CAREER: Formal Specifications as a Foundation for Aligned and Automated Software Engineering

PI: Juan Zhai, Assistant Professor, University of Massachusetts Amherst

Overview

The rise of large language model based agents is transforming software engineering, marking a new era where autonomous systems can participate in development tasks like code generation, debugging, and testing. This shift promises unprecedented productivity gains and wider access to software creation. However, as these agents become more powerful, the fundamental challenge in software engineering remains unchanged: ensuring that software systems behave as intended.

At the heart of this challenge is the persistent absence of formal specifications which are precise, unambiguous definitions of expected behavior. Formal specifications are essential not only for traditional activities like debugging, testing, and verification, but also for enabling large language model agents to generate correct and reliable code. They serve as the semantic anchor that allows tools and agents to reason meaningfully about behavior. Despite their importance, formal specifications remain rare in practice due to the complexity of authoring and maintaining them, especially in fast-evolving, real-world systems.

This proposal envisions a future where formal specifications are not just documentation, but first-class artifacts actively powering both traditional and AI-assisted software development. The PI will develop new techniques and infrastructure to: (1) construct the first repository-level dataset linking natural language descriptions, formal specifications, and code snippets, along with an automated evaluation pipeline for rigorous and reproducible assessment of specification accuracy; (2) automatically infer formal specifications from human-AI collaboration artifacts such as natural language documentation and code structure, and maintain these specifications as code and documentation evolve; and (3) harness specifications as dynamic tools for guiding and validating development, enabling tools and agents to produce behaviorally correct code and identify potential issues at the earliest stages. Together, these efforts aim to make formal specifications more accessible, reliable, and actionable in real-world software development.

Intellectual Merit

1. This project will produce the first repository-level dataset connecting natural language descriptions, formal specifications, and code, supporting reproducible research, model development, benchmarking, and broader advancements in specification-driven software engineering.
2. This project will develop techniques to infer specifications from natural language, propagate them via code dependencies, and maintain them as code and comments evolve, ensuring specifications stay accurate, up-to-date, and aligned with intended behavior.
3. This project will advance specification-driven tools for alignment and assurance, improving both code generation and software testing. It will enable agents to produce semantically accurate code and detect behavioral issues early in the development process.

Broader Impacts

1. The proposed techniques will help developers and AI agents build reliable software at scale, enabling safer autonomous systems, reducing failure costs, and improving accessibility. These advances are particularly impactful in safety-critical domains like healthcare and finance. Adoption will be accelerated through industry collaborations and tracked via tool integration, partner feedback, and open-source engagement metrics.
2. The PI will organize workshops and tutorials to disseminate datasets, developed techniques and benchmarks to the community. The PI has a good track record of open-sourcing research outcomes.
3. For a UMass graduate-level course, the PI will redesign the graduate software engineering curriculum for the agent era, focusing on specification-driven development and human-AI collaboration, delivered through active, collaborative learning, real-world examples, and timely feedback.
4. The PI will engage graduate, undergraduate, and K–12 students in specification research through independent studies, the UMass ERSP program, and STEM workshops. The PI has a strong record of publishing with undergraduates, many of whom have entered top Ph.D. programs, which the project will further support.

Introduction

The landscape of software engineering (SE) is undergoing a profound transformation. In the emerging era of agentic coding, developers increasingly communicate their intent through specifications which are then interpreted by large language models (LLMs) or autonomous coding agents to synthesize code and automate development tasks [1–3]. This paradigm shift holds the promise of dramatically accelerating productivity and democratizing access to software creation. However, it also introduces urgent new challenges, not only for ensuring that AI-generated code is correct and aligned with human intent, but also for enabling AI agents to reason about software behavior and perform development tasks effectively.

This transition is already reshaping how experts envision the future of SE. OpenAI has described the future as "simply specification" where specifications, rather than code, serve as the primary interface between developers and AI systems, enabling alignment, instruction, and reasoning [4, 5]. Anthropic CEO Dario Amodei projects that AI will generate 90% of code within six months, with full automation on the horizon [6]. Yet this acceleration brings risk: a recent study by Uplevel Data Labs found that developers using AI coding assistants introduced 41% more bugs than those without [7]. These developments underscore the urgency of rethinking core SE foundations centered not around code, but around precise, actionable specifications that can support both human oversight and automated reasoning.

However, several obstacles stand in the way of making specification-driven agentic development a reality.

First, AI agents often misinterpret developer intent, producing code that appears correct but behaves incorrectly, such as omitting access checks or hallucinating logic [8–12]. These misalignments often stem from the absence of clear, structured behavioral guidance. Today's agents lack deep semantic understanding and cannot guarantee that generated code faithfully implements the intended behavior. To address this, we need high-quality datasets that connect code, documentation, and formal specifications at the repository level, capturing the scale and complexity of real-world software systems. These datasets will be extremely useful in training models to improve their alignment in specification understanding. For example, for cases that specifications contain contradictions or requirements are underspecified, the agent will proactively interact with developers to ensure precise understanding.

Second, all real-world code, whether human-written or AI-generated, must be validated. Even if agents fully understand human intent, we cannot trust their outputs without grounding them in formal specifications that intensive existing tools can reason about. Specifications provide precise, logical descriptions of expected behavior and remain essential for testing [13–17], debugging [18, 19], and formal verification [20–22]. However, most codebases lack such specifications [14, 15, 23–32]. This calls for new methods to continuously extract and maintain formal specifications from human-AI collaboration artifacts including comments, commit logs, bug reports, and code, transforming scattered traces of intent into durable, verifiable contracts.

Third, even when specifications are available, current tools and AI agents struggle to use them effectively. Traditional workflows rely on post hoc validation [33, 34], which falls short in agentic development where agents rapidly generate and modify large volumes of code. Without real-time reasoning, errors propagate and become costly to detect. High computational costs and limited context windows further constrain agents' ability to enforce global constraints. To be effective, specifications must shift from passive checks to active, in-the-loop guidance that enables early error detection and scalable, reliable development.

This proposal directly addresses these challenges by building the infrastructure needed to make specification-driven development practical and effective in the agentic era. We will construct the first repository-scale dataset that connects natural language (NL), code, and formal specifications, enabling both rigorous evaluation and the training of intent-aware models. We will develop methods to automatically infer and maintain specifications from diverse development artifacts, ensuring that agents remain grounded as software evolves. Finally, we will turn specifications into active engines for alignment and assurance: training agents to generate semantically accurate code and detect behavioral issues early in development. Together, these efforts will elevate specifications from passive documentation to the foundation of safe, reliable, and intelligent AI-assisted SE.

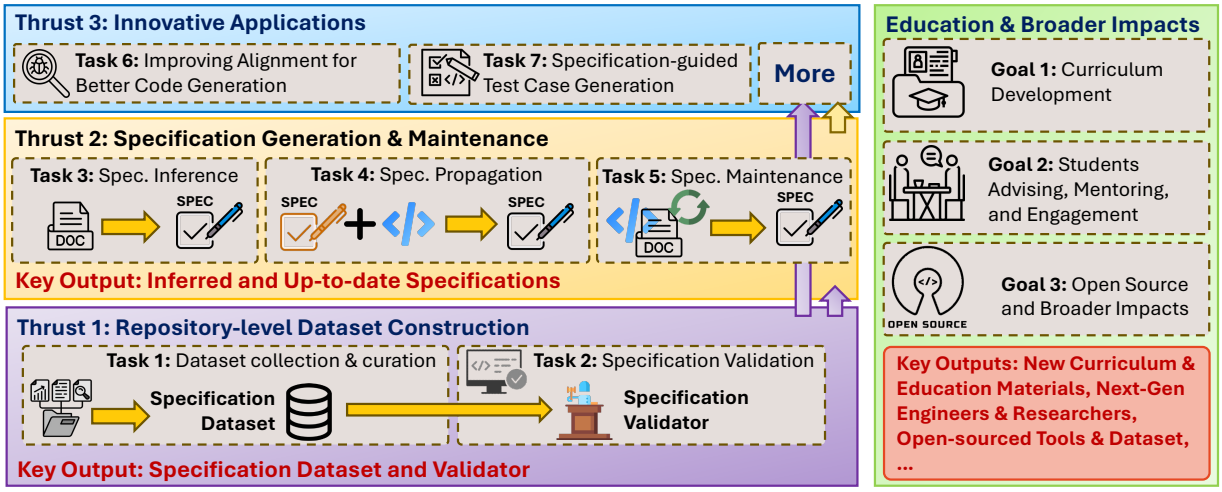


Figure 1: Overview of the Proposed Work

Intellectual Merit

This project advances knowledge at the intersection of SE, formal methods, and AI by making formal specifications actionable in agentic workflows. It is organized into three thrusts (overview in Figure 1).

Thrust 1 fills a critical gap in specification-driven agentic development by creating the first repository-scale datasets linking code, documentation, and formal specifications. This dataset will enable reproducible benchmarking of specification inference models and fine-tuning of behaviorally aligned code generation models. **Task 1** constructs a comprehensive and diverse dataset by combining existing resources, leveraging LLMs for data distillation, and incorporating human annotations through crowdsourcing. **Task 2** develops a validation pipeline consisting of symbolic validation, automated testing, and expert review to validate specifications.

Thrust 2 develops techniques and tools to generate and maintain formal specifications that serve as actionable, up-to-date guidance for AI agents and developers throughout the software lifecycle. **Task 3** fine-tunes a small, open-source language model to generate formal specifications from NL descriptions, using retrieval-augmented generation (RAG) to incorporate relevant code context and improve contextual accuracy. Generating specification for methods without NL descriptions is beyond the capabilities of existing methods. **Task 4** addresses this by analyzing static dependency relationships and leveraging program analysis to propagate specifications based on code semantics. Maintaining specifications is under explored. **Task 5** tracks and maintains the consistency between code, documentation and specifications, ensuring specifications can be accurately and timely updated upon code or documentation change.

Thrust 3 elevates specifications from passive artifacts to active drivers of software quality. It develops tools that fine-tune models for behaviorally aligned code generation and enable both AI agents and SE systems to reason with specifications in real time, accelerating development and enhancing robustness. **Task 6** develops techniques to improve code generation by fine-tuning models on specification-rich datasets, enabling them to better align with intended behavior and produce semantically correct, trustworthy code. **Task 7** develops new techniques to improve test case generation by supporting complex specification constructs through a divide-and-conquer strategy, while mitigating path explosion to ensure scalability and efficiency.

Broader Impacts

Societal. The proposed research aims to make formal specifications more accessible, maintainable, and actionable. By enabling agents to reason about behavior and validating their outputs through specifications, this project improves the reliability of AI-generated code while reducing the cost of testing, verification, and maintenance. Broader adoption of specification-driven practices across industry and research can elevate software quality and support safer, more rigorous development at scale.

Technical. This work advances the state of the art in specification synthesis, maintenance, and code/test gen-

eration, with strong potential for industrial impact. The PI will engage the community through publications, workshops, tutorials, and internships to support adoption, feedback, and evaluation on large-scale projects.
Educational. The PI will continue the integration of research and education. The PI designed and taught the undergraduate-level SE course at Rutgers and is teaching the graduate-level SE course at UMass Amherst. Based on the proposed research, the PI will integrate advances in AI-assisted SE and formal specifications into upper-level courses through hands-on, collaborative activities. Students will build prototypes for tasks such as agent-guided code generation and bug detection, and apply formal methods to analyze and verify real-world systems. A core goal is to prepare students for the agent era of SE by equipping them with practical skills in specification-driven development, an increasingly vital yet underemphasized area in current curricula.

The PI has a strong record of mentoring graduate and undergraduate students through honors programs, independent studies, and research centers [23, 24, 35–38], with several alumni now pursuing Ph.D.s at top institutions (e.g., Georgia Tech). The PI currently advises three senior students, two of whom are working on projects directly related to this proposal.

