TASKS, CHALLENGES, AND PATHS TOWARDS AI FOR SOFTWARE ENGINEERING

Alex Gu
MIT CSAIL
gua@mit.edu

Naman Jain[⋆]
University of California, Berkeley
naman_jain@berkeley.edu

Wen-Ding Li[⋆]
Cornell University
wl678@cornell.edu

Manish Shetty[⋆]
University of California, Berkeley
manishs@berkeley.edu

Yijia Shao
Stanford University
shaoyj@cs.stanford.edu

Ziyang Li
University of Pennsylvania
liby99@seas.upenn.edu

Diyi Yang
Stanford University
diyiy@cs.stanford.edu

Kevin Ellis
Cornell University
kellis@cornell.edu

Koushik Sen
University of California, Berkeley
ksen@berkeley.edu

Armando Solar-Lezama
MIT CSAIL
asolar@csail.mit.edu

ABSTRACT

AI for software engineering has made remarkable progress, becoming a notable success within generative AI. Despite this, achieving fully automated software engineering is still a significant challenge, requiring research efforts across both academia and industry. In this position paper, our goal is threefold. First, we provide a taxonomy of measures and tasks to categorize work towards AI software engineering. Second, we outline the key bottlenecks permeating today's approaches. Finally, we highlight promising paths towards making progress on these bottlenecks to guide future research in this rapidly maturing field.

1 INTRODUCTION

AI for software engineering has made remarkable progress, becoming a notable success within generative AI. Despite this, achieving fully automated software engineering remains a significant challenge, requiring significant research efforts across academia and industry. In this paper, we provide an opinionated view of the tasks, challenges, and promising directions towards achieving this goal.

Many existing surveys focus on particular topics discussed in this work. Liang et al. (2024) and Sergeyuk et al. (2025) survey the successes and challenges of AI programming assistants, (Wang et al., 2024a) survey using LLMs for software testing, and Joel et al. (2024) survey using LLMs in low-resource and domain-specific languages, and Zhang et al. (2023) focus on automated program repair, both with and without LLMs. Finally, Yang et al. (2024) is a roadmap for formal mathematical reasoning and has some overlap with our discussion on software verification. In addition, many works discuss the current state, challenges, and future of AI for software engineering (Fan et al., 2023; Ozkaya, 2023; Wong et al., 2023; Zheng et al., 2023; Hou et al., 2024; Jin et al., 2024; Wan et al., 2024). Our work draws inspiration from them, and we recommend that the reader consult them for alternative perspectives.

In this position paper, our goal is threefold. In Sec. 2, we provide a structured way to think about progress in the field and list tasks important in AI for software engineering. We also provide three measures for categorizing concrete realizations of the task: the scale of the problem, the amount of algorithmic complexity, and the level of human intervention. In Sec. 3, we highlight nine challenges in the field that today's models face, each of which applies to several tasks. In Sec. 4, we posit five research themes that are promising to tackle the aforementioned challenges. We hope that through our paper, the reader can appreciate the progress the field has made, understand the shortcoming of today's state-of-the-art models, and gain concrete research ideas for tackling these challenges.

2 Tasks in AI Software Engineering

We first provide a taxonomy of tasks in AI software engineering. To provide a structured way to consider concrete realizations of each task, we define three measures that apply across them: scope, algorithmic complexity, and level of human intervention. To achieve an AI software engineer, we strive for AI to be capable across all three measures.

Scope Measure: We define three levels of scope, the extent of changes that the AI makes to the codebase. *Function-level* scope refers to single, self-contained functions such as in HumanEval (Chen et al., 2021). *Self-contained unit* scope goes beyond singular functions and to larger chunks of code such as entire files and classes. Finally, *project-level* scope refers to larger codebases such as entire repositories, such as in Commit0 (Zhao et al., 2024) and SWE-Bench (Jimenez et al., 2024).

Algorithmic Complexity Measure: Tasks require a wide range of reasoning abilities when it comes to devising algorithms to solve them. *Low algorithmic complexity* tasks require little to no reasoning, such as writing CRUD (create, read, update, delete) applications or using APIs. *Medium algorithmic complexity* tasks include most LeetCode problems, finding inputs to fuzz simple programs, and reasoning about execution behavior of multithreaded programs. *High algorithmic complexity* tasks require meticulous and challenging levels of algorithmic reasoning, either because the algorithm is complex or because the problem requires clever thinking and insights. This includes difficult competition programming problems, writing large thread-safe concurrent programs, cracking cryptographic ciphers, and solving SMT-like problems. Many popular coding benchmarks are function-level, medium-high algorithmic complexity, such as APPS (Hendrycks et al., 2021), CodeContests (Li et al., 2022), and LiveCodeBench (Jain et al., 2024b).

