

# Compiling Large Multi-modal Requirement Documents into Runnable Software Systems: From an Agentic Test-Driven Perspective

WEIYU KONG, Shanghai Jiao Tong University, China

YUN LIN\*, Shanghai Jiao Tong University, China

XIWEN TEOH, National University of Singapore, Singapore

DUC-MINH NGUYEN, Shanghai Jiao Tong University, China

RUOFEI REN, Shanghai Jiao Tong University, China

JIAXIN CHANG, Shanghai Jiao Tong University, China

HAOXU HU, Shanghai Jiao Tong University, China

HAOYU CHEN, Shanghai Jiao Tong University, China

Large Language Models (LLMs) have significantly improved programming efficiency by parsing natural language into code snippets. However, their performance degrades significantly as requirements scale; when faced with multi-modal documents containing hundreds of scenarios, LLMs often produce incorrect implementations or omit crucial constraints. Observing LLMs' ever-evolving capability and their persistent stochastic hallucination, we raise a question, i.e., whether it is possible to make LLM-based agentic programming go beyond *code generation* to *requirement compilation*, i.e., whether programmers can produce a runnable system by only accomplishing (non-trivial) requirement documents?

In this work, we take a first step by proposing the ARC (Agentic Requirement Compilation) technique to parse a multi-modal requirement document, describing hundreds of scenarios in a DSL format, into a runnable software system. In addition to the source code, ARC also generates software engineering artifacts including (1) a modular design that spans the user interface, API interface, and database, (2) enriched test cases for each interface (including unit tests, modular tests, and integration tests), and (3) detailed traceability across all artifacts for software maintenance. Our approach employs a bidirectional test-driven agentic loop: (1) a **top-down architecture phase** that decomposes requirements into UI, API, and database interfaces, each of which is equipped with verifiable test suites, and (2) a **bottom-up implementation phase** where agents generate code that must satisfy the generated tests. Throughout this process, ARC maintains strict traceability across requirements, design, and code to facilitate intelligent asset reuse and follow-up maintenance. We evaluate ARC on two complementary benchmarks, i.e., a depth-oriented benchmark of 6 runnable web systems spanning 50–200 requirement scenarios, and the breadth-oriented AppForge benchmark comprising 101 Android app generation tasks. Across 3 independent trials, ARC outperforms all state-of-the-art LLM-based baselines, with the generated web systems passing on average 50.6% more GUI tests, and achieving 100% compile success and 68.3% test case pass rate on AppForge. In addition, a user study with 21 participants

\*Corresponding author.

Authors' Contact Information: [Weiyu Kong](#), Shanghai Jiao Tong University, Shanghai, China, [kwy160034@sjtu.edu.cn](mailto:kwy160034@sjtu.edu.cn); [Yun Lin](#), Shanghai Jiao Tong University, Shanghai, China, [lin\\_yun@sjtu.edu.cn](mailto:lin_yun@sjtu.edu.cn); [Xiwen Teoh](#), National University of Singapore, Singapore, Singapore, [xiwen.teoh@u.nus.edu](mailto:xiwen.teoh@u.nus.edu); [Duc-Minh Nguyen](#), Shanghai Jiao Tong University, Shanghai, China, [minh.nguyen@sjtu.edu.cn](mailto:minh.nguyen@sjtu.edu.cn); [Ruofei Ren](#), Shanghai Jiao Tong University, Shanghai, China, [renruofei0120@sjtu.edu.cn](mailto:renruofei0120@sjtu.edu.cn); [Ji-axin Chang](#), Shanghai Jiao Tong University, Shanghai, China, [cjx001234@sjtu.edu.cn](mailto:cjx001234@sjtu.edu.cn); [Haoxu Hu](#), Shanghai Jiao Tong University, Shanghai, China, [andyHu42@sjtu.edu.cn](mailto:andyHu42@sjtu.edu.cn); [Haoyu Chen](#), Shanghai Jiao Tong University, Shanghai, China, [chysunday@sjtu.edu.cn](mailto:chysunday@sjtu.edu.cn).



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2994-970X/2026/10-ARTISSTA105

<https://doi.org/10.1145/3832196>

shows that participants with limited programming experience successfully write DSL-based documents consisting of 50 to 174 scenarios, within 5.6 hours on average, to generate a runnable system such as a real-world ticket-booking system of around 10K lines of code with maintainable architecture.

CCS Concepts: • **Software and its engineering** → **Requirements analysis**; **Source code generation**; *Domain specific languages*; *Software testing and debugging*; *Formal software verification*.

Additional Key Words and Phrases: Software Engineering, Code Generation, Software Testing

### ACM Reference Format:

Weiyu Kong, Yun Lin, Xiwen Teoh, Duc-Minh Nguyen, Ruofei Ren, Jiaxin Chang, Haoxu Hu, and Haoyu Chen. 2026. Compiling Large Multi-modal Requirement Documents into Runnable Software Systems: From an Agentic Test-Driven Perspective. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA105 (October 2026), 24 pages. <https://doi.org/10.1145/3832196>

## 1 Introduction

The software engineering landscape has entered a transformative era where Large Language Models (LLMs) have evolved from simple autocompletion assistants into proactive coding partners capable of planning and executing complex development workflows [59, 71]. The integration of these models into the software development life cycle has broadened access to code creation, shifting the developer’s role from a low-level code producer to a high-level curator and architect [46, 65]. However, while LLMs excel at generating localized code snippets, their performance faces a “complexity wall” when tasked with large-scale projects. As requirement documents transition from simple text to multi-modal specifications spanning hundreds of scenarios, traditional agentic generative approaches frequently suffer from requirement drift and omission and persistent hallucinations [30, 33, 42]. This widening gap between intent and implementation raises a fundamental question in the age of agentic programming, i.e., *how far are we from shifting the paradigm from mere code generation to a more rigorous process of requirement compilation?*

The research community has primarily explored the following paradigms to address large-scale code generation from textual descriptions [33]:

- **Role-Based Orchestration:** These frameworks [29, 56] such as MetaGPT simulate a traditional software company by assigning specialized roles (e.g., Product Manager, Architect, Programmer) to different agents. By following Standard Operating Procedures (SOPs), they decompose high-level requirements into intermediate artifacts like PRDs and system designs before implementation. While their waterfall approaches [29] can output a complete repository structure, they remain procedural rather than verifiable, i.e., a misinterpretation in the initial PRD generates a structurally sound but functionally broken repository.
- **Reactive Action-Loops (OpenHands [68] / SWE-agent [73]):** These frameworks focus on tight environment interaction. OpenHands, in particular, utilizes an event-driven architecture where a centralized “EventStream” coordinates interactions between the user, the agent, and a secure docker-based runtime. By this means, these tools enable the agents to autonomously search complex directory structures, execute shell commands, and iteratively refine code based on real-time feedback from compilers or test execution results.

Despite these sophisticated structures, generating a runnable system from a multi-modal document remains difficult for the following reasons. First, most existing tools treat requirements as natural language prompts. However, natural language is inherently ambiguous and lacks the explicit hierarchy needed for repository-scale dependency tracking. MetaGPT’s waterfall approach [29] is procedural but less trustworthy in practice. For example, if a “Product Manager” agent misinterprets a requirement node, that error propagates into the architecture and code without an early-stage gate to stop the cascade. Second, reactive agents (e.g., OpenHands [68]) are excellent

at fixing but poor at compiling. They often struggle with a blank-slate repository because they cannot proactively build a global data contract (e.g., among UI, API, and database) across hundreds of scenarios. This results in fragmented implementations where individual files are created but fail to integrate into a cohesive, runnable system.

To bridge this gap, we propose ARC (Agentic Requirement Compilation), a framework that shifts the paradigm from stochastic code generation to a rigorous process of requirement compilation. Unlike existing “role-play” or “reactive” systems, ARC treats multi-modal requirement documents as a high-level source language, structured via a light-weight, graph-based Domain Specific Language (DSL). By transforming dense specifications into a formal graph of requirement scenarios, we provide a deterministic blueprint which reduces requirement drift and implementation omission typical of natural language prompts. This structured foundation enables a novel bidirectional, test-driven agentic loop, i.e.,

- **Testable Architecture Construction:** A top-down phase that first decomposes the graph into verifiable UI, API, and database interfaces, accompanied by reliable tests for each interface. As a result, the architecture is modularized as a set of interactable interfaces, each of which is equipped with a set of tests in the form of unit tests, component tests, and integration tests.
- **Constrained Code Generation:** A bottom-up implementation phase where agents generate code that must satisfy these predefined test suites as constraints. Specifically, ARC implements the architecture by satisfying low-level unit tests first, then moves to satisfy component tests and integration tests in the higher level.

Generally, this dual-phase approach simulates the workflow of the classical V-model [22] in a test-driven manner, which solves the “complexity wall” by enforcing a continuous feedback loop between design and code. By prioritizing the generation of interface tests before application logic, ARC creates a grounding mechanism that “gates” agent creative output, ensuring every generated module is strictly verified against its functional requirement. Furthermore, ARC maintains an explicit *traceability record* across the requirements, designs, and tests, allowing the system to intelligently reuse existing assets and maintain global consistency even as project scale increases. This systematic alignment ensures that the final output is not just a collection of code snippets, but a cohesive web system that is consistent with the original multi-modal document. The maintained traceability record can also facilitate software maintainability such as incremental requirement compilation. Moreover, we track the agentic requirement compiling process with the version control system (e.g., Git), allowing us to intervene the process by checking out their designated revisions.

We evaluated ARC by compiling 6 complex web systems from multi-modal requirement documents containing 50 to 200 distinct scenarios. To further assess generalizability beyond web systems, we additionally evaluated ARC on the AppForge benchmark [57], which comprises 101 Android app generation tasks. We discuss the selection rationale in Section 5.1. Our experimental results demonstrate that by replacing the stochastic one-shot generation of vanilla LLMs and the rigid waterfall roles of frameworks like MetaGPT with our bidirectional compilation loop, we enhance system reliability. Specifically, ARC outperforms state-of-the-art baselines by generating repositories that pass on average 50.6% in GUI tests and successfully eliminating the fragmented implementation errors common in existing agentic workflows. Furthermore, to assess the accessibility of our lightweight DSL, we conducted a user study involving 21 participants, including novice users with limited programming experience. The study demonstrates that the structured, graph-based DSL serves as an effective medium for human-AI collaboration. Participants successfully specify an average of 107 scenarios per project within 5.6 hours, and existing requirement

documents could be transformed into the DSL format with LLM assistance in under 10 minutes. Finally, our contributions are as follows:

- **A Test-Driven Requirement Compilation Paradigm:** We introduce a novel agentic software engineering paradigm that shifts from stochastic code generation to rigorous requirement compilation. This approach leverages a lightweight, graph-based DSL to provide formal grounding and a bidirectional, test-driven loop that ensures high reliability. By maintaining an explicit traceability record across requirements, design, and implementation, our paradigm significantly enhances the long-term maintainability of AI-generated systems.
- **The ARC Framework:** We design and implement ARC, an agentic framework capable of transforming multi-modal documents into fully runnable, repository-scale web systems. The framework features a unique top-down architecture construction phase and a bottom-up implementation generation phase, creating a reliable feedback loop that eliminates requirement drift and omission.
- **Extensive Evaluation:** We conduct a comprehensive evaluation of ARC by generating 6 real-world web systems. Our results demonstrate significant improvements in runnability and GUI test pass rates (up to 108.1%) compared to state-of-the-art agentic baselines. Furthermore, a user study with 21 participants confirms that the framework is accessible to novice users, enabling them to successfully compile production-grade repositories from scratch.

