes in Kotlin.
2. A ByteArray in Kotlin is an array of bytes.
3. ByteArray is a data structure in Kotlin that represents a sequence of bytes.
SUMMARIZATION RESULT
ByteArray is used to create and manage arrays of bytes in Kotlin.

4.6.6 RQ6: Sensitivity to Different LLMs

In addition to GPT-4o, we also experimented with using other LLMs as the underlying LLM for AutoDoc. We selected three open-sourced LLMs, including Llama 2-70B-Chat, Llama 3-70B-Instruct, and Mistral-7B-v0.3. We evaluated the performance of these three variants of AutoDoc on 10 APIs randomly sampled from our benchmark and showed the results in Table 4.17. After shifting from GPT-4o to open-sourced LLMs, the percentage of correct knowledge snippets and unique knowledge snippets both dropped to different extents. This is expected because GPT-4o is one of the state-of-the-art LLMs. Among the three open-sourced LLMs, Llama 3-70B-Instruct generated the most correct and unique API documents. This is because Llama 3-70B-Instruct was pre-trained on 8.3 times more tokens than Llama 2-70B-Chat [197] and has way more parameters than Mistral-7B-v0.3. Surprisingly, despite being the smallest model, Mistral-7B-v0.3 achieved similar performance with Llama 2-70B-Chat and outputted the most number of knowledge snippets (21.8). This is likely due to the Mixture of Experts (MoE) architecture [198] in Mistral-7B-v0.3, which compensates for its small size to some extent. Also, API documents with more knowledge snippets are not necessarily better. LLMs can generate short bullet points, each of which will be labeled as a knowledge snippet. These bullet points, although correct, may not bear much useful information (e.g., *"Operating system: Android"*).

Table 4.17. Performance of AutoDoc with open-sourced LLMs.

	Correctness	Uniqueness	Overlap	# of Snippets
with Llama 3	91.4%	77.6%	64.7%	22
with Llama 2	83%	80.3%	70.4%	17.1
with Mistral	84.9%	81.6%	65.9%	21.8
AutoDoc	96%	87.6%	66.7%	17.2

We further examined the API documentation outputs generated by different LLMs to analyze their behavioral similarities and differences. In terms of ***similarities***, all models were able to produce well-structured API documentation with seven sections. This suggests that all models can understand the knowledge types explained in our prompt. In terms of ***differences***, the models differ in the use of code examples and consistency of formatting. Specifically, GPT-4o and Mistral-7B-v0.3 both included code examples in their outputs (3 and 2 examples respectively across 10 API documents), whereas Llama 2 and Llama 3 did not include any. For consistency, all models except Mistral adhered to a uniform heading format within each section (e.g., using # or consistent markdown-like markers), while Mistral's output exhibited erratic section header styling. This irregularity may be attributed to Mistral's smaller model size, leading to reduced consistency in generated documents.

4.6.7 RQ7: Temperature Experiments

AutoDoc by default uses a temperature of 0.8 for GPT-4o to extract, validate, and summarize API knowledge. Because temperature is a key factor for LLM performance, we did a small-scale analysis on 10 randomly sampled APIs to investigate the AutoDoc's sensitivity to temperature. We used AutoDoc to generate documents for the 10 APIs under two other temperature settings: 0 and 0.5. For each API, we ran AutoDoc twice to generate two documents since LLMs may generate different outputs each time being prompted. Thus, 10 APIs $\times$ 2 temperatures $\times$ 2 trials = 40 documents were generated and labeled by the same two labelers following the same procedure described in Section 4.5.3. Our labeling results (Table 4.18) show that API documents generated by AutoDoc remain stable across different trials and different temperatures.

Table 4.18. Temperature experiment results.

	Correctness	Uniqueness	Overlap	# of Snippets
T=0/Trial 1	95.3%	95%	65.2%	19
T=0/Trial 2	93.9%	96.4%	66.3%	18
T=0.5/Trial 1	95.7%	95.5%	62.2%	21
T=0.5/Trial 2	94.5%	94.2%	62.9%	20

4.7 User Study

4.7.1 User Study Design

In addition to the quantitative experiments, we conducted a within-subjects user study to evaluate the quality of API documents generated by AutoDoc. We recruited 12 students through the CS department mailing list from a local university. Participants had an average of 5.2 years of programming experience. We randomly sampled 12 APIs from our benchmark. In each user study, we selected 4 out of the 12 APIs and asked the participants to review the API documents generated by AutoDoc and the SISE and GPT-4o (ZSP). We counterbalanced the API document assignment so that each API document was evaluated by four different participants.

For each API, participants were provided with the name of the API, a link to the official API documents, and documents generated by AutoDoc, GPT-4o (ZSP), and SISE. The participants were then asked to read all three API documents and evaluate the quality of these summaries by answering the following five multiple-choice questions. The five dimensions were borrowed from [177, 199]. For each question, the participants selected one of the API documents generated by SISE, GPT-4o (ZSP), and AutoDoc.

1. *Which API document provides the most helpful information?*
2. *Which API document provides the best overview of the API?*
3. *Which API document is most comprehensive?*
4. *Which API document is the most concise without being incomplete?*

5. *Which API document do you prefer to see in practice?*

To mitigate bias, the order of API documents was randomized, and we also did not reveal which model generated which of the three API documents. At the end of the user study, participants answered two open-ended questions:

1. *Do you wish to see API documents generated from SO posts when learning new APIs?*

2. *What are ideal API documents like?*

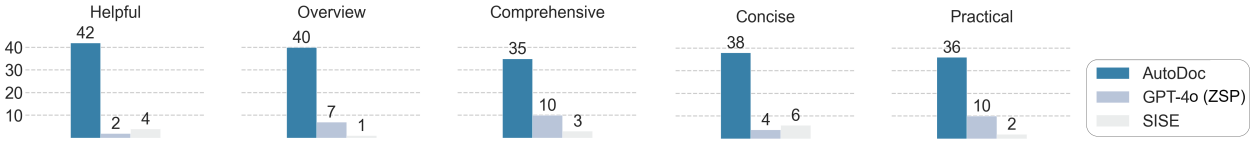

Figure 4.2. Participants' choices over API documents generated by different methods in five aspects

4.7.2 User Study Results

Figure 4.2 shows the choice of participants over the API documents generated by AutoDoc, GPT-4o (ZSP), and SISE in five aspects. On average, 80% of participants preferred API documents generated by AutoDoc over GPT-4o (ZSP) and SISE in all five dimensions. The participants' preference of AutoDoc over GPT-4o (ZSP) and SISE is particularly significant in helpfulness (88%) and overview (83%). Specifically, P2 said: "*I particularly enjoy how well-sectioned the API document generated by one of the three methods is (AutoDoc).*" P9 said, "*Compared to other methods, this method (AutoDoc) generates fewer knowledge snippets that are understandable and not too overbearing.*" This aligns with our hypothesis since SISE extracts insightful sentences that are sometimes hard to understand without seeing the post, and GPT-4o (ZSP) provides too general and shallow knowledge.

In the final survey, 7 out of 12 participants confirmed they would like to see API documents generated from SO posts when working with APIs. For example, P1 said: "*API documents generated from SO posts contain insights that I have never seen in the official document.*" On the other hand, *being well-sectioned* is mentioned by four participants as

one of the most important features an ideal API document should possess. For example, P9 said, "*Ideal API documents should be well-structured so I can find what I want easily*". Meanwhile, P7 said, "*Having well-defined section headers as navigation cues is one of the most important features an ideal API document should possess.*" Participants also wanted functionalities that AutoDoc do not currently possess. For example, two participants said they would like to see the original post from which a knowledge snippet was extracted to have more faith in the generated API document. However, AutoDoc cannot provide the post from which a knowledge snippet was extracted because the knowledge summarization component erased all traceability information.

4.8 Discussion

In this chapter, we demonstrate the effectiveness of leveraging insights from Stack Overflow (SO) to generate high-quality API documents. AutoDoc can effectively extract API knowledge that is not covered in the official API documents from SO posts. Our evaluation shows that 34.4% knowledge snippets in API documents generated by AutoDoc are not covered in the official API documents. This finding suggests that AutoDoc can help developers to learn new APIs with poorly maintained official API documents without having to navigate lengthy SO posts themselves.

Furthermore, our evaluation shows the success of AutoDoc relies heavily on the knowledge validation component. Hallucination in LLMs' output remains an unresolved concern [9, 10, 192]. Programmers may not trust LLM-generated API documents because they cannot distinguish the hallucinated content. Our evaluation demonstrates the feasibility of detecting hallucinated content in LLM-generated API documents by leveraging the LLM's self-correction ability. Our experiments showed that the knowledge validation component can remove 3.7% incorrect knowledge snippets in the generated documents. This result helps to boost user trust in AutoDoc and API documents generated by LLMs in general.

Finally, our experiment shows that only applying a general passage retrieval model, such as the DPR model, is insufficient. When being fine-tuned on the SO data dump, the retrieval accuracy of the DPR model is increased by 13% on average. This may carry a more general

implication—as we reuse models from NLP or ML on SE tasks, we should consider fine-tuning them to account for the unique characteristics of SE tasks. For example, SO posts contain unique vocabulary, including many technical terms. Therefore, training the DPR model to understand these terms via fine-tuning can help it better identify posts relevant to an API-related query.

Toward more concise documents. While the knowledge summarization component already makes AutoDoc's documents 34–47% shorter than those produced by directly prompting GPT-4o (Table 4.7), conciseness can be pushed further. One direction is to perform the summarization step with a model trained to be concise. The verbosity of an LLM is not fixed by task but shaped by its alignment training: preference-based fine-tuning is known to systematically inflate output length, and different models exhibit markedly different length biases [200–202]. This suggests that selecting—or fine-tuning with length-controlled preference signals—a model whose training rewards brevity could shorten the generated documents without sacrificing knowledge coverage. A complementary and cheaper direction is prompt engineering: the summarization prompt can impose explicit length or entity-density constraints, for example through chain-of-density-style iterative rewriting, which repeatedly compresses a draft while forcing it to retain and add salient information [203]. Since AutoDoc's pipeline treats the LLM as a black box, both directions can be adopted by swapping the model or the prompt of the summarization component, leaving the retrieval and validation safeguards untouched.

Limitations and Future Work. First, despite the knowledge validation component, AutoDoc still generated API documents with incorrect information, especially for less popular APIs. In practice, it is hard for a developer to distinguish incorrect information from the rest of the document. Therefore, the developers may lose faith in AutoDoc and resort to official API documents instead. As a future work, we plan to develop a component that provides traceability to each knowledge snippet (e.g., by providing a link to the original post) to boost user trust. For now, AutoDoc does not possess such functionality because, during the summarization process, the LLMs would merge similar knowledge and paraphrase the extracted knowledge. Secondly, many knowledge snippets in API documents generated by AutoDoc can be found in the official documents. In the user study, two participants

complained about the overlap between the official documents and documents generated by AutoDoc. For example, P6 said, *"I think most of the knowledge can be found in official documents. Why don't I just read the official documents?"*. In the future, we would like to develop a knowledge comparison component that automatically filters or marks the shared knowledge between documents by AutoDoc and the official documents so developers can only focus on the additional knowledge provided by the auto-generated documents.

4.9 Threats to Validity

In terms of **internal validity**, the evaluation process involves manual labeling, which can introduce bias in the benchmark. To mitigate this threat, two labelers iteratively established a set of labeling standards with two labeling rounds. At the end of the second labeling round, they achieved high Cohen's Kappa scores for all three metrics. Furthermore, although we tried to mitigate data leakage by removing SO posts that mention the 48 APIs, given the size of the dataset——144K question posts and 175K answer posts, it is impractical to manually verify each post for relevance.

In terms of **external validity**, AutoDoc was evaluated on 48 APIs with different popularity levels. Therefore, we cannot guarantee that the effectiveness of AutoDoc can be generalized to any APIs. In particular, AutoDoc may not generalize well for newer or less popular APIs, since these APIs may not be frequently discussed on Stack Overflow. As shown in Table 4.12, the correctness and uniqueness of knowledge snippets for unpopular APIs drop by 8% and 7% and the overlap with official API documentation increases by 12%, compared to popular APIs.

In terms of **construct validity**, the DPR model and GPT-4o can make mistakes. The DPR model may retrieve irrelevant posts. To mitigate this threat, for DPR, we used GPT-4o to check the relevance of posts retrieved by the DPR model. Specifically, we allowed GPT-4o to output nothing in case it cannot extract any API knowledge from the given post. For GPT-4o, we implemented a knowledge validation component to detect hallucinations in its output.

Our user study design also has several limitations. First, our user study may not fully reflect how users would utilize the knowledge extracted by AutoDoc, since we only asked participants to choose between several API documents instead of solving real-world problems. Second, participants were asked to select one preferred document, though sometimes they may not have a strong preference. To mitigate this issue, we randomized the order of the API document options presented to each participant. In this way, even if participants perceived no clear differences and chose randomly, each option would still have the same probability of being selected.

4.10 Summary

In this chapter, we presented AutoDoc, an LLM-based pipeline that generates API documentation covering seven types of API knowledge from crowdsourced Stack Overflow discussions. AutoDoc grounds knowledge extraction in posts retrieved by a fine-tuned dense passage retrieval model, validates every extracted knowledge snippet through LLM-based self-checking, and consolidates the validated knowledge through summarization. Across 48 APIs from Java, Android, Kotlin, and TensorFlow, AutoDoc generated documents that are up to 77.7% more accurate and 9.5% less redundant than five baselines, while surfacing 34.4% knowledge absent from official documentation. The ablation results isolate the contribution of each safeguard: removing the self-validation component measurably degrades correctness, and removing summarization inflates redundancy. Notably, with the full pipeline in place, even a 7B-parameter open-source model can produce documents approaching GPT-4o quality, suggesting that carefully engineered guidance and self-reflection can substitute for raw model scale.

With respect to the thesis statement, AutoDoc demonstrates that retrieval grounding (external guidance) and LLM-based validation (self-reflection) together make free-form generation trustworthy enough for a high-stakes artifact like API documentation. Yet both Assort and AutoDoc treat the model as a black box: they constrain its inputs and audit its outputs, but neither explains *why* the model hallucinates in the first place. After building these two systems, we came to realize that we lacked a fundamental understanding of

hallucination—without it, mitigation remains a collection of empirical safeguards. The next chapter addresses this gap by looking inside the model, using its attention as an implicit signal to understand how LLMs process natural language instructions and why their outputs sometimes deviate from them.

5. UNDERSTANDING HALLUCINATION THROUGH IMPLICIT SIGNALS: ATTENTION ALIGNMENT BETWEEN LARGE LANGUAGE MODELS AND HUMAN PROGRAMMERS

The two systems presented so far, ASSORT and AUTODOC, mitigate hallucination from the outside: they ground the model in external evidence and audit its outputs through self-checking. These safeguards are effective, but building them made a limitation increasingly clear—we were treating hallucination as a symptom to suppress, without understanding its cause. We could not answer why a model, given a perfectly clear instruction, sometimes produces content the instruction never asked for. We decided that making further progress required a fundamental understanding of how LLMs process natural language input, and that this understanding had to come from signals *inside* the model. This chapter therefore turns from building systems to studying models, and develops the third pillar of the thesis statement: understanding hallucination through *implicit signals*. We conduct the first large-scale empirical study of whether code generation models attend to the same parts of a task description as human programmers, across six LLMs, twelve attention calculation methods, and 1,111 programming tasks. The attention patterns we uncover explain a substantial portion of code generation errors—including hallucinated code semantics—and point toward hallucination-aware diagnostics for generative models.

5.1 Introduction

Large Language Models (LLMs) have made significant process on code generation in recent years [7, 64, 69, 72, 76, 204–207]. A recent study [77] shows that GPT-4, the state-of-the-art LLM with 1.7 trillion parameters, can correctly solve 84% of the Python programming tasks from HuamnEval [7], a widely used benchmark for code generation. Despite this great progress, it remains unclear why and how LLMs can generate correct code from natural language descriptions.

Model attention analysis is a common methodology to understand how a model works. It has been widely adopted in computer vision [20, 208–212] and autonomous driving [213–215] to investigate whether a model pays attention to the salient parts of an image or a traffic scene when making a decision. In particular, recent studies find that aligning model attention with human attention can effectively enhance model performance [20, 216, 217]. For instance, Huang et al. [20] show that the performance of Conv-4-based image classification models can be increased by up to 23% when they are trained to align with human attention (i.e., attending important areas in an image labeled by human annotators). Furthermore, previous studies also suggest that users have more confidence and trust in human-aligned models [21, 218–220]. For instance, Boggust et al. [21] find that users determine the trustworthiness of a model by checking whether the model makes predictions based on features they consider important.

These findings lead to an important scientific question for LLM-based code generation—*do LLMs attend to similar parts of a task description like human programmers in code generation?* We choose to compare model attention with human attention, since it can help us determine whether LLMs grasp the deep semantics in a task description like humans or whether they just learn superficial patterns from training data, which is known as a common problem in machine learning. Furthermore, by comparing human and model attention patterns, we seek to investigate whether the attention differences can be used to explain some code generation errors and inform new opportunities to improve LLMs for code generation.