Level of Human Intervention Measure: AI coding is a collaborative task. Following the autonomy taxonomy in Morris et al. (2023), we define three levels of human intervention. *Low autonomy* is when the human has full control over the task and uses AI to automate simple sub-tasks. *Medium autonomy* is when there is a similar amount of human-AI collaboration, with interactive coordination of goals and tasks. *High autonomy* is when AI drives the tasks, with the human providing feedback.

Next, with our taxonomy of measures in place, we turn to the set of tasks that are reflective of the tasks and capabilities of a human software engineer.

2.1 Code Generation

Code generation is the task of generating code from a specification. In *code completion* (e.g. Github Copilot), the specification takes the form of a pre-existing code snippet. In *natural language to code*, where the specification is a natural language description containing information such as the task description, input-output examples, libraries to use, and other requirements about the code. The difficulty of a code generation task depends highly on its scope and algorithmic complexity.

2.2 Code Transformation

Code Optimization: Transforming programs to improve performance characteristics while maintaining functional correctness is a critical software task. These optimizations span multiple dimensions, including execution time, memory usage, energy consumption, and parallel efficiency. The challenge lies not only in identifying opportunities for optimization but also in reasoning about complex performance trade-offs across different hardware architectures and usage scenarios.

Code Translation: Code translation is the task of translating code from a source language to a target language. This task is difficult because it requires understanding, in depth, how constructs expressed in the source language map to the target language. This requires reasoning at a very high level of abstraction. In the industry, there are many large-scale code translation attempts (like C to Rust, or Python to C), and this task is also useful for porting legacy code to modern programming languages.

Code Refactoring: Code refactoring presents a unique challenge in code transformation because success extends far beyond functional correctness or metrics. The goal is often to improve code maintainability, readability, or extensibility—qualities that are inherently difficult to quantify and highly context-dependent. For instance, extracting shared functionality into helper methods presents trade-offs between modularity and cognitive complexity (Parnas, 1972). These challenges are further compounded by the need to understand implicit trade-offs customized to specific codebases, respect conventions, and reason about the long-term maintenance implications of structural changes.

2.3 Code Understanding

Understanding the moving parts of a codebase is an important skill for being a stellar engineer. Code can often be difficult to understand due to many wrapper functions, error-handling boilerplate, deep call stacks, and sometimes even poor code cleanliness. We identify four practical code-understanding tasks that we find practical and important:

Code Summarization and Documentation: To ensure maintainability, readability, and ease of collaboration, code must be documented well. Writing good documentation is challenging because it needs to be written cleanly and crisply, describing both what the function does and how the function works. Importantly, it must also anticipate and address any misunderstandings that a programmer might have, such as potential side effects or special cases.

Code Navigation: It is challenging to navigate and get accustomed to a mature codebase for new joiners. Code navigation means finding where relevant functionality is implemented. Doing this well requires understanding both the high-level layout of where every functionality lies in the codebase and the low-level understanding of which helper functions are used to implement each functionality.

Code Question Answering: The holy grail of code understanding is the ability to answer arbitrarily complex questions about a codebase, which requires sophisticated code understanding and reasoning abilities. Models should not hallucinate or give incorrect information that skews a developer's mental model of the code. Beyond other tasks mentioned in this section, developers might commonly ask questions related to data flow (when and where data structures get mutated), code functionality (whether there are any side effects), performance characteristics (determining the runtime and memory complexity of a function), or error handling (whether certain corner cases are handled).

2.4 Code Debugging

Software inevitably will have bugs that vary in scope and algorithmic complexity. Minor bugs might miss a few corner cases and require only a few lines of modification, e.g. *simple, stupid bugs* (Mosolygó et al., 2021). Complex bugs (e.g. concurrency bugs) might have very high algorithmic complexity. Because bugs can often pass tests and syntax checks, even detecting them can be tricky and requires thorough testing both during development and production work. The difficulty of fixing a bug depends on scope and algorithmic complexity. Function-level, low algorithmic complexity bugs can easily fixed by feeding in the error information. Repository-level, high algorithmic complexity bugs could require locating the bug, re-thinking the algorithm, and restructuring or refactoring the codebase. These changes must be general and should not break the functionality of other parts of the codebase.

2.5 Scaffolding and Meta-Code

For a software system to work, the core logic must be written well, but that is not enough. Many infrastructural aspects must be in place to support the software. We group these into two main categories: *scaffolding* and *meta-code*. We define *scaffolding* to be code that is important to make the system work but does not actually participate in the execution of its main logic. Examples of scaffolding include test harnesses, configuration files, CI/CD code, Makefiles, Dockerfiles, sandbox databases, and preprocessors. In contrast, we define *meta-code* as a task outside of the code that must be done to get the software running the property. Examples of meta-code include setting up Google authentication, subscribing to APIs, managing file storage, and generating API tokens.