Project Scope and PI Qualification

PI Zhai's long-term career goal is to develop innovative SE techniques to build high-quality software, focusing on addressing one fundamental gap in software development: *the real implementation cannot be easily verified against its intended behavior*. Her prior work combines program analysis and NLP to jointly analyze code and comments, advancing both techniques and enabling tools that improve software quality. This proposal presents a systematic approach to integrating program analysis with recent AI advances to generate, maintain, and apply formal specifications for improved code generation and software testing. PI Zhai's expertise in SE and AI uniquely qualifies her to lead this research. Her highly relevant top-tier publications span SE, security, and machine learning venues like ICSE, FSE, USENIX Security, NeurIPS, and ACL. The topics include specification synthesis [23, 24, 37, 39], software testing [35, 36, 40–49], and benchmark curation [50, 51]. Her research has uncovered over 1,500 bugs in real-world software systems [37, 41, 43–45, 47–49, 52–54], with more than 290 confirmed or fixed by developers. Her work has significantly impacted the community, with over 3,597 citations (h-index: 24) as of July 2025, and has been recognized with the USENIX Security Distinguished Paper Award and the Research Council Award. Additionally, her group actively engages with the community, publicly releasing code or datasets for over 30 projects. As the co-director of the UMass Amherst Laboratory for Advanced Software Engineering Research (LASER) [55], PI Zhai is committed to advancing education, technique adoption, and broadening participation in computing, with strong support and mentoring from the college and community.

Thrust 1: Repository-level Dataset Construction

This thrust builds a comprehensive, repository-level dataset to support specification generation and maintenance in practical SE settings. By grounding specifications in real-world code and documentation, it enables rigorous evaluation and facilitates the development of tools that integrate seamlessly into modern, agent-assisted development workflows. Each data instance is anchored to a method with a specific NL description that focuses on a particular behavioral aspect of this method, such as its return value or exception handling. Aligned with this description, the instance includes a corresponding code snippet that implements the relevant behavior, and a formal specification that captures it precisely. This fine-grained design supports both generation and maintenance tasks, particularly in evolving software.

Constructing the proposed high-quality dataset entails a few challenges. First, most open-source repositories lack formal specifications. Existing datasets primarily focus on code generation [56–61] and summarization [62–64], rather than specifications. When specifications do exist, they are typically limited to stand-alone methods which are rare in practice. Most methods are part of larger codebases with dependencies, making it essential to understand their behavior in context. Second, aligning formal specifications with code and NL documentation at fine-grained granularity is challenging. Existing datasets typically aligns an NL

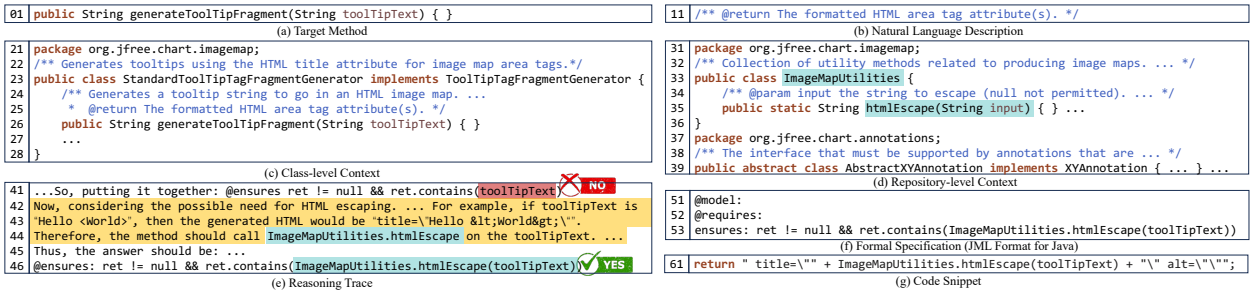


Figure 2: A Representative Dataset Instance. Each Subfigure Shows an Element; Line Numbers are for Reference Only

description with the entire method implementation. In contrast, fine-grained alignment where the description targets a specific aspect of behavior such as exception handling requires precise segmentation and careful annotation, which are difficult to automate and labor-intensive to curate. Third, validating the correctness of specifications requires accurately interpreting both the intended behavior and the specification logic, which demands domain expertise and makes the process time-consuming and difficult to scale.

We will develop an approach to construct a fine-grained specification-centric repository-level dataset that directly addresses the challenges outlined above. First, we will leverage program analysis, embedding-based retrieval, and LLMs to automatically generate initial data samples with fine-grained correspondence between the NL description, specification, and code snippet (Task 1). This will be followed by automated validation using symbolic reasoning, automated testing, and subsequently refined through expert review (Task 2).

Task 1: Dataset Collection & Curation. This task aims to build a fine-grained, repository-level dataset centered on program dependency. Our dataset instance is a 7-tuple $x = (m, d, s, c_{\text{class}}, c_{\text{repo}}, r, c_{\text{impl}})$, where m is the method signature, d is the NL description, s is the formal specification in a widely-adopted format, such as Java Modeling Langauge (JML [33]) for Java and icontract [65] for Python. c_{class} and c_{repo} denote the class-level and repository-level structural contexts respectively, r is the reasoning trace linking NL description d to specification s, and c_{impl} is the code snippet corresponding to the described behavior.

Our specifications capture intra- and inter-class dependencies, include references to standard libraries, and preserve pre-state information. For example, the specification `Arrays.deepEquals(\old(this.items()), \result.items())` is accompanied by `@model import java.util.Arrays` to reflect its reliance on the standard library method `deepEquals` from `Arrays`. The use of `\old(this.items())` preserves the value of `this.items()` prior to method execution. The context c_{class} and c_{repo} reflect structural and semantic context within the class and repository that directly inform and constrain the specification. The contextual information is restricted to class and method skeletons, along with associated comments, while excluding code implementations to avoid relying on implementation semantics. The reasoning trace r offers step-by-step logical guidance to help the model infer correct specifications. Figure 2 presents a representative data instance, with each subfigure corresponding to one element of the tuple.

This dataset is designed for flexibility and broad applicability. It supports both specification-centric research and general SE tasks. A subset such as $x = (m, d, s, c_{\text{class}}, c_{\text{repo}}, r)$ can train or evaluate models for specification generation, while $x = (m, d, s)$ suffices for evaluating specification inference. Beyond specification work, the dataset also enables applications such as code generation using $x = (m, d, s, c_{\text{class}}, c_{\text{repo}}, c_{\text{impl}})$, supporting workflows that require grounding code synthesis in developer intent (Task 6).

To construct this dataset, we propose to develop a curation tool consisting of four components as shown in Figure 3. The rest of this section will describe each step in detail.

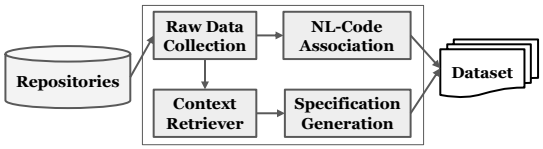


Figure 3: The Initial Dataset Construction Approach

① *Raw Data Collection.* The first step is selecting well-documented projects that offer comprehensive test coverage. To ensure diversity, we will cover a broad range of sources, including open-source repositories like GitHub projects [66] and established code datasets [56–61]. We will also incorporate existing open-source specifica-

tions [67] and prior research datasets [14, 15, 23, 25]. The proposed dataset will cover multiple languages, including Java, Python, C++, JavaScript and Solidity, to ensure broad applicability and coverage.

② *Retrieving Augmented Context.* Augmented context consists of class-level and repository-level information. The class-level context includes the skeleton of the class containing the target method, with all method signatures and associated comments. Our tool extracts this context using an abstract syntax tree (AST) parser, such as Java Development Tools (JDT) [68], that isolates the structural elements of the target class while excluding implementation details. Figure 2 (c) shows the class-level context of the method in Figure 2 (a).

To construct repository-level context, our curation tool extracts AST-based skeletons from all classes in the repository and encodes them using a pre-trained representation model, such as CodeBERT [69]. The resulting embeddings are indexed via a high-dimensional similarity structure, e.g., FAISS [70], to support efficient retrieval. For each target method, we embed a structured query, comprising its signature and its enclosing class's signature, and retrieve semantically similar classes and methods to enrich the context. This augmented context improves specification relevance and accuracy. Figure 2 (d) shows the repository-level context for the method in Figure 2 (a).

③ *Generating Candidate Specifications with Reasoning Traces.* Our tool combines the method signature, NL description, and retrieved class- and repository-level context into a unified prompt for an LLM to generate formal specifications with reasoning traces. These traces guide the logical derivation process, enabling self-correction and refinement, and improving the precision and interpretability of specification synthesis.

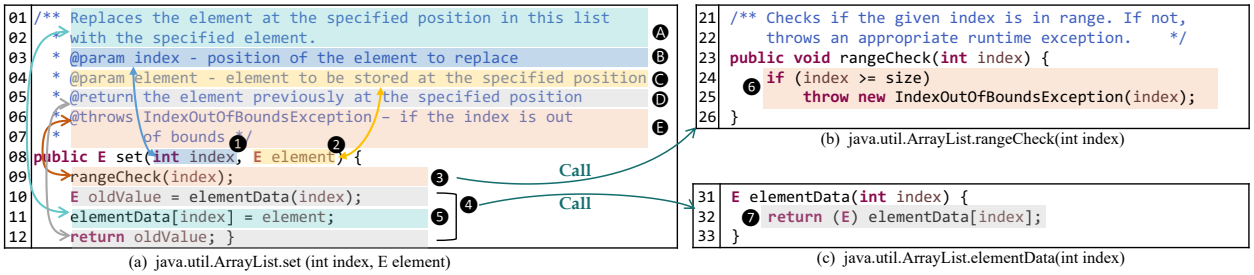
Figure 2 (e) shows a reasoning trace for generating a specification from the NL description in Figure 2 (b) for the method in Figure 2 (a). Initially, the model produced an incorrect specification `ret != null && ret.contains(toolTipText)` (line 41), same as the output of the state-of-the-art [32]. This checks if the returned string is non-null and contains the input `toolTipText`. The reasoning trace prompts the model to question whether special HTML characters require escaping (line 42). Recognizing that input like "Hello <World>" should yield "Hello <World>" (line 43), the model refines its output to correctly verify the presence of the escaped tooltip rather than the raw input, and produce the correct specification (line 46).

④ *Associating NL Descriptions with Code Snippets.* As NL descriptions of code, comments are inherently tied to code. We seek fine-grained associations where each description corresponds to a specific code snippet rather than an entire method or module. Such fine-grained alignment is essential for tracking and updating specifications as software evolves, especially since most changes typically affect localized behavior. However, establishing such associations is challenging due to the lack of explicit alignment. A method's comment often spans multiple concerns, such as return values, parameters, or exception handling. Moreover, a single description may describe behavior distributed across multiple statements or control paths.

We will leverage program slicing [71, 72] and keywords matching to automatically associate a NL description with a code snippet. After the previous three steps, we have established a fine-grained association between NL descriptions and specifications. This step extends the association to include the corresponding code snippets, resulting in a complete alignment among NL descriptions, specifications, and code.

Modern documentation tools, such as JavaDoc [73], use standard tags like `@return`, `@param`, and `@throws` to describe return values, parameters, and exceptions, with remaining text summarizing functionality. Similar formats exist in JSDoc [74] for JavaScript and Docstring [75] for Python. Here we use Java as an example. The basic idea is to perform program slicing on the code from given keywords, and then associate the code snippets with the corresponding comments. `@param` comments are directly related to the parameter names and `@return` is associated with the `return`-statement slice. For `@throws` comments, we first locate the exceptions of the same type in the code (which can be in the callees) and associate the comments with slices obtained by backward slicing from the `throw` statements. Since an exception can be thrown in different parts of the code under varying conditions, multiple `throw` slices can be associated to the same `@throws`. The rest of the code is considered to be the core functionality.

Figure 4 shows how we associate code with NL comments. Ⓐ – Ⓔ are comments and ❶ – ❼ are code



(a) java.util.ArrayList.set (int index, E element)

(b) java.util.ArrayList.rangeCheck(int index)

(c) java.util.ArrayList.elementData(int index)

Figure 4: Associating a Code Snippet with a NL Comment Based on Program Slicing and Javadoc Annotation

slices. ❶ and ❷ are straightforwardly associated to their corresponding parameter descriptions Ⓑ and Ⓒ, annotated with `@param` with parameter names. Slice ❸ calls the *rangeCheck()* method defined in ❻, which handles the `IndexOutOfBoundsException` (lines 24–25). As such, ❻ is associated with Ⓔ, the `@throws` comment. Slice ❹ (lines 10 and 12) deals with the return value, and is associated with comment Ⓓ. The remaining code ❺ forms the main functionality, and is associated with the core functionality description Ⓐ.