To bridge the knowledge gap, we made the first effort to unveil the code generation process of LLMs by analyzing *which parts of human language an LLM attends to when generating code*. We present a large-scale study that examines the attention alignment between six LLMs and human programmers on two popular code generation benchmarks—OpenAI's HumanEval benchmark [7] and Google's MBPP benchmark [221]. The hypothesis is that LLMs should generate code based on salient words from an NL description similar to human programmers, rather than generating code based on trivial tokens (e.g., prepositions, delimiters). Specifically, we investigated the following research questions in this study:

RQ1 To what extent is model attention aligned with human attention?

RQ2 Can attention explain errors of code generation models?

RQ3 What is the impact of different attention calculation methods on attention alignment?

RQ4 Which attention calculation method is most preferred by programmers?

Since none of the existing code generation benchmarks contain programmer attention information (i.e., which words or phrases a programmer pays attention to when writing code), we created the first programmer attention dataset for the 1,111 programming tasks from HumanEval [7] and MBPP [221]. We chose these two datasets because they are widely used benchmarks and they contain test cases to assess the functional correctness of generated code. We captured programmers' attention by asking two experienced programmers to manually label words and phrases that they considered essential to solving each programming task. Their labels are validated by a third programmer who independently labeled 164 programming tasks and achieved a substantial agreement with the previous two programmers. On the other hand, to capture model attention, we implemented and experimented with twelve different attention calculation methods in three categories—*self-attention based*, *gradient based*, and *perturbation based*. To ensure our findings generalize across different LLMs, we analyzed the attention of six LLMs with different sizes, including GPT-4 [222], InCoder-6.7B [72], CoderGen-2.7B [74], PolyCoder-2.7B [73], CodeParrot-1.5B [223], and GPT-J-6B [207].

Our study reveals several important insights into the code generation process of LLMs. First, we find a consistent *misalignment* between the LLM and human programmers in all six LLMs, regardless of the attention calculation methods. Furthermore, we performed an in-depth analysis of the attention patterns of 211 incorrect code generated by the two best models in our study, CodeGen and GPT-4. We found that 27% of the code generation errors could be explained by five attention patterns. Finally, among the 12 different attention calculation methods, we find that a perturbation-based method achieves the highest alignment with human attention and also receives the most preference from human programmers in a user study with 22 participants. Our findings highlight the need for developing human-aligned LLMs and also provide practical guidelines for improving the robustness of code generation models and calculating model attention.

In summary, this chapter makes the following contributions:

- We conducted the first study on the attention alignment of LLMs and human programmers on 1,111 programming tasks from HumanEval and MBPP datasets.
- Our study highlights the importance of understanding the code generation process of LLMs and identifies opportunities for improving code generation models through attention analysis.
- We conducted the first comparative analysis of 12 different attention calculation methods for code generation models through both quantitative experiments and a user study with 22 participants. Our results indicate that a perturbation-based method achieves the best performance.
- We made publicly available the first programmer attention dataset of 1,111 Python tasks, which can be used to develop new human-aligned models and evaluate new Explainable AI (XAI) methods for code generation models. Our code and data have been made available in an anonymous supplementary website: https://sites.google.com/view/codellmxai.

5.2 Motivation and Preliminaries

In this section, we first explain the motivation for performing attention analysis for code generation models. Then, we introduce popular benchmarks and metrics for code generation. Finally, we define model attention and describe different kinds of attention calculation methods for LLMs.

5.2.1 Motivation

Model attention analysis is a common task in several domains, such as computer vision [209–212], neural machine translation [224], and autonomous driving [213–215]. Specifically, several studies have demonstrated that aligning model attention with human attention during the training process can significantly improve the model performance [20, 216, 217].

For instance, Huang et al. [20] show that by aligning the attention of Conv-4-based and ResNet-based models to human attention on images, the performance of these models on image classification can be increased by up to 23% in the one-shot setting and 10% in the five-shot setting. In the domain of autonomous driving, many studies have demonstrated that adding attention alignment constraints can help the autonomous driving system to drive safer [213–215]. This motivates us to investigate the attention alignment between LLMs and human programmers in the domain of code generation.

There have been several recent attempts for attention analysis of code models for code summarization, program repair, and method name prediction [217, 225, 226]. Paltenghi et al. [225] studied the attention alignment between neural model attention and programmer attention on code summarization. However, their study focuses on CNN and a transformer model trained from scratch instead of LLMs. Bansal et al. [217] showed that aligning the attention of neural code summarization model with human attention can effectively improve model performance. Rabin et al. [226] found that pre-trained code models rely heavily on just a few syntactic features in the prompts to perform method name prediction and variable misuse detection. To the best of our knowledge, none of the existing studies have investigated the attention alignment between models and human programmers for code generation tasks or large language models with billions of parameters. Therefore, we conducted the first comprehensive empirical study on how the attention of six LLMs with different sizes aligns with programmer attention.

Finally, another motivation behind this study is the fact that there is still no consensus on how to calculate the attention score of LLM-based code models. This is largely attributed to the complexity of the multi-head, multi-layer attention mechanism adopted by these models. For example, some studies only considered model attention from the first transformer layer and stated that the first layer encodes lexical-level information [87, 227], while other studies summed up the attention from all transformer layers to incorporate long-distance token relationships [89, 100]. To bridge this gap, we conducted the first comprehensive study on 12 attention calculation methods and systemically evaluated them with quantitative experiments and a user study with 22 participants.

5.2.2 Code Generation Benchmarks and Metrics

codegen_2b_mono_output.py

```
    from typing import List

    def has_close_elements(numbers, threshold) -> bool:
        """
        Check if there are two numbers in a list that are
        closer to each other than a given threshold.
        """
+       for i in range(len(numbers) - 1):
+           if numbers[i + 1] - numbers[i] < threshold:
+               return True
+       return False
```

Figure 5.1. A Python function generated by CodeGen [74]. The generated code is highlighted in green.

In this chapter, we focus on code generation tasks that generate a function from a natural language description. Since the breakthrough from OpenAI Codex and GitHub Copilot in 2021, this kind of code generation task has become increasingly popular in the research community. In this task setting, given a function header and a task description in natural language, an LLM is expected to complete the function according to the task description. Figure 5.1 shows an example.

Code generation models are often evaluated on crowd-sourced programming benchmarks. OpenAI developed a programming benchmark called HumanEval and used it to evaluate the original Codex model and its variants [7]. HumanEval [7] includes 164 Python programming tasks and ground-truth solutions. It is by far the most popular benchmark and has been used to evaluate most LLM-based code generation models. Besides, MBPP is a large benchmark [221] with 947 crowd-sourced programming tasks and solutions. Both HumanEval and MBPP include test cases to evaluate the functional correctness of generated code. CodeXGLUE is another large benchmark that not only covers code generation tasks but also code translation and code summarization tasks [228]. However, CodeXGLUE does not include any test cases.

Two types of metrics are often used to evaluate the performance of LLM-based code generation models. First, if a code generation benchmark includes test cases, one can simply run the test cases to evaluate the correctness of generated code. A typical metric in this category is *Pass@k*, which is initially proposed by OpenAI [229]. *Pass@k* measures the

number of correctly solved programming tasks where k refers to the number of code samples generated by the LLM. If any of the k samples pass all test cases in a task, the task is considered correctly solved. For example, the Codex model achieves 28.8% in *Pass@1* and 70.2% in *Pass@100*. Since the number of test cases is limited, passing all test cases of a task does not necessarily mean the generated code is fully correct. Thus, prior work also measures the similarity between generated code and a ground-truth solution as a proxy for correctness. BLEU [230] and CodeBLEU [231] are commonly adopted similarity metrics. BLEU is a metric commonly used for evaluating the quality of machine-generated text. It evaluates the quality of machine-generated text by comparing the presence of n-grams in the generated text to those in reference texts. BLEU calculates a score between 0 and 1, with 1 indicating perfect similarity. CodeBLEU is designed to adapt BLEU specifically for evaluating generated code. While BLEU primarily considers word-level similarity, CodeBLEU considers the structure and correctness of the generated code, which are crucial aspects in code generation tasks.

5.2.3 Model Attention

In this chapter, we use *model attention* to refer to how important a token in a natural language task description is considered by an LLM during code generation. It implies which parts of the input the model "attends" to during code generation. This idea resembles *feature importance* [232], *saliency map* [233], and *feature attribution* [234] in the XAI literature. We describe three kinds of model attention calculation methods as follows.

Self-attention-based Methods.

The self-attention mechanism allows transformers to weigh the importance of different parts of the input when making predictions. By focusing on the most relevant tokens with the highest self-attention scores, the transformers can make better predictions by capturing relationships and dependencies in the input.

An LLM includes multiple *transformer layers*, each of which includes multiple *attention heads*. These attention heads independently calculate self-attention scores between different input tokens. Therefore, a transformer model could have multiple sources of model attention

calculated within different transformer layers and attention heads. For example, Figure 5.2 shows the self-attention scores of the first three attention heads of the first transformer layer in CodeGen-2.7B when generating token *"numbers"* from input sequence *"Return the max between two"*. The self-attention scores calculated by different attention heads differ for the same token. Each head represents different kinds of "focus" from the model.

To calculate model attention on the input sequence, vectors of self-attention scores from different layers and different heads need to be aggregated into a single vector to represent the overall importance of each input token to the model prediction. However, although self-attention scores have been generally used as model attention [235–238], there is still no consensus on how to aggregate self-attention scores from different layers and attention heads. For example, some studies sum up the attention scores attention from all transformer layers and all attention heads in each layer [99] as the final attention scores. These studies argue that this summation strategy can capture long-distance token relationships. Some other studies only use the attention scores from the first transformer layer and argue that the first layer captures lexical-level dependencies [87, 227]. To reveal how different ways to aggregate self-attention affect the alignment between model and human attention, we experimented with six self-attention-based methods in this study (detailed in Section 4.2.1).

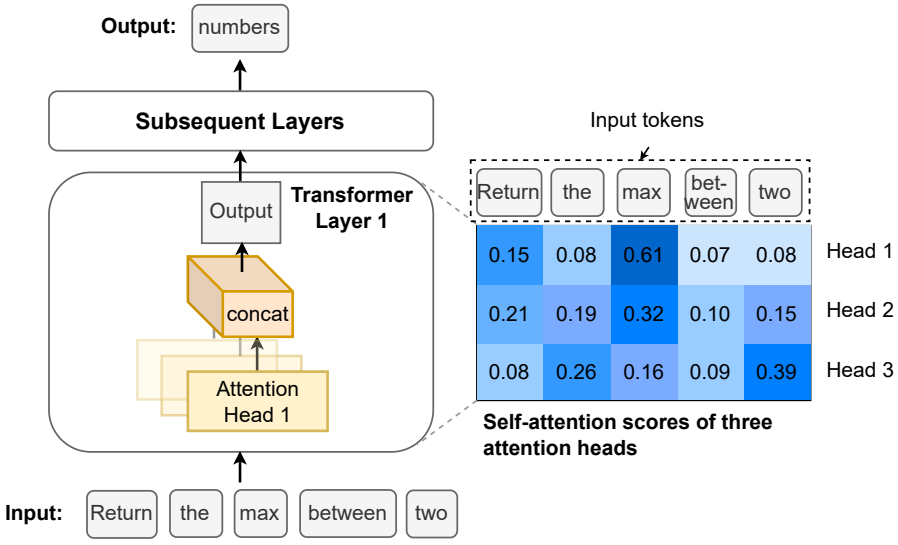

Figure 5.2. Attention matrix of the first attention head of the transformer layer in CodeGen-2.7B

Gradient-based Methods.

Gradient-based methods leverage the gradients of the model's predictions concerning the input features to calculate the model's attention. The calculation of gradient-based methods includes two different steps: (1) perform a forward pass of input of interest, and (2) calculate gradients using backpropagation through the neural network's layers. By analyzing the magnitudes of these gradients, these methods can identify which input tokens are most influential in determining the output. For example, *Integrated Gradients* is a gradient-based method that computes the integral of the gradients of the model's output concerning each input feature [239–241]. In this study, we experimented with two gradient-based methods used in previous work [241, 242] with details in Section 4.2.2.

Perturbation-based Methods.

Different from the previous two categories of methods, perturbation-based methods [238, 243] are model-agnostic. In other words, they do not require access to the internal information of a model. Perturbation-based methods are particularly useful for calculating the attention of commercial models such as GPT-4, since these models do not reveal their self-attention layers or gradients to users.

Perturbation-based methods first mutate the input and then calculate the model's attention based on the output differences. LIME [244] and SHAP [245] are two popular perturbation-based methods. LIME [244] generates a local explanation by approximating the specific model predictions with a simpler model (e.g., a linear classifier). SHAP [245] enhances LIME by perturbing the input based on game theory and uses Shapely value to estimate different tokens' importance.

A limitation of these two methods is that they often require a large number of perturbated samples to ensure estimation accuracy. This is costly for calculating the attention for GPT-4, since we need to query GPT-4 many times. Furthermore, LIME and SHAP only mutate an input by deleting tokens, which may significantly change the meaning or the structure of an input. To address this limitation, more recent perturbation-based methods choose to substitute tokens with similar or semantically related tokens in the context [243, 246]. They

often use a masked language model such as BERT [75] to predict similar or semantically related tokens to substitute existing tokens in an input. Then, they measure the influence of these substitutions on the output. In this study, we experimented with SHAP [245] and the BERT masking-based method [243] (detailed in Section 4.2.3).

5.3 The Construction of the Programmer Attention Dataset

Since none of the existing code generation benchmarks contain programmer attention information (i.e., which words or phrases a programmer considers important when writing code), we created the first programmer attention dataset based on the 1,111 programming tasks from HumanEval [7] and MBPP [221]. We select these two datasets since they provide test cases for each programming task, which is important for calculating the correctness of model-generated code.

The first two authors, who have more than five years of programming experience in Python, manually labeled the words and phrases they considered important to solve a programming task in each task description. Before the labeling process, the two labelers went through 164 programming tasks in HumanEval to familiarize themselves with the programming tasks and the code solutions. Then, they first independently labeled the first 20 task descriptions in HumanEval. This first round of labeling had a Cohen's Kappa score of 0.68. The two labelers discussed the disagreements and summarized four kinds of keywords that both of them considered important. The four keyword types are summarized below:

- **Data types**: Words or phrases that describe the types of data that the code should input or output, such as *"string"*, *"number"*, or *"list"*.
- **Operators**: Words or phrases that describe the operations that the code should perform on the data, such as *"compare"*, *"sort"*, *"filter"*, or *"search"*.
- **Conditionals**: Words or phrases that specify the conditions under which the code should execute, such as phrases after *"if"* and *"when"* in a task description.

- **Properties**: Important properties of the manipulated data and operations, such as quantifiers (e.g., *"all"*, *"one"*), adjectives (e.g., *"first"*, *"closer"*), and adverbs (e.g., *"every"*, *"none"*).

Although there are only four types of keywords, each type is designed to be high-level and inclusive. For instance, the "operator" type refers to any kind of operation on the data, such as *"sort a list"*, *"connect a database"*, and *"plot a graph"*. With this labeling standard, the two labelers proceed to label the remaining 144 task descriptions in the HumanEval dataset. The Cohen's Kappa score of this round of labeling increased to 0.72, indicating a substantial agreement [155]. Then, they discussed and resolved all disagreements.

To verify these labels, the third author, who was not involved in the previous labeling process, independently labeled the 164 tasks descriptions from the HumanEval dataset. Since Cohen's Kappa can only calculate the agreement level between two labelers, we used Fleiss' Kappa to measure the agreement between the third labeler and the initial labels from the first two labelers. The Fleiss' Kappa score is 0.64, indicating a substantial agreement [247]. This result shows labels made by the first two labelers are reasonable and can be accepted by other programmers. Then, the first two labelers continued to label 947 programming tasks in the MBPP dataset independently, which resulted in a Cohen's Kappa score of 0.73. Finally, they resolved all disagreements and used the final set of labels as the programmer's attention dataset. The entire labeling process takes 32 person-hours.

On average, each task description has 44 words, among which 8.1 words are considered important by both labelers. Among all four types of keywords, *property* keywords (31.6%) are labeled the most frequently by the two labelers, followed by *operators* (21.3%), *conditional* (28.1%), and *data types* (19%). Figure 5.3 shows two examples of labeled task descriptions. The four types of keywords are labeled in different colors.

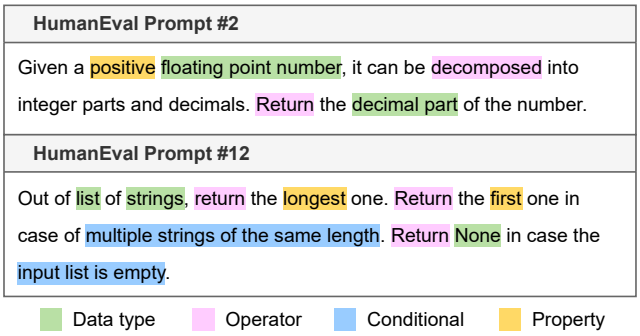

Figure 5.3. Two examples of labeled prompts from our dataset.

5.4 Methodology

This section describes the study design to answer the research questions listed in Section 5.1.

5.4.1 Code Generation Models

In this study, we select six LLMs with different sizes and different model performances on code generation tasks. Table 5.1 shows the size and model performance of these models. We describe each model below.

- **InCoder-6.7B** [72] is a recently published open-source transformer language model from Facebook AI. It is trained on 159 GB permissively licensed code from GitHub, GitLab, and Stack Overflow. Compared with other LLMs, it adopts a new causal masking objective, which allows it to infill blocks of code conditioned on the arbitrary left and right contexts. We used the largest pre-trained model released by Facebook, including 6.7B parameters and 32 transformer layers.