Given the space limit, all the videos, tool documentation, and experiment details are available on our project website [54], and an interactive online demonstration of ARC is available at ARC-Bench [5].

## 2 Preliminaries

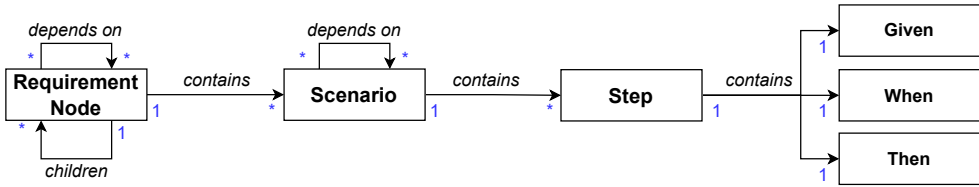


Fig. 1. The meta-model (or schema) of the multi-modal requirement of ARC. Each requirement node is equipped with a multi-modal description (via text and picture). Each requirement node and scenario is assigned with an identifier for reference. Each step is described with action and expectation via *Given*, *When*, and *Then* keywords.

### 2.1 Formal Definition and Meta-Model of Requirement

We define a requirement document as a directed acyclic graph (DAG) of Requirement Nodes. As illustrated in our meta-model (see Figure 1), each node acts as a container for full-stack specifications, spanning from high-level context at the root to specific UI, API, and DB constraints at the leaves. The core structure of our DSL is built upon three primary entities:

- **Requirement Node:** Each node represents a unique feature. It explicitly defines multi-modal description, *Dependencies* (prerequisite requirement) and *Children* (hierarchical sub-features), ensuring the framework can track repository-scale impacts.
- **Scenario:** Each requirement node contains one or more workflows that represent end-to-end user journeys. One scenario can specify a set of prerequisite scenarios, indicating its dependency.

- **Step:** A scenario is decomposed into an ordered sequence of atomic steps expressed in a Gherkin-like manner. Each step captures the test context (*Given*), the action under test (*When*), and the expected, verifiable outcome (*Then*), which are used to generate test cases.

For the full formal grammar of our DSL, readers may refer to the BNF specification on our project website [54]. The design of this DSL is not merely for organization. It is the fundamental grounding mechanism that allows ARC to overcome the complexity wall of large-scale systems.

- **Deterministic Blueprinting:** By converting ambiguous natural language into a dependency-aware graph, we provide the agentic loop with a deterministic path for top-down Architecture Construction (see Section 4).
- **Test-Driven Grounding:** Every step requires a paired action and expectation; the DSL serves as a direct source for generating GUI and API test suites. These expectations act as the “gates” that prevent the requirement drift and hallucinations common in existing models.
- **Traceability:** The unique identifiers (IDs) within the DSL allow ARC to maintain a Traceability Record. This ensures that every line of generated code in the Bottom-Up phase (see Section 4.3) is provably linked back to a specific requirement expectation.

While formal, our DSL is designed to be lightweight and accessible to both human architects and AI agents. In addition, we design a visualization tool to enable users to design the requirement graph in a more intuitive way. Interested readers can check our project website [54] for how the tool is used to draft requirement documents.

## 2.2 An Example

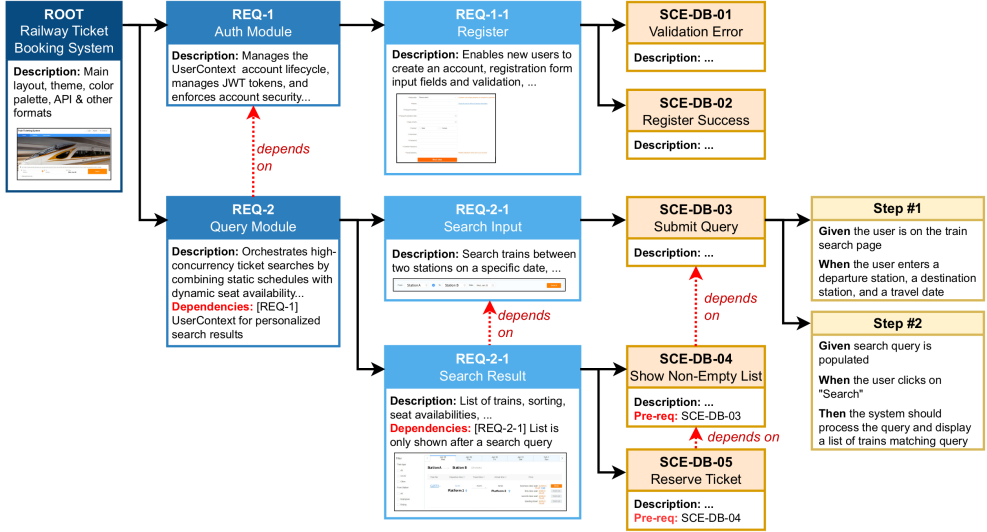


Fig. 2. An instance of multi-modal requirement document conforming to our DSL. **Blue boxes** (ROOT, REQ-) represent requirement nodes, **orange boxes** (SCE-) represent scenarios, and **yellow boxes** represent steps.

Figure 2 shows an instance of a requirement document that is in accordance with the meta-model in Figure 1. It describes a railway ticket booking system with two modules: authentication (login & registration) and query (ticket search). Each node includes multi-modal descriptions of appearance and behavior to support unit-level testing. At the *system level*, the requirement specifies

inter-feature dependencies. For example, accessing search results depends on submitting a query (REQ-2-1  $\rightarrow$  REQ-2-2), and performing a search requires a user account. These constraints can be translated into integration tests. At the *use-case level*, the scenarios capture the required interaction order for achieving specific functionalities. For instance, a user must log in before searching for tickets, and must search before reserving a ticket (SCE-DB-03  $\rightarrow$  SCE-DB-04  $\rightarrow$  SCE-DB-05). Detailed operations are recorded as step sequences, which can further serve as system-level and acceptance-level test cases (e.g., end-to-end GUI tests).

### 3 Problem Statement

Then, we define our problem statement as follows. Let  $\mathcal{G}_r = (V_r, E_r)$  be a multi-modal requirement graph, where  $V_r$  represents the set of nodes of requirements/scenarios and  $E_r \subseteq V_r \times V_r$  represents the relations between them. We define the Requirement Compilation Problem as synthesizing a realized software system  $\mathcal{S}$  and a traceability mapping  $\mathcal{M}$ , defined as follows:

**The Realized System ( $\mathcal{S}$ ).** The system is a tuple  $\mathcal{S} = (V_r, \mathcal{I}, \mathcal{T}, \mathcal{C}, \mathcal{E}_{call}, \mathcal{E}_{imp}, \mathcal{E}_{ver})$ , where:

- $\mathcal{I}$  is the set of generated Interfaces (API signatures, UI components, DB schemas).
- $\mathcal{T}$  is the set of Test Suites, strictly derived from the expectations specified in  $V_r$ .
- $\mathcal{C}$  is the set of executable Code Implementations.
- $\mathcal{E}_{call} \subseteq \mathcal{I} \times \mathcal{I}$  represents the Call Graph, denoting dependency relations between interfaces.
- $\mathcal{E}_{imp} \subseteq \mathcal{I} \times \mathcal{C}$  represents the Implementation Relation, mapping an abstract interface to its concrete logic.
- $\mathcal{E}_{ver} \subseteq \mathcal{C} \times \mathcal{T}$  represents the Verification Relation, denoting that implementation  $c$  is validated by test  $t$ .

**The Traceability Mapping ( $\mathcal{M}$ ).** We define  $\mathcal{M}$  as a provenance relation

$$\mathcal{M} \subseteq V_r \times \mathcal{I} \times \mathcal{T} \times \mathcal{C},$$

where each traceability tuple  $(r, \mathcal{I}_r, \mathcal{T}_r, c_r) \in \mathcal{M}$  signifies that:

- $\mathcal{I}_r$  is the set of interfaces projected from a requirement node or scenario  $r$ .
- $\mathcal{T}_r$  is the set of test cases derived from  $r$ .
- $c_r \in \mathcal{C}$  is an implementation that conforms to an interface  $i \in \mathcal{I}_r$  and satisfies the tests in  $\mathcal{T}_r$ .

**Objective.** The goal of ARC is to synthesize  $\mathcal{S}$  and  $\mathcal{M}$  to optimize two competing objectives:

- **Maximize Alignment:** Maximize the structural consistency between the requirement space  $V_r$  and the test space  $\mathcal{T}$  via the interface layer  $\mathcal{I}$ . Intuitively, all the requirements are translated into tests runnable upon the designed interfaces.
- **Minimize Implementation Errors (Execution Phase):** Minimize the cardinality of test failures in the generated code.

$$\min \sum_{(c,t) \in \mathcal{E}_{ver}} \mathbb{1}(\text{ExecuteTest}(c, t) \neq \text{PASS})$$

### 4 Approach

Figure 3 shows the overview of the ARC framework, which is designed as a rigorous requirement compiler that transforms high-level, multi-modal specifications into validated software systems. The framework takes a multi-modal requirement document, which has been structured into a lightweight, graph-based Domain Specific Language (DSL), serving as the source code of the intent, capturing functional scenarios and visual constraints. The final output is a fully runnable web system. Along with the system, ARC produces three critical artifacts: a software architecture consisting of UI, API, and DB interfaces, a comprehensive test suite for each interface, and a traceability record that maps the provenance from requirement to code. ARC adopts a dual-phase, i.e.,

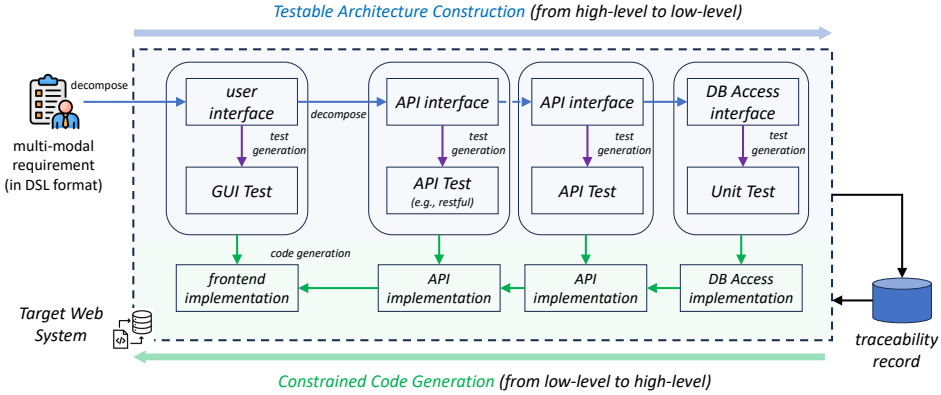


Fig. 3. An overview of ARC to parse a multi-modal requirement into a runnable web system. In addition to the web system, the resultant software artifacts include (1) a software architecture consisting of a set of interfaces (UI, API, and database), (2) a test suite for each generated interface, and (3) a traceability record to capture all the provenance from the requirement to the design, test, and implementation.