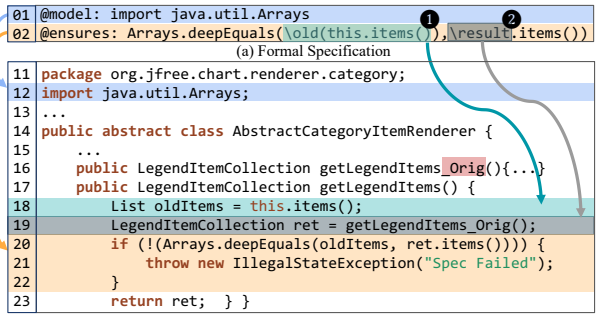
Task 2: Data Validation. We will develop a specification validation pipeline, which consists of symbolic validation, automated testing, and expert review. This pipeline not only refines our dataset from Task 1, but also serves as a reusable tool for evaluating external specification-generation methods.

We perform symbolic validation to ensure specifications are logically sound. Specifications are encoded into solver-compatible formats (e.g., SMT-LIB for Z3 [76]) by resolving scopes, modeling pre- and post-state values, and enforcing types. The solver checks for contradictions (e.g., `x > 0` $\Rightarrow$ `x < 0`) and tautologies (e.g., `true == true`), filtering out ill-formed or uninformative specifications to improve dataset quality.

To validate specifications through automated testing, we will build a tool that translates them into executable checks and instruments them into source code. The instrumented program will run against existing test suites, which we treat as reliable due to prior use in unit and regression testing. Any runtime violations will be flagged for review. We instrument source code rather than test code for two reasons: (1) to ensure specifications are checked for all methods, including private or indirectly invoked ones in test code, and (2) to avoid the complexity and incompleteness of identifying all test call sites, particularly under dynamic dispatch.

Translating specifications into executable code is non-trivial. Using JML as an example, constructs like quantified expressions (e.g., `\forall`, `\exists`) and implications (e.g., $e_1 \Rightarrow e_2$) require translation into loops and conditionals. Pre-state references (e.g., `\old()`) demand storing values before method execution. Our tool addresses these by automatically generating supporting code based on construct semantics.

Figure 5 shows how a JML specification (a) is translated and injected into the source code (b). The required `import java.util.Arrays` (line 01) is added if missing. The pre- and post-state expressions, `\old(this.items())` (❶) and `\result` (❷), are separately translated into variable definitions (lines 18–19). The original method is renamed (line 16), and a wrapper with the original signature (lines 17–24) invokes it and checks the translated specification via an `if` statement (line 20). Violations raise an exception (line 21), enabling runtime detection.



(a) Formal Specification

(b) Injected Source Code

Figure 5: Specification Injection Example

Human expert review is the final validation step in dataset construction. Specifications are evaluated for (1) semantic consistency with the code and (2) alignment with the NL description. Reasoning traces are additionally assessed for (3) logical coherence and clarity. Each sample is independently reviewed by multiple experts; disagreements are resolved through discussion or escalated when needed. If LLM outputs are invalid or incomplete, experts provide high-quality replacements to ensure dataset integrity.

Preliminary Results: The feasibility of the proposed research is strongly supported by PI Zhai's prior work [37, 50]. PI Zhai introduced the first fine-grained comment taxonomy and method for associating comments with code [37], demonstrating expertise directly applicable to designing the proposed specification dataset. In [50], PI Zhai developed EVALGPTFIX, a dataset for evaluating ChatGPT's performance on automated program repair, further establishing experience in dataset construction and rigorous evaluation.

Evaluation Plan: For Task 1, dataset evaluation will focus on two aspects: (1) Correctness, random samples will undergo expert validation to verify the accuracy of NL-specification-code alignments, with inter-annotator agreement (e.g., Fleiss' Kappa [77]) measuring reviewer consistency; and (2) Coverage, we will assess diversity across property categories, dependency scopes, logical complexities, and state dependencies to ensure broad representation. For Task 2, we will measure precision and recall in detecting invalid or incomplete specifications and reasoning traces, along with efficiency improvements over purely manual validation. The dataset will be updated based on community feedback and field advancements.

Thrust 2: Specification Generation & Maintenance

This thrust develops techniques to generate and maintain specifications for agentic development workflows. We focus on three core tasks: (1) inferring specifications from NL documentation, (2) propagating them across codebases via static dependencies, and (3) maintaining them in response to evolving code or documentation. These capabilities ensure specifications remain accurate and actionable, enabling AI agents and tools to reason effectively about software behavior throughout development and evolution.

Task 3: Specification Inference. This task builds an automated system to generate formal specifications from NL documentation. Prior pattern- and search-based methods [14, 15, 23, 28, 30] require heavy manual effort and lack generalizability. While LLM-based approaches show promise [31, 32, 78, 79], they remain limited: 1) they are resource-intensive, limiting accessibility, 2) they lack safeguards against hallucinations, producing unreliable outputs, and 3) they omit pre-state information, reducing expressiveness and accuracy.

We will develop a four-stage pipeline. First, we fine-tune a small open-source language model using the dataset from Thrust 1. Second, for a given method and NL description, we construct an augmented prompt using retrieved code structures and comments via the RAG module developed in Task 1 and use the fine-tuned model to generate formal specifications with reasoning traces. Third, specifications are validated through the automated pipeline from Task 2. Finally, all outputs are reviewed via an interactive user interface (UI) that supports optional human feedback. As outlined in Task 1, the specifications will capture intra- and inter-class dependencies, reference standard libraries, and preserve pre-state information.

We apply supervised fine-tuning to train a model to generate specifications from NL prompts enriched with structural code context. Each training instance includes a prompt, a reasoning trace, and a target specification. The prompt integrates the method signature, NL description, structural context at both class and repository levels, and task-specific instructions. Recent studies [80–83] show that introducing an explicit reasoning phase, where a model first produces structured logical steps, significantly improves performance. Following this, our model is trained to generate a reasoning trace followed by the final specification. While the reasoning path may vary, the final output must align with ground truth, ensuring both interpretability and accuracy.

In the inference stage, given a method signature and its NL description, our approach retrieves related code structure and comments via program analysis, constructs a context-rich prompt, and uses a fine-tuned model to generate candidate specifications with reasoning traces. By incorporating a RAG module to capture inter-method and inter-class dependencies, it produces more accurate, context-aware specifications while reducing hallucination. Figure 6 presents an example from JGraphT [84]. Given a method and its comment (lines 1–

```
1 /** Returns a set containing the edges
  ↪  associated with the specified vertex. */
2 public Set edgeOf(Object vertex){ ... }
3 //@ ensures (\forall Object e;
  ↪  \result.contains(e) <==>
  ↪  graph.containsEdge(vertex,e));
4 //@ ensures (\forall Object e;
  ↪  \result.contains(e) <==>
  ↪  vertex.equals(graph.getEdgeSource(e) ||
  ↪  vertex.equals(graph.getEdgeTarget(e));
```

Figure 6: RAG Example from JGraphT

2), Code Llama [85] hallucinates a non-existent method (`containsEdge(vector, e)`) in the generated specification (line 3). To mitigate this, our RAG-based approach retrieves relevant project context to ground the model's output. Here, it surfaces two related methods, `getEdgeSource(E e)` and `getEdgeTarget(E e)`, whose comments indicate they collectively provide the intended functionality. Using this context, our model generates a corrected, accurate specification (line 4), avoiding hallucination and improving reliability.

Our approach will validate generated specifications using the symbolic evaluation and testing process established in Task 2. In real-world development, including open-source and industry settings, test cases are commonly available through manual effort, test-driven practices, or automation. Benchmarks like Defects4J [86] confirm their prevalence, making them valuable for specification validation. When tests are unavailable, validation relies solely on symbolic analysis.

We will build an interactive UI that displays inferred specifications with their method signatures, NL descriptions, and code context. Users can review, validate, and optionally provide feedback, enabling human-in-the-loop refinement. This improves precision, captures edge cases, and supports iterative model tuning.

Preliminary Results: PI Zhai's prior research [23, 24] on specification synthesis from NL descriptions offers valuable insights that can be applied to this task. Her recent work [87] also offers guidance for efficient model fine-tuning relevant to this task.

Evaluation Plan: We will evaluate our system on accuracy, reasoning quality, generalization, contextual effectiveness, and efficiency. Part of the dataset constructed in Thrust 1 will be used to fine-tune the model, while the remaining data will serve as held-out test sets for evaluation. Accuracy will be assessed using precision and recall against ground truth, and reasoning traces will be reviewed for logical coherence with inter-annotator agreement (e.g., Fleiss' Kappa). Generalization will be tested on unseen projects, and we will measure the system's ability to capture cross-class dependencies and pre-state conditions. Ablation studies will isolate the contribution of class-level context, repository-level context, and reasoning traces. Comparisons against state-of-the-art methods will demonstrate improvements in specification accuracy. Finally, we will report model size and inference time to assess efficiency.

Task 4: Specification Propagation. This task develops a method to propagate specifications across the codebase using static dependency relationships, assigning confidence scores based on behavioral consistency and structural prevalence. The key insight is that *methods are not isolated; their behavior is shaped by callers and calees, inheritance, and interface contracts, which encode implicit assumptions about inputs, outputs, and exceptions*. By analyzing these relationships, we infer specifications even without explicit NL descriptions. However, not all inferred specifications are equally robust. Some capture general behavioral contracts, while others reflect narrow, context-specific assumptions. We assign confidence scores to quantify their generality and support, offering interpretable guidance for downstream use. All propagated specifications will be validated via symbolic analysis and testing (Task 2). As our approach relies solely on structural context, these specifications serve as independent expectations, enabling inconsistency checks against implementation.

For caller-callee relationships, we propagate specifications by analyzing how a method is used across its call sites. Using program slicing, our approach extracts relevant code fragments around these call sites to infer behavioral expectations. Specifications are then propagated forward from the caller to infer preconditions and backward from callee usage to infer postconditions. Figure 7 illustrates an example of specification propagation in `java.lang.reflect.Array`. The method `newInstance()` (lines 4–7) calls the native method `newArray()` (line 9), which lacks comments from which to infer specifications. Based on the comment (lines 1–2), our tool infers the precondition `length >= 0` (line 3), which holds at the call site (line 5). The inferred precondition is propagated to `newArray()` (line 8),

```
01 /** @throws NegativeArraySizeException if the
02   * specified code length is negative.*/
03 /** @requires length >= 0 */                    Inference
04 public Object newInstance(Class<?> componentType,
       int length) throws NegativeArraySizeException {
05   /** @requires length >= 0 */   Propagation ①
06   return newArray(componentType, length);
07 }
08 /** @requires length >= 0 */   Propagation ②
09 private static native Object newArray(
       Class<?> componentType, int length)
       throws NegativeArraySizeException;
```

Figure 7: Specification Propagation Example

providing it with a derived specification. During propagation, actual parameters from the caller are mapped to the callee's formal parameters to ensure specifications align with the callee's scope and naming. Here, the two methods share the parameter name `length`, so the propagated preconditions appear identical.

For inheritance and interface contracts, we propagate specifications bidirectionally: from parent classes or interfaces to subclasses and from subclasses back to their supertypes. In downward propagation, inherited specifications may initially be too weak for subclasses but still convey meaningful behavioral constraints; these can be strengthened using available specifications at the subclasses, either inferred from NL descriptions or propagated via caller-callee dependencies. For instance, a superclass might require that an input be non-null, while a subclass refines this to additionally require that the input is non-empty. In upward propagation, specifications aggregated from multiple subclasses may be overly restrictive for the general parent class. To mitigate this, we generalize constraints by identifying common behavioral guarantees across subclasses, employing SAT solvers to reason about logical entailment and synthesize the weakest valid specification.

While logical reasoning ensures formal soundness, not all inferred specifications are equally reliable or broadly applicable. To capture this variability, we will assign confidence scores based on the prevalence and consistency of inferred constraints across dependencies. Specifications supported by diverse, independent evidence will be scored higher, providing a principled measure of their robustness. For example, if 90% of call sites validate that an input is positive and 10% perform no checks, the inferred precondition would be assigned a high confidence score reflecting its broad support.

Preliminary Results: The feasibility of this proposed research has been demonstrated by PI Zhai's one recent project that propagates NL comments from one code entity to another based on program semantics [37].

Evaluation Plan: We will evaluate the propagated specifications on three dimensions: correctness, coverage, and robustness. Correctness will be assessed through expert review with inter-annotator agreement to ensure consistency. Coverage will measure the proportion of methods with successfully inferred specifications across different dependency types. Robustness will be evaluated by correlating confidence scores with expert-validated quality, ensuring higher-scored specifications have lower error rates. Comparisons against state-of-the-art work will validate improvements in specification accuracy.