- **PolyCoder-2.7B** [73] is an open-source model from CMU. It is based on the GPT-2 architecture and is designed to be the open-source counterpart of OpenAI Codex, since Codex is not open-sourced. It is trained on 249GB of code and has 2.7B parameters.

- **CodeGen-Mono-2.7B** [74] is an open-source model from Salesforce Research. It follows a standard transformer autoregressive model architecture with rotary position

embedding. We use the CodeGen model that is trained on 217GB Python code and has 2.7B parameters.

- **CodeParrot-1.5B** [223] is another open-source effort of training a GPT-2 model for code generation. It is trained on 180GB Python Code and has 1.5B parameters.
- **GPT-J-6B** [207] is an open-source model from EleutherAI. It adopts a transformer architecture similar to GPT-3. It is trained on 825 GB text data, which includes 95GB code from GitHub.
- **GPT-4** [222] is the state-of-the-art language model developed by OpenAI. Since the internal structure of GPT-4 is not public, we do not include the number of layers and heads of GPT-4 in Table 5.1. It is reported that GPT-4 has about 1.76 trillion parameters [248]. We used the API (`gpt-4`) provided by OpenAI to query GPT-4.

Table 5.1. Code generation models included in this study.

Model	Layer	Head	Pass@1
InCoder-6.7B	32	32	15.20%
PolyCoder-2.7B	32	32	5.59%
CodeGen-2.7B	32	32	23.70%
CodeParrot-1.5B	48	25	3.58%
GPT-J-6B	28	16	11.62%
GPT-4	-	-	67%

5.4.2 Model Attention Calculation

We experimented with twelve attention calculation methods from three different categories: six *self-attention-based* methods, four *gradient-based* methods, and two *perturbation-based* methods.

Self-attention-based methods.

Given that LLMs have multiple attention layers, there is currently no consensus on what is the right way to aggregate those self-attentions to explain LLMs. Zeng et al. [87] show that the first attention layer is indicative of which tokens the model attends to, while Wan et al. [100] show that deeper attention layers are better at capturing long-distance dependencies and program structure. To perform a comprehensive analysis, we decide to experiment with three settings: (1) only using the first attention layer (denoted as *first*), (2) only using the last attention layer (denoted as *last*), and (3) using all attention layers (denoted as *all*).

To aggregate self-attentions across different attention heads in a layer, we follow the previous work [99] by summing the attention values from different heads. Finally, since LLMs generate code in an autoregressive manner, their attention changes in each step as they read more tokens from the input and as they generate more code. We are curious about what the model has attended as they read the input as well as what the model has attended as they generate code. So we consider two experiment settings: (1) summing up the attention scores assigned to each token during the input reading process (denoted as *READING*), and (2) summing up the attention scores assigned to each token during the code generation process (denoted as *CODING*). Given the three settings in layer-wise attention aggregation and the two settings in step-wise attention aggregation, we have a total of six experiment settings: *READING_first*, *CODING_first*, *READING_last*, *CODING_last*, *READING_all*, and *CODING_all*.

Gradient-based methods.

We consider two different methods to calculate gradient-based model attention: (1) *Saliency* [242], and (2) *Input*×*Gradient* [241]. The saliency method calculates the model's attention by computing model gradients with respect to the input. Given a LLM $\mathcal{F}$, suppose an input is $X = [x_1, x_2, \ldots, x_n]$, where n is the length of the input. The attention s_i on x_i is calculated as $s_i = \frac{\partial \mathcal{F}(X)}{\partial x_i}$. Different from the saliency method, *Input*×*Gradient* further multiplies gradients with the input's embedding values. The attention s_i on x_i is calculated as $s_i = x_i \cdot \frac{\partial \mathcal{F}(X)}{\partial x_i}$.

Similar to self-attentions, gradients also change constantly at each generative step. Thus, we also experimented with the two step-wise attention aggregation settings as in the self-attention methods. The combination of the two gradient calculation methods with the two step-wise aggregation settings results in four gradient-based methods—*Input*× *Gradient_reading*, *Input* × *Gradient_coding*, *Saliency_reading*, and *Saliency_coding*.

Perturbation-based methods.

We consider two different perturbation-based methods: (1) *SHAP* [245], which masks the input prompt through deleting some tokens, and (2) *BERT Masking* [243], which masks the input prompt through substituting a token with its masked language modeling prediction result from BERT.

- *SHAP*. We use its official implementation's default settings to calculate model's attention (SHAP scores) on different tokens. We aggregate SHAP scores on predicting different tokens by summing them up.
- *BERT Masking*. We first use a general pre-trained BERT model's prediction to replace one token from the input prompts. Then we calculate the model's attention on this token by calculating the BLEU score [230] between the prediction results with and without *BERT Masking*. We iterate through all tokens in the input prompts to obtain model's attention on different tokens.

5.4.3 Attention Alignment Measurement

We measured the attention alignment between models and human programmers using two robust metrics—San Martino's token overlapping metrics [249] and Krippendorff's alpha [250]. We also experimented with two simple metrics, Cohen's kappa and keyword coverage rate, and obtained similar results. Due to the page limit, we only reported the results of the first two metrics in this chapter. The results of the other two metrics are included in Supplementary Material.

San Martino's token overlapping metrics [249]

calculate the overlap between two sets of tokens in terms of precision, recall, and F-1 score. It was initially designed for sequence labeling tasks in NLP [249] and has been recently adopted in SE studies as a robust metric for toxicity detection in code review comments [251]. Following the recent SE study [251], we also perform token-level comparison rather than character-level comparison as in [249], since token-level comparison provides more explainability. For human attention, a token is considered "highly attended" if it is labeled as an important word by human programmers, as described in Section 5.3. For model attention, since the attention calculation methods in Section 5.4.2 compute a continuous score for each token, it is hard to determine a universal threshold to decide which token is highly attended by a model. Thus, we rank the tokens in a programming task description based on their attention scores and select the top K tokens as the "highly attended" tokens by a model. Since the average of human-labeled important words is 8.1 per task description, we experiment with $K = 5, 10, 20$ in our study. Given a set of highly attended tokens by human programmers and a set of highly attended tokens by a model, we follow the equations in [251] to compute precision, recall, and F-1 score. Due to the page limit, we only report the F-1 scores in this chapter.

Note that LLMs perform subword tokenization to handle out-of-vocabulary words while a manageable vocabulary size. For example, in Figure 5.4, the word "separate" is tokenized into two tokens: "separ" and "ate" through byte pair encoding (BPE). During inference, an LLM will compute attention scores separately for these two subtokens, though they are from the same English word. This makes it hard for us to directly perform token-to-token comparisons between highly attended tokens of an LLM and important words labeled by human programmers.

To address this challenge, we develop a method to automatically map LLM tokens back to NL words and define LLM's attention on each NL word. For an input text, suppose the human labeler divides it into N words: $\{t_{h,1}, t_{h,2}, \ldots, t_{h,N}\}$, while the model's tokenizer splits

it into M tokens: $\{t_{m,1}, t_{m,2}, \ldots, t_{m,M}\}$. We calculate the model's attention on ith keyword $t_{h,i}$ as the sum of attention of all tokens that overlap with $t_{h,i}$:

$$s_{h,i} = \sum_{\{t_{m,j} \cap t_{h,i} \neq \emptyset\}} s_{m,j} \tag{5.1}$$

The model's attention on each natural language word is the sum of the attention scores of all LLM tokens that overlap with this particular word. For example, the attention of the keyword *separate* ($t_{h,1}$) is the sum of LLM's attention on two tokens *separ* ($t_{m,1}$) and *ate* ($t_{m,2}$) in Figure 5.4. We have also experimented with two alternative methods—(1) averaging the attention on all its sub-tokens and (2) simply using the sub-token with the max attention. Our experiment results show that these three methods resulted in similar alignment scores. The detailed results are shown on the website.[1]

Krippendorff's alpha [252]

is a robust statistical measure for inter-labeler agreement. It can be used to measure the agreement level between human programmers and LLMs on important words in a programming task description. Similar to San Martino's metrics, we compute the Krippendorff's alpha coefficients based on the tokens labeled as important by human programmers and the top K tokens attended by an LLM. $\alpha = 1$ indicates perfect agreement, $\alpha = 0$ indicates no agreement, and $\alpha = -1$ indicates complete disagreement. Compared to other inter-rater agreement metrics such as Cohen's kappa [253], Krippendorff's alpha is more robust to the number of coders, missing data, and sample size.

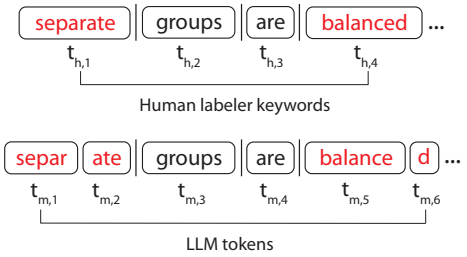

Figure 5.4. Mapping LLM sub-tokens back to NL words

[1] ↑https://sites.google.com/view/codellmxai

5.4.4 User Study Design

To answer RQ4, we conducted a user study to evaluate the different attention calculation methods [2]. We recruited 22 students (18 males and 4 females) through the department mailing lists in a CS department. According to our user study, all participants have an average of 5.62 years of programming experience. All of them are familiar with basic concepts of self-attention mechanisms in transformers.

We randomly selected 8 task descriptions from our dataset. For each task, we leveraged CodeGen, the best-performing model on HumanEval among the five open-source LLMs in our experiment, to calculate its model attention. We selected the top 3 attention calculation methods that have the best alignment with our programmer attention dataset according to the quantitative evaluation results (Sec. 5.5.3): $CODING\_last$, $Input \times Gradient\_all$, and *SHAP*.

In each user study, participants first read the task description to understand the programming task and then read the code generated by CodeGen. For each attention method, we render a highlighted version of the task description similar to Figure 5.3, where the top 10 attended tokens are highlighted based on the attention scores computed by this method. We chose to render the top 10 important keywords since it is close to the average number of important words (8.1) labeled in our dataset. Participants were then asked to rate each attention method by indicating their agreement with the following three statements on a 7-point Likert scale (1—completely disagree, 7—completely agree).

Q1 *The model-attended keywords align with my attention when reading the natural language task description.*

Q2 *This attention explains why the model succeeded in or failed at generating the correct code.*

Q3 *I want to see this attention when working with code generation models in real life.*

[2]↑Our user study questionnaire and participants' responses are available in an anonymous supplementary website: https://sites.google.com/view/codellmxai.

The order of different attention calculation methods is randomized to mitigate the learning effect. We also do not reveal the names of the attention calculation methods to reduce bias. At the end of the user study, participants answer three open-ended questions about different attention calculation methods and the user study design. We ask these open-ended questions to study the correlation between model explainability and user trust. These questions include:

Q4 *Are you interested to know how the LLM generates the code?*

Q5 *What do you want to find out about the internal code generation process in LLM?*

Q6 *Do you trust this LLM? What do you need to know to improve the trust of the LLM?*

5.5 Results

5.5.1 RQ1: To what extent is model attention aligned with human attention?

To answer this question, we collect the top K keywords that the six LLMs attend to and compare them with the keywords labeled as important in the programmer attention dataset. As described in Section 5.4.3, we use San Martino's token overlapping metrics [249] and Krippendorff's alpha [250] to measure the attention alignment between LLMs and human programmers. When running models where the max output length can be set, we set the token limit to 250.

Table 5.2a, Table 5.2b, and Table 5.3 present the attention alignment results when computing attention scores with *perturbation-based*, *gradient-based*, and *self-attention-based* methods, respectively. We find that the two perturbation-based methods consistently achieve better alignment with programmer attention by a large margin compared with methods in the other two categories for all models where a comparison is possible. Because GPT-4 does not reveal its internal states during runtime, only perturbation-based methods are applicable. Section 5.5.3 compares different attention calculation methods in detail. Here, we focus on interpreting the results of perturbation-based methods for brevity.

As K increases from 5 to 20, San Martino's F1 score also increases since more tokens are considered when computing the attention overlap between models and human programmers.

However, the Krippendorff's alpha coefficients do not change much and remain below 0.2, indicating little agreement between model attention and human attention. Among these models, GPT-4 showed the highest attention overlap with human attention in terms of San Martino's F1 score. This is not surprising since GPT-4 is the best-performing and the most recent model in this study. However, even with $K = 20$, San Martino's F1 scores for all six models are still below 66%. In addition, Krippendorff's alpha is always below 0.2 except for CodeParrot, which indicates little or no alignment [155]. These results suggest a consistent attention misalignment between LLMs and programmers when generating code.

Finding 1

There is a consistent misalignment between LLM attention and programmer attention in all settings, indicating that LLMs do not reason programming tasks like human programmers.

5.5.2 RQ2: Can attention explain errors of code generation models?

To answer this question, the first two authors manually analyzed code generation errors made by the best two models in our study—GPT-4 and CodeGen-2.7B. In total, these two models generated 920 incorrect code solutions on the two benchmarks. We randomly sampled 211 incorrect solutions. The sample size is statistically significant with 90% confidence level and 5% margin of error.

The first two authors started with the first 50 code generation errors and independently checked whether the attention pattern of each code could explain the error in it. For simple tasks, it takes about five minutes to check each one. For complicated tasks (e.g., tasks that require an understanding of specific math concepts), it takes around 10 to 15 minutes, since the authors need to manually debug the code and inspect its runtime values to understand the error first. After analyzing 50 errors, they discussed these errors with the other authors and summarized six common attention patterns that can be used to explain code generation errors:

- *Missing attention to critical conditions.* The task description mentions certain conditions or corner cases to handle. However, the model misses or incorrectly handles one or more such conditions, since it does not attend to the words or phrases that describe the corresponding conditions.

- *Missing attention to important descriptive words of an operation or a data object.* The task description mentions an important property of an operation or a data object, such as "*largest* element" and "*ascending* order". However, the model does not attend to these descriptive words and thus generates a code solution with incorrect logic.

- *Missing attention to operation descriptions.* The task description mentions an operation or action, such as *open a file* and *sort a list*. However, the model fails to attend to the verb words or phrases. Therefore, the generated code performs the wrong action or does not perform the action.

- *Missing attention to data types.* Programming tasks often explicitly mention the expected type of inputs and outputs. Some task description also mentions the data type of some intermediate results. However, the model does not attend to some data type descriptions. This can lead to different types of errors, e.g., initializing a variable with the wrong type of data, returning the wrong type of data, calling a method on the wrong type of object, etc.

- *Incorrect mapping between NL words and code elements.* In some cases, we observe the model attends to an important word or phrase correctly but the model maps it to a wrong method call, variable, parameter, value, or logic, potentially due to some conceptual misunderstanding of the semantics meaning of the NL words.

Then, they independently labeled the remaining errors, discussed their labeling with the other authors, and resolved the conflicts. In total, we found that 57 of the 211 manually inspected errors (27%) could be explained by one of the six attention patterns mentioned above. This finding suggests that neural attention analysis can be potentially applied to locate and repair a non-trivial portion of errors in LLM-generated code. Despite this, we

acknowledge that there are still many errors that cannot be easily explained by model attention. Such errors include syntax errors, undefined variable names, incorrect API usage, incorrect array index, infinite loop, etc. This result indicates that further analysis is required to understand the root causes of these errors.

We show the distribution and examples for each type of error below. In these examples, we highlight keywords with high attention scores from the model in blue.

Missing attention to critical conditions.

The example below asks for a function that matches a given word at the end of a string. The task description mentions a condition that if a punctuation symbol exists at the end of the string, it should be ignored. However, GPT-4 does not pay high attention to this phrase and does not handle optional punctuation at the end of a string. 6 incorrect solutions fall into this category.

```
# Write a function that matches a word at the end of a string, with optional punctuation.
def text_match_word(s, word):
    pattern = r'\b' + re.escape(word) + r'\W*$'
    if re.search(pattern, s, re.IGNORECASE):
        return True
    else:
        return False
```

Missing attention to important descriptive words of an operation or a data object.

In the example below, the programming task asks to sort rows in a matrix according to the *sum* of values in each row. However, CodeGen-2.7B does not pay high attention to the word, "sum". Thus, it fails to generate the logic of sorting based on the sum of values in each row. 13 incorrect solutions fall into this category.

```
# Write a function to sort a given matrix in ascending order according to the sum of its row .
def sort(matrix):
    rows = len(matrix)
    cols = len(matrix[0])
    for i in range(cols):
        matrix = sorted(matrix, key=lambda x: x[i])
```

```
    for i in range(rows):
        matrix = sorted(matrix, key=lambda x: x[i])
    return matrix
```

Missing attention to operation descriptions.

The example below asks for a function that r, given a binary string (i.e., a string with 0s and 1s), finds all substrings, counts the number of 0s and 1s in each substring, and finds the substring with the maximum difference between the number of 0s and 1s. Our attention analysis shows that GPT-4 does not pay high attention to words that describe the first two operations, e.g., "the number of ", "sub-string", etc. Therefore, the generated code does not perform the first two operations. 6 incorrect solutions fall into this category.

```
# Write a function to find the maximum difference between the number of 0s and number of 1s in any
    sub-string of the given binary string.
def find_length(binary_string):
    max_diff = 0 # Maximum difference
    current_diff = 0 # Current difference

    for char in binary_string:
        # Convert '0' to 1, '1' to -1
        value = 1 if char == '0' else -1

        # Add the current value to the running total or reset it, if it becomes negative
        current_diff = max(value, current_diff + value)

        # Update the maximum difference found so far
        max_diff = max(max_diff, current_diff)

    return max_diff
```

Missing attention to data types.