2.6 Formal Verification

The task of formal verification involves generating checkable, mechanized proofs that can guarantee that a piece of code works as intended. Formal verification of software is necessary in mission-critical applications such as aircraft software, where it is crucial that code is correct with absolute certainty. Over the years, there have been countless programming languages designed specifically for formal verification. Some of the popular ones include TLA (Lamport, 1994), Coq (The Coq Development Team, 2024), Lean (De Moura et al., 2015), Dafny (Leino, 2010), Isabelle (Nipkow et al., 2002), and Verus (Lattuada et al., 2024).

In the formal methods literature, there are two major classes of formal verification: full functional verification (FFV) and property verification (PV). In FFV, the goal is to design a complete and precise formal specification that captures the desired behavior of the implementation. While FFV provides a complete set of guarantees, it is usually sufficient to opt for PV, where a few key properties of a system are proven correct, such as ensuring that two threads do not simultaneously enter a critical section of a program. Unlike in FFV, the specification is simpler to write and the challenge is finding the right domain-specific tool and algorithms to write the proof.

3 Challenges

While the field of AI for code has made fruitful progress, cutting-edge AI still struggles with SWE tasks, especially at larger scopes and higher levels of algorithmic complexity. Next, we discuss ten key challenges in AI for code. Each challenge spans multiple tasks, and progress on any can lead to improvements on many tasks at once. We order these roughly by how easy it is to resolve them.

3.1 Evaluation and Benchmarks

Our taxonomy of tasks and measures highlights some of the shortcomings of today's evaluations and benchmarks. For example, the majority of today's coding evaluations have no level of human intervention, with a few, such as Copilot-Arena (Chi et al., 2024), having low to medium autonomy. HumanEval, MBPP, APPS, CodeContests, and LiveCodeBench are all at function-level scope, with low to medium-high algorithmic complexity. Commit0 and SWE-Bench, TestGenEval (Jain et al., 2024a), RefactorBench (Gautam et al., 2024) are at project-level scope with low to medium algorithmic complexity. Today, we lack benchmarks 1) at project-level scope with high algorithmic complexity, 2) for many tasks other than code generation, and 3) requiring human intervention. Next, we note that current coding evaluations primarily focus on the code generation task. Most of the tasks discussed in Sec 3 are either not studied such as Code QA or only studied in self-contained scopes like EvalPerf (Liu et al., 2024), formal verification (Sun et al., 2024).

3.2 Intelligent Tool Usage

Software engineering has witnessed the development of various open and proprietary tooling support for programming, debugging, analysis, and code management throughout time. For example, linters and type-checkers provide assurances on static code correctness. Print statements and debuggers are used for dynamically analyzing and debugging programs. Beyond programming, such tools are richly integrated into the entire software development lifecycle, e.g. code navigation or search, reviewing code, CI testing. While many efforts combine LLMs and agents with tools, they do not achieve fully dynamic and intelligent software engineering tool usage. There are a few challenges towards this goal: first, the agent must identify which tools could potentially be useful for the task at hand. Second, the agent then needs to decide when to invoke the tool. Third, the agent then must figure out how to best make use the tool. Finally, the agent needs to incorporate the output provided by the tool in order to inform its next steps.

3.3 Human-AI Collaboration

While AI coding assistants are increasingly more powerful, the majority of AI coding assistants are still at a low to medium autonomy level, with constant human supervision required. We identify a few key challenges of today's AI coding systems that prevent these systems from working with humans effectively at higher levels of autonomy.

Lack of controllability: When programmers use AI coding assistants, they are often looking for a specific result and functionality. However, they currently do not have reliable ways of steering LLMs to generate a very specified chunk of code. Often, they rely on a guess-and-check like approach where the LLM is repeatedly sampled or given feedback until outputting a piece of code the programmer likes. As a result, humans still need to spend a lot of time reviewing and modifying the code to ensure that it performs their desired functionality (Weisz et al., 2024).

Identifying when human input is necessary: LLMs rarely defer to humans for clarification, while developers often ask questions to clarify the description of a task provided by their peers. One example is a product manager refining a requirements document: developers confused about the requirements or the scope ask questions and add comments to the document, which are typically resolved by the manager with the goal of iteratively disambiguating the requirements (Nahar et al., 2022). Based on its knowledge of existing software, AI should be able to incorporate implicit priors from a specification while keeping the user in the loop. For example, when designing an academic website for someone, there are implicit requirements, such as including a person's list of publications and contact information, but whether to include the person's GPA may be a clarification point.