Task 5: Specification Maintenance. Maintaining specifications as code and documentation evolve is one of the most persistent and overlooked challenges in SE. In both traditional and agentic workflows, specifications quickly become outdated as software changes, undermining their role in testing, debugging, and automated reasoning. Without robust maintenance, even high-quality specifications degrade, leading to silent misalignments that erode trust in analysis tools and AI agents.

Specification maintenance entails a few key challenges: (1) identifying which specifications are affected by a given change; (2) handling the ripple effects of changes across callers, callees, and type hierarchies; and (3) validating that updates remain correct and consistent with intent. To address them, we will develop a systematic approach that combines test execution, fine-grained association models, and static dependency analysis to identify obsolete specifications. When specifications require revision, we will regenerate them using our inference techniques from Tasks 3 and 4, and validate them using symbolic and runtime checks from Task 2. For cases where code changes without updated NL descriptions, we will automatically revise the associated comments using our LLMCup system [88], complemented by existing models when appropriate [89–92], ensuring fresh and accurate input to the inference pipeline. Finally, all updated specifications will be presented through an iterative UI to support optional human review and feedback.

To identify impacted specifications, we combine test execution, learning-based alignment models, and static dependency analysis. First, when tests are available, we instrument specifications using the tool from Task 2 and run test cases to detect behavioral deviations; failures suggest the associated specifications may be invalid. Second, our approach identifies directly affected specifications using classifiers trained to assess alignment between specifications and their associated NL descriptions or code snippets. Specifically, to estimate semantic alignment, one classifier is trained on pairs of NL and specification, while another is

trained on pairs of code snippet and specification. The dataset curated in Task 1 will be used for training. Upon a change in NL description, the first classifier estimates which specification is associated with the original description and then check whether this specification still aligns with the new description. For a code change, our approach extracts the corresponding code slice that contains the code change via program slicing and applies the second classifier to determine which specification was associated with the original code snippet and whether this specification still aligns with the new code snippet. Third, our approach uses static dependency analysis to trace transitive effects on specifications of related methods, including callers, callees, type hierarchies, and methods that incorporate the changed method into their own specifications.

Figure 8 illustrates an example. The method `checkRange()` (line 11) had the specification `index >= 0 && index < this.getEditableColumns()` at line 9, assuming that `getEditableColumns()` returns the number of editable columns, as described and implemented (lines 1–3). Following a behavioral change, `getEditableColumns()` was modified to return the editable columns themselves rather than their count (lines 4–6). As a result, the specifications of both `getEditableColumns()` and `checkRange()` must be updated to reflect the new behavior.

```
1  - /** Returns the number of editable columns. */
2  - public int getEditableColumns(){
3  -   return m_editableColumns.size();
4  + /** Returns the editable columns. */
5  + public List<TableProperty> getEditableColumns() {
6  +   return m_editableColumns;
7  }
8  /** The specified index should be non-negative and
   ↪  smaller than the number of the editable columns.
9  -  * @requires index >= 0 && index <
   ↪  this.getEditableColumns()
10 +  * @requires index >= 0 && index <
   ↪  this.getEditableColumns().size() */
11 public void checkRange(int index){}
```

Figure 8: Updating Specifications from OpenCms

We will develop a UI to help developers inspect and manage specifications affected by code or NL changes. The UI will highlight modified code/NL and corresponding specifications, and provide side-by-side comparisons of old and regenerated specifications. It will allow users to approve, reject, or revise suggested updates, enabling a human-AI collaboration workflow that combines model-generated insights with human oversight for improved reliability.

Preliminary Results: PI Zhai's prior work on comment propagation via program analysis [37], along with recent work on detecting and updating outdated comments [88], provides a strong foundation for identifying and updating outdated specifications through static dependency analysis.

Evaluation Plan: We will evaluate the proposed maintenance work across outdated specification detection accuracy, regeneration quality, and end-to-end effectiveness. We will reuse an existing dataset from our prior comment update work [88], which includes old/new code and NL descriptions, and augment it with corresponding specifications and annotations indicating which specifications are affected by each change. Using this dataset, we will measure the system's ability to detect outdated specifications based on precision and recall. Regeneration quality will be assessed via expert review and the validation pipeline in Task 2. We will perform an ablation study across the three detection strategies to quantify the individual contribution of each component to detection performance.

Thrust 3: Innovative Applications

This thrust develops tools and agent capabilities that leverage formal specifications to enhance the quality and reliability of AI-generated code and tests. In software development, aligning code with intended behavior and detecting flaws early are long-standing challenges. In agentic workflows, where AI agents autonomously generate and revise large volumes of code, these challenges become more acute. Misalignments can propagate rapidly, and undetected errors may compound across iterations. This thrust addresses both issues: first, by fine-tuning models on specification-rich datasets to improve behavioral alignment in code generation (Task 6); second, by integrating formal specifications into testing workflows as executable assertions to catch errors early and prevent regressions (Task 7).

Task 6: Improving Alignment for Better Code Generation. This task focuses on fine-tuning models to generate behaviorally aligned code using richly annotated datasets from Thrust 1. In agentic workflows

where AI agents synthesize or modify code from user prompts, maintaining semantic alignment with intended behavior is critical. Our dataset contains fine-grained pairs of NL descriptions, formal specifications, and code snippets, which we will use to train models that can generate code that is not only syntactically correct but also semantically aligned with the specified behavior grounded by formal specifications.

We fine-tune a base code generation model (e.g., DeepSeek Coder [93]) using a dataset of aligned tuples $(\text{nl}, \text{spec}, \text{code}, \text{context}_c, \text{context}_r)$, where nl is a NL description of the intended behavior, spec is the corresponding formal specification, code is the target implementation, and context_c and context_r provides class- or repository-level context. During training, the model takes as input the concatenation of NL description and specification, and is optimized to generate code autoregressively. An example data sample is shown in Figure 2; part (e) is excluded for this task as it is used for inferring specifications rather than code.

The model is trained to minimize a primary objective of autoregressive cross-entropy between the predicted code $\hat{\text{code}}_i$ and the reference implementation code_i. To further promote semantic and formal alignment, we incorporate auxiliary losses that relate the generated code to the NL description and the specification. The overall loss is defined as: $\mathcal{L} = \lambda_{\text{code}} \cdot \mathcal{L}_{\text{code}}(\hat{\text{code}}_i, \text{code}_i) + \lambda_{\text{nl}} \cdot \mathcal{L}_{\text{nl}}(\hat{\text{code}}_i, \text{nl}_i) + \lambda_{\text{spec}} \cdot \mathcal{L}_{\text{spec}}(\hat{\text{code}}_i, \text{spec}_i)$, where $\mathcal{L}_{\text{code}}$ is the standard token-level cross-entropy loss, $\mathcal{L}_{\text{nl}}$ encourages consistency with the NL description (e.g., via contrastive embedding similarity [94]), and $\mathcal{L}_{\text{spec}}$ enforces alignment with the specification (e.g., via logical entailment). It preserves a decoder-only, autoregressive architecture while incorporating supervision from both informal and formal intent signals. Hyperparameters, including the loss weights $\lambda_{\text{code}}, \lambda_{\text{nl}}, \lambda_{\text{spec}}$, are selected based on validation performance across syntactic accuracy and specification compliance. Learning rates are tuned via grid search for stability and convergence, typically within 10^{-5} to 10^{-3}. Fine-tuning is performed with LoRA [95] adapters for memory efficiency, using batch sizes appropriate to available hardware.

Figure 9 illustrates the impact. Given the NL description "`return the first element from this list`" (line 01), which is a common pattern in documentation for languages like Java and Python, the baseline model incorrectly returns the first element *after* removal (line 08 with red background), misinterpreting the intent. In contrast, the correct behavior is to return the original head element *before* modification. After fine-tuning on our specification dataset, the model generates the correct implementation (lines 06 and 09 with green background), demonstrating how formal specifications enable precise behavioral grounding and semantically aligned code generation.

```
01  /**Removes and returns the first element from this list.*/
02  public E removeFirst() {
03      final Node<E> f = first;
04      if (f == null)
05          throw new NoSuchElementException();
06  +   final E element = f.item;
07      ...
08  -   return (f == null) ? null : f.item;
09  +   return element;
10  }
```

Figure 9: Fine-Tuning with Formal Specifications Improves Code Correctness. Red: Baseline Output. Green: Correct Output After Fine-Tuning.

Preliminary Results: PI Zhai's prior work provides key insights and expertise for this task, including generating simplied code from NL [39], assessing LLM maturity via formal specifications [51], evaluating the quality of code generated by LLMs [96], testing embedding robustness [40], and efficient model fine-tuning [87].

Evaluation Plan: We will evaluate the fine-tuned model along three key dimensions: (1) syntactic correctness, measured via compilation success and exact match against reference code; (2) semantic alignment, assessed through embedding-based similarity (e.g., CodeBERTScore [97]) and human judgments on adherence to NL intent; and (3) specification compliance, verified using formal methods such as SMT solving or property-based testing. Comparisons will be made against strong baselines (e.g., DeepSeek Coder, Code LLaMA), with additional ablations to isolate the effects of specification-guided training. We will report results on held-out in-domain and out-of-domain tasks, and include human evaluation for qualitative assessment.

Task 7: Specification-guided Test Case Generation. This task enhances automated test generation by integrating formal specifications into tools like EvoSuite [98]. In agentic workflows where AI agents can generate or modify large volumes of code rapidly, specifications must be leveraged during development to catch errors on the fly. Without real-time detection, debugging becomes prohibitively difficult, and regressions

or critical flaws may go unnoticed. Integrating specifications into the generation loop can help catch errors early, prevent regressions, and guide agent behavior toward satisfying intended functionality.

Current methods rely on heuristics to explore unconstrained input and execution spaces and infer weak oracles from runtime behavior, leading to path explosion and limited behavioral coverage. In agentic workflows, these inefficiencies are further magnified: heuristic-based generation requires expensive inference cycles and extended planning, exacerbating already high computational costs. Moreover, agents operate with limited context windows, often missing broader project constraints, causing them to generate shallow or misaligned tests that overlook critical logic.

We propose to extend EvoSuite by grounding test generation in precise specifications to reduce the search space, strengthen test oracles, and improve relevance and efficiency under real-world constraints. In agentic workflows, where AI agents synthesize and revise code at scale, such specification-guided testing enables on-the-fly validation, catching behavioral errors early before they propagate. Preconditions will restrict input generation to valid usage scenarios, while postconditions will be translated into runtime assertions that serve as precise test oracles, enabling the detection of subtle behavioral deviations.

Several technical challenges arise in this integration. EvoSuite only supports a limited subset of JML and cannot handle quantified constraints (e.g., `\forall`, `\exists`), which are essential for reasoning over collections. Java's short-circuit evaluation may skip relevant conditions in compound expressions, reducing observability. Furthermore, deeply nested or complex logical expressions degrade search efficiency by hindering EvoSuite's ability to satisfy test goals.

Our approach builds on specification inference techniques from Thrust 2 and translates the inferred specifications into executable assertions using Task 2's infrastructure. To improve observability, our approach applies a divide-and-conquer strategy that decomposes compound specifications into atomic predicates with intermediate assertions. This exposes each condition during execution, avoids short-circuiting issues, and provides EvoSuite with fine-grained feedback to guide its search more effectively. Finally, the executable checks will be instrumented into source code using Task 2's technique, enabling EvoSuite to generate valid, semantically meaningful test cases with improved coverage, oracle strength, and overall quality.