Testable Architecture Construction phase (in blue in Figure 3) and Constrained Code Generation phase (in green in Figure 3).

- The first phase follows a **top-down decomposition strategy** to translate abstract requirements into a blueprint for validation. Starting from the DSL-based requirements, ARC systematically decomposes the project into hierarchical layers: the User Interface (UI), the API interfaces, and the Database (DB) Access interfaces. For every interface defined, ARC immediately generates a corresponding test suite, including GUI Tests, RESTful API Tests, Integration Tests, and Unit Tests, before any application logic is written.
- The second phase operates in a **bottom-up implementation** direction, where agentic creative output is strictly “gated” by the previously defined architecture. Agents begin by implementing the lowest level, the DB Access layer, and move upward through the API implementation to the frontend implementation. Crucially, each generated code module is immediately executed against its predefined test suite. If a module fails its test, the agent enters a reactive loop to fix the code until it satisfies the interface contract.

Throughout both phases, ARC maintains a traceability record. This captures the link between a specific scenario in the original document and its corresponding test and code implementation. This record allows ARC to decide whether to reuse, modify, or create a new interface. In addition, such a record serves as a blueprint for the generated system, allowing developers to understand exactly why a piece of code exists and how it relates to the original multi-modal intent.

#### 4.1 Overall Algorithm

To be more specific, we introduce the detailed algorithm to parse the DSL-based requirement document. Algorithm 1 serves as the central engine of the ARC framework, utilizing a recursive Depth-First Search (DFS) strategy to traverse the requirement graph  $\mathcal{G}_r$ <sup>1</sup>. This traversal orchestrates the construction of the target software system  $\mathcal{S} = \langle \mathcal{I}, \mathcal{T}, \mathcal{C} \rangle$  by managing the lifecycle of each requirement node  $r$  through three distinct states:

<sup>1</sup>An illustrative animation of our algorithm is available at [54].

**Algorithm 1** Main Compilation Driver

---

**Require:** Requirement Graph  $\mathcal{G}_r = (V_r, E_r)$   
**Ensure:** System  $\mathcal{S}$ , Traceability  $\mathcal{M}$

```

1: Global  $\mathcal{S} \leftarrow (\emptyset, \dots, \emptyset)$ ,  $\mathcal{M} \leftarrow \emptyset$ 
2: Global  $State \leftarrow \{r : \text{UNPROCESSED} \mid \forall r \in V_r\}$ 
3: procedure COMPILENODE( $r$ )
4:    $State[r] \leftarrow \text{WORKING}$ 
5:   ▷ Phase 1: Top-Down Interface Design
6:    $\langle \mathcal{J}_r, \mathcal{T}_r \rangle \leftarrow \text{SynthesizeInterface}(r)$  ▷ See Alg. 2
7:    $\mathcal{S}.\mathcal{I} \leftarrow \mathcal{S}.\mathcal{I} \cup \mathcal{J}_r$ ;  $\mathcal{S}.\mathcal{T} \leftarrow \mathcal{S}.\mathcal{T} \cup \mathcal{T}_r$ 
8:   ▷ Recursive Step & Call Graph Construction
9:   for all  $v \in \text{Children}(r)$  do
10:    if  $State[v] = \text{UNPROCESSED}$  then
11:      COMPILENODE( $v$ )
12:    end if
13:     $\mathcal{S}.\mathcal{E}_{call} \leftarrow \mathcal{S}.\mathcal{E}_{call} \cup \{(i_r, i_v) \mid i_r \in \mathcal{J}_r, i_v \in v.\text{interfaces}\}$ 
14:  end for
15:   ▷ Phase 2: Bottom-Up Implementation
16:   $c_r \leftarrow \text{GenerateImplementation}(\mathcal{J}_r, \mathcal{T}_r, \mathcal{S}.\mathcal{E}_{call}, \mathcal{M})$  ▷ See Alg. 3
17:   $\mathcal{S}.\mathcal{C} \leftarrow \mathcal{S}.\mathcal{C} \cup \{c_r\}$ 
18:   $\mathcal{M} \leftarrow \mathcal{M} \cup \{(r, i, t, c_r) \mid i \in \mathcal{J}_r, t \in \mathcal{T}_r\}$ 
19:   $State[r] \leftarrow \text{DONE}$ 
20: end procedure
21: Start:
22: for all  $root \in \text{Roots}(\mathcal{G}_r)$  do
23:   COMPILENODE( $root$ )
24: end for
25: return  $(\mathcal{S}, \mathcal{M})$ 

```

---

- **UNPROCESSED:** The initial state for all nodes, indicating that neither the design nor the implementation for this requirement has been attempted.
- **WORKING (Top-Down Phase):** The node is currently being visited. In this state, the framework prioritizes Architecture Design, synthesizing the Interface ( $\mathcal{J}_r$ ) and Test Suite ( $\mathcal{T}_r$ ) before addressing dependencies. This prevents architectural drift by establishing a contract first.
- **DONE (Bottom-Up Phase):** The node and all its dependencies have been successfully processed. The generated implementation ( $c_r \in \mathcal{C}$ ) has been verified against  $\mathcal{T}_r$ , and integrated into  $\mathcal{S}$ .

Upon visiting an UNPROCESSED node  $r$ , the algorithm marks it as WORKING, invokes `SynthesizeInterface` (Algorithm 2), and adds  $\mathcal{J}_r$  and  $\mathcal{T}_r$  to  $\mathcal{S}.\mathcal{I}$  and  $\mathcal{S}.\mathcal{T}$ , locking in design constraints before code generation (Lines 6-8). It then recursively compiles each child  $v$  and, as recursion unwinds, links parent and child interfaces in  $\mathcal{S}.\mathcal{E}_{call}$  to align requirement dependencies with the software call structure (Lines 10-15). Once all children are processed (i.e., their interfaces are stable), the algorithm enters the implementation phase. It calls `GenerateImplementation` (Algorithm 3), passing the now-defined call graph  $\mathcal{S}.\mathcal{E}_{call}$  to ensure the generated code  $c_r$  correctly invokes its dependencies. If the code passes verification, it is added to  $\mathcal{S}.\mathcal{C}$ , and the provenance is recorded in  $\mathcal{M}$ . Finally, the node is marked as DONE, signaling to its own parents that it is ready for integration (Lines 17-23).

## 4.2 Top-down: Requirement & Design Decomposition

In this section, we describe the top-down phase in more detail (Algorithm 2). We refer to a node's requirements as an aggregation of its multi-modal descriptions, scenarios, and step-level behaviors. Before generating new interfaces, ARC queries the node's ancestors via `GetAncestors` (line

1 in Algorithm 2) to check whether existing interfaces can be adapted to satisfy part or all of the node's requirements using `AdaptInterfaceSignature` (line 5 in Algorithm 2). If applicable, it will modify/reuse these interfaces and update their test cases, producing a set of adapted interfaces  $\mathcal{J}_{\text{adapted}}$ . Finally, `GenInterfaceSignature` (line 11 in Algorithm 2) takes the node's requirements and  $\mathcal{J}_{\text{adapted}}$  to decide whether new interfaces are needed, generating  $\mathcal{J}_r$  and calling `GenTestSuite` (line 12 in Algorithm 2) to produce their corresponding test suite  $\mathcal{T}_r$ .

**4.2.1 Generating Interfaces.** In `GenInterfaceSignature` (line 11 in Algorithm 2), ARC follows a formalized principle to determine what interfaces to add. ARC uses *events* to track data flows between two interfaces that interact (e.g., via function calls). An *event* is described as  $e = (\text{name}, \text{payload})$ , where *name* identifies the event, and *payload* contains structured data in JSON standard (e.g., `{username: string, password: string}`). We first formally define the behaviors of each type of interface:

- **UI:** Defined as both a producer and a consumer of events,  $i_{\text{UI}} = (E_{\text{out}}, E_{\text{in}})$ , where  $E_{\text{out}}$  is the set of events generated by user actions and  $E_{\text{in}}$  is the set of events received from APIs.
- **API:** Defined as mapping events to operations and emits events,  $i_{\text{API}} := (e_{\text{in}}, \mathcal{O}, e_{\text{out}})$ , where  $e_{\text{in}}$  is the incoming event,  $\mathcal{O}$  is a data operation, and  $e_{\text{out}}$  is the event emitted to the UI or other APIs.
- **DB:** Defined as a set of data entities,  $i_{\text{DB}} := \{D_i\}$ , where each entity  $D_i = (\text{name}, \{a_1, a_2, \dots, a_n\})$  consists of a name and a set of attributes.

To determine the interfaces required by a node  $r$ , ARC parses both textual descriptions and images (via an image captioning vision-language model). Based on the formal definitions of interface types, it generates a set of interfaces  $i \in \mathcal{J}_r$  that collectively capture all requirements. Each interface signature specifies the type (**UI**, **API**, or **DB**), the name, and the expected location in the system  $\mathcal{S}$ . For **UI** interfaces, the signature includes layout information inferred from images, as well as the events produced and consumed. For **API** interfaces, it includes the incoming events, the data operations performed (expressed as function headers with parameters), and the outgoing events emitted to other interfaces. For **DB** interfaces, it includes the data entity name and its attributes. When two interfaces reference the same event (i.e., one produces it and another consumes it), a data flow is established between them, which serves as a reference for the bottom-up phase.

Using the login feature in Figure 2 as an example, the agent generates a **UI** interface that produces a `LoginClicked(username, password)` event and consumes a `LoginResponse(success, message)` event, with layout information inferred from the screenshot. It also generates an **API** interface that maps the incoming `LoginClicked` event to the `authService.login(username, password)` data operation (function header) and emits the corresponding `LoginResponse` event. Finally, the **DB** interface defines a `User` entity with attributes `username` and `passwordHash`. The detailed agent prompt and more detailed animated demonstrations are available at [54].

**4.2.2 Generating Test Suites.** After generating interfaces, ARC calls `GenTestSuite` to produce test scripts for the node  $r$ . The inputs to this process include: (1) The scenarios in the node,  $r.\text{scenarios}$ , and their steps,  $\text{scenario.steps}$ . (2) Interface signatures  $i \in \mathcal{J}_r$  to be tested. (3) Ancestor tests  $\mathcal{T}'_a$ , which can serve as test fixtures.

For test types, the agent follows a formal principle: **end-to-end tests** validate **UI**  $\leftrightarrow$  **API** interactions; **integration tests** validate **API**  $\leftrightarrow$  **API** and **API**  $\leftrightarrow$  **DB** interactions; and **unit tests** validate individual **UI**, **API**, **DB**, and the data operators of **API**.