The example below asks for a function that, given a list of tuples, computes the average value of elements in each tuple in the list. However, GPT-4 only attends to the second mention of "tuples" and does not attend to the first mention of "tuple". Thus, it thinks that it needs to calculate the average of values across all tuples, not each tuple. 7 errors belong to this category.

```
# Write a function to find the average value of the numbers in a given tuple of tuples.
def average_tuple(tuples):
    total_sum = 0
    count = 0
    for inner_tuple in tuples:
        for number in inner_tuple:
            total_sum += number
            count += 1
    if count == 0:
        return None
    else:
        average = total_sum / count
        return average
```

Incorrect mapping between NL words and code elements.

The example below asks for a function that counts the occurrences of substring *"std"* in a given string. The code generated by GPT-4 instead counts and returns the occurrences of *"s"*, *"t"*, and *"d"*. Although GPT-4 manages to attend to token *"std"*, it maps it to three individual characters, rather than a sequence of characters, which leads to the code generation error. 25 of the incorrect solutions belong to this category.

```
# Write a function to find the occurrence of characters 'std' in the given string.
def count_occurance(input_string):
    s_count = t_count = d_count = 0
    for char in input_string:
        if char == 's':
            s_count += 1
        elif char == 't':
            t_count += 1
        elif char == 'd':
            d_count += 1
    return s_count, t_count, d_count
```

Finding 2

Attention analysis can explain 27% of generation errors in a benchmark of 211 errors. These errors provide insights into the limitations of LLMs.

5.5.3 RQ3: What is the impact of attention calculation methods on the alignment?

According to our results in RQ1, perturbation-based methods consistently produce much higher attention alignment than other methods. In RQ3, we further compare the performance of different methods within each category. As shown in Table 5.2a, *SHAP* produces higher Krippendorff's alpha in most cases. Table 5.2b and Table 5.3 show that all gradient-based and self-attention-based methods perform similarly.

Between self-attention-based and gradient-based methods, self-attention-based methods are better for all K values in terms of San Martino's F1 score. For example, when selecting $K = 20$, self-attention-based methods outperform gradient-based methods in all settings. While neither of the two attention aggregation techniques for self-attention-based methods trumps the other, we do observe that aggregating the attention scores across all generative steps (i.e., *CODING*) outperforms only aggregating the attention scores during the input reading process (i.e., *READING*) for *Saliency* methods.

To confirm there is indeed no significant difference among different self-attention-based and gradient-based methods, we performed Wilcoxon signed-rank tests. The experiment results can be found on our website.[3]

Finding 3

SHAP produces the best attention alignment to human programmers among all methods. Saliency_coding is a better choice within gradient-based methods, while none of the self-attention methods trump others.

5.5.4 RQ4: Which attention calculation method is the most preferred?

Figure 5.5 shows participants' assessment about *SHAP*, *InputXGradient_coding*, and *CODING_all* in terms of the alignment with their own attention, the explainability of the computed attention, and their own preference (Q1-Q3 in Section 5.4.4). Overall, *SHAP* is favored over the other two methods in all three aspects. The average rating on the attention

[3]↑https://sites.google.com/view/codellmxai

alignment of *SHAP* is 5.13, while the average for *InputXGradient_coding* and *CODING_all* are 4.59 and 3.62, respectively. The mean differences between *SHAP* and *InputXGradient_coding* and between *SHAP* and *CODING_all* are both statistically significant (Welch's t-test, $p = 0.05$ and $p = 0.004$). These results are consistent with our findings in RQ1 and RQ3, where *SHAP* outperforms all other methods in attention alignment.

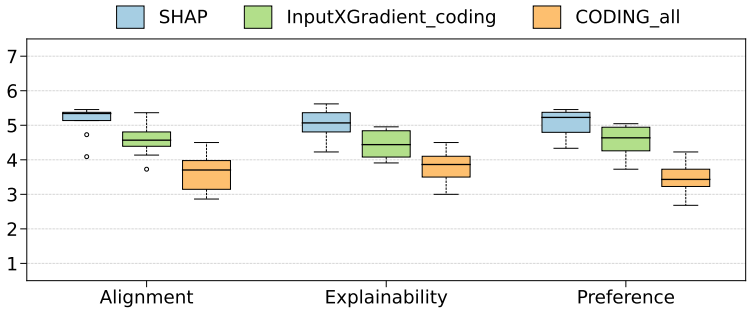

Figure 5.5. Participants' choices over different attention calculation methods in three dimensions.

Furthermore, participants suggested that *SHAP* has better explainability than *InputXGradient_coding* (mean difference: 0.59, $p = 0.02$) and *CODING_all* (mean difference: 1.26, $p = 0.00001$). However, according to participants' responses about their trust in LLMs (Q6 in Section 5.4.4), 14 out of 22 participants expressed a lack of confidence and trust, even after seeing the attention-based explanations. For example, P3 wrote, *"I want to know whether the LLM model can provide the reasons why they generate the code. For instance, the reference code."* Participants also asked for more fine-grained attention analysis. Specifically, they wished to see which parts of the input are responsible of generating which parts of the output. For instance, P10 said, *"[I want to know] how the LLM determines the input and output, what are the discriminative words that drive different generations."* Participants also showed more preference for *SHAP* over the other two methods. The mean preference differences between *SHAP* and *InputXGradient_coding* and between *SHAP* and *CODING_all* is 5.04 vs. 4.56 (Welch's t-test, $p = 0.05$) and 5.04 vs. 3.48 (Welch's t-test, $p = 0.00009$).

Finding 4

Overall, participants preferred the perturbation-based method over the gradient-based and the self-attention-based methods. However, participants still felt a lack of trust in LLMs after seeing the attention-based explanations and wished to see richer explanations such as reference code and fine-grained attention mapping between text and code.

5.6 Implications and Opportunities

Our study has several significant implications that benefit the development of more reliable and more accurate LLMs for code generation in the future.

In RQ1, we discovered a consistent misalignment between human and model attention in all six models. While it is possible that LLMs use an entirely different method to reason about task descriptions compared with human programmers, it is arguable whether such a method is indeed good for programmers due to concerns about interpretability, robustness, and trust. Many studies have shown that human-aligned models are perceived as more trustworthy by humans [20, 213–217]. Furthermore, in practice, LLMs today can reliably solve a limited number of simple programming tasks and struggle to handle more sophisticated or custom tasks, which indicates a huge space for improvement. Thus, we believe investigating the attention patterns of LLMs is a worthwhile effort to help us understand how LLMs generate code and why LLMs make some mistakes and also inform new opportunities to improve LLMs.

In RQ2, we manually analyzed the attention of 211 incorrect generations of the best two models in our paper (GPT-4 and CodeGen) and found 27% of the errors can be explained by incorrect attention alignment. Our finding suggests the potential of fixing these generation errors by adjusting the attention alignment. Furthermore, previous work in computer vision (CV) has shown that the performance of transformers can be improved if we force their attention to align with humans [209–212]. One potential solution is to extend the loss function of LLMs with a penalty term that measures the KL divergence between model

attention and human attention. During training, the loss will increase when model attention deviates from human attention. As a result, the LLM will be trained to distribute attention in a way similar to human programmers. We have open-sourced our human attention dataset to facilitate future work on attention alignment.

Furthermore, the model attention patterns in RQ3 can help to improve the robustness of code generation models by developing new attention-based adversarial training methods. Most adversarial training methods only apply small perturbations to random tokens in a prompt, without considering the importance of a token to the model [254–257]. Our results show that it is worthwhile to prioritize important tokens during perturbation. Perturbing these important words and asking the model to generate code for these perturbed prompts may reveal robustness issues more efficiently.

In RQ3, we compared 12 attention calculation methods with quantitative experiments. Therefore, our study provides practical guidelines for choosing the attention calculation method for future research on LLM-based code generation models. In our study, we experimented with 12 attention calculation methods in three categories. We compared these methods by calculating the token overlaps between attention maps generated by different methods and human attention. Our experiment results indicate developers who want to measure the attention of code generation models should first consider *perturbation-based* methods such as *SHAP* [245], since *SHAP* consistently achieves the best alignment with human attention in all settings. For self-attention-based and gradient-based methods, we cannot find one aggregation mechanism that trumps all others. This suggests that if practitioners want to leverage self-attention-based and gradient-based methods to interpret LLMs, they should experiment with different aggregation mechanisms first and select the one that suits their LLM the best.

In RQ4, we conducted a user study to compare developers' perceptions on different explanation methods. Most participants considered SHAP as the best XAI method for LLM code generation models. This finding suggests that future researchers may want to use SHAP as the default XAI method for LLM-based code generation. Furthermore, in the post-study survey, participants also asked for more fine-grained attention analysis, such as revealing the

association between individual input and output tokens. This highlights the need for new XAI methods for interpreting LLM-based code generation models.

In the future, we would also like to explore how to teach humans how LLMs interpret code. Understanding how LLMs interpret code can inspire human programmers to craft prompts that are more understandable by LLMs and thus guide LLMs to generate better code.

5.7 Threats to Validity

Internal validity. One potential threat lies in the manual labeling process. Two programmers manually labeled the important words to understand a task description and implement the correct function. Since this criterion of keywords is highly subjective, the words selected by the two labelers may not represent the choice of a large pool of programmers. To mitigate this threat, the authors established a labeling standard through discussion and achieved substantial agreement on the labeling. Furthermore, we invite a third labeler to validate our annotated dataset by independently labeling the 164 prompts from the HumanEval dataset. We calculated the Fleiss' Kappa score among the labels of three labelers and the result (0.64) indicates a significant agreement.

External validity. One potential threat to external validity is that we have only experimented with one programming language, Python. We cannot guarantee that our findings generalize to another language.

Construct validity. One potential threat to construct validity lies in the survey design. As we know, problems are mentally demanding to solve and programmers may have different views on which part of the prompt should the model attend to. However, in the current design, only 22 participants were involved. The small sample size may result in findings that are not generalizable. To combat this threat, we only invited participants who are experienced in Python programming and have used code generation LLMs at least once to control the quality of the user study.

5.8 Summary

In this chapter, we presented the first large-scale empirical study of attention alignment between LLM-based code generation models and human programmers. Using a newly constructed programmer attention dataset of 1,111 programming tasks from HumanEval and MBPP, we compared the attention of six LLMs, computed by twelve different methods, against the words and phrases human programmers consider essential. The study yielded three findings that matter for this dissertation. First, model attention is *consistently misaligned* with human attention across all models and attention computation methods—LLMs routinely fixate on trivial tokens while overlooking information humans deem essential. Second, this misalignment is not benign: 27% of the 211 analyzed incorrect code generations, including cases where models hallucinate semantics that the task never specified, can be explained by five recurring attention patterns. Third, among the twelve attention computation methods, a perturbation-based method (SHAP) aligns best with human attention and is most preferred by programmers, providing practical guidance for building interpretability tooling.

With respect to the thesis statement, this study demonstrates that implicit signals are a workable lens for *understanding* hallucination: attention deviations correlate with, and often explain, the moments where generation goes wrong. Together with the previous two chapters, this completes the arc of the dissertation—external guidance and self-reflection to mitigate hallucination, and implicit signals to understand it. The next chapter revisits the thesis statement in light of all three works, distills broader insights from building and studying these systems, and outlines directions for future research.

Table 5.2. Human-model attention alignment calculated by different methods in terms of San Martino's F1 score and Krippendorff's alpha score (in parenthesis). Among the 12 different attention calculation methods, the one that computes the highest Krippendorff's alpha score is highlighted in yellow for each model and each K value. Note that we only experimented with perturbation-based methods on GPT-4, since GPT-4 does not provide access to its self-attention layers and gradients.

(a) Perturbation-based methods.

Method	*BERT masking*			*SHAP*		
	Top 5	**Top 10**	**Top 20**	**Top 5**	**Top 10**	**Top 20**
Incoder	31.1%(0.0)	49.3%(0.06)	59.5%(0.08)	36.5%(0.0)	47.1%(0.07)	64.2%(0.12)
CodeGen	37.1%(-0.06)	62.1%(0.11)	61.1%(0.07)	40.8%(0.02)	57.7%(0.12)	64.7%(0.11)
CodeParrot	29.3%(-0.02)	57.1%(0.13)	61.9%(0.08)	38.4%(0.02)	58.2%(0.11)	65%(0.12)
GPT-J-6B	42.1%(-0.01)	51.5%(0.12)	63.8%(0.04)	39.5%(0.0)	46.5%(0.09)	65.2%(0.1)
PolyCoder	29.3%(-0.04)	58.1%(0.02)	61.1%(-0.05)	35.1%(-0.05)	56.5%(-0.01)	64.7%(0.07)
GPT-4	36.7%(-0.02)	52.1%(-0.002)	61.5%(-0.15)	28.7% (-0.16)	53.65%(0.03)	65.15% (-0.04)

(b) Gradient-based methods

Method	Top 5				Top 10				Top 20			
	P	**R**	**F1**	**KA**	**P**	**R**	**F1**	**KA**	**P**	**R**	**F1**	**KA**
Input×Gradient_reading												
Incoder	62.4%	38.4%	47.6%	-0.21	69.7%	67.1%	68.4%	-0.01	71.7%	94.4%	81.5%	0.02
CodeGen	59.9%	37.6%	46.2%	-0.23	68.2%	65.6%	66.9%	-0.03	70.8%	93.3%	80.5%	0.02
CodeParrot	42.2%	27.5%	33.3%	-0.18	47.8%	55.3%	51.3%	-0.13	53.6%	90.7%	67.4%	-0.12
GPT-J-6B	46.2%	30.1%	36.4%	-0.11	49.0%	57.3%	52.8%	-0.07	53.1%	91.4%	67.2%	-0.1
PolyCoder	70.3%	45.4%	55.2%	0.14	67.2%	73.6%	70.2%	0.24	60.9%	94.6%	74.1%	0
Input×Gradient_coding												
Incoder	55.5%	32.1%	40.7%	-0.3	66.6%	61.7%	64.1%	-0.06	70.5%	93.3%	80.4%	0.02
CodeGen	51.7%	32.1%	39.6%	-0.32	63.9%	62.7%	63.3%	-0.09	68.3%	93.8%	79.1%	-0.02
CodeParrot	71.2%	46.7%	56.4%	0.17	65.8%	71.5%	68.5%	0.22	60.2%	94.4%	73.5%	0.01
GPT-J-6B	73.3%	48.2%	58.2%	0.21	66.0%	72.2%	68.9%	0.23	60.5%	94.5%	73.8%	0.03
PolyCoder	77.7%	51.6%	62.0%	0.27	71.9%	78.4%	75.0%	0.37	62.0%	95.3%	75.1%	0.06

Method	Top 5				Top 10				Top 20			
	P	**R**	**F1**	**KA**	**P**	**R**	**F1**	**KA**	**P**	**R**	**F1**	**KA**
Saliency_reading												
Incoder	68.2%	43.8%	53.4%	0.11	65.1%	71.2%	68.0%	0.19	60.3%	94.1%	73.5%	-0.01
CodeGen	49.2%	33.0%	39.5%	-0.1	53.6%	60.8%	57.0%	-0.02	55.9%	91.7%	69.5%	-0.07
CodeParrot	48.2%	30.2%	37.2%	-0.14	55.2%	62.5%	58.6%	0	56.8%	92.5%	70.4%	-0.06
GPT-J-6B	50.7%	32.7%	39.7%	-0.11	55.8%	62.4%	58.9%	-0.01	57.7%	92.5%	71.0%	-0.06
PolyCoder	51.3%	32.7%	39.9%	-0.11	55.8%	62.1%	58.8%	-0.01	58.0%	93.2%	71.5%	-0.06
Saliency_coding												
Incoder	77.8%	51.1%	61.7%	0.26	71.6%	77.4%	74.4%	0.35	62.2%	95.2%	75.3%	0.06
CodeGen	59.2%	40.4%	48.0%	0.04	58.8%	66.1%	62.2%	0.1	57.1%	92.6%	70.6%	-0.03
CodeParrot	62.2%	40.5%	49.0%	0.06	59.9%	65.7%	62.7%	0.11	58.1%	92.5%	71.4%	-0.01
GPT-J-6B	55.0%	36.1%	43.6%	-0.03	57.6%	64.6%	60.9%	0.07	57.2%	93.9%	71.1%	-0.03
PolyCoder	54.0%	35.9%	43.1%	-0.05	57.3%	64.0%	60.5%	0.06	57.5%	93.7%	71.3%	-0.03

Table 5.3. Human–model attention alignment calculated by the six self-attention-based methods in terms of San Martino's F1 score and Krippendorff's alpha score (in parentheses). The method that computes the highest Krippendorff's alpha score is highlighted in yellow for each model and each K value.