Figure 10 shows how specifications can enhance automated test generation. Line 1 presents the specification for the method `remove(Object o)` (line 2). It states that if the input object `o` exists in the original list, then after executing `remove(Object)`, all elements before its original index remain unchanged, while elements after it are shifted one position to the left. Without specification guidance, EvoSuite generates the test (lines 4–7 and 9–10) that only checks whether the method returns `true` (line 10), missing critical behavioral validations. By integrating our inferred specification, EvoSuite could generate additional test logic highlighted in lines 8 and 11–16. The first loop (lines 12–13) asserts elements preceding the input String "`banana`" remain unchanged, while the second (lines 14–15) verifies subsequent elements are shifted to left correctly. This enhancement significantly improves oracle quality and behavioral coverage compared to the baseline test.

```
1  //@ ensures \old(this.contains(o)) ==> \forall int i; i>=0 &&
   ↪  i<\old(this.indexOf(o)); (this.get(i) == null &&
   ↪  \old(this.get(i)) == null) || (this.get(i) != null &&
   ↪  this.get(i).equals(\old(this.get(i)))) && \forall int i;
   ↪  i>\old(this.indexOf(o)) && i<\old(this.size());
   ↪  (this.get(i-1) == null && \old(this.get(i)) == null) ||
   ↪  (this.get(i-1) != null &&
   ↪  this.get(i-1).equals(\old(this.get(i))))
2  public boolean remove(Object o){...}
3
4  ArrayList list = new ArrayList();
5  list.add("apple");
6  list.add("banana");
7  list.add("orange");
8  ArrayList oldList = list.clone();
9  boolean ret = list.remove("banana");
10 org.junit.Assert.assertTrue(ret);
11 boolean flag = true;
12 for(int i = 0; i >= 0 && i < 1; i++)
13         flag = flag && list.get(i).equals(oldList.get(i));
14 for(int i = 0; i > 1 && i < 3; i++)
15         flag = flag && list.get(i-1).equals(oldList.get(i));
16 org.junit.Assert.assertTrue(flag);
```

Figure 10: Test Case Generation Example from JDK

Preliminary Results: PI Zhai has extensive experience in software testing [23, 35–37, 40–45]. Her research

has detected over 1,500 bugs in real-world software systems [37, 41, 43–45, 47–49, 52–54], with more than 290 confirmed or fixed by developers. Some of them were identified using specifications automatically inferred from NL comments [37].

Evaluation Plan: We will evaluate our tool with and without an agentic workflow, comparing it to EvoSuite and Randoop on real-world projects. Metrics include line, branch, and mutation coverage, assertion diversity, and bug detection. Ablation studies will isolate the impact of specifications by removing input filtering and output checking. We will also conduct case studies on methods with rich behavioral contracts, such as list operations. It will demonstrate how inferred specifications improve assertion quality, reduce path explosion, and guide agents toward generating more meaningful and semantically grounded tests.

Education & Broader Impacts

The research, education, and broader impact components will evolve together (Figure 1). The research findings will be incorporated into initiatives via the following distinct efforts.

[Goal 1] Curriculum Development. At Rutgers, PI Zhai regularly taught CS431 *Software Engineering*, CS112 *Data Structures*, and CS111 *Introduction to Computer Science*, with a total enrollment of 1,000 students per year. She designed a new curriculum of CS431, incorporating recent advances such as AI for SE and SE for AI. She also involved in the revamping of CS112, updating learning objectives, slides, assignments, recitation questions, exams, and the course website. At UMass Amherst, PI Zhai regularly teaches the UMass graduate-level courses CS520 *Theory and Practice of Software Engineering* with about 150 students per semester and CS692P *Hot Topics in Software Engineering Research* every two years, and she will teach CS621 *Advanced Software Engineering: Analysis and Evaluation* starting from Fall 2026. PI Zhai has integrated AI techniques into SE practices, which has been well received by students. Two students from her CS520 class are now working with her on research projects. One is co-authoring a paper expected to be submitted to FSE 2026. PI Zhai is an experienced instructor with high course evaluations.

PI Zhai will modernize SE education to reflect the transformative role of AI agents in software development. Beyond traditional instruction, she will incorporate hands-on experiences that demonstrate how formal specifications guide AI-driven tools and agents, fostering deeper understanding and improving software quality. By embedding these concepts in the classroom, PI Zhai aims to prepare a new generation of software engineers who not only appreciate the value of specifications but are also equipped to design and apply them in modern, AI-augmented environments. Example curriculum enhancements include: ① **Modernized content:** Students will learn recent advances in AI for SE, including testing, debugging, and automation, through real-world case studies and tools for specification-centric development in agentic workflows. Student outcomes will be measured using pre- and post-assessments on specification understanding, engagement metrics, and course feedback. Materials and tools will be released publicly to support integration into SE curricula at other institutions. ② **Joyful hands-on experiences:** PI Zhai will design engaging hands-on projects where students write formal specifications for small software components, use automated tools to generate specifications, and leverage those specifications to identify and fix bugs in provided code samples. The experience of using AI for SE and the accomplishment of finding and fixing bugs will be rewarding feedback for students, as evidenced by her previous experiences. ③ **Learning by doing, together:** PI Zhai plans to incorporate group projects into the curriculum. It will enable students to work together on creating and analyzing formal specifications, and foster a deeper understanding through collaboration. Students will select complex SE problems that require formal specifications, and take on different roles (e.g., specification writer, verifier, and tester). This ensures a balanced workload and allows students to specialize in their areas of interest. ④ **Self assessment and peer feedback:** Students will participate in multiple review cycles, both as reviewers and reviewees, to gain diverse feedback and perspectives. PI Zhai will provide initial sessions on giving and receiving constructive feedback, emphasizing positive and actionable comments. These sessions will also clarify review criteria, focusing on the correctness and clarity of specifications, as well as the effectiveness of their implementation and testing.

[Goal 2] Students Advising, Mentoring, and Engagement. Another crucial aspect of this project involves mentoring graduate and undergraduate students. This proposal will provide funding for one Ph.D. student who will be working on formal specifications.

Advising Graduate Students. In the past, PI Zhai has advised one master's student and two Ph.D. students [24, 44, 99]. She was also the thesis advisor for six master's students. At UMass Amherst, PI Zhai is now advising 1 Ph.D. student, 1 incoming Ph.D. student and 2 master's students. Her students, Gehao Zhang (Ph.D.) and Po-Hsiang Wang (Master), are collaborating on generating formal specifications, aligning with this proposal and slated for submission to FSE 2026. Additionally, PI Zhai collaborates with students from other institutions, leading to publications at ICSE, FSE, ISSTA, ASE, TOSEM, and TDSC [36, 40–42, 46, 47, 49, 100–103]. One student has worked on comment maintenance, which directly supports Task 5, and has submitted a related research paper. She will continue these efforts following the detailed mentoring plan attached.

Advising Undergraduate Students. PI Zhai has a strong track record of doing research with undergraduates. One of her undergraduate students, Xiangzhe Xu, worked on informal specification propagation, and won the runner-up award of ASE 2020 Student Research Competition. Xiangzhe is now pursuing a Ph.D. degree at Purdue University. PI Zhai worked as a research advisor at Rutgers Aresty Research Center through Rutgers Aresty Research Program [104] for three years, advising one undergraduate each academic year. At UMass Amherst, she is advising two senior students and one senior intern from another institution. PI Zhai will continue to actively engage undergraduate students in research activities, including participating in the ERSP (Early Research Scholars Program) program [105] at UMass and working with undergraduate students on independent study projects. PI Zhai will apply for the NSF REU program [106] to support undergraduate students in research once this proposal gets funded.

K–12 Outreach. Many talented students are lost to STEM fields before they reach college, making it essential to engage younger students and encourage their exploration of STEM. PI Zhai has actively participated in events aimed at attracting and retaining people in STEM at both Rutgers and UMass. She shared and presented her career path with a strong belief that role models are important in recruiting and retaining. As part of this project, PI Zhai plans to organize workshops for local middle- and high-school students. These workshops will include, but are not limited to, the following topics: ① guest speaker sessions where successful people from various STEM fields will share their career paths; ② hands-on activities such as coding bootcamps and engineering design projects; ③ pairing students with mentors who are STEM professionals or university students; and ④ helping with college applications. In particular, PI Zhai plans to organize two workshops during the project: one in year 2 and another in year 5. In Year 2, she will introduce the basic concept of specifications and demonstrate their practical applications through the Soda Straw Bridge project. In Year 5, she will demonstrate how formal specifications guide code generation and support AI agents, exposing students to core computing concepts and inspiring STEM interest through real-world AI and logic applications.

[Goal 3] Open Source and Broader Impacts. PI Zhai will actively promote open-source development and collaboration to maximize the impact of her research just like what she did in the past [35, 36, 40, 46–49, 54, 96, 100, 101, 107–114]. She will organize workshops and tutorials to educate users on the datasets and tools, ensuring users are well-equipped to contribute effectively. She will also establish a leaderboard for benchmarking (Thrust 1), encouraging the community to evaluate and compare different tools on standardized datasets. Active community participation will be further encouraged through forums and structured feedback mechanisms, and she plans to host them on GitHub. Regular evaluations and updates will be performed to ensure that the datasets remain relevant, meaningful, and up-to-date. All data collection and usage will adhere to strict ethical guidelines, ensuring compliance with privacy laws and respect for intellectual property rights. Data will be anonymized where necessary, and explicit permissions will be obtained for any proprietary data used. These measures will safeguard user privacy and maintain the integrity of the datasets.

Impact on Other Fields. The research builds bridges across AI, programming languages, formal verification,

and security communities. In machine learning, specifications can define precise behavioral contracts for data pipelines, training workflows, and model-serving APIs, enhancing reliability and accountability. In cybersecurity, specifications offer a principled way to enforce and validate security invariants, such as access control. In programming languages and verification, inferred specifications can be fed into theorem provers, model checkers, and type systems to automate reasoning about correctness. For safety-critical domains, specifications help ensure safety and fault tolerance where manual specification is impractical.

Technology Transfer. PI Zhai will pursue technology transfer by engaging with industry and open-source partners who build developer tools, IDEs, and continuous integration pipelines, ensuring that research outcomes benefit both academia and practice. By enabling the automated alignment of code and intent, this work aims to shift how reliability and correctness are ensured in software development and maintenance at scale.

Evaluation Plan UMass Amherst has a systematic evaluation plan for the educational components of this project, including evaluations and feedback from in-class students, mentees, faculty peers with rich expertise on teaching and education, faculty mentors, industrial partners, etc. PI Zhai will fully leverage them to evaluate the effectiveness of the proposed educational activities and collect feedback for further improvement. She will also actively seek for feedback on the open-source tools and datasets, and evaluate the impact of the project on the broader community in annual basis.

Related work

Besides the related work mentioned in each thrust, the proposed research is also related to the following. *LLMs for SE* LLMs have been used for many SE tasks, such as requirement engineering [115, 116], code generation [58, 117–123], test generation [124–129, 129–134], documentation generation [135–137], and software maintenance and evolution [138–146]. *Formal Specification Generation* There are approaches of generating specifications from source code, based on static analysis techniques [147–152], dynamic analysis approaches [34, 153–156], and large-scale repository mining [157–161]. *Code-Comment Inconsistency Detection* Researchers develop methods to identify potential code-comment inconsistencies using NLP techniques and program analysis techniques [27–29, 162–171]. *NL Artifacts Utilization* NL artifacts have been analyzed to detect bugs [27, 29, 30, 37, 172–180], recommend code examples for using API [181], synthesize code [39, 182–187], improve code search [188–190], mine semantically similar words in software [191–194], generate code description [195–197], and recommend code changes [198–202].

Management

Each task will have three phases: *(i)* design the proposed technique, implement a prototype, and obtain a dataset necessary for later phases; *(ii)* implement the proposed technique; *(iii)* evaluate extensively, prepare a report, and release sources of the tools, as she has done in the past [35, 36, 40, 46–49, 54, 87, 100, 101, 103, 107–114]. Table 1 shows the planned distribution of tasks throughout the project. Every year, she will spend time preparing and revising new educational materials based on feedback from students and users. Deliverables will consist of *(i)* conference and journal publications; *(ii)* software implementations and documentations; *(iii)* empirically collected datasets; and *(iv)* educational software modules and materials. She will make all the deliverables publicly online on GitHub.

Table 1: Project Timeline; Darker Means More Effort

Thrusts and Tasks	Y_1	Y_2	Y_3	Y_4	Y_5
Dataset					
Dataset Curation					
Data Validation					
Generation & Maintenance					
Specification Inference					
Specification Propagation					
Specification Maintenance					
Innovative Applications					
Code Generation					
Test Case Generation					
Edu & Broader Impact					
Curriculum					
Advising, Mentoring					
Community					

Results from Prior NSF Support

The PI has no prior NSF support.

References

[1] OpenAI Launches an Agentic, Web-Based Coding Tool. https://www.wired.com/story/openai-launches-an-agentic-web-based-coding-tool/?utm_source=chatgpt.com, 2025.