For test content, the agent interprets each scenario step: **Given** establishes test fixtures or prerequisites, potentially reusing ancestor tests  $\mathcal{T}'_a$ ; **When** specifies actions or events to perform; and

**Algorithm 2** Phase 1: Synthesize Interface (Top-Down)**Require:** Node  $r$ **Ensure:** Interfaces  $\mathcal{I}_r$ , Test Suite  $\mathcal{T}_r$ 

```

1:  $Ancestors \leftarrow \text{GetAncestors}(r)$ 
2:  $AncestorInterfaces \leftarrow \bigcup_{a \in Ancestors} a.interfaces$  ▷ Collect interfaces synthesized in ancestors
3:  $\mathcal{I}_{adapted} \leftarrow \emptyset$ 
4: for all  $i_a \in AncestorInterfaces$  do
5:    $\langle i'_a, \mathcal{T}'_a \rangle \leftarrow \text{AdaptInterfaceSignature}(i_a, r)$  ▷ Try to modify or reuse ancestor interface
6:   if  $i'_a \neq \text{null}$  then
7:      $\mathcal{I}_{adapted} \leftarrow \mathcal{I}_{adapted} \cup \{i'_a\}$ 
8:   end if
9: end for
10: ▷ Generate new interfaces for any parts of  $r$  not covered by adapted interfaces
11:  $\mathcal{I}_r \leftarrow \text{GenInterfaceSignature}(r, \mathcal{I}_{adapted})$ 
12:  $\mathcal{T}_r \leftarrow \text{GenTestSuite}(r.scenarios, \mathcal{T}'_a, \mathcal{I}_r)$  ▷ Generate tests for the new interfaces
13: return  $\langle \mathcal{I}_r, \mathcal{T}_r \rangle$ 

```

**Then** specifies expected outcomes, which the agent translates into concrete assertions or verification statements.

Using the same example, ARC generates end-to-end tests that verify a UI-initiated interaction (i.e., through submitting a login UI form) corresponding to `LoginClicked(username, password)` results in a response consistent with `LoginResponse(success, message)`. Integration tests validate the **API**  $\leftrightarrow$  **DB** interaction by ensuring that the **API** correctly queries the User entity, verifies the stored passwordHash, and properly handles both successful and failed login scenarios. Unit tests focus on the **API** logic by exercising `authService.login(username, password)` under varied inputs, while **DB** unit tests validate the schema and constraints of the User entity.

### 4.3 Bottom-up: Constrained Code Generation

The second phase of the ARC framework operates in a bottom-up direction, transforming the testable architecture into a realized software system. As described in Algorithm 3, this phase ensures that agentic creative output is strictly gated by the predefined interfaces and test suites, preventing architectural drift. The implementation process follows the recursive unwinding of the DFS traversal, distinguishing between leaf and non-leaf nodes.

*Implementation of Leaf Nodes.* For a leaf node  $r \in V_r$  (where  $\text{Children}(r) = \emptyset$ ), the implementation focus is localized. The input to the agent consists of the set of interfaces  $I_r$  generated during the top-down phase. The objective is to synthesize a code implementation  $c_r$  that fulfills the contract defined by  $I_r$  and satisfies its local test suite  $\mathcal{T}_r$ . In this case, the dependency set is empty, i.e.,  $\mathcal{D}_r = \emptyset$ . The agent is prompted to generate code  $c_r$  such that  $c_r \models \mathcal{T}_r$ .

*Implementation of Non-Leaf Nodes.* For a non-leaf node  $r$ , the implementation begins only after all its children  $v \in \text{Children}(r)$  have been processed. At this stage, ARC must not only implement the interfaces  $I_r$  directly associated with the current node but also correctly invoke the realized interfaces generated by its children. The dependency interfaces are defined as  $\mathcal{D}_r = \bigcup_{v \in \text{Children}(r)} I_v$ . The implementation  $c_r$  is thus conditioned on both the target interfaces  $I_r$  and the implemented child interfaces in  $\mathcal{D}_r$ . This ensures structural alignment between the requirement hierarchy and the software's internal call structure.

*The Reactive Verification Loop.* The implementation phase is essentially a test-driven development (TDD) loop that functions as a grounding gate for agentic output. As shown in Algorithm 3,

---

**Algorithm 3** Phase 2: Generate Implementation (Bottom-Up)

---

**Require:** Interfaces  $\mathcal{I}_r$ , Test Suite  $\mathcal{T}_r$ , Call Graph  $\mathcal{E}_{call}$ , Traceability Record  $\mathcal{M}$

**Ensure:** Code  $c_r$ , Boolean *passed*

```

1:  $b \leftarrow \text{MaxBudget}$ 
2:  $passed \leftarrow \text{False}$ 
3:  $\mathcal{D}_r \leftarrow \bigcup_{i_r \in \mathcal{I}_r} \{i_v \mid (i_r, i_v) \in \mathcal{E}_{call}\}$  ▷ Retrieve realized interfaces from children
4:  $Feedback \leftarrow \emptyset$ 
5: while  $b > 0 \wedge \neg passed$  do
6: ▷ Generate candidate code conditioned on target
interfaces, child dependencies, and feedback
7:    $c_r \leftarrow \text{GenCode}(\mathcal{I}_r, \mathcal{D}_r, \mathcal{M}, Feedback)$ 
8: ▷ Verify implementation against the node's test
suite
9:    $passed, Feedback \leftarrow \text{ExecuteTest}(c_r, \mathcal{T}_r)$ 
10:   $b \leftarrow b - 1$ 
11: end while
12: return  $\langle c_r, passed \rangle$ 

```

---

the generation process is constrained by the local test suite  $\mathcal{T}_r$  and a maximum iteration budget  $b$ , i.e.,  $c_r \models \mathcal{T}_r$  and  $\text{Attempts} \leq b$ . In each iteration, ARC invokes *GenCode* to produce a candidate implementation  $c_r$  based on the target interfaces  $\mathcal{I}_r$ , the dependency interfaces  $\mathcal{D}_r$  from children, and the current traceability context  $\mathcal{M}$ . The candidate is immediately verified by *ExecuteTest*. If the tests fail, the resulting *Feedback* (comprising compiler error logs or test assertion failures) is fed back into the next generation cycle as a refinement prompt. To ensure compilation efficiency and prevent infinite optimization loops for highly complex scenarios, we limit the number of attempts to the budget  $b$ . If the budget is exhausted before passing all tests, the framework retains the last version of  $c_r$  to maintain the workflow. This iterative mechanism ensures that the generated code is not only architecturally compliant but also functionally verified against the original requirement expectations.

More details about the agent's prompt are provided on our project website [54].

## 5 Evaluation

We design our experiments to answer the following research questions:

- **RQ1: Effectiveness.** Can ARC correctly implement web systems of different scales from complex multi-modal requirements, in comparison to state-of-the-art approaches?
- **RQ2: Overhead Analysis.** What is the overhead introduced by ARC, measured in token consumption and development time?
- **RQ3: Ablation Study.** How much does each individual component (DFS search strategy, test suite  $\mathcal{T}$ , traceability record  $\mathcal{M}$ ) contribute to the overall performance of ARC?

### 5.1 Benchmark

We evaluate ARC on two complementary benchmarks, i.e., (1) a depth-oriented benchmark consisting of six complex web systems, each specified by hundreds of requirement scenarios, and (2) a breadth-oriented benchmark, AppForge [57], consisting of 101 Android apps, each involving fewer than 10 scenarios. The former stresses repository-scale synthesis under deeply hierarchical, inter-dependent requirements, whereas the latter emphasizes generalization across a large number of smaller and more diverse app-generation tasks.

**5.1.1 Depth-oriented Web Systems.** We constructed a benchmark that consists of a set of systems  $\mathcal{S} = \{\mathcal{S}_1, \dots, \mathcal{S}_n\}$ . For each system  $\mathcal{S}_i$ , we prepared (1) a multi-modal requirements document  $\mathcal{G}_r^i$  that

Table 1. Overview of web systems in the benchmark.

| Scale  | Web System     | #REQ | #TEST | Link                                                              | Type                          |
|--------|----------------|------|-------|-------------------------------------------------------------------|-------------------------------|
| Small  | BookStack      | 32   | 30    | <a href="https://demo.bookstackapp.com">demo.bookstackapp.com</a> | Knowledge Base                |
|        | Keep           | 35   | 45    | <a href="https://keep.google.com">keep.google.com</a>             | Interactive Productivity Tool |
| Medium | Stack Overflow | 66   | 60    | <a href="https://stackoverflow.com">stackoverflow.com</a>         | Community Q&A Platform        |
|        | PrestaShop     | 86   | 75    | <a href="https://demo.prestashop.com">demo.prestashop.com</a>     | E-commerce Platform           |
| Large  | 12306          | 142  | 80    | <a href="https://12306.cn">12306.cn</a>                           | Public Ticketing System       |
|        | Ctrip          | 152  | 80    | <a href="https://ctrip.com">ctrip.com</a>                         | Service Aggregation Platform  |

specifies the intended functionalities, and (2) a GUI test suite  $\mathcal{T}^i$  used to validate the correctness of the implemented system  $\mathcal{S}_i$ .

*Selection of Target Systems.* In this study, we selected six benchmark web systems based on two criteria: (1) *Maturity*, the system shall be in operation for at least 10 years; and (2) *Popularity*, the system shall have either at least 100 million users or be adopted by companies among the top 100 by revenue. For each system, we evaluated a subset of its visible and functional features (see Table 1 and our project website [54] for details).

- (1) **BookStack.** [10] A wiki platform supporting hierarchical content organization and collaborative CRUD operations, representing content-heavy knowledge-management systems.
- (2) **Keep.** [36] A note-taking system supporting multimodal inputs and synchronized user interactions, representing responsive user-centric applications.
- (3) **Stack Overflow.** [62] A community Q&A platform supporting search, voting, commenting, authentication, and reputation management, representing socially interactive multi-user systems.
- (4) **PrestaShop.** [52] An e-commerce platform supporting shopping carts, product catalogs, payment processing, and order management, representing transaction-heavy applications.
- (5) **12306.** [1] A railway ticket booking system supporting route, schedule, and seat booking workflows, representing large-scale high-concurrency transactional systems.
- (6) **Ctrip.** [15] A travel booking platform integrating flights, hotels, and train services, representing heterogeneous multi-service applications.

*Construction of Requirement Documents.* For each selected system  $\mathcal{S}_i$ , we manually constructed its corresponding requirement graph  $\mathcal{G}_r^i$ . We recruited six annotators (graduate students with computer science backgrounds) to construct the DSL-based requirement documents. Instead of manually writing the DSL format, we provided them with a development requirement modeling tool (available on our project website [54]) to visualize hierarchical requirement nodes and scenarios. We also provided detailed guidelines and examples for the quality of the documents (see [54] for details). For each system, one annotator spent two weeks on exploring the system and writing the DSL-based requirement documents. All documents were then reviewed in group meetings and further refined based on feedback from another two annotators. The full construction process took approximately from 1200 man-hours.

*Construction of Test Suites* We evaluate functionalities of generated systems by constructing end-to-end GUI tests for each web system. The annotators were instructed to create one GUI test case for each leaf-node scenario in the requirement documents. Each test case was *manually* constructed, independent of using ARC, according to the actual function of real-world web system.

**5.1.2 Breadth-oriented App Systems.** To complement our depth-oriented web benchmark, we additionally evaluate ARC on AppForge [57], a breadth-oriented benchmark consisting of 101 Android

app development tasks derived from real-world applications. In contrast to our primary benchmark, which emphasizes repository-scale synthesis under deeply hierarchical and interdependent requirements, AppForge emphasizes cross-app diversity: each app is relatively small, typically involving fewer than 10 features, while collectively covering a broad spectrum of app functionalities and implementation challenges.