Method	*READING_first*			*CODING_first*		
	Top 5	**Top 10**	**Top 20**	**Top 5**	**Top 10**	**Top 20**
Incoder	31.9%(-0.10)	55.1%(0.05)	63.8%(-0.19)	37.9%(0.00)	54.9%(0.05)	63.4%(-0.2)
CodeGen	20%(-0.25)	46.3%(-0.07)	63.8%(-0.06)	20.6%(-0.25)	48.0%(-0.04)	64.0%(-0.06)
CodeParrot	37.9%(0.01)	55.9%(0.08)	65.2%(-0.03)	53.4%(0.24)	66.5%(0.30)	65.3%(-0.03)
GPT-J-6B	32.0%(-0.08)	50.4%(0.05)	61.3%(0.00)	36.7%(-0.01)	50.7%(0.06)	63.0%(0.05)
PolyCoder	41.1%(0.07)	60.5%(0.20)	65.1%(-0.03)	33.9%(-0.07)	55.4%(0.08)	64.7%(-0.04)
Method	*READING_last*			*CODING_last*		
	Top 5	**Top 10**	**Top 20**	**Top 5**	**Top 10**	**Top 20**
Incoder	32.8%(-0.01)	54.9%(0.05)	63.4%(-0.20)	37.1%(-0.02)	53.2%(0.02)	63.1%(-0.21)
CodeGen	37.4%(-0.01)	57.9%(0.12)	64.4%(-0.05)	30.2%(-0.02)	37.6%(-0.01)	64.4%(-0.05)
CodeParrot	52.9%(0.23)	67.5%(0.32)	65.4%(-0.03)	50.9%(0.20)	65.0%(0.27)	65.2%(-0.03)
GPT-J-6B	31%(-0.10)	50.4%(0.04)	62.1%(0.01)	34.1%(-0.05)	49.6%(0.03)	62.2%(0.01)
PolyCoder	30.3%(-0.13)	53.4%(0.03)	64.7%(-0.05)	33.4%(-0.07)	54.4%(0.07)	64.6%(-0.04)
Method	*READING_all*			*CODING_all*		
	Top 5	**Top 10**	**Top 20**	**Top 5**	**Top 10**	**Top 20**
Incoder	35.8%(-0.04)	50.1%(-0.06)	62.9%(-0.23)	38.1%(0.00)	54.9%(0.04)	63.4%(-0.21)
CodeGen	27.4%(-0.16)	44.2%(-0.13)	63.7%(-0.06)	38.3%(0.00)	56.2%(0.08)	64.5%(-0.04)
CodeParrot	39.8%(0.02)	58.2%(0.12)	65%(-0.04)	52.1%(0.22)	63.8%(0.24)	0.65(-0.04)
GPT-J-6B	24.1(-0.21)	44.8%(-0.06)	60.5%(-0.03)	29.3%(-0.13)	47.9%(-0.02)	61.3%(-0.02)
PolyCoder	31.7%(-0.11)	55.7%(0.08)	64.9%(-0.04)	39.6%(0.02)	60.6%(0.18)	65.1%(-0.03)

6. CONCLUSION AND FUTURE WORK

This dissertation set out to address one of the most consequential failure modes of generative models in software engineering: hallucination. Our thesis statement is that *hallucination of large language models in software engineering can be understood and mitigated by external guidance, self-reflection, and implicit signals.* This chapter revisits the statement in light of the three works, distills broader insights from building and studying these systems, and discusses future research directions.

6.1 Revisiting the Thesis Statement

Each work in this dissertation substantiates one or more pillars of the thesis statement.

External guidance mitigates hallucination (Chapters 3 and 4). ASSORT made the role of guidance explicit: an out-of-domain summarizer left alone produces hallucinated terminology and narratives under domain shift, but the same summarizer becomes trustworthy when a pre-trained natural language inference model—an external guide—re-anchors its output to sentences of the original post. Without any labeled training data, this indirectly supervised method outperforms six prior techniques, earns developer preference on par with a fully supervised counterpart, and even overtakes that counterpart when labeled data is scarce. AUTODOC extended the same principle to multi-document generation: by retrieving evidence with a fine-tuned dense passage retrieval model and extracting knowledge only from that evidence, it kept an LLM grounded while consolidating knowledge from thousands of Stack Overflow posts into API documents that are up to 77.7% more accurate than five baselines.

Self-reflection mitigates hallucination (Chapter 4). AUTODOC's knowledge validation component prompts the LLM to check each extracted knowledge snippet against its source post and to discard unsupported content. The ablation study confirmed that this self-checking step measurably improves the correctness of generated documents, and that a subsequent self-summarization step removes the redundancy that grounded extraction inevitably introduces. Notably, these safeguards allowed even a 7B-parameter open-source

model to approach the output quality of GPT-4o, showing that reflection can substitute for scale.

Implicit signals help us understand hallucination (Chapter 5). Our attention alignment study looked inside six code generation models and found their attention consistently misaligned with that of human programmers. The misalignment is diagnostic: five recurring attention patterns explain 27% of the analyzed incorrect code generations, including hallucinated code semantics. The study also identified which of twelve attention computation methods best matches human attention, providing a practical foundation for attention-based interpretability and debugging tools.

Taken together, the three works support the thesis statement from both directions: the first two demonstrate *mitigation* through external guidance and self-reflection in deployed, evaluated systems, and the third demonstrates *understanding* through implicit signals, closing the loop between suppressing hallucination and explaining it.

6.2 Insights

Beyond the individual results, building and studying these systems yielded several takeaways that we believe generalize.

Faithfulness can be designed for, not just trained for. The most reliable anti-hallucination mechanisms in this dissertation were architectural, not statistical. ASSORT is extractive by construction; AUTODOC only permits knowledge that survives a validation check against retrieved evidence. In both cases the guarantee comes from the shape of the pipeline rather than from hoping a model internalizes faithfulness during training. When correctness matters, constraining what a model *can* say is more dependable than improving what it *tends to* say.

Domain shift is a primary trigger of hallucination in software engineering. Across all three works, models faltered precisely where the software domain diverges from general text: general-domain summarizers hallucinated terminology on SO posts, LLMs hallucinated API knowledge for unpopular APIs that are underrepresented in their training data, and code models misread task descriptions whose salient tokens differ from natural

prose. Practitioners should treat low-resource corners of the domain—rare APIs, niche tags, unusual task phrasings—as hallucination hot spots that deserve extra grounding.

Verification is cheaper than generation. AutoDoc's validation step and Assort$_{IS}$'s NLI-based alignment both exploit the same asymmetry: checking whether a statement is supported by a source is easier than generating a correct statement outright. This asymmetry is what makes self-reflection profitable, and it explains why a small model wrapped in a generate-validate-summarize loop can rival a much larger unguarded model.

Applying NLP techniques to software engineering raises challenges that model scale alone will not solve. The three works repeatedly ran into obstacles that stem from the nature of the software domain rather than from any weakness of a particular model. The most fundamental one is the tension between *knowledge cutoff* and the pace of software evolution: an LLM's parametric knowledge is frozen at training time, while the software world it describes is not—libraries deprecate APIs, change default behaviors, and release new major versions on cycles far shorter than model training cycles. We saw this concretely in AutoDoc, where crowdsourced posts contain knowledge that is correct for one library version and obsolete for the next, and any parametric answer risks describing an API that no longer exists in that form. Related challenges include the scarcity of labeled data in software subdomains, terminology and discourse conventions that differ sharply from the general-domain corpora NLP models are trained on, and the fact that software claims have an executable ground truth that pure text training never touches. Looking forward, we expect continued progress in foundation models to dissolve some of these obstacles but not others. Larger and better-trained models will likely close the gaps that are, at bottom, about linguistic competence: understanding developer jargon, handling long and messy discussion threads, and generalizing from fewer labeled examples—the domain-shift problems that motivated much of Assort's design. What scaling cannot do is make a static parametric memory track a moving target: no matter how capable the model, knowledge minted after its cutoff—a new API, a changed default, a freshly discovered caveat—is simply absent, and a model trained to be fluent will fill that absence with plausible fabrication. Staying faithful to fast-moving software knowledge therefore requires the architectural mechanisms advocated in this dissertation—retrieval that injects up-to-date evidence at inference time, validation

that checks generated claims against that evidence, and ultimately verification against the software itself—regardless of how powerful the underlying foundation model becomes.

Attention misalignment is an early-warning signal. The attention study suggests that a model that stops attending to the tokens humans consider essential is a model about to go wrong. This opens the possibility of *predictive* hallucination detection: rather than auditing outputs after the fact, systems could monitor attention during generation and flag deviations before an incorrect artifact reaches the developer.

6.3 Future Work

The works in this dissertation also open several directions that we consider important next steps.

From post summarization to trustworthy knowledge curation at scale. ASSORT summarizes one post at a time. The natural next step is multi-document summarization over entire discussion threads and, beyond that, continuous curation of a living knowledge base: as questions, answers, and library versions evolve, a summarization system should detect when previously distilled knowledge becomes stale, reconcile conflicting answers, and preserve attribution so every claim remains traceable to its source. Achieving this would transform crowdsourced forums from searchable archives into self-maintaining, hallucination-resistant knowledge infrastructure for the entire developer community.

From generated documents to self-updating documentation ecosystems. AUTODOC generates a document as a snapshot. We envision documentation that maintains itself: pipelines that continuously ingest new crowdsourced discussions, bug trackers, and release notes; validate candidate knowledge against executable evidence (e.g., by synthesizing and running probe programs against the actual API); and negotiate updates with human maintainers. Grounding validation in execution rather than text alone would raise the bar from "supported by a post" to "verified against the software itself," effectively eliminating an entire class of hallucination in documentation.

From observing attention to aligning it. Our study measured attention alignment; the ambitious next step is to *optimize* for it. Human attention labels could serve as a train-

ing signal—through attention-regularized fine-tuning or preference optimization—to produce code models that provably attend to what matters. Attention could also be exploited at inference time: decoding strategies that detect attention drift mid-generation and re-focus the model on neglected constraints, or IDE integrations that visualize model attention so developers can spot misreadings before accepting a suggestion. Ultimately, we envision attention alignment as a standard evaluation axis for code LLMs, alongside functional correctness.

Toward hallucination-aware AI software engineers. The three pillars of this dissertation were developed in separate systems; unifying them is the larger prize. An AI programming assistant of the future should ground every claim in retrievable evidence, reflect on its own outputs before presenting them, and monitor its internal signals for early signs of deviation—all at once, and transparently to the developer. As LLM-based agents take on increasingly autonomous roles in software development, such hallucination-aware design will not be optional: it will be the foundation on which developer trust, and therefore adoption, rests. This dissertation provides evidence that each ingredient works; the next decade of research will determine how they compose.

6.4 Concluding Remarks

Hallucination is often framed as an inevitable tax on generative AI. The results of this dissertation argue otherwise. When generation is grounded in external evidence, checked through self-reflection, and illuminated by the model's own implicit signals, large language models can produce software engineering artifacts—summaries, documentation, and code—that developers measurably prefer and can justifiably trust. We hope the systems, datasets, and findings presented here serve as a foundation for the community's pursuit of trustworthy AI for software engineering.

REFERENCES

[1] B. Xu, Z. Xing, et al., "Answerbot: Automated generation of answer summary to developers' technical questions," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2017, pp. 706–716. [Online]. Available: https://doi.org/10.1109/ase.2017.8115681

[2] E. Aghajani et al., "Software documentation issues unveiled," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, 2019, pp. 1199–1210. [Online]. Available: https://doi.org/10.1109/icse.2019.00122

[3] OpenAI, *Gpt-4 technical report*, 2023. arXiv: 2303.08774 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2303.08774

[4] Anthropic, *The claude 3 model family: Opus, sonnet, haiku*, https://www.anthropic.com/claude-3-model-card, 2024.

[5] P. Vaithilingam et al., "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *CHI conference on human factors in computing systems extended abstracts*, 2022, pp. 1–7. [Online]. Available: https://doi.org/10.1145/3491101.3519665

[6] S. I. Ross, F. Martinez, S. Houde, M. Muller, and J. D. Weisz, "The programmer's assistant: Conversational interaction with a large language model for software development," in *Proceedings of the 28th International Conference on Intelligent User Interfaces*, 2023, pp. 491–514. [Online]. Available: https://doi.org/10.1145/3581641.3584037

[7] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, et al., "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[8] Z. Ji et al., "Survey of hallucination in natural language generation," *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–38, 2023. [Online]. Available: https://doi.org/10.1145/3571730

[9] L. Huang et al., "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions," *arXiv preprint arXiv:2311.05232*, 2023. [Online]. Available: https://arxiv.org/abs/2311.05232

[10] V. Rawte, A. Sheth, and A. Das, "A survey of hallucination in large foundation models," *arXiv preprint arXiv:2309.05922*, 2023. [Online]. Available: https://arxiv.org/abs/2309.05922

[11] F. Liu et al., "Beyond functional correctness: Exploring hallucinations in LLM-generated code," *IEEE Transactions on Software Engineering*, 2025. [Online]. Available: https://doi.org/10.1109/tse.2026.3657432

[12] Y. Tian et al., "CodeHalu: Investigating code hallucinations in LLMs via execution-based verification," in *Proceedings of the 39th AAAI Conference on Artificial Intelligence (AAAI)*, vol. 39, 2025, pp. 25 300–25 308. [Online]. Available: https://doi.org/10.1609/aaai.v39i24.34717

[13] S. Kabir, D. N. Udo-Imeh, B. Kou, and T. Zhang, "Who answers it better? an in-depth analysis of chatgpt and stack overflow answers to software engineering questions," *arXiv preprint arXiv:2308.02312*, 2023. [Online]. Available: https://arxiv.org/abs/2308.02312

[14] M. Kazemitabaar, X. Hou, A. Henley, B. J. Ericson, D. Weintrop, and T. Grossman, "How novices use llm-based code generators to solve cs1 coding tasks in a self-paced learning environment," in *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research*, 2023, pp. 1–12. [Online]. Available: https://doi.org/10.1145/3631802.3631806

[15] N. Gardella, R. Pettit, and S. L. Riggs, "Performance, workload, emotion, and self-efficacy of novice programmers using ai code generation," in *Proceedings of the 2024 Innovation and Technology in Computer Science Education V. 1*, 2024, pp. 290–296. [Online]. Available: https://doi.org/10.1145/3649217.3653615

[16] S. Nadi and C. Treude, "Essential sentences for navigating stack overflow answers," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2020, pp. 229–239. [Online]. Available: https://doi.org/10.1109/saner48275.2020.9054828

[17] B. Kou, Y. Di, M. Chen, and T. Zhang, "Sosum: A dataset of stack overflow post summaries," in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, IEEE, 2022, pp. 247–251. [Online]. Available: https://doi.org/10.1145/3524842.3528487

[18] W. Maalej and M. P. Robillard, "Patterns of knowledge in api reference documentation," *IEEE Transactions on software Engineering*, vol. 39, no. 9, pp. 1264–1282, 2013. [Online]. Available: https://doi.org/10.1109/tse.2013.12

[19] V. Karpukhin et al., "Dense passage retrieval for open-domain question answering," *arXiv preprint arXiv:2004.04906*, 2020. [Online]. Available: https://arxiv.org/abs/2004.04906

[20] S. Huang, M. Zhang, Y. Kang, and D. Wang, "Attributes-guided and pure-visual attention alignment for few-shot recognition," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, 2021, pp. 7840–7847. [Online]. Available: https://doi.org/10.1609/aaai.v35i9.16957

[21] A. Boggust, B. Hoover, A. Satyanarayan, and H. Strobelt, "Shared interest: Measuring human-ai alignment to identify recurring patterns in model behavior," in *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, 2022, pp. 1–17. [Online]. Available: https://doi.org/10.1145/3491102.3501965

[22] R. F. Silva, C. K. Roy, et al., "Recommending comprehensive solutions for programming tasks by mining crowd knowledge," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, IEEE, 2019, pp. 358–368. [Online]. Available: https://doi.org/10.1109/icpc.2019.00054

[23] H. Wang, X. Xia, et al., "Automatic solution summarization for crash bugs," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, IEEE, 2021, pp. 1286–1297. [Online]. Available: https://doi.org/10.1109/icse43902.2021.00117

[24] Z. Gao, X. Xia, et al., "Generating question titles for stack overflow from mined code snippets," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–37, 2020. [Online]. Available: https://doi.org/10.1145/3401026

[25] Z. Gao, X. Xia, D. Lo, J. Grundy, and Y.-F. Li, "Code2que: A tool for improving question titles from mined code snippets in stack overflow," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1525–1529. [Online]. Available: https://doi.org/10.1145/3468264.3473114

[26] S. Rastkar, G. C. Murphy, and G. Murray, "Summarizing software artifacts: A case study of bug reports," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, IEEE, vol. 1, 2010, pp. 505–514. [Online]. Available: https://doi.org/10.1145/1806799.1806872

[27] H. Liu, Y. Yu, et al., "Bugsum: Deep context understanding for bug report summarization," in *Proceedings of the 28th International Conference on Program Comprehension*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 94–105, ISBN: 9781450379588. [Online]. Available: https://doi.org/10.1145/3387904.3389272

[28] S. Rastkar, G. C. Murphy, et al., "Automatic summarization of bug reports," *IEEE Transactions on Software Engineering*, vol. 40, no. 4, pp. 366–380, 2014. DOI: 10.1109/TSE.2013.2297712

[29] S. Mani, R. Catherine, V. S. Sinha, and A. Dubey, "Ausum: Approach for unsupervised bug report summarization," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–11. [Online]. Available: https://doi.org/10.1145/2393596.2393607

[30] X. Li, H. Jiang, D. Liu, Z. Ren, and G. Li, "Unsupervised deep bug report summarization," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, IEEE, 2018, pp. 144–14 411. [Online]. Available: https://doi.org/10.1145/3196321.3196326