[2] New Tools for Building Agents. https://openai.com/index/new-tools-for-building-agents/?utm_source=chatgpt.com, 2025.

[3] Ranjan Sapkota, Konstantinos I Roumeliotis, and Manoj Karkee. Vibe Coding vs. Agentic Coding: Fundamentals and Practical Implications of Agentic AI. *arXiv preprint arXiv:2505.19443*, 2025.

[4] How Specifications Can Become the Future of Programming. https://python.plainenglish.io/how-specifications-can-become-the-future-of-programming-0a553a78c286, 2025.

[5] The New Code Talk. https://www.youtube.com/watch?v=8rABwKRsec4, 2025.

[6] AI Will Write 90% of Code Within 6 Months. https://www.businessinsider.com/anthropic-ceo-ai-90-percent-code-3-to-6-months-2025-3?utm_source=chatgpt.com, 2025.

[7] Copilot Introduced 41% More Bugs. https://uplevelteam.com/blog/ai-for-developer-productivity?utm_source=chatgpt.com, 2025.

[8] Shihan Dou, Haoxiang Jia, Shenxi Wu, Huiyuan Zheng, Weikang Zhou, Muling Wu, Mingxu Chai, Jessica Fan, Caishuang Huang, Yunbo Tao, et al. What's Wrong With Your Code Generated by Large Language Models? an extensive study. *arXiv preprint arXiv:2407.06153*, 2024.

[9] QiHong Chen, Jiachen Yu, Jiawei Li, Jiecheng Deng, Justin Tian Jin Chen, and Iftekhar Ahmed. A Deep Dive Into Large Language Model Code Generation Mistakes: What and Why? *arXiv preprint arXiv:2411.01414*, 2024.

[10] Ziyao Zhang, Yanlin Wang, Chong Wang, Jiachi Chen, and Zibin Zheng. LLM Hallucinations in Practical Code Generation: Phenomena, Mechanism, and Mitigation. *arXiv preprint arXiv: 2409.20550*, 2025.

[11] Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, Li Zhang, Zhongqi Li, and Yuchi Ma. Exploring and Evaluating Hallucinations in LLM-Powered Code Generation. *arXiv preprint arXiv: 2404.00971*, 2024.

[12] Zhao Tian, Junjie Chen, and Xiangyu Zhang. Fixing Large Language Models' Specification Misunderstanding for Better Code Generation. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 645–645. IEEE Computer Society, 2025.

[13] Manish Motwani and Yuriy Brun. Automatically Generating Precise Oracles from Structured Natural Language Specifications. In *International Conference on Software Engineering (ICSE)*, pages 188–199, Montreal, QC, Canada, May 2019.

[14] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. Translating Code Comments to Procedure Specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 242–253, Amsterdam, Netherlands, July 2018. ACM.

[15] Alberto Goffi, Alessandra Gorla, Michael D Ernst, and Mauro Pezzè. Automatic Generation of Oracles for Exceptional Behaviors. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 213–224. ACM, 2016.

[16] Andreas Leitner, Ilinca Ciupa, Manuel Oriol, Bertrand Meyer, and Arno Fiva. Contract Driven Development = Test Driven Development-Writing Test Cases. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 425–434. ACM, 2007.

[17] Bertrand Meyer, Ilinca Ciupa, Andreas Leitner, and Lisa Ling Liu. Automatic Testing of Object-Oriented Software. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 114–129. Springer, 2007.

[18] James A Jones, Mary Jean Harrold, and John Stasko. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 467–477. IEEE, 2002.

[19] Andreas Zeller and Ralf Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[20] Osbert Bastani, Saswat Anand, and Alex Aiken. Specification Inference Using Context-Free Language Reachability. In *ACM SIGPLAN Notices*, pages 553–566. ACM, 2015.

[21] Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement Types for Haskell. In *ACM SIGPLAN Notices*, pages 269–282. ACM, 2014.

[22] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[23] Juan Zhai, Yu Shi, Minxue Pan, Guian Zhou, Yongxiang Liu, Chunrong Fang, Shiqing Ma, Lin Tan, and Xiangyu Zhang. C2S: Translating Natural Language Comments to Formal Program Specifications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 25–37, 2020.

[24] Shiyu Zhang, Juan Zhai, Bu Lei, Wang Linzhang, and Xuandong Li. Automated Generation of LTL Specifications For Smart Home IoT Using Natural Language. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020.

[25] Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W Godfrey. DocTer: Documentation-Guided Fuzzing for Testing Deep Learning API Functions. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 176–188, 2022.

[26] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. Inferring Method Specifications from Natural Language API Descriptions. In *Proceedings of the 34th International Conference on Software Engineering*, pages 815–825. IEEE Press, 2012.

[27] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T Leavens. tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 260–269. IEEE, 2012.

[28] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /* iComment: Bugs or Bad Comments? */. In *ACM SIGOPS Operating Systems Review*, pages 145–158. ACM, 2007.

[29] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. Analyzing APIs Documentation and Code to Detect Directive Defects. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 27–37. IEEE, 2017.

[30] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. aComment: Mining Annotations from Comments and Code to Detect Interrupt Related Concurrency Bugs. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 11–20. IEEE, 2011.

[31] Danning Xie, Byungwoo Yoo, Nan Jiang, Mijung Kim, Lin Tan, Xiangyu Zhang, and Judy S Lee. Impact of Large Language Models on Generating Software Specifications. *arXiv preprint arXiv:2306.03324*, 2023.

[32] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K Lahiri. Can Large Language Models Transform Natural Language Intent into Formal Method Postconditions? *Proceedings of the ACM on Software Engineering*, (FSE):1889–1912, 2024.

[33] Gary T Leavens, Albert L Baker, and Clyde Ruby. JML: A Notation for Detailed Design. *Behavioral Specifications of Businesses and Systems*, pages 175–188, 1999.

[34] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Sci. Comput. Program.*, 69:35–45, 2007.

[35] Xuanqi Gao, Juan Zhai, Shiqing Ma, Chao Shen, Yufei Chen, and Shiwei Wang. CILIATE: Towards Fairer Class-Based Incremental Learning by Dataset and Training Refinement. In René Just and Gordon Fraser, editors, *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, volume abs/2304.04222, pages 475–487. ACM, 2023.

[36] Xuanqi Gao, Juan Zhai, Shiqing Ma, Chao Shen, Yufei Chen, and Qian Wang. FairNeuron: Improving Deep Neural Network Fairness with Adversary Games on Selective Neurons. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, volume abs/2204.02567, pages 921–933. ACM, 2022.

[37] Juan Zhai, Xiangzhe Xu, Yu Shi, Minxue Pan, Shiqing Ma, Lei Xu, Weifeng Zhang, Lin Tan, and Xiangyu Zhang. CPC: Automatically Classifying and Propagating Natural Language Comments via Program Analysis. In *Software Engineering (ICSE), 2020 IEEE/ACM 42nd International Conference on*, pages 1359–1371. IEEE, 2020.

[38] Yang Zhang, Shuai Shao, Juan Zhai, and Shiqing Ma. FineLock: Automatically Refactoring Coarse-Grained Locks Into Fine-Grained Locks. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 565–568, 2020.

[39] Juan Zhai, Jianjun Huang, Shiqing Ma, Xiangyu Zhang, Lin Tan, Jianhua Zhao, and Feng Qin. Automatic Model Generation from Documentation for Java API Functions. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, ICSE'16, pages 380–391. IEEE, 2016.

[40] Weipeng Jiang, Juan Zhai, Shiqing Ma, Xiaoyu Zhang, and Chao Shen. COSTELLO: Contrastive Testing for Embedding-based Large Language Model as a Service Embeddings. In *Proceedings of the ACM International Conference on the Foundations of Software Engineering*, 2024.

[41] Zeqin Liao, Henglong Liang, Yuhong Nan, Sicheng Hao, Zibin Zheng, Juan Zhai, and Jiajing Wu. SmartAxe: Detecting Cross-Chain Vulnerabilities in Bridge Smart Contracts via Fine-Grained Static Analysis. In *Proceedings of the ACM International Conference on the Foundations of Software Engineering*, 2024.

[42] Quanjun Zhang, Juan Zhai, Chunrong Fang, Jiawei Liu, Weisong Sun, Haichuan Hu, and Qingyu Wang. Machine Translation Testing via Syntactic Tree Pruning. *ACM Transactions on Software Engineering and Methodology*, 33(5):1–39, 2024.

[43] Minxue Pan, Yifei Lu, Yu Pei, Tian Zhang, Juan Zhai, and Xuandong Li. Effective Testing of Android Apps Using Extended IFML Models. *Journal of Systems and Software*, 159:110433, 2020.

[44] Yifei Lu, Minxue Pan, Juan Zhai, Tian Zhang, and Xuandong Li. Preference-Wise Testing for Android Applications. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 268–278. ACM, 2019.

[45] Zhenhao Tang, Juan Zhai, Minxue Pan, Yousra Aafer, Shiqing Ma, Xiangyu Zhang, and Jianhua Zhao. Dual-Force: Understanding WebView Malware via Cross-Language Forced Execution. In *ASE*, pages 714–725, 2018.

[46] Xuanqi Gao, Weipeng Jiang, Juan Zhai, Shiqing Ma, Xiaoyu Zhang, and Chao Shen. Efficient DNN-powered Software with Fair Sparse Models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 2024*. ACM, 2024.

[47] Yinglong Zou, Juan Zhai, Chunrong Fang, Jiawei Liu, Tao Zheng, and Zhenyu Chen. Mutation-Based Deep Learning Framework Testing Method in JavaScript Environment. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 970–981, 2024.

[48] Yanzhou Mu, Juan Zhai, Chunrong Fang, Xiang Chen, Zhixiang Cao, Peiran Yang, Yinglong Zou, Tao Zheng, and Zhenyu Chen. DevMuT: Testing Deep Learning Framework via Developer Expertise-Based Mutation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1533–1544, 2024.

[49] Yanzhou Mu, Juan Zhai, Chunrong Fang, Xiang Chen, Zhixiang Cao, Peiran Yang, Kexin Zhao, An Guo, and Zhenyu Chen. Improving Deep Learning Framework Testing with Model-Level Metamorphic Testing. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):2158–2180, 2025.

[50] Quanjun Zhang, Tongke Zhang, Juan Zhai, Chunrong Fang, Bowen Yu, Weisong Sun, and Zhenyu Chen. A Critical Review of Large Language Model on Software Engineering: An Example from ChatGPT and Automated Program Repair. *arXiv preprint arXiv:2310.08879*, 2023.

[51] Fusen He, Juan Zhai, and Minxue Pan. Beyond Code Generation: Assessing Code LLM Maturity with Postconditions. *arXiv preprint arXiv:2407.14118*, 2024.

[52] Xuanqi Gao, Juan Zhai, Shiqing Ma, Siyi Xie, and Chao Shen. ASSURE: Metamorphic Testing for AI-powered Browser Extensions. *arXiv preprint arXiv: 2507.05307*, 2025.

[53] Yanzhou Mu, Rong Wang, Juan Zhai, Chunrong Fang, Xiang Chen, Zhiyuan Peng, Peiran Yang, Ruixiang Qian, Shaoyu Yang, and Zhenyu Chen. Deep Learning Framework Testing via Model Mutation: How Far Are We? *arXiv preprint arXiv: 2506.17638*, 2025.

[54] Yinglong Zou, Juan Zhai, Chunrong Fang, Yanzhou Mu, Jiawei Liu, and Zhenyu Chen. Deep Learning Framework Testing via Heuristic Guidance Based on Multiple Model Measurements. *arXiv preprint arXiv: 2507.15181*, 2025.

[55] Laboratory for Advanced Software Engineering Research. https://groups.cs.umass.edu/laser/, 2024.

[56] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, D. Song, and J. Steinhardt. Measuring Coding Challenge Competence With APPS. *NeurIPS Datasets and Benchmarks*, 2021.

[57] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. MultiPL-E: A Scalable and Extensible Approach to Benchmarking Neural Code Generation. *arXiv preprint arXiv: 2208.08227*, 2022.

[58] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, Suchir Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code. *ArXiv*, abs/2107.03374, 2021.

[59] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR, 2023.

[60] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. *Advances in Neural Information Processing Systems*, 36, 2024.

[61] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-Trained Models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12, 2024.