To adapt AppForge to the input format of ARC, we first convert each natural-language app specification into our DSL by mapping the described functionalities into requirement nodes and their relations. This conversion is structural rather than semantic, i.e., it preserves the original content while making the requirement organization explicit for ARC. However, we observe that some AppForge specifications do not fully enumerate all functions, interaction scenarios, or intermediate execution constraints needed for test-driven generation. We therefore further perform *requirement enhancement* by supplementing missing intermediate scenarios and finer-grained execution constraints. As a result, for each AppForge app, we prepare two versions of the requirement document, i.e., the *originally described* version obtained by structural conversion only, and the *fully described* version after requirement enhancement. The fully described documents are used in RQ1 to evaluate effectiveness under more complete requirements, while both versions are used in RQ3 to study how requirement completeness affects the performance of ARC.

## 5.2 Baselines

We selected four state-of-the-art baselines. The baselines are described below, along with details of how each was used to generate web systems in our benchmark.

- (1) **MetaGPT** [29]: a multi-agent repository-level CLI framework that generates code from high-level requirements via role-based agent coordination.
- (2) **OpenHands** [68]: a multi-agent repository-level IDE-based framework that performs code generation through iterative planning, execution, and feedback-driven refinement.
- (3) **Cursor** [16]: a single-agent repository-level IDE framework operated in agent mode, supporting interactive, multi-step code generation and editing with repository-wide context.
- (4) **Copilot** [14]: a single-agent repository-level IDE framework operated in agent mode, performing localized code generation and refinement at the line or function level.

For Cursor and Copilot, we initialized either a Web/Android project template and the requirement documents. Then we opened a new chat session within IDE to prompt them with the requirement documents. For OpenHands, we used the OpenHands GUI to upload the project template and requirement document in a new session, and prompted the system to generate the corresponding implementation. For MetaGPT, we provided the requirement document as input to its official Docker implementation, which then generated artifacts via its built-in four-agent configuration. Stepwise setup documentation is available on our project website [54].

## 5.3 Evaluation Metrics

Following prior benchmark studies [24, 34], we evaluated all methods using **Test Pass Rate (%)**. For each benchmark system, the provided test suite  $\mathcal{T}^i$  served as the ground-truth execution oracle (see Section 5.1.1), where  $\mathcal{T}$  denotes the set of all test cases and  $\mathcal{T}_{pass} \subseteq \mathcal{T}$  the passed test cases.

$$\text{Test Pass Rate (\%)} = \frac{|\mathcal{T}_{pass}|}{|\mathcal{T}|} \times 100$$

A test was considered *passed* if and only if all its test steps executed successfully on the system under test and all assertions were satisfied. Since each leaf node in the requirement tree was mapped to one test, every test corresponded to a concrete function or scenario to be implemented.

Table 2. GUI test case pass rates (%) over 3 independent trials (mean  $\pm$  std) indicating the functional correctness of implemented systems. ( $\uparrow x$ ) denotes absolute improvement over the best baseline.

| Benchmark System | Test Pass Rate (%)                                            |                 |                 |                 |                 |
|------------------|---------------------------------------------------------------|-----------------|-----------------|-----------------|-----------------|
|                  | Ours                                                          | MetaGPT         | OpenHands       | Cursor          | Copilot         |
| BookStack        | <b>91.1 <math>\pm</math> 5.1 (<math>\uparrow 14.4</math>)</b> | 76.7 $\pm$ 10.0 | 55.6 $\pm$ 24.6 | 54.4 $\pm$ 12.6 | 52.2 $\pm$ 15.0 |
| Keep             | <b>91.9 <math>\pm</math> 5.6 (<math>\uparrow 22.2</math>)</b> | 55.6 $\pm$ 13.3 | 40.0 $\pm$ 21.2 | 40.7 $\pm$ 26.9 | 69.6 $\pm$ 4.6  |
| Stack Overflow   | <b>86.7 <math>\pm</math> 6.0 (<math>\uparrow 33.3</math>)</b> | 53.3 $\pm$ 6.7  | 37.2 $\pm$ 8.6  | 35.0 $\pm$ 15.5 | 36.1 $\pm$ 6.7  |
| PrestaShop       | <b>85.8 <math>\pm</math> 1.5 (<math>\uparrow 17.8</math>)</b> | 48.0 $\pm$ 6.7  | 20.4 $\pm$ 12.6 | 68.0 $\pm$ 9.3  | 34.7 $\pm$ 6.7  |
| 12306            | <b>78.8 <math>\pm</math> 5.7 (<math>\uparrow 28.0</math>)</b> | 41.3 $\pm$ 7.5  | 31.7 $\pm$ 11.6 | 50.8 $\pm$ 10.6 | 45.0 $\pm$ 3.3  |
| Ctrip            | <b>75.4 <math>\pm</math> 7.6 (<math>\uparrow 39.2</math>)</b> | 18.8 $\pm$ 6.3  | 15.0 $\pm$ 5.7  | 36.3 $\pm$ 13.0 | 23.8 $\pm$ 11.9 |

We adopted this metric for repository-scale evaluation because it allows us to precisely evaluate how many required functions are successfully generated.

Unlike web systems where the system is still runnable (i.e., testable) regardless of partial compilation errors in some modules. The generated Android APKs need to be compilable before running any tests. Therefore, we use the following metrics for the evaluation. Let  $\mathcal{S}$  be the set of all Android apps,  $\mathcal{S}_{comp} \subseteq \mathcal{S}$  be the successfully compiled apps, and  $\mathcal{S}_{suite} \subseteq \mathcal{S}$  be the apps whose entire test suite passes (i.e., all test cases pass).

$$\text{Compile Succ. (\%)} = \frac{|\mathcal{S}_{comp}|}{|\mathcal{S}|} \times 100 \quad \text{Test Suite Pass (\%)} = \frac{|\mathcal{S}_{suite}|}{|\mathcal{S}|} \times 100$$

In addition, we further evaluate the per-app test case pass rate to capture partial functional completion. Let  $\mathcal{T}^s$  and  $\mathcal{T}_{pass}^s$  denote the test cases and passed test cases for app  $s$ , respectively.

$$\text{Test Case Pass (\%)} = \frac{1}{|\mathcal{S}_{comp}|} \sum_{s \in \mathcal{S}_{comp}} \frac{|\mathcal{T}_{pass}^s|}{|\mathcal{T}^s|} \times 100 \quad \text{Final Pass (\%)} = \frac{1}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} \frac{|\mathcal{T}_{pass}^s|}{|\mathcal{T}^s|} \times 100$$

Note that Final Pass (%) = Compile Succ.(%)  $\times$  Test Case Pass.(%)

## 5.4 Experimental Setup

ARC and all baselines used Google Gemini 3 Pro [26] with temperature 1.0, top- $p$  0.95, top- $k$  64, and *thinking level* as high. We pre-define the project template for Web application by React (frontend), Node.js (backend), SQLite3 (database), Vitest (unit and integration testing), and Playwright (end-to-end GUI testing). We pre-define the project template for Android application by Java, Gradle (build system), and Robolectric (unit, integration, and end-to-end testing), with JUnit as the test execution framework. Considering the non-determinism of LLMs, each experiment was run for 3 trials, and we reported the mean and standard deviation of the test pass rates.

## 5.5 RQ1: Effectiveness

Table 2 reports the results where ARC consistently outperforms all four baselines regarding all the measurement with the test-driven design. ARC outperforms the state-of-the-art baseline by 14.4–39.2% regarding the pass rates where ARC improves the test passing rate by on average 54.9%. In addition, we observe that the pass rates decrease as the system complexity increases. Nevertheless, ARC demonstrates stronger robustness comparing to the baselines. For example, the performance of OpenHands drops from 55.6% (BookStack) to 15.0% (Ctrip). In contrast, ARC still keeps its performance as 75.4% in generating the Ctrip project. Across all trials, ARC exhibits consistently low variance (std  $\leq$  7.6%), while baselines show substantially higher variance (e.g., Cursor std 26.9% on Keep, OpenHands 24.6% on BookStack), confirming that the test-driven design improves both the

Table 3. Compile success rate, test suite pass rate, test case pass rate, and final pass rate (%) over 3 independent trials (mean  $\pm$  std) on AppForge.

| Methods     | Compile Succ.<br>(%)              | Test Suite Pass<br>(%)           | Test Case Pass<br>(%)            | Final Pass<br>(%)                |
|-------------|-----------------------------------|----------------------------------|----------------------------------|----------------------------------|
| <b>Ours</b> | <b>100.0 <math>\pm</math> 0.0</b> | <b>31.2 <math>\pm</math> 0.5</b> | <b>68.3 <math>\pm</math> 1.4</b> | <b>68.3 <math>\pm</math> 1.4</b> |
| MetaGPT     | 23.3 $\pm$ 5.8                    | 1.6 $\pm$ 0.7                    | 13.9 $\pm$ 2.8                   | 3.2 $\pm$ 1.0                    |
| OpenHands   | 32.7 $\pm$ 9.3                    | 3.6 $\pm$ 1.2                    | 23.3 $\pm$ 5.4                   | 8.0 $\pm$ 3.6                    |
| Cursor      | 59.9 $\pm$ 2.2                    | 10.3 $\pm$ 0.4                   | 41.9 $\pm$ 1.0                   | 25.1 $\pm$ 0.9                   |
| Copilot     | 85.1 $\pm$ 1.6                    | 12.3 $\pm$ 2.1                   | 36.2 $\pm$ 1.0                   | 30.8 $\pm$ 2.1                   |

Table 4. Average execution time ("Time") and token consumption ("Cost") on AppForge.

| Methods     | Time<br>(min) | Cost<br>(M) |
|-------------|---------------|-------------|
| <b>Ours</b> | 83.18         | 3.07        |
| MetaGPT     | 23.50         | 0.30        |
| OpenHands   | 7.21          | 1.01        |
| Cursor      | 17.13         | 1.14        |
| Copilot     | 18.12         | 1.40        |

effectiveness and its performance stability. Table 3 reports the results on AppForge where ARC consistently outperforms all baselines, achieving 100% compile success, 31.2% test suite pass rate, and 68.3% test case and final pass rates. This demonstrates that the ARC framework also exhibits strong generalizability over diverse Android apps.

*Why can ARC outperform the baselines?* First, we observe that baselines often compress requirements into unstructured summaries, the resultant context is usually long and noisy, potentially leading to hallucination by missing or mistaking the crucial implementation. In contrast, ARC interprets each feature (i.e., requirement node) into a set of verifiable tests, providing a more reliable signal for LLM to produce code. Second, existing frameworks and AI IDEs (e.g., Cursor and Copilot) are likely to repetitively modify implemented code, even if their features are independent in the requirement documents. In contrast, ARC avoids these issues by parsing the requirements with the designed DSL where agents can strictly process one feature after another, isolating their implementation with minimal interference. As a result, ARC enforces orthogonality and non-regression in the process of code generation.