[31] Y. Liu, "Fine-tune BERT for extractive summarization," *arXiv preprint arXiv:1903.10318*, 2019. [Online]. Available: https://arxiv.org/abs/1903.10318

[32] S. Narayan, S. B. Cohen, et al., "Don't give me the details, just the summary!" *Topic-aware Convolutional Neural Networks for Extreme Summarization. In*, 2018. [Online]. Available: https://doi.org/10.18653/v1/D18-1206

[33] S. Narayan, S. B. Cohen, et al., "Ranking sentences for extractive summarization with reinforcement learning," *arXiv preprint arXiv:1802.08636*, 2018. [Online]. Available: https://arxiv.org/abs/1802.08636

[34] Y. Dong, Y. Shen, et al., "Banditsum: Extractive summarization as a contextual bandit," *arXiv preprint arXiv:1809.09672*, 2018. [Online]. Available: https://arxiv.org/abs/1809.09672

[35] S. Verma and V. Nidhi, "Extractive summarization using deep learning," *arXiv preprint arXiv:1708.04439*, 2017. [Online]. Available: https://arxiv.org/abs/1708.04439

[36] D. Miller, "Leveraging bert for extractive text summarization on lectures," *arXiv preprint arXiv:1906.04165*, 2019. [Online]. Available: https://arxiv.org/abs/1906.04165

[37] L. Ponzanelli, A. Bacchelli, and M. Lanza, "Seahawk: Stack overflow in the ide," in *2013 35th International Conference on Software Engineering (ICSE)*, IEEE, 2013, pp. 1295–1298. [Online]. Available: https://doi.org/10.1109/icse.2013.6606701

[38] L. Ponzanelli, G. Bavota, et al., "Mining stackoverflow to turn the ide into a self-confident programming prompter," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 102–111. [Online]. Available: https://doi.org/10.1145/2597073.2597077

[39] T. Ye, B. Xie, Y. Zou, and X. Chen, "Interrogative-guided re-ranking for question-oriented software text retrieval," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 115–120. [Online]. Available: https://doi.org/10.1145/2642937.2642953

[40] Y. Zou, T. Ye, Y. Lu, J. Mylopoulos, and L. Zhang, "Learning to rank for question-oriented software text retrieval (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2015, pp. 1–11. [Online]. Available: https://doi.org/10.1109/ase.2015.24

[41] M. Soliman, A. R. Salama, M. Galster, O. Zimmermann, and M. Riebisch, "Improving the search for architecture knowledge in online developer communities," in *2018 IEEE International Conference on Software Architecture (ICSA)*, IEEE, 2018, pp. 186–18 609. [Online]. Available: https://doi.org/10.1109/icsa.2018.00028

[42] M. M. Rahman and C. Roy, "Effective reformulation of query for code search using crowdsourced knowledge and extra-large data analytics," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2018, pp. 473–484. [Online]. Available: https://doi.org/10.1109/icsme.2018.00057

[43] H. Li, S. Li, et al., "Improving api caveats accessibility by mining api caveats knowledge graph," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2018, pp. 183–193. [Online]. Available: https://doi.org/10.1109/icsme.2018.00028

[44] C. Treude and M. P. Robillard, "Augmenting api documentation with insights from stack overflow," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, IEEE, 2016, pp. 392–403. [Online]. Available: https://doi.org/10.1145/2884781.2884800

[45] H. Li et al., "Improving api caveats accessibility by mining api caveats knowledge graph," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2018, pp. 183–193. [Online]. Available: https://doi.org/10.1109/icsme.2018.00028

[46] S. Subramanian, L. Inozemtseva, et al., "Live api documentation," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 643–652. [Online]. Available: https://doi.org/10.1145/2568225.2568313

[47] G. Uddin and F. Khomh, "Opiner: An opinion search and summarization engine for apis," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2017, pp. 978–983. [Online]. Available: https://doi.org/10.1109/ase.2017.8115715

[48] B. Lin, F. Zampetti, et al., "Pattern-based mining of opinions in q&a websites," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, 2019, pp. 548–559. [Online]. Available: https://doi.org/10.1109/icse.2019.00066

[49] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim, "Are code examples on an online q&a forum reliable?: A study of api misuse on stack overflow," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, IEEE, 2018, pp. 886–896. [Online]. Available: https://doi.org/10.1145/3180155.3180260

[50] A. Reinhardt, T. Zhang, et al., "Augmenting stack overflow with api usage patterns mined from github," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 880–883. [Online]. Available: https://doi.org/10.1145/3236024.3264585

[51] J. Zhou and R. J. Walker, "Api deprecation: A retrospective analysis and detection method for code examples on the web," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 266–277. [Online]. Available: https://doi.org/10.1145/2950290.2950298

[52] N. Zhang, Q. Huang, X. Xia, Y. Zou, D. Lo, and Z. Xing, "Chatbot4qr: Interactive query refinement for technical question retrieval," *IEEE Transactions on Software Engineering*, vol. 48, no. 4, pp. 1185–1211, 2022. DOI: 10.1109/TSE.2020.3016006

[53] G. Uddin, F. Khomh, and C. K. Roy, "Automatic api usage scenario documentation from technical q&a sites," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 3, pp. 1–45, 2021. [Online]. Available: https://doi.org/10.1145/3439769

[54] J. Stylos, A. Faulring, Z. Yang, and B. A. Myers, "Improving api documentation using api usage information," in *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, IEEE, 2009, pp. 119–126. [Online]. Available: https://doi.org/10.1109/vlhcc.2009.5295283

[55] U. Dekel and J. D. Herbsleb, "Improving api documentation usability with knowledge pushing," in *2009 IEEE 31st International Conference on Software Engineering*, IEEE, 2009, pp. 320–330. [Online]. Available: https://doi.org/10.1109/icse.2009.5070532

[56] C. Chen and K. Zhang, "Who asked what: Integrating crowdsourced faqs into api documentation," in *Companion Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 456–459. [Online]. Available: https://doi.org/10.1145/2591062.2591128

[57] M. Liu et al., "Generating query-specific class api summaries," in *Proceedings of the 2019 27th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2019, pp. 120–130. [Online]. Available: https://doi.org/10.1145/3338906.3338971

[58] S. Wang, N. Phan, Y. Wang, and Y. Zhao, "Extracting api tips from developer question and answer websites," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, IEEE, 2019, pp. 321–332. [Online]. Available: https://doi.org/10.1109/msr.2019.00058

[59] H. Yin, Y. Zheng, Y. Sun, and G. Huang, "An api learning service for inexperienced developers based on api knowledge graph," in *2021 IEEE International Conference on Web Services (ICWS)*, IEEE, 2021, pp. 251–261. [Online]. Available: https://doi.org/10.1109/icws53863.2021.00043

[60] M. Liu, X. Peng, A. Marcus, S. Xing, C. Treude, and C. Zhao, "Api-related developer information needs in stack overflow," *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4485–4500, 2021. [Online]. Available: https://doi.org/10.1109/tse.2021.3120203

[61] B. Xu, D. Ye, Z. Xing, X. Xia, G. Chen, and S. Li, "Predicting semantically linkable knowledge in developer online forums via convolutional neural network," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2016, pp. 51–62. [Online]. Available: https://doi.org/10.1145/2970276.2970357

[62] Y. Feng, R. Martins, J. Van Geffen, I. Dillig, and S. Chaudhuri, "Component-based synthesis of table consolidation and transformation tasks from examples," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 422–436, 2017. [Online]. Available: https://doi.org/10.1145/3140587.3062351

[63] J. Guo et al., "Towards complex text-to-sql in cross-domain database with intermediate representation," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019, pp. 4524–4535. [Online]. Available: https://doi.org/10.18653/v1/p19-1444

[64] Z. Feng et al., "Codebert: A pre-trained model for programming and natural language processing," in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, 2020, pp. 307–316. [Online]. Available: https://arxiv.org/abs/2002.08155

[65] Y. Zeng et al., "Recparser: A recursive semantic parsing framework for text-to-sql task.," in *IJCAI*, 2020, pp. 3644–3650. [Online]. Available: https://doi.org/10.24963/ijcai.2020/504

[66] S. Shen, X. Zhu, Y. Dong, Q. Guo, Y. Zhen, and G. Li, "Incorporating domain knowledge through task augmentation for front-end javascript code generation," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1533–1543. [Online]. Available: https://doi.org/10.1145/3540250.3558965

[67] B. Wang et al., "Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020, pp. 7567–7578. [Online]. Available: https://doi.org/10.18653/v1/2020.acl-main.677

[68] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 8696–8708. [Online]. Available: https://doi.org/10.18653/v1/2021.emnlp-main.685

[69] A. Andonian, Q. Anthony, et al., *GPT-NeoX: Large Scale Autoregressive Language Modeling in PyTorch*, version 0.0.1, Aug. 2021. DOI: 10.5281/zenodo.5879544 [Online]. Available: https://www.github.com/eleutherai/gpt-neox

[70] M. R. Parvez, W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Retrieval augmented code generation and summarization," in *Findings of the Association for Computational Linguistics: EMNLP 2021*, 2021, pp. 2719–2734. [Online]. Available: https://doi.org/10.18653/v1/2021.findings-emnlp.232

[71] L. Tunstall, L. von Werra, and T. Wolf, *Natural Language Processing with Transformers: Building Language Applications with Hugging Face*. O'Reilly Media, Incorporated, 2022, ISBN: 1098103246. [Online]. Available: https://www.oreilly.com/library/view/natural-language-processing-with-transformers/9781098103231/

[72] D. Fried et al., "Incoder: A generative model for code infilling and synthesis," in *The Eleventh International Conference on Learning Representations*, 2023. [Online]. Available: https://openreview.net/forum?id=hQwb-lbM6EL

[73] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "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. [Online]. Available: https://arxiv.org/abs/2202.13169

[74] E. Nijkamp, B. Pang, et al., "Codegen: An open large language model for code with multi-turn program synthesis," in *The Eleventh International Conference on Learning Representations*, 2023. [Online]. Available: https://arxiv.org/abs/2203.13474

[75] J. Devlin, M.-W. Chang, et al., "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018. [Online]. Available: https://arxiv.org/abs/1810.04805

[76] D. Guo et al., "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020. [Online]. Available: https://arxiv.org/abs/2009.08366

[77] S. Bubeck et al., "Sparks of artificial general intelligence: Early experiments with gpt-4," *arXiv preprint arXiv:2303.12712*, 2023. [Online]. Available: https://arxiv.org/abs/2303.12712

[78] G. Destefanis, S. Bartolucci, and M. Ortu, "A preliminary analysis on the code generation capabilities of gpt-3.5 and bard ai models for java functions," *arXiv preprint arXiv:2305.09402*, 2023. [Online]. Available: https://arxiv.org/abs/2305.09402

[79] Z. Zhang, H. Zhang, B. Shen, and X. Gu, "Diet code is healthy: Simplifying programs for pre-trained models of code," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1073–1084. [Online]. Available: https://doi.org/10.1145/3540250.3549094

[80] B. Chen et al., "Codet: Code generation with generated tests," *arXiv preprint arXiv:2207.10397*, 2022. [Online]. Available: https://arxiv.org/abs/2207.10397

[81] S. Chakraborty et al., "Natgen: Generative pre-training by "naturalizing" source code," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 18–30. [Online]. Available: https://doi.org/10.1145/3540250.3549162

[82] D. Zan, B. Chen, et al., "CERT: continual pre-training on sketches for library-oriented code generation," in *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, L. D. Raedt, Ed., ijcai.org, 2022, pp. 2369–2375. DOI: 10.24963/ijcai.2022/329 [Online]. Available: https://doi.org/10.24963/ijcai.2022/329

[83] J. Li, Y. Li, G. Li, Z. Jin, Y. Hao, and X. Hu, "Skcoder: A sketch-based approach for automatic code generation," *arXiv preprint arXiv:2302.06144*, 2023. [Online]. Available: https://arxiv.org/abs/2302.06144

[84] X. Liu, Y. Xia, and D. Lo, "An empirical study on the usage of transformer models for code completion," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2020, pp. 408–418. [Online]. Available: https://doi.org/10.1109/tse.2021.3128234

[85] D. Hendrycks et al., "Measuring coding challenge competence with apps," *NeurIPS*, 2021. [Online]. Available: https://arxiv.org/abs/2105.09938

[86] R. R. Rodrigues et al., "Studying the usage of text-to-text transfer transformer to support code-related tasks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, IEEE, 2021, pp. 327–336. [Online]. Available: https://doi.org/10.1109/icse43902.2021.00041

[87] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, and L. Zhang, "An extensive study on pre-trained models for program understanding and generation," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 39–51. [Online]. Available: https://doi.org/10.1145/3533767.3534390

[88] T. Y. Zhuo et al., "On robustness of prompt-based semantic parsing with large pre-trained language model: An empirical study on codex," in *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, 2023, pp. 1090–1102. [Online]. Available: https://doi.org/10.18653/v1/2023.eacl-main.77

[89] Y. Liu, C. Tantithamthavorn, Y. Liu, and L. Li, "On the reliability and explainability of automated code generation approaches," *arXiv preprint arXiv:2302.09587*, 2023. [Online]. Available: https://arxiv.org/abs/2302.09587

[90] O. Asare, M. Nagappan, and N. Asokan, "Is github's copilot as bad as humans at introducing vulnerabilities in code?" *arXiv preprint arXiv:2204.04741*, 2022. [Online]. Available: https://arxiv.org/abs/2204.04741

[91] H. Pearce et al., "Asleep at the keyboard? assessing the security of github copilot's code contributions," in *2022 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2022, pp. 754–768. [Online]. Available: https://doi.org/10.1109/sp46214.2022.9833571

[92] B. Yetistiren, I. Ozsoy, and E. Tuzun, "Assessing the quality of github copilot's code generation," in *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2022, pp. 62–71. [Online]. Available: https://doi.org/10.1145/3558489.3559072

[93] M. L. Siddiq et al., "An empirical study of code smells in transformer-based code generation techniques," in *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, IEEE, 2022, pp. 71–82. [Online]. Available: https://doi.org/10.1109/scam55253.2022.00014

[94] S. Barke, M. B. James, and N. Polikarpova, "Grounded copilot: How programmers interact with code-generating models," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 85–111, 2023. [Online]. Available: https://doi.org/10.1145/3586030

[95] C. Bird et al., "Taking flight with copilot: Early insights and opportunities of ai-powered pair-programming tools," *Queue*, vol. 20, no. 6, pp. 35–57, 2022. [Online]. Available: https://doi.org/10.1145/3582083

[96] M. Ciniselli, L. Pascarella, and G. Bavota, "To what extent do deep learning-based code recommenders generate predictions by cloning code from the training set?" In *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 167–178. [Online]. Available: https://doi.org/10.1145/3524842.3528440

[97] A. Karmakar and R. Robbes, "What do pre-trained code models know about code?" In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2021, pp. 1332–1336. [Online]. Available: https://doi.org/10.1109/ase51524.2021.9678927

[98] P. Liguori et al., "Can nmt understand me? towards perturbation-based evaluation of nmt models for code generation," in *2022 IEEE/ACM 1st International Workshop on Natural Language-Based Software Engineering (NLBSE)*, IEEE, 2022, pp. 59–66. [Online]. Available: https://doi.org/10.1145/3528588.3528653

[99] K. Zhang, G. Li, and Z. Jin, "What does transformer learn about source code?" *arXiv preprint arXiv:2207.08466*, 2022. [Online]. Available: https://arxiv.org/abs/2207.08466

[100] Y. Wan, W. Zhao, H. Zhang, Y. Sui, G. Xu, and H. Jin, "What do they capture? a structural analysis of pre-trained language models for source code," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2377–2388. [Online]. Available: https://doi.org/10.1145/3510003.3510050

[101] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-cam: Visual explanations from deep networks via gradient-based localization," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 618–626. [Online]. Available: https://doi.org/10.1109/iccv.2017.74

[102] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba, "Learning deep features for discriminative localization," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2921–2929. [Online]. Available: https://doi.org/10.1109/cvpr.2016.319

[103] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I 13*, Springer, 2014, pp. 818–833. [Online]. Available: https://arxiv.org/abs/1311.2901

[104] J. Chen, S. E. Li, and M. Tomizuka, "Interpretable end-to-end urban autonomous driving with latent deep reinforcement learning," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 6, pp. 5068–5078, 2021. [Online]. Available: https://doi.org/10.1109/tits.2020.3046646

[105] Z. Zhang et al., "LLM hallucinations in practical code generation: Phenomena, mechanism, and mitigation," *Proceedings of the ACM on Software Engineering*, vol. 2, no. ISSTA, 2025. [Online]. Available: https://doi.org/10.1145/3728894

[106] J. Spracklen, R. Wijewickrama, A. H. M. N. Sakib, A. Maiti, B. Viswanath, and M. Jadliwala, "We have a package for you! A comprehensive analysis of package hallucinations by code generating LLMs," in *Proceedings of the 34th USENIX Security Symposium (USENIX Security)*, 2025. [Online]. Available: https://arxiv.org/abs/2406.10279

[107] N. Jiang, Q. Li, L. Tan, and T. Zhang, "Collu-bench: A benchmark for predicting language model hallucinations in code," *arXiv preprint arXiv:2410.09997*, 2024. [Online]. Available: https://arxiv.org/abs/2410.09997

[108] P. Lewis et al., "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, 2020, pp. 9459–9474. [Online]. Available: https://arxiv.org/abs/2005.11401