[62] Alexander LeClair, Siyuan Jiang, and Collin McMillan. A Neural Model for Generating Natural Language Summaries of Program Subroutines. In *Proceedings of the 41st International Conference on Software Engineering*, pages 795–806. IEEE Press, 2019.

[63] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *arXiv preprint arXiv:1909.09436*, 2019.

[64] Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K Reddy. XLCoST: A Benchmark Dataset for Cross-Lingual Code Intelligence. *arXiv preprint arXiv:2206.08474*, 2022.

[65] icontract. https://github.com/Parquery/icontract, 2025.

[66] Github. https://github.com/, 2018.

[67] JML Specification Examples. http://www.eecs.ucf.edu/leavens/JML/examples.shtml, 2025.

[68] Eclipse Java Development Tools (JDT). https://www.eclipse.org/jdt/, 2025.

[69] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *arXiv preprint arXiv:2002.08155*, 2020.

[70] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The Faiss Library. *arXiv preprint arXiv:2401.08281*, 2024.

[71] Mark Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, (4):352–357, 2009.

[72] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[73] Java Documentataion. https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html, 2025.

[74] JSDoc. https://jsdoc.app/, 2015.

[75] Helping with Documentation. https://devguide.python.org/documentation/help-documenting/index.html#helping-with-documentation, 2025.

[76] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2008.

[77] Joseph L Fleiss. Measuring Nominal Scale Agreement Among Many Raters. *Psychological Bulletin*, 76(5):378, 1971.

[78] Matthias Cosler, Christopher Hahn, Daniel Mendoza, Frederik Schmitt, and Caroline Trippel. nl2spec: Interactively Translating Unstructured Natural Language to Temporal Logics with Large Language Models. In *International Conference on Computer Aided Verification*, pages 383–396. Springer, 2023.

[79] Jens U Kreber and Christopher Hahn. Generating Symbolic Reasoning Problems with Transformer GANs. *arXiv preprint arXiv:2110.10054*, 2021.

[80] OpenAI. Learning to Reason with LLMs, 2024.

[81] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *arXiv preprint arXiv:2501.12948*, 2025.

[82] Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. s1: Simple Test-Time Scaling. *arXiv preprint arXiv:2501.19393*, 2025.

[83] Ling Yang, Zhaochen Yu, Bin Cui, and Mengdi Wang. ReasonFlux: Hierarchical LLM Reasoning via Scaling Thought Templates. *arXiv preprint arXiv:2502.06772*, 2025.

[84] JGraphT: A Java Library of Graph Theory Data Structures and Algorithms. https://jgrapht.org/, 2024.

[85] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code Llama: Open Foundation Models for Code. *arXiv preprint arXiv:2308.12950*, 2023.

[86] René Just, Darioush Jalali, and Michael D Ernst. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440, 2014.

[87] Xiaoyu Zhang, Juan Zhai, Shiqing Ma, Chao Shen, Tianlin Li, Weipeng Jiang, and Yang Liu. STAFF: Speculative Coreset Selection for Task-Specific Fine-Tuning. In *The Thirteenth International Conference on Learning Representations*.

[88] Hua Ge, Juan Zhai, Minxue Pan, Fusen He, and Ziyue Tan. LLMCup: Ranking-enhanced comment updating with llms. *arXiv preprint arXiv:2507.08671*, 2025.

[89] Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. Source Code Summarization in the Era of Large Language Models. *arXiv preprint arXiv:2407.07959*, 2024.

[90] Sheena Panthaplackel, Pengyu Nie, Miloš Gligorić, Junyi Jessy Li, and R. Mooney. Learning to Update Natural Language Comments Based on Code Changes. *Annual Meeting of the Association for Computational Linguistics*, 2020.

[91] Zhongxin Liu, Xin Xia, Meng Yan, and Shanping Li. Automating Just-in-Time Comment Updating. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 585–597, 2020.

[92] Bo Lin, Shangwen Wang, Kui Liu, Xiaoguang Mao, and Tegawendé F Bissyandé. Automated Comment Update: How Far Are We? In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 36–46. IEEE, 2021.

[93] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

[94] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. Supervised Contrastive Learning. *Advances in neural information processing systems*, 33:18661–18673, 2020.

[95] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank Adaptation of Large Language Models. *ICLR*, 1(2):3, 2022.

[96] Weipeng Jiang, Xuanqi Gao, Juan Zhai, Shiqing Ma, Xiaoyu Zhang, Ziyan Lei, and Chao Shen. From Effectiveness to Efficiency: Uncovering Linguistic Bias in Large Language Model-based Code Generation. *arXiv preprint arXiv: 2406.00602*, 2025.

[97] Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. Codebertscore: Evaluating Code Generation with Pretrained Models of Code. *arXiv preprint arXiv:2302.05527*, 2023.

[98] Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 416–419, 2011.

[99] Ruihua Ji, Junyu Pei, Wenhua Yang, Juan Zhai, Minxue Pan, and Tian Zhang. Extracting Mapping Relations for Mobile User Interface Transformation. In *Internetware '19: The 11th Asia-Pacific Symposium on Internetware, 2019*, 2019.

[100] Xiaoyu Zhang, Juan Zhai, Shiqing Ma, and Chao Shen. AutoTrainer: An Automatic DNN Training Problem Detection and Repair System. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 359–371. IEEE, 2021.

[101] Xiaoyu Zhang, Juan Zhai, Shiqing Ma, Xiaohong Guan, and Chao Shen. DREAM: Debugging and Repairing AutoML Pipelines. *ACM Transactions on Software Engineering and Methodology*, 34(4):1–29, 2025.

[102] Xiuting Ge, Chunrong Fang, Xuanye Li, Weisong Sun, Daoyuan Wu, Juan Zhai, Shang-wei Lin, Zhihong Zhao, Yang Liu, and Zhenyu Chen. Machine Learning for Actionable Warning Identification: A Comprehensive Survey. *ACM Computing Surveys*, 57(2):1–35, 2024.

[103] Xiaoyu Zhang, Chao Shen, Shiqing Ma, Juan Zhai, and Chenhao Lin. An automated monitoring and repairing system for dnn training. *IEEE Transactions on Dependable and Secure Computing*, 2024.

[104] Rutgers Aresty Research Program. https://aresty.rutgers.edu/programs-funding/research-assistant-program, 2025.

[105] Early Research Scholars Program. https://sites.google.com/umass.edu/ersp-cics, 2025.

[106] Research Experiences for Undergraduates Program. https://www.nsf.gov/crssprgm/reu/, 2024.

[107] Zhenting Wang, Hailun Ding, Juan Zhai, and Shiqing Ma. Training with More Confidence: Mitigating Injected and Natural Backdoors During Training. In *Proc. of NeurIPS*, 2022.

[108] Zhenting Wang, Juan Zhai, and Shiqing Ma. BppAttack: Stealthy and Efficient Trojan Attacks against Deep Neural Networks via Image Quantization and Contrastive Adversarial Learning. In *Proc. of CVPR*, 2022.

[109] Zhenting Wang, Kai Mei, Hailun Ding, Juan Zhai, and Shiqing Ma. Rethinking the Reverse-Engineering of Trojan Triggers. In *Proc. of NeurIPS*, 2022.

[110] Hailun Ding, Shenao Yan, Juan Zhai, and Shiqing Ma. ELISE: A Storage Efficient Logging System Powered by Redundancy Reduction and Representation Learning. In *Proc. of USENIX Security*, 2021.

[111] Hailun Ding, Juan Zhai, Dong Deng, and Shiqing Ma. The Case for Learned Provenance Graph Storage Systems. In *Proceedings of the 32nd USENIX Conference on Security Symposium*, SEC '23, USA, 2023. USENIX Association.

[112] Hailun Ding, Juan Zhai, Yuhong Nan, and Shiqing Ma. AIRTAG: Towards Automated Attack Investigation by Unsupervised Learning with Log Texts. In *Proceedings of the 32nd USENIX Conference on Security Symposium*, SEC '23. USENIX Association, 2023.

[113] Xiaoyu Zhang, Juan Zhai, Shiqing Ma, Qingshuang Bao, Weipeng Jiang, Qian Wang, Chao Shen, and Yang Liu. The Invisible Hand: Unveiling Provider Bias in Large Language Models for Code Generation. *Annual Meeting of the Association for Computational Linguistics*, 2025.

[114] Weipeng Jiang, Zhenting Wang, Juan Zhai, Shiqing Ma, Zhengyu Zhao, and Chao Shen. An Optimizable Suffix Is Worth A Thousand Templates: Efficient Black-Box Jailbreaking without Affirmative Phrases via LLM as Optimizer. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 5471–5483, 2025.

[115] Xianchang Luo, Yinxing Xue, Zhenchang Xing, and Jiamou Sun. PRCBERT: Prompt Learning for Requirement Classification Using BERT-Based Pretrained Language Models. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13, 2022.

[116] Dipeeka Luitel, Shabnam Hassani, and Mehrdad Sabetzadeh. Improving Requirements Completeness: Automated Assistance Through Large Language Models. *Requirements Engineering*, 29(1):73–95, 2024.

[117] Jiawei Liu, Chun Xia, Yuyao Wang, and Lingming Zhang. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. *ArXiv*, abs/2305.01210, 2023.

[118] Ayush Kumar, Parth Nagarkar, Prabhav Nalhe, and Sanjeev Vijayakumar. Deep Learning Driven Natural Languages Text to SQL Query Conversion: A Survey. *ArXiv*, abs/2208.04415, 2022.

[119] Naihao Deng, Yulong Chen, and Yue Zhang. Recent Advances in Text-to-SQL: A Survey of What We Have and What We Expect. *ArXiv*, abs/2208.10099, 2022.

[120] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. InCoder: A Generative Model for Code Infilling and Synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

[121] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Haiquan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *International Conference on Learning Representations*, 2022.

[122] Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B Tenenbaum, and Chuang Gan. Planning with Large Language Models for Code Generation. *arXiv preprint arXiv:2303.05510*, 2023.

[123] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. Jigsaw: Large Language Models Meet Program Synthesis. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1219–1231, 2022.

[124] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. Generating Accurate Assert Statements for Unit Test Cases Using Pretrained Transformers. *2022 IEEE/ACM International Conference on Automation of Software Test (AST)*, pages 54–64, 2020.

[125] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. Adaptive Test Generation Using a Large Language Model. *ArXiv*, abs/2302.06527, 2023.

[126] Mohammed Latif Siddiq, Joanna Cecilia Da Silva Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinícius Carvalho Lopes. Using Large Language Models to Generate JUnit Tests: An Empirical Study. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, EASE 2024. ACM, June 2024.

[127] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-Trained Large Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 919–931. IEEE, 2023.

[128] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C Desmarais. Effective Test Generation Using Pre-Trained Large Language Models and Mutation Testing. *Information and Software Technology*, 171:107468, 2024.

[129] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. Large Language Models are Edge-Case Fuzzers: Testing Deep Learning Libraries via FuzzGPT. *arXiv preprint arXiv:2304.02014*, 2023.

[130] Sungmin Kang, Juyeon Yoon, and Shin Yoo. Large Language Models are Few-Shot Testers: Exploring LLM-Based General Bug Reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2312–2323. IEEE, 2023.

[131] Myeongsoo Kim, Tyler Stennett, Saurabh Sinha, and Alessandro Orso. A multi-agent approach for rest api testing with semantic graphs and llm-driven inputs. *arXiv preprint arXiv:2411.07098*, 2024.

[132] Venkata Sai Aswath Duvvuru, Bohan Zhang, Michael Vierhauser, and Ankit Agrawal. LLM-Agents Driven Automated Simulation Testing and Analysis of small Uncrewed Aerial Systems. *arXiv preprint arXiv:2501.11864*, 2025.

[133] Shiyao Zhou, Jincheng Wang, He Ye, Hao Zhou, Claire Le Goues, and Xiapu Luo. LWDIFF: an LLM-Assisted Differential Testing Framework for Webassembly Runtimes. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 769–769. IEEE Computer Society, 2025.

[134] Soneya Binta Hossain and Matthew Dwyer. TOGLL: Correct and Strong Test Oracle Generation with LLMs. *arXiv preprint arXiv:2405.03786*, 2024.

[135] Junjie Zhao, Xiang Chen, Guang Yang, and Yiheng Shen. Automatic Smart Contract Comment Generation via Large Language Models and In-Context Learning. *Information and Software Technology*, 168:107405, 2024.

[136] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. Large Language Models are Few-Shot Summarizers: Multi-Intent Comment Generation via In-Context Learning. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24. ACM, February 2024.