*When can ARC fail to pass the GUI tests?* By investigating the GUI tests that ARC failed to pass, we identify two main causes. First, implicit requirement scenarios may be missing from the documents. Since ARC treats the requirement graph as the source of truth, it may miss implicit constraints; for example, our 12306 document did not explicitly specify that user IDs cannot be duplicated during registration, causing failures in related test cases. Second, test feedback may provide insufficient guidance within the given budget. For instance, in PrestaShop's "Recently Viewed" product feature, the test only reports that "viewed product A is not shown," which gives limited information for the LLM to infer where and how to fix the problem.

Interested readers can find more successful and failing cases of ARC on our website [54].

## 5.6 RQ2: Overhead Analysis

**5.6.1 Experiment Design.** As described in Section 5.1 and 5.4, we measured the overhead over two dimensions: the end-to-end runtime (in minutes), denoted as **Time (min)**, and the token consumption (in millions of tokens), denoted as **Cost (M)**.

**5.6.2 Results.** Table 5 shows our results. ARC requires an average of 3 hours more than the state-of-the-art baseline, and consumes 6–21 million additional tokens. This additional cost is spent on (1) incrementally planning and implementing interfaces, and (2) rigorous feedback through test execution. As systems scale (from BookStack to Ctrip), we see that ARC's overhead increases accordingly. In contrast, the overhead of baselines does not increase. Our investigation shows that, while taking less overhead, the state-of-the-art solutions usually feed the requirement documents to LLM as a whole context, which can easily miss crucial information to generate code especially

Table 5. Execution time in minutes ("Time") and token consumption in millions ("Cost") of our framework and baseline methods for implementing web systems from requirements.

| Web System     | Ours          |             | MetaGPT       |             | OpenHands     |             | Cursor        |             | Copilot       |             |
|----------------|---------------|-------------|---------------|-------------|---------------|-------------|---------------|-------------|---------------|-------------|
|                | Time<br>(min) | Cost<br>(M) | Time<br>(min) | Cost<br>(M) | Time<br>(min) | Cost<br>(M) | Time<br>(min) | Cost<br>(M) | Time<br>(min) | Cost<br>(M) |
| BookStack      | 145.98        | 11.80       | 48.90         | 0.98        | 22.00         | 5.69        | 29.16         | 3.82        | 13.28         | 2.46        |
| Keep           | 161.04        | 17.50       | 37.83         | 0.38        | 22.35         | 6.70        | 13.58         | 1.84        | 22.78         | 3.95        |
| Stack Overflow | 231.52        | 21.38       | 42.38         | 0.67        | 52.00         | 14.43       | 16.48         | 3.47        | 22.50         | 5.14        |
| PrestaShop     | 253.95        | 22.76       | 42.98         | 0.89        | 34.72         | 3.94        | 19.73         | 4.45        | 14.12         | 2.85        |
| 12306          | 351.52        | 27.02       | 53.72         | 1.23        | 38.73         | 16.29       | 39.11         | 7.26        | 17.40         | 3.81        |
| Ctrip          | 363.27        | 26.38       | 35.00         | 0.90        | 26.68         | 5.24        | 16.21         | 2.74        | 18.51         | 3.35        |

when the document is complicated. Despite that we imposed no upper limit on tokens or runtime for any method, we observe that baselines frequently failed to fully utilize resources, which is a known limitation of LLMs in agentic settings [17, 18]. The similar results are consistent in AppForge. ARC takes more overhead while generating far more accurate code in return.

We further analyzed execution logs of ARC to estimate its average time and token consumption in each phase. As for time consumption, the code generation phase accounts for 84.3%, the test generation phase for 14.8%, and design generation phase for 0.9%. As for token consumption, the code generation phase dominates at 62% (including feedback iterations), followed by test generation phase at 36%, and design generation phase accounting for 2%. The gap between time and token consumption lies in that the code generation phase involves substantial non-LLM overhead (e.g., compilation and test execution), whereas test and design generation are dominated by the interaction with LLM.

## 5.7 RQ3: Ablation Study

**5.7.1 Experiment Design.** In this experiment, we adopted the settings described in Section 5.1 and 5.4. On the web benchmark, we compared three ablated variants of ARC: (1) *w.o. DFS strategy*, where ARC designs and implements requirement nodes in an arbitrary order; (2) *w.o. Test*, where no test suites  $\mathcal{T}$  are generated for interfaces during the top-down phase; and (3) *w.o. Traceability*, where code provenance  $\mathcal{M}$  is not recorded during the bottom-up phase. On the Android app benchmark, AppForge, we compared the performance on fully and partially described requirements (see Section 5.1). We used the same performance metrics as in RQ1 (Section 5.5).

**5.7.2 Results.** Table 6 shows our results. Without traceability, ARC spends more time gathering contextual information from the repository. Although the DFS strategy and testing contribute to increased time and cost, this is a necessary trade-off to support reliable development in large-scale systems, as evidenced by the higher pass rate. These results validate our claims in RQ1 and RQ2. On AppForge, ARC has better performance on the fully described documents. Test suite pass rate increases from 18.3% (partially described) to 31.7% (fully described). The improvement also applies in the test case pass rate and the final pass rate. Note that, the performance on partially described documents still outperforms the state-of-the-art baselines.

## 5.8 User Study

We conducted a study to evaluate how users can adopt ARC in practice to compile a web system by only writing DSL-based multi-modal requirements.

**Subject System.** In this study, we selected the system of 12306, a popular railway ticket booking system. We chose this system because all the participants indicated that they had used this system

Table 6. Ablation study of our framework. Columns show test pass rate (“Pass”), execution time in minutes (“Time”), and token consumption in millions (“Cost”) under different component removals.

| Web System     | Ours     |            |          | w.o. DFS Strategy |            |          | w.o. Test |            |          | w.o. Traceability |            |          |
|----------------|----------|------------|----------|-------------------|------------|----------|-----------|------------|----------|-------------------|------------|----------|
|                | Pass (%) | Time (min) | Cost (M) | Pass (%)          | Time (min) | Cost (M) | Pass (%)  | Time (min) | Cost (M) | Pass (%)          | Time (min) | Cost (M) |
| BookStack      | 91.11    | 145.98     | 11.80    | 63.33             | 84.32      | 3.09     | 60.00     | 112.77     | 11.74    | 90.00             | 164.78     | 15.92    |
| Keep           | 91.85    | 161.04     | 17.50    | 75.56             | 103.75     | 13.48    | 60.00     | 136.21     | 16.68    | 86.67             | 162.94     | 22.59    |
| Stack Overflow | 86.67    | 231.52     | 21.38    | 35.00             | 117.34     | 8.42     | 56.67     | 162.37     | 14.74    | 83.33             | 215.18     | 21.24    |
| PrestaShop     | 85.78    | 253.95     | 22.76    | 53.33             | 90.12      | 8.43     | 38.67     | 167.14     | 10.05    | 74.67             | 236.24     | 22.74    |
| 12306          | 78.75    | 351.52     | 27.02    | 36.25             | 210.41     | 20.21    | 67.5      | 220.13     | 23.63    | 77.5              | 429.08     | 25.45    |
| Ctrip          | 75.42    | 363.27     | 26.38    | 33.75             | 234.47     | 17.16    | 48.75     | 236.48     | 13.99    | 71.25             | 441.02     | 31.22    |

Table 7. Ablation study of our framework on partially described and fully described requirements in AppForge. We compare ARC applied to DSL-formatted requirements under both requirement completeness settings.

|                     | Compile Succ. (%) | Test Suite Pass (%) | Test Case Pass (%) | Final Pass (%) |
|---------------------|-------------------|---------------------|--------------------|----------------|
| Partially Described | 100.0 ± 0.0       | 18.3 ± 1.0          | 41.8 ± 1.7         | 41.8 ± 1.7     |
| Fully Described     | 100.0 ± 0.0       | 31.7 ± 0.9          | 68.0 ± 0.2         | 68.0 ± 0.2     |

in person, which made it more convenient for them to write requirement documents in this study. The participants were asked to specify five modules (e.g., login module, registration module, query module, ticket-booking module, and profile update module) in the system.

**Participants.** We advertised the study to undergraduate students in a software engineering course, allowing interested students to register via an online form. This population provided a representative proxy for novice developers. We selected the participants with prior knowledge in web development, at least one year of exposure to the subject system, and at least six months of experience with AI-assisted coding tools. As a result, we recruited 21 participants in total. Given the requirement is non-trivial, we organized those participants into 7 groups, each consisting of 3 participants to accomplish the documents to generate the web system.

**Task Procedure & Metrics.** Participants were first introduced to the requirements DSL. To ensure familiarity with the DSL before the study began, we first guided them to complete a small warm-up task for practice on a web system which is different from the subject system. Each group explored the subject system and documented its functionality using the DSL, after which they used ARC to replicate the system. We asked the participants to complete the task within one day. Upon completion, we collected all replicated systems and evaluated them based on the test pass rate (%) against our ground-truth test cases. Finally, participants completed a post-study survey.

**Results.** All 7 groups successfully transformed their requirements into runnable systems with test pass rates ranging from 80% to 96% (90.3% on average), completing their projects within one day. On average, each group spent 5.6 hours on DSL-based requirement specification and system generation, with completion times ranging from 3.1 to 9.8 hours. Overall, participants found the DSL easy to learn, requiring an average of 5 rounds of requirement updates to complete their tasks. Some participants drafted requirements informally first and translated them into the DSL with LLM assistance<sup>2</sup>, a process that took under 10 minutes including error correction. The compiled software projects contained 107 requirement nodes and 1,395 lines of DSL code on average. We further observed that the precision of requirement documents led to clear differences across groups.

<sup>2</sup>The prompt used for this translation is available on our website [54].

Groups that described functional details, validation rules, and intermediate interaction scenarios more precisely generally produced systems with higher test pass rates, whereas groups with more ambiguous or incomplete requirement documents were more likely to miss expected behaviors in the generated systems. This suggests that, beyond DSL learnability, the quality and completeness of requirement specification substantially affect the final compilation outcome.

**Post-Study Survey.** 6 out of 7 groups (86%) agreed that ARC is more effective and efficient than conventional approaches such as unstructured AI-assisted coding (vibe-coding) or fully manual implementation. Participants reported that ARC supports more accurate requirement implementation with fewer manual interactions, and that the traceability records linking requirements to interfaces, tests, and code helped them understand the generated systems. The remaining group did not agree that ARC was more effective in practice, not because the final system was unusable, but because debugging the agentic pipeline remained difficult once the generated system deviated from expectation. In particular, participants reported that feedback from the compilation process was relatively slow, making it hard to quickly identify which requirement, test, or implementation step caused the mismatch. Several participants also described their current debugging strategy as using `git checkout` to roll back to earlier generated versions and compare intermediate results. While this strategy was useful in practice, they still considered it inefficient and cumbersome for iterative debugging. This feedback suggests that future versions of ARC should provide more interactive debugging support and faster feedback when generated systems fail to satisfy user expectations.

Due to space constraints, detailed metric distributions, compiled artifacts, and demonstration videos are available on our project website [54].

## 5.9 Discussion

Beyond the conventional software generation, ARC also shows promise for supporting software evolution [61]. Its core advantage lies in the explicit traceability it maintains across requirements, interfaces, tests, and implementations, which helps agents reason about how local changes may affect interacting components in an existing system. Our experimental results further suggest that this design is effective in practice. Compared with state-of-the-art baselines, ARC consistently achieves higher pass rates and exhibits stronger stability on complex, repository-scale systems, indicating that requirement grounding and test-driven decomposition are beneficial.