[109] A. Eghbali and M. Pradel, "De-hallucinator: Mitigating LLM hallucinations in code generation tasks via iterative grounding," *arXiv preprint arXiv:2401.01701*, 2024. [Online]. Available: https://arxiv.org/abs/2401.01701

[110] A. Madaan et al., "Self-refine: Iterative refinement with self-feedback," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 36, 2023. [Online]. Available: https://doi.org/10.52202/075280-2019

[111] S. Dhuliawala et al., "Chain-of-verification reduces hallucination in large language models," in *Findings of the Association for Computational Linguistics: ACL 2024*, 2024, pp. 3563–3578. [Online]. Available: https://doi.org/10.18653/v1/2024.findings-acl.212

[112] P. Manakul, A. Liusie, and M. J. Gales, "Selfcheckgpt: Zero-resource black-box hallucination detection for generative large language models," *arXiv preprint arXiv:2303.08896*, 2023. [Online]. Available: https://arxiv.org/abs/2303.08896

[113] X. Wang et al., "Self-consistency improves chain of thought reasoning in language models," in *International Conference on Learning Representations (ICLR)*, 2023. [Online]. Available: https://arxiv.org/abs/2203.11171

[114] Y. Du, S. Li, A. Torralba, J. B. Tenenbaum, and I. Mordatch, "Improving factuality and reasoning in language models through multiagent debate," in *Proceedings of the 41st International Conference on Machine Learning*, 2024. [Online]. Available: https://arxiv.org/abs/2305.14325

[115] X. L. Li et al., "Contrastive decoding: Open-ended text generation as optimization," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2023, pp. 12 286–12 312. [Online]. Available: https://doi.org/10.18653/v1/2023.acl-long.687

[116] Y.-S. Chuang, Y. Xie, H. Luo, Y. Kim, J. Glass, and P. He, "Dola: Decoding by contrasting layers improves factuality in large language models," in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://arxiv.org/abs/2309.03883

[117] J. Zhang, D.-C. Juan, C. Rashtchian, C.-S. Ferng, H. Jiang, and Y. Chen, "Sled: Self logits evolution decoding for improving factuality in large language models," in *Advances in Neural Information Processing Systems*, 2024. [Online]. Available: https://arxiv.org/abs/2411.02433

[118] H. Zhang, H. Chen, M. Chen, and T. Zhang, "Active layer-contrastive decoding reduces hallucination in large language model generation," in *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, 2025. [Online]. Available: https://arxiv.org/abs/2505.23657

[119] K. Li, O. Patel, F. Viégas, H. Pfister, and M. Wattenberg, "Inference-time intervention: Eliciting truthful answers from a language model," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 36, 2023. [Online]. Available: https://doi.org/10.52202/075280-1797

[120] Z. Lan et al., "Factgen: Faithful text generation by factuality-aware pre-training and contrastive ranking fine-tuning," *Journal of Artificial Intelligence Research*, vol. 76, pp. 1281–1303, 2023. [Online]. Available: https://doi.org/10.1613/jair.1.14267

[121] K. Tian, E. Mitchell, H. Yao, C. D. Manning, and C. Finn, "Fine-tuning language models for factuality," in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://arxiv.org/abs/2311.08401

[122] S.-C. Lin et al., "Flame: Factuality-aware alignment for large language models," in *Advances in Neural Information Processing Systems*, 2024. [Online]. Available: https://doi.org/10.52202/079017-3671

[123] H. Zhang et al., "R-tuning: Instructing large language models to say 'i don't know'," in *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2024, pp. 7106–7132. [Online]. Available: https://doi.org/10.18653/v1/2024.naacl-long.394

[124] S. Kadavath, T. Conerly, A. Askell, T. Henighan, D. Drain, et al., "Language models (mostly) know what they know," *arXiv preprint arXiv:2207.05221*, 2022. [Online]. Available: https://arxiv.org/abs/2207.05221

[125] C. Burns, H. Ye, D. Klein, and J. Steinhardt, "Discovering latent knowledge in language models without supervision," in *The Eleventh International Conference on Learning Representations*, 2023. [Online]. Available: https://arxiv.org/abs/2212.03827

[126] A. Azaria and T. Mitchell, "The internal state of an LLM knows when it's lying," in *Findings of the Association for Computational Linguistics: EMNLP 2023*, 2023, pp. 967–976. [Online]. Available: https://doi.org/10.18653/v1/2023.findings-emnlp.68

[127] H. Duan, Y. Yang, and K. Y. Tam, "Do llms know about hallucination? an empirical investigation of llm's hidden states," *arXiv preprint arXiv:2402.09733*, 2024. [Online]. Available: https://arxiv.org/abs/2402.09733

[128] S. CH-Wang, B. Van Durme, J. Eisner, and C. Kedzie, "Do androids know they're only dreaming of electric sheep?" In *Findings of the Association for Computational Linguistics: ACL 2024*, 2024, pp. 4401–4420. [Online]. Available: https://doi.org/10.18653/v1/2024.findings-acl.260

[129] H. Orgad et al., "Llms know more than they show: On the intrinsic representation of llm hallucinations," in *The Thirteenth International Conference on Learning Representations*, 2025. [Online]. Available: https://arxiv.org/abs/2410.02707

[130] S. Farquhar, J. Kossen, L. Kuhn, and Y. Gal, "Detecting hallucinations in large language models using semantic entropy," *Nature*, vol. 630, pp. 625–630, 2024. [Online]. Available: https://doi.org/10.1038/s41586-024-07421-0

[131] J. Brandt, P. J. Guo, et al., "Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2009, pp. 1589–1598. [Online]. Available: https://doi.org/10.1145/1518701.1518944

[132] X. Xia, L. Bao, et al., "What do developers search for on the web?" *Empirical Software Engineering*, vol. 22, no. 6, pp. 3149–3185, 2017. [Online]. Available: https://doi.org/10.1007/s10664-017-9514-4

[133] Y. Wu, S. Wang, C.-P. Bezemer, and K. Inoue, "How do developers utilize source code from stack overflow?" *Empirical Software Engineering*, vol. 24, no. 2, pp. 637–673, 2019. [Online]. Available: https://doi.org/10.1007/s10664-018-9634-5

[134] R. Abdalkareem, E. Shihab, and J. Rilling, "What do developers use the crowd for? A study using stack overflow," *IEEE Software*, vol. 34, no. 2, pp. 53–60, 2017. [Online]. Available: https://doi.org/10.1109/ms.2017.31

[135] Q. Huang, X. Xia, et al., "Api method recommendation without worrying about the task-api knowledge gap," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2018, pp. 293–304. [Online]. Available: https://doi.org/10.1145/3238147.3238191

[136] D. Khashabi, S. Min, et al., "Unifiedqa: Crossing format boundaries with a single qa system," *EMNLP - findings*, 2020. [Online]. Available: https://doi.org/10.18653/v1/2020.findings-emnlp.171

[137] M. Lewis, Y. Liu, et al., "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," *arXiv preprint arXiv:1910.13461*, 2019. [Online]. Available: https://arxiv.org/abs/1910.13461

[138] Y. Liu and M. Lapata, "Text summarization with pretrained encoders," *arXiv preprint arXiv:1908.08345*, 2019. [Online]. Available: https://arxiv.org/abs/1908.08345

[139] *Document summarization on cnn/daily mail*, https://paperswithcode.com/sota/document-summarization-on-cnn-daily-mail, Accessed: 2022-3-29, 2022.

[140] M. P. Robillard and Y. B. Chhetri, "Recommending reference API documentation," *Empirical Software Engineering*, vol. 20, no. 6, pp. 1558–1586, 2015. [Online]. Available: https://doi.org/10.1007/s10664-014-9323-y

[141] G. Erkan and D. R. Radev, "LexRank: Graph-based lexical centrality as salience in text summarization," *Journal of Artificial Intelligence Research*, vol. 22, pp. 457–479, 2004. [Online]. Available: https://doi.org/10.1613/jair.1523

[142] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2009, pp. 1589–1598. [Online]. Available: https://doi.org/10.1145/1518701.1518944

[143] X. Xia, L. Bao, D. Lo, P. S. Kochhar, A. E. Hassan, and Z. Xing, "What do developers search for on the web?" *Empirical Software Engineering*, vol. 22, no. 6, pp. 3149–3185, 2017. [Online]. Available: https://doi.org/10.1007/s10664-017-9514-4

[144] S. Gupta and S. K. Gupta, "Abstractive summarization: An overview of the state of the art," *Expert Systems with Applications*, vol. 121, pp. 49–65, 2019. [Online]. Available: https://doi.org/10.1016/j.eswa.2018.12.011

[145] Y. Huang, X. Feng, X. Feng, and B. Qin, "The factual inconsistency problem in abstractive text summarization: A survey," *arXiv preprint arXiv:2104.14839*, 2021. [Online]. Available: https://arxiv.org/abs/2104.14839

[146] W. Kryściński, B. McCann, C. Xiong, and R. Socher, "Evaluating the factual consistency of abstractive text summarization," *arXiv preprint arXiv:1910.12840*, 2019. [Online]. Available: https://arxiv.org/abs/1910.12840

[147] Z. Cao, F. Wei, W. Li, and S. Li, "Faithful to the original: Fact aware neural abstractive summarization," in *thirty-second AAAI conference on artificial intelligence*, 2018. [Online]. Available: https://doi.org/10.1609/aaai.v32i1.11912

[148] W. Kryściński, N. S. Keskar, B. McCann, C. Xiong, and R. Socher, "Neural text summarization: A critical evaluation," *arXiv preprint arXiv:1908.08960*, 2019. [Online]. Available: https://arxiv.org/abs/1908.08960

[149] B. Goodrich, V. Rao, P. J. Liu, and M. Saleh, "Assessing the factual accuracy of generated text," in *proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, 2019, pp. 166–175. [Online]. Available: https://doi.org/10.1145/3292500.3330955

[150] T. Falke, L. F. Ribeiro, P. A. Utama, I. Dagan, and I. Gurevych, "Ranking generated summaries by correctness: An interesting but challenging application for natural language inference," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019, pp. 2214–2220. [Online]. Available: https://doi.org/10.18653/v1/p19-1213

[151] L. B. De Souza, E. C. Campos, et al., "Ranking crowd knowledge to assist software development," in *Proceedings of the 22nd International Conference on Program Comprehension*, 2014, pp. 72–82. [Online]. Available: https://doi.org/10.1145/2597008.2597146

[152] C. Treude, O. Barzilay, et al., "How do programmers ask and answer questions on the web?(nier track)," in *Proceedings of the 33rd international conference on software engineering*, 2011, pp. 804–807. [Online]. Available: https://doi.org/10.1145/1985793.1985907

[153] C. Rosen and E. Shihab, "What are mobile developers asking about? a large scale study using stack overflow," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1192–1223, 2016. [Online]. Available: https://doi.org/10.1007/s10664-015-9379-3

[154] M. Allamanis and C. Sutton, "Why, when, and what: Analyzing stack overflow questions by topic, type, and code," in *2013 10th Working conference on mining software repositories (MSR)*, IEEE, 2013, pp. 53–56. [Online]. Available: https://doi.org/10.1109/msr.2013.6624004

[155] M. L. McHugh, "Interrater reliability: The kappa statistic," *Biochemia medica*, vol. 22, no. 3, pp. 276–282, 2012. [Online]. Available: https://doi.org/10.11613/bm.2012.031

[156] J. Tabassum, M. Maddela, W. Xu, and A. Ritter, "Code and named entity recognition in stackoverflow," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2020. [Online]. Available: https://www.aclweb.org/anthology/2020.acl-main.443/

[157] C. Chen, Z. Xing, and W. Ximing, "Unsupervised software-specific morphological forms inference from informal discussions," in *The 39th International Conference on Software Engineering, Buenos Aires, Argentina*, IEEE, 2017. [Online]. Available: https://doi.org/10.1109/icse.2017.48

[158] *Bart large cnn*, https://huggingface.co/facebook/bart-large-cnn, Accessed: 2022-1-7, 2022.

[159] I. Dagan, O. Glickman, and B. Magnini, "The pascal recognising textual entailment challenge," in *Machine learning challenges workshop*, Springer, 2005, pp. 177–190. [Online]. Available: https://doi.org/10.1007/11736790_9

[160] W. Yin, D. Radev, and C. Xiong, "Docnli: A large-scale dataset for document-level natural language inference," *arXiv preprint arXiv:2106.09449*, 2021. [Online]. Available: https://arxiv.org/abs/2106.09449

[161] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014. [Online]. Available: https://arxiv.org/abs/1412.6980

[162] Q. Zhou, N. Yang, F. Wei, S. Huang, M. Zhou, and T. Zhao, "Neural document summarization by jointly learning to score and select sentences," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2018, pp. 654–663. [Online]. Available: https://doi.org/10.18653/v1/p18-1061

[163] A. See, P. J. Liu, and C. D. Manning, "Get to the point: Summarization with pointer-generator networks," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2017, pp. 1073–1083. [Online]. Available: https://doi.org/10.18653/v1/p17-1099

[164] A. Celikyilmaz, A. Bosselut, X. He, and Y. Choi, "Deep communicating agents for abstractive summarization," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, 2018, pp. 1662–1675. [Online]. Available: https://doi.org/10.18653/v1/n18-1150

[165] E. Sandhaus, "The new york times annotated corpus," *Linguistic Data Consortium, Philadelphia*, vol. 6, no. 12, e26752, 2008. [Online]. Available: https://catalog.ldc.upenn.edu/LDC2008T19

[166] *Stack exchange data explorer*, https://pypi.org/project/lexrank/, Accessed: 2022-8-15, 2022.

[167] J. Carbonell and J. Goldstein, "The use of mmr, diversity-based reranking for reordering documents and producing summaries," in *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, 1998, pp. 335–336. [Online]. Available: https://doi.org/10.1145/3130348.3130369

[168] H. Zhong and H. Mei, "An empirical study on api usages," *IEEE Transactions on Software Engineering*, vol. 45, no. 4, pp. 319–334, 2017. [Online]. Available: https://doi.org/10.1109/tse.2017.2782280

[169] G. Uddin, O. Baysal, L. Guerrouj, and F. Khomh, "Understanding how and why developers seek and analyze api-related opinions," *IEEE Transactions on Software Engineering*, vol. 47, no. 4, pp. 694–735, 2019. [Online]. Available: https://doi.org/10.1109/tse.2019.2903039

[170] M. P. Robillard and R. DeLine, "A field study of api learning obstacles," *Empirical Software Engineering*, vol. 16, pp. 703–732, 2011. [Online]. Available: https://doi.org/10.1007/s10664-010-9150-8

[171] C. Treude, J. Middleton, and T. Atapattu, "Beyond accuracy: Assessing software documentation quality," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1509–1512. [Online]. Available: https://doi.org/10.1145/3368089.3417045

[172] G. Uddin and M. P. Robillard, "How api documentation fails," *Ieee software*, vol. 32, no. 4, pp. 68–75, 2015. [Online]. Available: https://doi.org/10.1109/ms.2014.80

[173] T. Huang et al., "Planning and editing what you retrieve for enhanced tool learning," in *NAACL*, 2024. [Online]. Available: https://doi.org/10.18653/v1/2024.findings-naacl.61

[174] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar, "Inferring method specifications from natural language api descriptions," in *2012 34th international conference on software engineering (ICSE)*, IEEE, 2012, pp. 815–825. [Online]. Available: https://doi.org/10.1109/icse.2012.6227137

[175] D. Fucci, A. Mollaalizadehbahnemiri, and W. Maalej, "On using machine learning to identify knowledge in api reference documentation," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 109–119. [Online]. Available: https://doi.org/10.1145/3338906.3338943

[176] H. Li et al., "Improving API caveats accessibility by mining API caveats knowledge graph," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2018, pp. 183–193. [Online]. Available: https://doi.org/10.1109/icsme.2018.00028

[177] C. Treude and M. P. Robillard, "Augmenting api documentation with insights from stack overflow," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 392–403. DOI: 10.1145/2884781.2884800

[178] B. Xu, Z. Xing, X. Xia, and D. Lo, "Answerbot: Automated generation of answer summary to developers' technical questions," in *2017 32nd IEEE/ACM international conference on automated software engineering (ASE)*, IEEE, 2017, pp. 706–716. [Online]. Available: https://doi.org/10.1109/ase.2017.8115681

[179] S. Nadi and C. Treude, "Essential sentences for navigating stack overflow answers," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2020, pp. 229–239. [Online]. Available: https://doi.org/10.1109/saner48275.2020.9054828

[180] H. Wang, X. Xia, D. Lo, J. Grundy, and X. Wang, "Automatic solution summarization for crash bugs," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1286–1297. DOI: 10.1109/ICSE43902.2021.00117

[181] B. Kou, *Github repository*, Accessed: March 2025, 2025. [Online]. Available: https://github.com/BonanKou/icse2026

[182] K. Thayer, S. E. Chasins, and A. J. Ko, "A theory of robust api knowledge," *ACM Transactions on Computing Education (TOCE)*, vol. 21, no. 1, pp. 1–32, 2021. [Online]. Available: https://doi.org/10.1145/3444945

[183] M. Meng, S. Steinhardt, and A. Schubert, "Application programming interface documentation: What do software developers want?" *Journal of Technical Writing and Communication*, vol. 48, no. 3, pp. 295–330, 2018. [Online]. Available: https://doi.org/10.1177/0047281617721853

[184] X. Ren, J. Sun, Z. Xing, X. Xia, and J. Sun, "Demystify official api usage directives with crowdsourced api misuse scenarios, erroneous code examples and patches," in *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, 2020, pp. 925–936. [Online]. Available: https://doi.org/10.1145/3377811.3380430

[185] M. Monperrus, M. Eichberg, E. Tekes, and M. Mezini, "What should developers be aware of? an empirical study on the directives of api documentation," *Empirical Software Engineering*, vol. 17, pp. 703–737, 2012. [Online]. Available: https://doi.org/10.1007/s10664-011-9186-4

[186] J. Zhang, H. Jiang, Z. Ren, T. Zhang, and Z. Huang, "Enriching api documentation with code samples and usage scenarios from crowd knowledge," *IEEE Transactions on Software Engineering*, vol. 47, no. 6, pp. 1299–1314, 2019. [Online]. Available: https://doi.org/10.1109/tse.2019.2919304

[187] C. R. De Souza, D. Redmiles, L.-T. Cheng, D. Millen, and J. Patterson, "How a good software practice thwarts collaboration: The multiple roles of apis in software development," in *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, 2004, pp. 221–230. [Online]. Available: https://doi.org/10.1145/1029894.1029925

[188] S. Exchange, *Stack exchange data dump on internet archive*, https://archive.org/details/stackexchange, Accessed: 2024-09-09, 2024.