3.4 Vague Specifications and User Misalignment

When using code LLMs, we typically prompt them with a natural language specification. This can include a natural language description of the desired code, input-output examples, relevant code snippets, and other functional requirements. However, there is a gap in the level of abstraction between English and code, leading to incomplete or ambiguous specifications. For longer programs, the number of ambiguous decision points also increases, and choices traditionally made by a human are now implicitly incorporated into the LLM's generated code. This often leads to user misalignment due to vague specifications.

Inherent trade-offs in software development: Designing large software systems always surfaces trade-offs between various desiderata such as readability, scalability, performance, maintainability, reliability, security, etc. These trade-offs are often context dependent. A long-term and rapidly moving project may be willing to trade off some performance to have simplicity and readability. Performance-critical applications may completely sacrifice readability to eke out every millisecond of speed. Finding the sweet spot among these trade-offs can often involve extensive prototyping and benchmarking to understand the performance characteristics of different approaches. User specifications rarely include details about these trade-offs, nor do models often take them into account.

Formal specifications: Autoformalization can mitigate underspecification by translating user intent into formal specifications. Ideally, a formal language could completely and unambiguously capture the desired software behavior. However, autoformalization may also suffer from the misalignment problem by misinterpreting user intent, resulting in a formal specification that does not accurately reflect the user's actual requirements. Therefore, users must understand and carefully verify the formalized specification generated by autoformalization tools.

3.5 Long-Horizon Code Planning

When working on large projects, engineers and tech leads often make complex decisions about how to design and structure the code to best support the various functionalities that will eventually be needed. To build a long-lasting software system, an engineer must know the potential paths that the system's evolution might take. This requires domain expertise and experience in how different code structures require different forms of extension.

Designing good abstractions: One instance of long-horizon code planning is choosing the right abstractions from the onset. An API designed with good abstractions will allow new features to be implemented seamlessly with minimal user overhead, while an API designed with poor abstractions may lead to excessive code duplication, refactoring, or even debugging. One example of this is *Library learning*: designing APIs and libraries with useful abstractions often leads to more code reuse and more intuitive interfaces. The challenge of library learning is to derive a library of useful abstractions from a corpus of programs by abstracting out common, reusable computations Ellis et al. (2021); Stengel-Eskin et al. (2024). While the traditional library learning literature has focused primarily on code reuse, a truly effective library must also prioritize ease of use and maintainability, as well as be robust and adaptable to future extensions. Another challenge is *data representation*, as the choice between data structures leads to a variety of trade-offs when it comes to performance aspects such as memory usage and processing speed. For example, database engineers need to decide between various data models, storage formats, and indexing methods to balance performance.

3.6 Large Scope and Long Contexts

The tasks in Sec. 2 become significantly more difficult at the repository-level scope, as engineering with large codebases is a multi-step task. For code generation, there are naturally many decision points where specifications can be vague. Because these decisions compound, it is tricky to generate the right code that the user wants. In code refactoring, modifications will touch multiple parts of the codebase, and it can be tricky to keep the repository consistent. In both code debugging and code navigation, as repositories get bigger, localization becomes more difficult. In code debugging, functions can be large and bugs can be nested deeply within stacks of function calls. In code navigation, because there are so many functions interacting in various ways, it can be difficult to know where each piece of functionality is implemented and how the code is put together.

The limits of long-context models: Software engineering often requires dealing with very large codebases–for example, Google has repositories with over a billion lines of code (Potvin & Levenberg, 2016). Dealing with large context lengths has long been an outstanding challenge for LLMs, and progress has been rapid with Gemini providing over 1M tokens of context. In coding, Magic has even trained models with 100M token context. While these models show good performance in toy benchmarks such as needle retrieval, we have not seen evidence of how this performance translates to large real-world code-bases.

The limits of retrieval-augmented generation (RAG): Retrieval-based algorithms have been the predominant way to deal with long-context coding issues. First, the retriever retrieves relevant functions. Then, the generator leverages the retrieval to improve generation. While RAG has proven effective in many NLP tasks such as question answering (Gao et al., 2023; Lewis et al., 2020), the code domain provides new challenges for these methods. In the code domain, for the *retrieval step*, it is often unclear what the correct item to retrieve is. In addition, code embeddings generally group code together via syntactic similarity (Ma et al., 2024; Utpala et al., 2023), which can make it hard to retrieve crucial snippets that are semantically relevant but syntactically unrelated. For the *generation step*, in NLP tasks it often is a straightforward application of the retrieved information. However, in code, writing a new function requires more than copying and pasting retrieved code snippets. Rather, the relevant snippets must first be analyzed, an algorithm using these code snippets must be developed, and be pieced together coherently in a precise manner.