At the same time, the experiments also reveal important limitations of the current framework. First, ARC is primarily optimized for functional correctness, and therefore may overlook non-functional requirements such as security, performance, scalability, and reliability, which are equally important in practice. Second, ARC remains fundamentally bounded by the completeness of the requirement document and the quality of the generated tests. When key scenarios are missing or the feedback signal from tests is weak, the system may still produce partially incorrect implementations. These observations point to several directions for future work. One direction is to extend ARC from pure functional compilation to multi-objective requirement compilation, where functional requirements are treated as hard constraints and non-functional properties are optimized as soft objectives. Another is to improve the strength of the testing and feedback loop, for example by generating more complete tests, richer debugging signals, and more informative failure localization.

## 5.10 Threats to Validity

**Internal Validity.** The first threat concerns the use of graduate students to construct benchmark artifacts, which may introduce variability due to limited professional software engineering experience. To mitigate this, we provided detailed construction guidelines to standardize the drafting

process (see [54]), and two annotators independently reviewed and refined all artifacts to resolve ambiguities and ensure consistency. Nevertheless, requirements are inherently incomplete, ambiguous, and evolving [48], motivating future work on specification refinement. The second threat concerns reliance on proprietary LLMs, which may be updated at any time without disclosure, making precise versioning and full replication of results difficult. We acknowledge this as an inherent limitation, and anticipate that future work will evaluate open-source or locally hosted models. The last threat is LLM non-determinism, which we mitigated by running each approach for 3 independent trials and reporting mean and standard deviation on all results. We will extend the study with more number of trials in the future.

**External Validity.** There are two main factors that may limit the generalizability of our results. First, we selected six web systems for our benchmark. While these systems capture standard enterprise and transactional web applications, they do not fully represent the diversity of web system behaviors. For example, media-heavy streaming applications (e.g., YouTube, Discord), real-time collaboration tools (e.g., Google Docs), and single-page applications (e.g., Google Maps) exhibit complex behaviors that are difficult to automatically test and therefore challenging to systematically benchmark. Furthermore, such systems often involve ambiguity in implementation choices, such as whether to use WebSockets for real-time communication, or other architectural decisions. Addressing these complexities could be an avenue for future work. Second, ARC was evaluated using Gemini 3 Pro as the underlying model, which raises questions about transferability to other models. However, as long as the underlying model is trained on coding data, the framework remains applicable. Our approach is a general organizational and instructional framework that is similar to multi-agent or other system frameworks, which do not rely on any model-specific advantages. It is designed to leverage already capable models in a structured way.

## 6 Related Work

**Code Generation.** Early automatic-programming systems such as PROW generated executable code from formal logical specifications [9, 67]. Later heuristic and probabilistic methods include grammar-based generation [4, 72] and specialized statistical models [19, 32], but are often rigid and hard to scale. Transformer-based models support longer-context reasoning and more complex program generation, including Codex [13], MBPP [7], APPS [28], and competitive-programming systems such as AlphaCode [39]. Stronger foundation models such as ChatGPT and Deepseek [2, 27] enable semantic code understanding for IDE suggestions such as Cursor and Copilot [14, 16, 40], and real-world repair on SWE-bench [34, 74], with Trae Agent reaching near 80% [64]. These works motivate the shift from function-level code generation to repository-level generation [31, 43, 63].

**Repository Generation.** Repository-level generation is difficult because long-context business logic spans many files. Existing work follows three axes: (1) *Interactive feedback*: multi-round planning with external feedback, such as SDE-I [77], which uses static analysis and unit tests for library generation but requires reliable pre-existing tests, and CodePlan [8], Cursor [16], and Copilot [14] Plan Mode, which derive repository-aware plans before code generation; (2) *Memory augmentation*: RepoCoder [76] and CodeChain [37] retrieve code through dependency graphs, code-search methods align natural-language queries with code fragments [44], and RepoGraph [49] models code-line-level repository structure; and (3) *Task delegation*: multi-agent systems such as OpenHands [68] and HyperAgent [51] distribute planning, navigation, and editing across agents, with communication and context-management overhead [70]. Effective for incremental generation over existing repositories, these approaches lack explicit system-level requirement modeling; new-feature changes can therefore affect unrelated components or invalidate passing tests. In contrast, ARC tracks requirement-implementation alignment throughout the development lifecycle.

**End-to-end SE Agents.** Extending repository generation, *waterfall frameworks* model development as a linear pipeline, including Self-Collaboration [21], ChatDev [56], and MetaGPT [29], whereas *agile frameworks* such as AgileCoder [45] use iterative sprints. Existing agents still lack enforced requirement alignment. ARC treats such alignment as a first-class objective, keeping new implementations consistent with system requirements without regressing prior behavior.

**Test Generation.** Test generation is central to software reliability and spans three complementary classes: (1) *Program-analysis-based approaches*: symbolic execution [12, 25, 58], including Klee [12] and DART [25], and search-based techniques [6, 23, 50], including EvoSuite [23] and Randoop [50], systematically explore program behaviors but often require precise models or extensive search; (2) *Hybrid approaches*: program analysis with learning-based guidance improves exploration and prioritizes semantically relevant executions [11, 38], yet remains implementation-centric; and (3) *LLM-based approaches*: recent work frames test generation as code synthesis from focal methods or natural language descriptions [3, 20, 35, 47, 60, 66, 69, 75]. Requirement-intention-driven testing generates project-specific tests from explicit validation objectives, preconditions, and expected results [55]. Recent issue-localization work also treats test suites as executable proxies for requirements [41]. Although these methods capture semantics, they typically operate at the unit level and generate tests independently of system-wide specifications. Test generation is *complementary* to ARC: rather than proposing a new test generator, ARC treats tests as first-class artifacts derived from structured requirements and uses them as correctness gates in repository-level compilation.

## 7 Conclusion

Repository-scale development is challenging because maintaining correctness and managing dependencies in large, evolving systems is difficult. We frame this as a requirements compilation problem, translating multi-modal specifications into runnable web systems. We propose ARC, which operates in two phases. The top-down phase designs formalized UI, API, and DB interfaces, with dependencies enforced by initially failing tests. The bottom-up phase implements and validates these interfaces. On a depth-oriented benchmark of six web systems, ARC outperforms state-of-the-art approaches by up to 108.1% in test pass rates; on the breadth-oriented AppForge benchmark of 101 Android apps, ARC achieves a 68.3% test case pass rate on average, outperforming all baselines. A user study further confirms its effectiveness and usability.

## Acknowledgments

This research is supported in part by the National Natural Science Foundation of China (62572300), the Minister of Education, Singapore (MOE-T2EP20124-0017, MOET32020-0004), the National Research Foundation, Singapore and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN), DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG2-GC-2023-008-1B), and Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme and CyberSG R&D Cyber Research Programme Office, and partially by HUAWEI's AI Hundred Schools Program using the Ascend AI technology stack. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, Cyber Security Agency of Singapore as well as CyberSG R&D Programme Office, Singapore.

## Data Availability

The datasets and code used in this study are available on our public repository [53]. Supplementary materials are available on our project website [54], and an interactive online version of ARC is available at [arc-bench.com](https://arc-bench.com).

## References

- [1] 12306 China Railway 2011. <https://www.12306.cn/en/index.html>
- [2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altmenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).
- [3] Toufique Ahmed, Jatin Ganhotra, Rangeet Pan, Avraham Shinnar, Saurabh Sinha, and Martin Hirzel. 2025. Otter: generating tests from issues to validate SWE patches. In *Proceedings of the 42nd International Conference on Machine Learning* (Vancouver, Canada) (ICML '25). JMLR.org, Article 30, 20 pages. doi:10.48550/arXiv.2502.05368
- [4] Miltiadis Allamanis and Charles Sutton. 2014. Mining idioms from source code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) (FSE 2014). Association for Computing Machinery, New York, NY, USA, 472–483. doi:10.1145/2635868.2635901
- [5] ARC-Bench Online Demonstration 2026. <http://arc-bench.com>
- [6] Andrea Arcuri and Xin Yao. 2008. Search based software testing of object-oriented containers. *Inf. Sci.* 178, 15 (Aug. 2008), 3075–3095. doi:10.1016/j.ins.2007.11.024
- [7] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732 [cs.PL] <https://arxiv.org/abs/2108.07732>
- [8] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D. C., Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. 2024. CodePlan: Repository-Level Coding using LLMs and Planning. *Proc. ACM Softw. Eng.* 1, FSE, Article 31 (July 2024), 24 pages. doi:10.1145/3643757
- [9] R. Balzer. 1985. A 15 Year Perspective on Automatic Programming. *IEEE Transactions on Software Engineering* SE-11, 11 (1985), 1257–1268. doi:10.1109/TSE.1985.231877
- [10] BookStack 2015. <https://www.bookstackapp.com/>
- [11] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. 2017. Combining symbolic execution and search-based testing for programs with complex heap inputs. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) (ISSTA 2017). Association for Computing Machinery, New York, NY, USA, 90–101. doi:10.1145/3092703.3092715
- [12] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, Vol. 8. 209–224.
- [13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri 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, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] <https://arxiv.org/abs/2107.03374>
- [14] Copilot 2023. <https://github.com/features/copilot>
- [15] Ctrip 2017. <https://trip.com/>
- [16] Cursor 2023. <https://cursor.com/>
- [17] Cursor. 2026. Expanding our long-running agents research preview. <https://cursor.com/blog/long-running-agents>. Accessed: 2026.
- [18] Cursor. 2026. Scaling long-running autonomous coding. <https://cursor.com/blog/scaling-agents>. Accessed: 2026.
- [19] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340. doi:10.1007/978-3-540-78800-3\_24
- [20] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K. Lahiri. 2022. TOGA: a neural method for test oracle generation. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) (ICSE '22). Association for Computing Machinery, New York, NY, USA, 2130–2141. doi:10.1145/3510003.3510141
- [21] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2024. Self-Collaboration Code Generation via ChatGPT. *ACM Trans. Softw. Eng. Methodol.* 33, 7, Article 189 (Sept. 2024), 38 pages. doi:10.1145/3672459
- [22] Kevin Forsberg and Harold Mooz. 1991. The Relationship of System Engineering to the Project Cycle. *IN-COSE International Symposium* 1, 1 (1991), 57–65. arXiv:<https://incose.onlinelibrary.wiley.com/doi/pdf/10.1002/j.2334-5837.1991.tb01484.x> doi:10.1002/j.2334-5837.1991.tb01484.x
- [23] Gordon Fraser and Andrea Arcuri. 2011. 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*. 416–419. doi:10.1145/2025113.2025179