[137] Chia-Yi Su and Collin McMillan. Distilled GPT for Source Code Summarization. *Automated Software Engineering*, 31(1):22, 2024.

[138] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A Generic Method for Automatic Software Repair. *Ieee Transactions on Software Engineering*, 38(1):54–72, 2011.

[139] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 172–184, 2023.

[140] Toufique Ahmed, Supriyo Ghosh, Chetan Bansal, Thomas Zimmermann, Xuchao Zhang, and Saravan Rajmohan. Recommending Root-Cause and Mitigation Steps for Cloud Incidents Using Large Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1737–1749. IEEE, 2023.

[141] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. InferFix: End-to-End Program Repair with LLMs. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1646–1656, 2023.

[142] Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. TFix: Learning to Fix Coding Errors with a Text-to-Text Transformer. In *International Conference on Machine Learning*, pages 780–791. PMLR, 2021.

[143] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. Impact of Code Language Models on Automated Program Repair. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1430–1442, 2023.

[144] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. CodeT5: Identifier-Aware Unified Pre-Trained Encoder-Decoder Models for Code Understanding and Generation. *ArXiv*, abs/2109.00859, 2021.

[145] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. Repairagent: An autonomous, llm-based agent for program repair. *arXiv preprint arXiv:2403.17134*, 2024.

[146] Aidan ZH Yang, Sophia Kolak, Vincent J Hellendoorn, Ruben Martins, and Claire Le Goues. Revisiting Unnaturalness for Automated Program Repair in the Era of Large Language Models. *arXiv preprint arXiv:2404.15236*, 2024.

[147] Junjie Chen, Yanwei Bai, Dan Hao, Lingming Zhang, Lu Zhang, Bing Xie, and Hong Mei. Supporting Oracle Construction via Static Analysis. *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 178–189, 2016.

[148] Sharon Shoham, Eran Yahav, Stephen J. Fink, and Marco Pistoia. Static Specification Mining Using Automata-Based Abstractions. *IEEE Transactions on Software Engineering*, 34:651–666, 2007.

[149] Cormac Flanagan and K. Rustan M. Leino. Houdini, an Annotation Assistant for ESC/Java. In *FME*, 2001.

[150] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: Securing Software by Blocking Bad Input. *ACM SIGOPS Operating Systems Review*, 41(6):117–130, 2007.

[151] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. Automatic Inference of Necessary Preconditions. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 128–148. Springer, 2013.

[152] Mohamed Nassim Seghir and Daniel Kroening. Counterexample-Guided Precondition Inference. In *European Symposium on Programming*, pages 451–471. Springer, 2013.

[153] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[154] Jeremy W Nimmer and Michael D Ernst. Automatic Generation of Program Specifications. *ACM SIGSOFT Software Engineering Notes*, 27(4):229–239, 2002.

[155] Angello Astorga, Siwakorn Srisakaokul, Xusheng Xiao, and Tao Xie. PreInfer: Automatic Inference of Preconditions via Symbolic Analysis. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 678–689. IEEE, 2018.

[156] Sudheendra Hangal and Monica S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 291–301, 2002.

[157] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Static Specification Inference Using Predicate Mining. In *ACM SIGPLAN Notices*, pages 123–134. ACM, 2007.

[158] Hoan Anh Nguyen, Robert Dyer, Tien N Nguyen, and Hridesh Rajan. Mining Preconditions of APIs in Large-Scale Code Corpus. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 166–177. ACM, 2014.

[159] Xujie Si, Aaditya Naik, Hanjun Dai, M. Naik, and Le Song. Code2Inv: A Deep Learning Framework for Program Verification. *Computer Aided Verification*, 12225:151–164, 2020.

[160] Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Sekhar Jana. CLN2INV: Learning Loop Invariants with Continuous Logic Networks. *ArXiv*, abs/1909.11542, 2019.

[161] Facundo Molina, Renzo Degiovanni, Pablo Ponzio, Germán Regis, Nazareno Aguirre, and Marcelo Fabian Frias. Training Binary Classifiers as Data Structure Invariants. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 759–770, 2019.

[162] Inderjot Kaur Ratol and Martin P Robillard. Detecting Fragile Comments. In *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*, pages 112–122. IEEE, 2017.

[163] Fazle Rabbi and Md Saeed Siddik. Detecting Code Comment Inconsistency Using Siamese Recurrent Network. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 371–375, 2020.

[164] Fazle Rabbi, Md Nazmul Haque, Md Eusha Kadir, Md Saeed Siddik, and Ahmedul Kabir. An Ensemble Approach to Detect Code Comment Inconsistencies using Topic Modeling. In *SEKE*, pages 392–395, 2020.

[165] Sheena Panthaplackel, Junyi Jessy Li, Milos Gligoric, and Raymond J Mooney. Deep Just-in-Time Inconsistency Detection between Comments and Source Code. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 427–435, 2021.

[166] Theo Steiner and Rui Zhang. Code Comment Inconsistency Detection with BERT and Longformer. *arXiv preprint arXiv:2207.14444*, 2022.

[167] Zhengkang Xu, Shikai Guo, Yumiao Wang, Rong Chen, Hui Li, Xiaochen Li, and He Jiang. Code Comment Inconsistency Detection Based on Confidence Learning. *IEEE Transactions on Software Engineering*, 2024.

[168] Haiyang Yang, Hao Chen, Zhirui Kuai, Shuyuan Tu, and Li Kuang. ASKDetector: An AST-Semantic and Key Features Fusion Based Code Comment Mismatch Detector. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, pages 392–402, 2024.

[169] Sicheng Hao, Yuhong Nan, Zibin Zheng, and Xiaohui Liu. SmartCoCo: Checking Comment-Code Inconsistency in Smart Contracts via Constraint Propagation and Binding. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 294–306. IEEE, 2023.

[170] Guoping Rong, Yongda Yu, Song Liu, Xin Tan, Tianyi Zhang, Haifeng Shen, and Jidong Hu. Code Comment Inconsistency Detection and Rectification Using a Large Language Model. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 432–443. IEEE Computer Society, 2024.

[171] Yichi Zhang, Zixi Liu, Yang Feng, and Baowen Xu. Leveraging large language model to assist detecting rust code comment inconsistency. In *Proceedings of the 39th IEEE/ACM international conference on automated software engineering*, pages 356–366, 2024.

[172] Jeffy Jahfar Poozhithara, Hazeline U Asuncion, and Brent Lagesse. Keyword Extraction From Specification Documents for Planning Security Mechanisms. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1661–1673. IEEE, 2023.

[173] Juan C Alonso, Sergio Segura, and Antonio Ruiz-Cortés. AGORA: Automated Generation of Test Oracles for REST APIs. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1018–1030, 2023.

[174] Hao Zhong and Zhendong Su. Detecting API Documentation Errors. In *ACM SIGPLAN Notices*, pages 803–816. ACM, 2013.

[175] Cindy Rubio-González and Ben Liblit. Expect the Unexpected: Error Code Mismatches between Documentation and the Real World. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 73–80. ACM, 2010.

[176] Yingying Zhang and Daqing Hou. Extracting Problematic API Features from Forum Discussions. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pages 142–151. IEEE, 2013.

[177] Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N Nguyen. A Topic-Based Approach for Narrowing the Search Space of Buggy Files from a Bug Report. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 263–272. IEEE Computer Society, 2011.

[178] Xin Ye, Razvan Bunescu, and Chang Liu. Learning to Rank Relevant Files for Bug Reports Using Domain Knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 689–699. ACM, 2014.

[179] Phong Minh Vu, Tam The Nguyen, Hung Viet Pham, and Tung Thanh Nguyen. Mining User Opinions in Mobile App Reviews: A Keyword-Based Approach (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 749–759. IEEE, 2015.

[180] Phong Minh Vu, Hung Viet Pham, Tam The Nguyen, and Tung Thanh Nguyen. Phrase-Based Extraction of User Opinions in Mobile App Reviews. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 726–731. ACM, 2016.

[181] Collin McMillan, Denys Poshyvanyk, and Mark Grechanik. Recommending Source Code Examples via API Call Usages and Documentation. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, pages 21–25. ACM, 2010.

[182] Meet Shah, Rajat Shenoy, and Radha Shankarmani. Natural Language to Python Source Code Using Transformers. In *2021 International Conference on Intelligent Technologies (CONIT)*, pages 1–4. IEEE, 2021.

[183] Anh Tuan Nguyen, Peter C Rigby, Thanh Van Nguyen, Mark Karanfil, and Tien N Nguyen. Statistical Translation of English Texts to API Code Templates. In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*, pages 331–333. IEEE, 2017.

[184] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep API Learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642. ACM, 2016.

[185] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. SWIM: Synthesizing What I Mean-Code Search and Idiomatic Snippet Synthesis. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 357–367. IEEE, 2016.

[186] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. Bimodal Modelling of Source Code and Natural Language. In *International Conference on Machine Learning*, pages 2123–2132, 2015.

[187] Tihomir Gvero and Viktor Kuncak. Synthesizing Java Expressions from Free-Form Queries. In *Acm Sigplan Notices*, pages 416–432. ACM, 2015.

[188] Thanh Van Nguyen, Anh Tuan Nguyen, Hung Dang Phan, Trong Duc Nguyen, and Tien N Nguyen. Combining word2vec with Revised Vector Space Model for Better Code Retrieval. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 183–185. IEEE Press, 2017.

[189] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Chen Fu, and Qing Xie. Exemplar: A Source Code Search Engine for Finding Highly Relevant Applications. *IEEE Transactions on Software Engineering*, 38(5):1069–1087, 2012.

[190] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. Automatic Query Reformulations for Text Retrieval in Software Engineering. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 842–851. IEEE Press, 2013.

[191] Shaowei Wang, David Lo, and Lingxiao Jiang. Inferring Semantically Related Software Terms and Their Taxonomy by Leveraging Collaborative Tagging. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 604–607. IEEE, 2012.

[192] Matthew J Howard, Samir Gupta, Lori Pollock, and K Vijay-Shanker. Automatically Mining Software-Based, Semantically-Similar Words from Comment-Code Mappings. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 377–386. IEEE Press, 2013.

[193] Jinqiu Yang and Lin Tan. SWordNet: Inferring Semantically Related Words from Software Context. *Empirical Software Engineering*, 2013.

[194] Yuan Tian, David Lo, and Julia Lawall. SEWordSim: Software-Specific Word Similarity Database. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 568–571. ACM, 2014.

[195] Sebastiano Panichella, Jairo Aponte, Massimiliano Di Penta, Andrian Marcus, and Gerardo Canfora. Mining Source Code Descriptions from Developer Communications. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 63–72. IEEE, 2012.

[196] Carmine Vassallo, Sebastiano Panichella, Massimiliano Di Penta, and Gerardo Canfora. CODES: Mining Source Code Descriptions from Developers Discussions. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 106–109. ACM, 2014.

[197] Cong Chen and Kang Zhang. Who Asked What: Integrating Crowdsourced FAQs into API Documentation. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 456–459. ACM, 2014.

[198] Andrea Di Sorbo, Sebastiano Panichella, Carol V Alexandru, Junji Shimagaki, Corrado A Visaggio, Gerardo Canfora, and Harald C Gall. What Would Users Change in My App? Summarizing App Reviews for Recommending Software Changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 499–510. ACM, 2016.

[199] Andrea Di Sorbo, Sebastiano Panichella, Carol V Alexandru, Corrado A Visaggio, and Gerardo Canfora. SURF: Summarizer of User Reviews Feedback. In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*, pages 55–58. IEEE, 2017.

[200] Adelina Ciurumelea, Andreas Schaufelbühl, Sebastiano Panichella, and Harald C Gall. Analyzing Reviews and Code of Mobile Apps for Better Release Planning. In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*, pages 91–102. IEEE, 2017.

[201] Sebastiano Panichella, Andrea Di Sorbo, Emitza Guzman, Corrado A Visaggio, Gerardo Canfora, and Harald C Gall. How Can I Improve My App? Classifying User Reviews for Software Maintenance and Evolution. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 281–290. IEEE, 2015.

[202] Sebastiano Panichella, Andrea Di Sorbo, Emitza Guzman, Corrado A Visaggio, Gerardo Canfora, and Harald C Gall. ARdoc: App Reviews Development Oriented Classifier. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1023–1027. ACM, 2016.