3.7 Global Understanding of Codebases

After working with a codebase for a while, programmers have a global understanding and mental model of the codebase. This includes overall codebase structure, different algorithms, stylistic aspects, data representation, program invariants, package versions, test coverage, and the interplay between different functions. This holistic understanding is necessary for performing many tasks.

LLMs struggle at global understanding of codebases for several reasons. First, the way that code is pieced together can be relatively intricate, and understanding all these complex relationships can be difficult. Second, code can have units with high algorithmic complexity with custom algorithms that may never have appeared anywhere in the training data. Finally, because a disproportionately large number of LLM training tokens are spent on code generation rather than other coding tasks, models may lack a holistic awareness and world model of code. Generalizing across coding tasks may not be as simple as training: Gu et al. (2024) found that fine-tuning coding models on additional problems and solutions led to significant improvements on code generation but not code execution. Here, additional training on coding data did not transfer to improving code understanding and execution.

3.8 Low-Resource Languages and Custom Libraries

As we adapt code LLMs to individual codebases, generating correct code in out of distribution (OOD) scenarios becomes crucial. Much of software development in business contexts revolves around proprietary codebases, a distribution shift from the open-source code that dominates LLM training data (Ahmed et al., 2024). Examples include domain-specific languages (DSLs), custom internal libraries, low-resource APIs, and company-specific coding styles/conventions.

Syntactic failures and poor semantic understanding: Models often hallucinate constructs from higher resource languages when working in low-resource languages. Blinn et al. (2024) found that when writing Hazel, LLMs often borrowed syntax and library functions from OCaml and Elm, higher-resource languages. When writing Triton, models often use syntactically incorrect constructs such as array indexing. In addition, models have less exposure to the various language constructs. Therefore, they have a weaker semantic understanding of the language. Many studies reveal that code LLMs perform poorly when asked to write code in low-resource languages. Due to the lack of training data in these OOD domains, models may struggle to write common primitives or piece together functionality coherently. On HumanEval, Qwen 2.5 Coder Instruct (32B) (Hui et al., 2024) has an accuracy of 83% in Python but only 27% in D.[1]

[1]As reported by the BigCode Models Leaderboard on the MultiPL-E benchmark (Cassano et al., 2023)

Library usage failures: In OOD scenarios, LLMs lack awareness of the libraries and functions available for use. In new codebases using custom libraries, many functions appear only a few times, providing limited training data for AI models to learn their usage. This scarcity can lead to overfitting, where models fail to recognize an effective use-case of these functions. Models also frequently hallucinate non-existent functions based on patterns that it infers.

3.9 Library and API Version Updates

Continual learning, the idea of training an AI system to take in new information continually, has been a long-standing challenge in AI and NLP (Wu et al., 2024; Wang et al., 2024b). In software engineering, codebases are continuously changing as new features are supported and awkward design patterns are reworked. While backwards compatibility is often prioritized in software design, it inevitably becomes broken as codebases evolve further. Therefore, programming libraries have version releases, each release supporting and deprecating features in the last version.

Continous adaptation: Language models are not as dynamic as codebases, and new features and APIs released yesterday will not be in the training data of today's code LMs. Code written with new APIs and new versions of existing libraries are low-resource. Even when there are new and simplified ways to write features, LLMs may rely on older, more cumbersome approaches because those are the ones seen more frequently in the model's training data.

Version-specific constructs: For fast-changing libraries that are not backward compatible, it can be difficult for LLMs to implicitly keep track of which constructs are associated with each version. This makes consistently using constructs from the right version difficult. Therefore, LLMs may write code that mixes and matches API constructs from different versions of the same library.

3.10 High Algorithmic Complexity: OOD Domains

Some programming tasks are challenging for even the best human programmers, requiring approaches with a very high algorithmic complexity. Examples of tasks that fall into this category include superoptimizing programs, discovering attacks for purportedly secure code, writing performant compilers, optimizing GPU kernels (Ouyang et al., 2025), and writing very error-prone and very technical code. Because they are hard for humans, these tasks are very rarely in the training data of today's language models. They have unique, domain-specific, challenges that making generalizing from existing data difficult. For these problems, language models rely heavily on feedback-driven search algorithms (Mankowitz et al., 2023), and it can be difficult to navigate the search space effectively. In addition, many of these tasks lack feedback mechanisms, which is crucial for AI to pick up learning signals. When designing a complex algorithm or data structure, it is often hard to know if you are on the right track until you get to the correct result. When writing code for a large multithreaded operation, it may be hard to know if the algorithm has concurrency issues until all the parts are fully fleshed out. Without feedback, incremental improvement is nearly impossible.