[189] J. Gao et al., "Collabcoder: A lower-barrier, rigorous workflow for inductive collaborative qualitative analysis with large language models," in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–29. [Online]. Available: https://doi.org/10.1145/3613904.3642002

[190] I. Singh et al., "Progprompt: Generating situated robot task plans using large language models," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2023, pp. 11 523–11 530. [Online]. Available: https://doi.org/10.1109/icra48891.2023.10161317

[191] A. Jabb, *How to cite stack overflow data dump?* Answer on Stack Overflow, accessed March 12, 2025, 2020. [Online]. Available: https://stackoverflow.com/a/62606723

[192] Y. Zhang et al., "Siren's song in the ai ocean: A survey on hallucination in large language models," *arXiv preprint arXiv:2309.01219*, 2023. [Online]. Available: https://arxiv.org/abs/2309.01219

[193] Z. Ji, T. Yu, Y. Xu, N. Lee, E. Ishii, and P. Fung, "Towards mitigating llm hallucination via self reflection," in *Findings of the Association for Computational Linguistics: EMNLP 2023*, 2023, pp. 1827–1843. [Online]. Available: https://doi.org/10.18653/v1/2023.findings-emnlp.123

[194] F. Boudin, M. El-Bèze, and J.-M. Torres-Moreno, "A scalable mmr approach to sentence scoring for multi-document update summarization," in *Coling 2008: Companion volume: Posters*, 2008, pp. 23–26. [Online]. Available: https://aclanthology.org/C08-2006/

[195] S. Robertson, H. Zaragoza, et al., "The probabilistic relevance framework: Bm25 and beyond," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009. [Online]. Available: https://doi.org/10.1561/1500000019

[196] L. Wang et al., "Text embeddings by weakly-supervised contrastive pre-training," *arXiv preprint arXiv:2212.03533*, 2022. [Online]. Available: https://arxiv.org/abs/2212.03533

[197] A. Grattafiori et al., "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024. [Online]. Available: https://arxiv.org/abs/2407.21783

[198] R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton, "Adaptive mixtures of local experts," *Neural computation*, vol. 3, no. 1, pp. 79–87, 1991. [Online]. Available: https://doi.org/10.1162/neco.1991.3.1.79

[199] B. Kou, M. Chen, and T. Zhang, "Automated summarization of stack overflow posts," *arXiv preprint arXiv:2305.16680*, 2023. [Online]. Available: https://arxiv.org/abs/2305.16680

[200] P. Singhal, T. Goyal, J. Xu, and G. Durrett, "A long way to go: Investigating length correlations in rlhf," in *First Conference on Language Modeling*, 2024. [Online]. Available: https://arxiv.org/abs/2310.03716

[201] K. Saito, A. Wachi, K. Wataoka, and Y. Akimoto, "Verbosity bias in preference labeling by large language models," *arXiv preprint arXiv:2310.10076*, 2023. [Online]. Available: https://arxiv.org/abs/2310.10076

[202] Y. Dubois, B. Galambosi, P. Liang, and T. B. Hashimoto, "Length-controlled alpacaeval: A simple way to debias automatic evaluators," *arXiv preprint arXiv:2404.04475*, 2024. [Online]. Available: https://arxiv.org/abs/2404.04475

[203] G. Adams, A. Fabbri, F. Ladhak, E. Lehman, and N. Elhadad, "From sparse to dense: Gpt-4 summarization with chain of density prompting," in *Proceedings of the 4th New Frontiers in Summarization Workshop*, 2023, pp. 68–74. [Online]. Available: https://doi.org/10.18653/v1/2023.newsum-1.7

[204] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2021, pp. 2655–2668. [Online]. Available: https://doi.org/10.18653/v1/2021.naacl-main.211

[205] A. Elnaggar et al., "Codetrans: Towards cracking the language of silicon's code through self-supervised deep learning and high performance computing," *arXiv preprint arXiv:2104.02443*, 2021. [Online]. Available: https://arxiv.org/abs/2104.02443

[206] P. Jain and A. Jain, "Contrastive code representation learning," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021. [Online]. Available: https://arxiv.org/abs/2007.04973

[207] B. Wang and A. Komatsuzaki, *GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model*, https://github.com/kingoflolz/mesh-transformer-jax, May 2021.

[208] H. Zhao, W. Zhou, X. Hou, and H. Zhu, "Double attention for multi-label image classification," *IEEE Access*, vol. 8, pp. 225 539–225 550, 2020. [Online]. Available: https://doi.org/10.1109/access.2020.3044446

[209] R. C. Fong, W. J. Scheirer, and D. D. Cox, "Using human brain activity to guide machine learning," *Scientific reports*, vol. 8, no. 1, p. 5397, 2018. [Online]. Available: https://doi.org/10.1038/s41598-018-23618-6

[210] S. Jia et al., "Biometric recognition through eye movements using a recurrent neural network," in *2018 IEEE International Conference on Big Knowledge (ICBK)*, IEEE, 2018, pp. 57–64. [Online]. Available: https://doi.org/10.1109/icbk.2018.00016

[211] C. Melício et al., "Object detection and localization with artificial foveal visual attention," in *2018 Joint IEEE 8th international conference on development and learning and epigenetic robotics (ICDL-EpiRob)*, IEEE, 2018, pp. 101–106. [Online]. Available: https://doi.org/10.1109/devlrn.2018.8761032

[212] A. Nunes, R. Figueiredo, and P. Moreno, "Learning to search for objects in images from human gaze sequences," in *Image Analysis and Recognition: 17th International Conference*, Springer, 2020, pp. 280–292. [Online]. Available: https://doi.org/10.1007/978-3-030-50347-5_25

[213] A. Stocco et al., "Thirdeye: Attention maps for safe autonomous driving systems," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12. [Online]. Available: https://doi.org/10.1145/3551349.3556968

[214] C. Hazard et al., "Importance is in your attention: Agent importance prediction for autonomous driving," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 2532–2535. [Online]. Available: https://doi.org/10.1109/cvprw56347.2022.00284

[215] I. Kotseruba, A. Rasouli, and J. K. Tsotsos, "Joint attention in autonomous driving (jaad)," *arXiv preprint arXiv:1609.04741*, 2016. [Online]. Available: https://arxiv.org/abs/1609.04741

[216] Y. Gao et al., "Aligning eyes between humans and deep neural network through interactive attention alignment," *Proceedings of the ACM on Human-Computer Interaction*, vol. 6, no. CSCW2, pp. 1–28, 2022. [Online]. Available: https://doi.org/10.1145/3555590

[217] A. Bansal, B. Sharif, and C. McMillan, "Towards modeling human attention from eye movements for neural source code summarization," *Proceedings of the ACM on Human-Computer Interaction*, vol. 7, no. ETRA, pp. 1–19, 2023. [Online]. Available: https://doi.org/10.1145/3591136

[218] D. Hendrycks et al., "Aligning ai with shared human values," *arXiv preprint arXiv:2008.02275*, 2020. [Online]. Available: https://arxiv.org/abs/2008.02275

[219] D. Shin, "The effects of explainability and causability on perception, trust, and acceptance: Implications for explainable ai," *International Journal of Human-Computer Studies*, vol. 146, p. 102 551, 2021. [Online]. Available: https://doi.org/10.1016/j.ijhcs.2020.102551

[220] K. Weitz et al., "" do you trust me?" increasing user-trust by integrating virtual agents in explainable ai interaction design," in *Proceedings of the 19th ACM International Conference on Intelligent Virtual Agents*, 2019, pp. 7–9. [Online]. Available: https://doi.org/10.1145/3308532.3329441

[221] J. Austin, A. Odena, et al., "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021. [Online]. Available: https://arxiv.org/abs/2108.07732

[222] *Chatgpt*, http://chat.openai.com, 2023.

[223] *Codeparrot*, https://github.com/huggingface/transformers/tree/main/examples/research_projects/codeparrot, 2022.

[224] W. Chen, E. Matusov, S. Khadivi, and J.-T. Peter, "Guided alignment training for topic-aware neural machine translation," *arXiv preprint arXiv:1607.01628*, 2016. [Online]. Available: https://arxiv.org/abs/1607.01628

[225] M. Paltenghi and M. Pradel, "Thinking like a developer? comparing the attention of humans with neural models of code," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2021, pp. 867–879. [Online]. Available: https://doi.org/10.1109/ase51524.2021.9678712

[226] M. R. I. Rabin, V. J. Hellendoorn, and M. A. Alipour, "Understanding neural code intelligence through program simplification," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 441–452. [Online]. Available: https://doi.org/10.1145/3468264.3468539

[227] J. Bensemann et al., "Eye gaze and self-attention: How humans and transformers attend words in sentences," in *Proceedings of the Workshop on Cognitive Modeling and Computational Linguistics*, 2022, pp. 75–87. [Online]. Available: https://doi.org/10.18653/v1/2022.cmcl-1.9

[228] S. Lu et al., "Codexglue: A machine learning benchmark dataset for code understanding and generation," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021. [Online]. Available: https://arxiv.org/abs/2102.04664

[229] T. Chen, L. Li, and X. Li, "Code generation from natural language: A survey," *arXiv preprint arXiv:2101.00162*, 2021. [Online]. Available: https://arxiv.org/abs/2101.00162

[230] K. Papineni et al., "Bleu: A method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318. [Online]. Available: https://doi.org/10.3115/1073083.1073135

[231] S. Ren et al., "Codebleu: A method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020. [Online]. Available: https://arxiv.org/abs/2009.10297

[232] S. Hooker, D. Erhan, P.-J. Kindermans, and B. Kim, "Evaluating feature importance estimates," *arXiv preprint arXiv:1806.10758*, 2018. [Online]. Available: https://arxiv.org/abs/1806.10758

[233] E. Niebur, "Saliency map," *Scholarpedia*, vol. 2, no. 8, p. 2675, 2007. [Online]. Available: http://www.scholarpedia.org/article/Saliency_map

[234] C. Molnar, *Interpretable Machine Learning: A Guide for Making Black Box Models Explainable*. Lulu.com, 2020. [Online]. Available: https://christophm.github.io/interpretable-ml-book/

[235] K. Clark, U. Khandelwal, O. Levy, and C. D. Manning, "What does bert look at? an analysis of bert's attention," in *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, 2019, pp. 276–286. [Online]. Available: https://doi.org/10.18653/v1/w19-4828

[236] A. Galassi, M. Lippi, and P. Torroni, "Attention in natural language processing," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 10, pp. 4291–4308, 2021. [Online]. Available: https://doi.org/10.1109/TNNLS.2020.3019893

[237] J. Li, W. Monroe, and D. Jurafsky, "Understanding neural networks through representation erasure," *arXiv preprint arXiv:1612.08220*, 2016. [Online]. Available: https://arxiv.org/abs/1612.08220

[238] S. Vashishth, S. Upadhyay, G. S. Tomar, and M. Faruqui, "Attention interpretability across nlp tasks," *arXiv preprint arXiv:1909.11218*, 2019. [Online]. Available: https://arxiv.org/abs/1909.11218

[239] M. Sundararajan, A. Taly, and Q. Yan, "Axiomatic attribution for deep networks," in *International conference on machine learning*, PMLR, 2017, pp. 3319–3328. [Online]. Available: https://arxiv.org/abs/1703.01365

[240] M. Denil, A. Demiraj, and N. de Freitas, "Extraction of salient sentences from labelled documents," *arXiv preprint arXiv:1412.6815*, 2014. [Online]. Available: https://arxiv.org/abs/1412.6815

[241] A. Shrikumar, P. Greenside, and A. Kundaje, "Learning important features through propagating activation differences," in *International conference on machine learning*, PMLR, 2017, pp. 3145–3153. [Online]. Available: https://arxiv.org/abs/1704.02685

[242] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional networks: Visualising image classification models and saliency maps," *arXiv preprint arXiv:1312.6034*, 2013. [Online]. Available: https://arxiv.org/abs/1312.6034

[243] Z. Wu et al., "Perturbed masking: Parameter-free probing for analyzing and interpreting bert," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020, pp. 4166–4176. [Online]. Available: https://doi.org/10.18653/v1/2020.acl-main.383

[244] M. T. Ribeiro et al., "" why should i trust you?" explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, 2016, pp. 1135–1144. [Online]. Available: https://doi.org/10.18653/v1/n16-3020

[245] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, 2017, pp. 4765–4774. [Online]. Available: https://arxiv.org/abs/1705.07874

[246] S. Liu, Z. Li, T. Li, V. Srikumar, V. Pascucci, and P.-T. Bremer, "NLIZE: A perturbation-driven visual interrogation tool for analyzing and interpreting natural language inference models," *IEEE Transactions on Visualization and Computer Graphics*, vol. 25, no. 1, pp. 651–660, 2019. [Online]. Available: https://doi.org/10.1109/tvcg.2018.2865230

[247] J. L. Fleiss, "Measuring nominal scale agreement among many raters.," *Psychological bulletin*, vol. 76, no. 5, p. 378, 1971. [Online]. Available: https://doi.org/10.1037/h0031619

[248] *Gpt-4 parameters: Unlimited guide nlp's game-changer*, https://medium.com/@mlubbad/the-ultimate-guide-to-gpt-4-parameters-everything-you-need-to-know-about-nlps-game-changer-109b8767855a, 2023.

[249] G. Da San Martino et al., "Fine-grained analysis of propaganda in news article," in *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)*, Association for Computational Linguistics, 2019, pp. 5636–5646. [Online]. Available: https://doi.org/10.18653/v1/d19-1565

[250] K. Krippendorff, *Content Analysis: An Introduction to Its Methodology*, 4th. SAGE Publications, 2018. [Online]. Available: https://us.sagepub.com/en-us/nam/content-analysis/book258450

[251] J. Sarker, S. Sultana, S. R. Wilson, and A. Bosu, "Toxispanse: An explainable toxicity detection in code review comments," in *2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, IEEE, 2023, pp. 1–12. [Online]. Available: https://doi.org/10.1109/esem56168.2023.10304855

[252] K. Krippendorff, "Reliability in content analysis: Some common misconceptions and recommendations," *Human communication research*, vol. 30, no. 3, pp. 411–433, 2004. [Online]. Available: https://doi.org/10.1093/hcr/30.3.411

[253] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, 1960. [Online]. Available: https://doi.org/10.1177/001316446002000104

[254] N. Yefet, U. Alon, and E. Yahav, "Adversarial examples for models of code," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020. [Online]. Available: https://doi.org/10.1145/3428230

[255] H. Zhang, Z. Li, G. Li, L. Ma, Y. Liu, and Z. Jin, "Generating adversarial examples for holding robustness of source code processing models," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, 2020, pp. 1169–1176. [Online]. Available: https://doi.org/10.1609/aaai.v34i01.5469

[256] P. Bielik and M. Vechev, "Adversarial robustness for code," in *International Conference on Machine Learning*, PMLR, 2020, pp. 896–907. [Online]. Available: https://arxiv.org/abs/2002.04694

[257] S. Srikant et al., "Generating adversarial computer programs using optimized obfuscations," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://arxiv.org/abs/2103.11882

VITA

Bonan Kou received his PhD in Computer Science from Purdue University, West Lafayette, Indiana, advised by Dr. Tianyi Zhang. His research focuses on trustworthy AI for software engineering, in particular understanding and mitigating hallucination of large language models in developer-facing tasks. His work has been published at premier software engineering venues, including ICSE and FSE.

Selected publications:

- Bonan Kou, Muhao Chen, Tianyi Zhang. "Automated Summarization of Stack Overflow Posts." In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE 2023)*.
- Bonan Kou, Shengmai Chen, Zhijie Wang, Lei Ma, Tianyi Zhang. "Do Large Language Models Pay Similar Attention Like Human Programmers When Generating Code?" In *Proceedings of the ACM International Conference on the Foundations of Software Engineering (FSE 2024)*.
- Bonan Kou, Zijie Zhou, Muhao Chen, Tianyi Zhang. "Automating API Documentation from Crowdsourced Knowledge." In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE 2026)*.