- [24] Lingyue Fu, Bolun Zhang, Hao Guan, Yaoming Zhu, Lin Qiu, Weiwen Liu, Xuezhi Cao, Xunliang Cai, Weinan Zhang, and Yong Yu. 2026. Automatically Benchmarking LLM Code Agents through Agent-driven Annotation and Evaluation. In *Proceedings of the 25th International Conference on Autonomous Agents and Multiagent Systems* (Paphos, Cyprus) (AAMAS '26). International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 874–882. doi:10.65109/HJFB4234
- [25] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 213–223. doi:10.1145/1065010.1065036
- [26] Google Gemini Pro 3 2025. <https://ai.google.dev/gemini-api/docs/models/gemini-3-pro-preview>
- [27] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025). doi:10.1038/s41586-025-09422-z
- [28] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring Coding Challenge Competence With APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, J. Vanschoren and S. Yeung (Eds.), Vol. 1. [https://datasets-benchmarks-proceedings.neurips.cc/paper\\_files/paper/2021/file/c24cd76e1ce41366a4bbe8a49b02a028-Paper-round2.pdf](https://datasets-benchmarks-proceedings.neurips.cc/paper_files/paper/2021/file/c24cd76e1ce41366a4bbe8a49b02a028-Paper-round2.pdf)
- [29] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiwu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. 2024. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*. doi:10.48550/arXiv.2308.00352
- [30] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.* 33, 8, Article 220 (Dec. 2024), 79 pages. doi:10.1145/3695988
- [31] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. In *The Thirteenth International Conference on Learning Representations*. <https://openreview.net/forum?id=chfJJYC3IL>
- [32] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 215–224. doi:10.1145/1806799.1806833
- [33] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2026. A Survey on Large Language Models for Code Generation. *ACM Trans. Softw. Eng. Methodol.* 35, 2, Article 58 (Jan. 2026), 72 pages. doi:10.1145/3747588
- [34] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770* (2023). doi:10.48550/arXiv.2310.06770
- [35] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models are Few-Shot Testers: Exploring LLM-Based General Bug Reproduction. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) (ICSE '23). IEEE Press, 2312–2323. doi:10.1109/ICSE48619.2023.00194
- [36] Keep 2013. <https://keep.google.com/>
- [37] Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. 2024. CodeChain: Towards Modular Code Generation Through Chain of Self-revisions with Representative Sub-modules. In *The Twelfth International Conference on Learning Representations*. <https://openreview.net/forum?id=vYhglxSj8j>
- [38] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-Trained Large Language Models. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) (ICSE '23). IEEE Press, 919–931. doi:10.1109/ICSE48619.2023.00085
- [39] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097. arXiv:<https://www.science.org/doi/pdf/10.1126/science.abq1158> doi:10.1126/science.abq1158
- [40] Chenyan Liu, Yufan Cai, Yun Lin, Yuhuan Huang, Yunrui Pei, Bo Jiang, Ping Yang, Jin Song Dong, and Hong Mei. 2024. Coedpilot: Recommending code edits with learned prior edit relevance, project-wise awareness, and interactive nature. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 466–478. doi:10.1145/3650212.3652142

- [41] Jiawei Liu, Yun Lin, Chenyan Liu, Yu Qian, Yiming Liu, Jiaxin Chang, Weinan Zhang, and Linpeng Huang. 2026. IssueExec: A Test-Driven Approach for Localizing Software Engineering Issues. *arXiv preprint arXiv:2607.17286* (2026). doi:10.48550/arXiv.2607.17286
- [42] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173. doi:10.1162/tacl\_a\_00638
- [43] Tianyang Liu, Canwen Xu, and Julian McAuley. 2024. RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems. In *The Twelfth International Conference on Learning Representations*. <https://openreview.net/forum?id=pPjZlOuQuF>
- [44] Yiming Liu, Ruofan Liu, Yun Lin, Zicong Zhang, Weiyu Kong, Pengnian Qi, Xiao Cheng, Weinan Zhang, Qianxiang Wang, and Linpeng Huang. 2026. XSearch: Explainable Code Search via Concept-to-Code Alignment. *arXiv preprint arXiv:2605.16046* (2026). arXiv:2605.16046 [cs.SE] doi:10.48550/arXiv.2605.16046 Accepted to ISSTA 2026.
- [45] Minh Huynh Nguyen, Thang Phan Chau, Phong X. Nguyen, and Nghi D. Q. Bui. 2025. AgileCoder: Dynamic Collaborative Agents for Software Development based on Agile Methodology. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*. 156–167. doi:10.1109/Forge66646.2025.00026
- [46] Nhan Nguyen and Sarah Nadi. 2022. An empirical evaluation of GitHub copilot’s code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories (Pittsburgh, Pennsylvania) (MSR ’22)*. Association for Computing Machinery, New York, NY, USA, 1–5. doi:10.1145/3524842.3528470
- [47] Pengyu Nie, Rahul Banerjee, Junyi Jessie Li, Raymond J. Mooney, and Milos Gligoric. 2023. Learning Deep Semantics for Test Completion. In *Proceedings of the 45th International Conference on Software Engineering (Melbourne, Victoria, Australia) (ICSE ’23)*. IEEE Press, 2111–2123. doi:10.1109/ICSE48619.2023.00178
- [48] Bashar Nuseibeh and Steve Easterbrook. 2000. Requirements engineering: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering (Limerick, Ireland) (ICSE ’00)*. Association for Computing Machinery, New York, NY, USA, 35–46. doi:10.1145/336512.336523
- [49] Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. 2025. RepoGraph: Enhancing AI Software Engineering with Repository-level Code Graph. In *The Thirteenth International Conference on Learning Representations*. <https://openreview.net/forum?id=dw9VUsHGB>
- [50] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816. doi:10.1145/1297846.1297902
- [51] Huy Nhat Phan, Phong X Nguyen, and Nghi DQ Bui. 2024. HyperAgent: Generalist Software Engineering Agents to Solve Coding Tasks at Scale. *arXiv preprint arXiv:2406.11912* (2024). doi:10.48550/arXiv.2409.16299
- [52] PrestaShop 2007. <https://prestashop.com/>
- [53] Project Repository 2026. <https://github.com/code-philias/ARC>
- [54] Project Website 2026. <https://sites.google.com/view/agentic-req-compilation/home>
- [55] Binhang Qi, Yun Lin, Xinyi Weng, Yuhuan Huang, Chenyan Liu, Hailong Sun, Zhi Jin, and Jin Song Dong. 2025. Generating Project-Specific Test Cases with Requirement Validation Intention. *arXiv preprint arXiv:2507.20619* (2025). arXiv:2507.20619 [cs.SE] doi:10.48550/arXiv.2507.20619 Accepted at ISSTA 2026.
- [56] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023. ChatDev: Communicative Agents for Software Development. *arXiv preprint arXiv:2307.07924* (2023). doi:10.48550/arXiv.2307.07924
- [57] Dezhi Ran, Yuan Cao, Mengzhou Wu, Simin Chen, Yuzhe Guo, Jun Ren, Zihe Song, Hao Yu, Jialei Wei, Linyi Li, Wei Yang, Baishakhi Ray, and Tao Xie. 2026. From Assistant to Independent Developer — Are GPTs Ready for Software Development?. In *The Fourteenth International Conference on Learning Representations*. <https://openreview.net/forum?id=XrP8dp1rCg>
- [58] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A concolic unit testing engine for C. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 263–272. doi:10.1145/1081706.1081750
- [59] Aditya S Shethiya. 2024. Engineering with Intelligence: How Generative AI and LLMs Are Shaping the Next Era of Software Systems. *Spectrum of Research* 4, 1 (2024).
- [60] Jiho Shin, Sepehr Hashtroudi, Hadi Hemmati, and Song Wang. 2024. Domain Adaptation for Code Model-Based Unit Test Case Generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (Vienna, Austria) (ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1211–1222. doi:10.1145/3650212.3680354
- [61] Vanshika Singh, Caglar Korlu, and Wesley K. G. Assunção. 2025. Experiences on Using Large Language Models to Re-Engineer a Legacy System at Volvo Group. In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 102–112. doi:10.1109/SANER64311.2025.00018
- [62] Stack Overflow 2008. <https://stackoverflow.com/>

- [63] Xiangru Tang, Yuliang Liu, Zefan Cai, Yanjun Shao, Junjie Lu, Yichi Zhang, Zexuan Deng, Helan Hu, Kaikai An, Ruijun Huang, Shuzheng Si, Sheng Chen, Haozhe Zhao, Liang Chen, Yan Wang, Tianyu Liu, Zhiwei Jiang, Baobao Chang, Yin Fang, Yujia Qin, Wangchunshu Zhou, Yilun Zhao, Arman Cohan, and Mark Gerstein. 2024. ML-Bench: Evaluating Large Language Models and Agents for Machine Learning Tasks on Repository-Level Code. *arXiv:2311.09835* [cs.CL] <https://arxiv.org/abs/2311.09835>
- [64] Trae Research Team, Pengfei Gao, Zhao Tian, Xiangxin Meng, Xincheng Wang, Ruida Hu, Yuanan Xiao, Yizhou Liu, Zhao Zhang, Junjie Chen, Cuiyun Gao, Yun Lin, Yingfei Xiong, Chao Peng, and Xia Liu. 2025. Trae Agent: An LLM-based Agent for Software Engineering with Test-time Scaling. *arXiv:2507.23370* [cs.SE] <https://arxiv.org/abs/2507.23370>
- [65] Valerio Terragni, Partha Roop, and Kelly Blincoe. 2024. The Future of Software Engineering in an AI-Driven World. *arXiv:2406.07737* [cs.SE] <https://arxiv.org/abs/2406.07737>
- [66] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617* (2020). doi:10.48550/arXiv.2009.05617
- [67] Richard J. Waldinger and Richard C. T. Lee. 1969. PROW: a step toward automatic program writing. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence* (Washington, DC) (IJCAI'69). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 241–252.
- [68] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. 2024. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741* (2024). doi:10.48550/arXiv.2407.16741
- [69] Jin Wen, Qiang Hu, Yuejun Guo, Maxime Cordy, and Yves Le Traon. 2025. Variable Renaming-Based Adversarial Test Generation for Code Model: Benchmark and Enhancement. *ACM Trans. Softw. Eng. Methodol.* 35, 1, Article 6 (Dec. 2025), 28 pages. doi:10.1145/3723353
- [70] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryan W White, Doug Burger, and Chi Wang. 2023. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. *arXiv:2308.08155* [cs.AI] <https://arxiv.org/abs/2308.08155>
- [71] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. 2025. The rise and potential of large language model based agents: A survey. *Science China Information Sciences* 68, 2 (2025), 121101. doi:10.1007/s11432-024-4222-0
- [72] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 416–426. doi:10.1109/ICSE.2017.45
- [73] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems* 37 (2024), 50528–50652. doi:10.52202/079017-1601
- [74] John Yang, Carlos E Jimenez, Alex L Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R Narasimhan, et al. 2024. Swe-bench multimodal: Do ai systems generalize to visual software domains? *arXiv preprint arXiv:2410.03859* (2024). doi:10.48550/arXiv.2410.03859
- [75] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and Improving ChatGPT for Unit Test Generation. *Proc. ACM Softw. Eng.* 1, FSE, Article 76 (jul 2024), 24 pages. doi:10.1145/3660783
- [76] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 2471–2484. doi:10.18653/v1/2023.emnlp-main.151
- [77] Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M Rush. 2025. Commit0: Library Generation from Scratch. In *The Thirteenth International Conference on Learning Representations*. <https://openreview.net/forum?id=MMwaQEVsAg>

Received 2026-01-30; accepted 2026-06-25