4 Paths Forward

4.1 Training: New Objectives and Synthetic Data

Reinforcement Learning (RL): RL has delivered considerable improvements in isolated but challenging algorithmic problems (DeepSeek-AI et al., 2025). A promising direction is to scale such reinforcement learning approaches to more mature real-world tasks by collecting execution-assisted gym-like programming environments (Jain et al., 2024c; Pan et al., 2024). Particularly, we believe it will be important to incorporate tasks with high construct validity and consider ways of mitigating and model task ambiguity (Shao et al., 2024).

Synthetic Data: In code, synthetic data is a promising method for gathering more data. In contrast to text, code contains strong, verifiable feedback such as test cases, program execution engines, and other symbolic tools. For example, to generate code with interesting program invariants, we can sample a large batch of programs and filter using an invariant detector to keep the ones with interesting invariants. Successful synthetic data can also be used to seed the generation of more complex synthetic data. Code is also compositional, allowing the possibility of combining individual building blocks to even generate synthetic data with repository-level scope. In DSLs, it is often possible to generate programs with desired properties via sampling. This technique has been successfully applied to difficult reasoning tasks such as ARC-AGI (Li et al., 2024) and math olympiad problems (Trinh et al., 2024; Google, 2024).

4.2 Developing a World Model for Code

In Sec. 3.7, we discussed how LLMs do not have a global understanding of codebases. We believe it is important for a coding language model to have a good "world model" for code in the same way that students completing a programming course will gain a holistic understanding of coding.

Augmenting Data with Program Information: While code is currently often treated as pure tokens, we can actually extract a lot of information about a program by using the wide variety of programming language tools available. These tools can be used to augment existing programs in the training set with available information describing properties the code. We believe that being able to see code alongside its properties will enhance a model's overall world model of code. These properties can include static analysis (abstract syntax trees, data flow analyses), dynamic analysis (program states, call stacks), program instrumentation (memory consumption, code coverage), and formal verification (concurrency analysis, program invariants).

Training with More Tasks: Another way to steer code models to have a more general understanding of code is to widen the distribution of tasks models are exposed to during training, such as including the tasks in Sec. 2. Training should include more tasks that improve a model's semantic understanding, such as describing errors in subtle bugs or predicting the execution states of a program. This training could be done in a curriculum-like manner to gradually improve world model capabilities.

4.3 Integration with SWE development frameworks

Integrating AI with SWE development frameworks is critical for practical applications and impact on developer workflows. While software development is inherently integrated with tools, workflows, and processes, scaffolding and meta-code (Sec. 3.8) is often absent from source code and scarce in AI training data. Ensuring that AI deeply understands software deployment beyond code editing is crucial, as writing code is only a small part of the development cycle. In continuous integration and continuous deployment (CI/CD), automated pipelines are the backbone for building, testing, and deploying code changes. CI/CD accelerates feedback cycles and minimizes integration issues. AI offers several integration points within CI/CD. AI-powered code review tools can be incorporated into CI pipelines to automatically identify and flag style violations, potential security vulnerabilities, and code smells before human reviewers are involved. Furthermore, AI can provide intelligent deployment risk assessments. By analyzing code changes, test outcomes, and historical deployment data, AI can predict the likelihood of deployment issues, informing decisions about whether to proceed with automated deployment or mandate manual verification steps. Finally, AI can automate the generation of release notes by summarizing commit messages, issue tracker data, and relevant code modifications within the CI/CD process.

4.4 Agents and Tool Integration

Learned Tool Usage: To improve the tool usage of AI agents, we should teach agents to understand the intricacies of tools so that they can autonomously invoke them as needed when writing code. Drawing inspiration from learning how to play games, we imagine a RL approach where the model can learn the strengths and weaknesses of each tool by trying it out at various points in code writing.

Neurosymbolic Approaches: Today, the majority of research in AI for code do not take into account the deep symbolic properties of code. We believe that LMs and ymbolic tools should be more deeply integrated with each other. This could include using information about program structure in training (e.g. ASTs) or using the grammar of the programming language to do constrained decoding.

4.5 Context Adaptation and Continual Learning

Low-resource languages (Sec. 3.8), custom APIs, library version updates (Sec. 3.9), and large codebases (Sec. 3.6) all surface the fact that code LMs struggle to adapt to specialized contexts.

Test-time training: One recent paradigm is test-time training, the idea of adapting to a specific problem instance by training on a narrow set of in-distribution examples. This allows a model to adapt to a specific codebase, new domain, or unseen API. The model could also use synthetic data to create in-distribution examples and annotate them with symbolic (e.g. compiler) feedback to gain a more global understanding of the current environment.

Controllable forgetting: If the correct repository version can be identified, the model can also be trained on the specific version of each API, reducing version-related hallucinations. This can also be combined with algorithms that can induce controllable forgetting (Wu et al., 2024).

Retrieval-augmentation: In these domains, the challenge is purely syntactic rather than algorithmic, making RAG-based methods ideal. When using APIs with multiple versions, providing retrievals in the current version can steer the model. These retrievals can be in the form of documentation, API function definitions, or example use cases.

5 Conclusion

In this position paper, we have identified key tasks at the heart of AI for software engineering, critical cross-cutting challenges that permeate throughout many tasks, and promising research directions for alleviating these challenges and advancing AI towards being a more capable software engineer. We hope this work provides a valuable framework for understanding the current landscape and encourages future research in these directions. By building on these insights, we can work toward developing AI-driven solutions that better support software engineers in real-world settings.

References

Toufique Ahmed, Christian Bird, Premkumar Devanbu, and Saikat Chakraborty. Studying llm performance on closed-and open-source data. *arXiv preprint arXiv:2402.15100*, 2024.

Andrew Blinn, Xiang Li, June Hyung Kim, and Cyrus Omar. Statically contextualizing large language models with typed holes. *Proceedings of the ACM on Programming Languages*, 8 (OOPSLA2):468–498, 2024.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. Multipl-e: a scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691, 2023.

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

Wayne Chi, Valerie Chen, Wei-Lin Chiang, Anastasios N. Angelopoulos, Naman Jain, Tianjun Zhang, Ion Stoica, Chris Donahue, and Ameet Talwalkar. Copilot arena, 2024.

Leonardo De Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*, pp. 378–388. Springer, 2015.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL `https://arxiv.org/abs/2501.12948`.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B Tenenbaum. Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd acm sigplan international conference on programming language design and implementation*, pp. 835–850, 2021.

Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pp. 31–53. IEEE, 2023.

Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2023.

Dhruv Gautam, Spandan Garg, Jinu Jang, Neel Sundaresan, and Roshanak Zilouchian Moghaddam. Refactorbench: Evaluating stateful reasoning in language agents through code. In *NeurIPS 2024 Workshop on Open-World Agents*, 2024.

Google. Ai achieves silver-medal standard solving international mathematical olympiad problems. https://deepmind.google/discover/blog/ai-solves-imo-problems-at-silver-medal-level/, 2024.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I Wang. CRUXEval: A Benchmark for Code Reasoning, Understanding and Execution. *arXiv preprint arXiv:2401.03065*, 2024.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. *NeurIPS*, 2021.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–79, 2024.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Kush Jain, Gabriel Synnaeve, and Baptiste Rozière. Testgeneval: A real world unit test generation and test completion benchmark. *arXiv preprint arXiv:2410.00752*, 2024a.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code, 2024b. URL `https://arxiv.org/abs/2403.07974`.

Naman Jain, Manish Shetty, Tianjun Zhang, King Han, Koushik Sen, and Ion Stoica. R2e: Turning any github repository into a programming agent environment. In *Forty-first International Conference on Machine Learning*, 2024c.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL `https://openreview.net/forum?id=VTF8yNQM66`.

Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv preprint arXiv:2408.02479*, 2024.

Sathvik Joel, Jie JW Wu, and Fatemeh H Fard. A survey on llm-based code generation for low-resource and domain-specific programming languages. *arXiv preprint arXiv:2410.03981*, 2024.

Leslie Lamport. Introduction to tla. 1994.

Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, et al. Verus: A practical foundation for systems verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pp. 438–454, 2024.

K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*, pp. 348–370. Springer, 2010.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33: 9459–9474, 2020.

Wen-Ding Li, Keya Hu, Carter Larsen, Yuqing Wu, Simon Alford, Caleb Woo, Spencer M Dunn, Hao Tang, Michelangelo Naim, Dat Nguyen, et al. Combining induction and transduction for abstract reasoning. *arXiv preprint arXiv:2411.02272*, 2024.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Jenny T Liang, Chenyang Yang, and Brad A Myers. A large-scale survey on the usability of ai programming assistants: Successes and challenges. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pp. 1–13, 2024.

Jiawei Liu, Songrun Xie, Junhao Wang, Yuxiang Wei, Yifeng Ding, and Lingming Zhang. Evaluating language models for efficient code generation. In *First Conference on Language Modeling*, 2024. URL `https://openreview.net/forum?id=IBCBMeAhmC`.

Wei Ma, Shangqing Liu, Mengjie Zhao, Xiaofei Xie, Wenhang Wang, Qiang Hu, Jie Zhang, and Yang Liu. Unveiling code pre-trained models: Investigating syntax and semantics capacities. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–29, 2024.

Daniel J Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, et al. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618(7964):257–263, 2023.

Meredith Ringel Morris, Jascha Sohl-Dickstein, Noah Fiedel, Tris Warkentin, Allan Dafoe, Aleksandra Faust, Clement Farabet, and Shane Legg. Levels of agi: Operationalizing progress on the path to agi. *arXiv preprint arXiv:2311.02462*, 2023.

Balázs Mosolygó, Norbert Vándor, Gábor Antal, and Péter Hegedűs. On the rise and fall of simple stupid bugs: a life-cycle analysis of sstubs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pp. 495–499. IEEE, 2021.

Nadia Nahar, Shurui Zhou, Grace Lewis, and Christian Kästner. Collaboration challenges in building ml-enabled systems: Communication, documentation, engineering, and process. In *Proceedings of the 44th international conference on software engineering*, pp. 413–425, 2022.

Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517*, 2025.

Ipek Ozkaya. Application of large language models to software engineering tasks: Opportunities, risks, and implications. *IEEE Software*, 40(3):4–8, 2023.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym, 2024. URL `https://arxiv.org/abs/2412.21139`.

David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

Rachel Potvin and Josh Levenberg. Why google stores billions of lines of code in a single repository. *Communications of the ACM*, 59(7):78–87, 2016.

Agnia Sergeyuk, Yaroslav Golubev, Timofey Bryksin, and Iftekhar Ahmed. Using ai-based coding assistants in practice: State of affairs, perceptions, and ways forward. *Information and Software Technology*, 178:107610, 2025.

Yijia Shao, Vinay Samuel, Yucheng Jiang, John Yang, and Diyi Yang. Collaborative gym: A framework for enabling and evaluating human-agent collaboration, 2024. URL `https://arxiv.org/abs/2412.15701`.

Elias Stengel-Eskin, Archiki Prasad, and Mohit Bansal. Regal: refactoring programs to discover generalizable abstractions. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. Clover: Closed-loop verifiable code generation. In *International Symposium on AI Verification*, pp. 134–155. Springer, 2024.

The Coq Development Team. The Coq reference manual – release 8.19.0. `https://coq.inria.fr/doc/V8.19.0/refman`, 2024.

Trieu H Trinh, Yuhuai Wu, Quoc V Le, He He, and Thang Luong. Solving olympiad geometry without human demonstrations. *Nature*, 625(7995):476–482, 2024.

Saiteja Utpala, Alex Gu, and Pin Yu Chen. Language agnostic code embeddings. *arXiv preprint arXiv:2310.16803*, 2023.

Yao Wan, Zhangqian Bi, Yang He, Jianguo Zhang, Hongyu Zhang, Yulei Sui, Guandong Xu, Hai Jin, and Philip Yu. Deep learning for code intelligence: Survey, benchmark and toolkit. *ACM Computing Surveys*, 2024.

Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 2024a.

Liyuan Wang, Xingxing Zhang, Hang Su, and Jun Zhu. A comprehensive survey of continual learning: theory, method and application. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2024b.

Justin D Weisz, Shraddha Kumar, Michael Muller, Karen-Ellen Browne, Arielle Goldberg, Ellice Heintze, and Shagun Bajpai. Examining the use and impact of an ai code assistant on developer productivity and experience in the enterprise. *arXiv preprint arXiv:2412.06603*, 2024.

Man-Fai Wong, Shangxin Guo, Ching-Nam Hang, Siu-Wai Ho, and Chee-Wei Tan. Natural language generation and understanding of big code for ai-assisted programming: A review. *Entropy*, 25(6):888, 2023.

Tongtong Wu, Linhao Luo, Yuan-Fang Li, Shirui Pan, Thuy-Trang Vu, and Gholamreza Haffari. Continual learning for large language models: A survey. *arXiv preprint arXiv:2402.01364*, 2024.

Kaiyu Yang, Gabriel Poesia, Jingxuan He, Wenda Li, Kristin Lauter, Swarat Chaudhuri, and Dawn Song. Formal mathematical reasoning: A new frontier in ai. *arXiv preprint arXiv:2412.16075*, 2024.

Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. A survey of learning-based automated program repair. *ACM Transactions on Software Engineering and Methodology*, 33(2):1–69, 2023.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M Rush. Commit0: Library generation from scratch. *arXiv preprint arXiv:2412.01769*, 2024.

Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. A survey of large language models for code: Evolution, benchmarking, and future trends. *arXiv preprint arXiv:2311.10372*, 2023.