How Much Static Structure Do Code Agents Need? A Study of Deterministic Anchoring

ANONYMOUS AUTHOR(S)

LLM-based code agents navigate repositories through keyword search but miss the structural relationships, such as call graphs, inheritance hierarchies, and configuration dependencies, that define how software actually works. This makes agent navigation stochastic and difficult to reproduce across runs. We investigate whether lightweight static analysis can provide *deterministic anchors* for these agents: stable structural facts injected as plain-text comments that constrain probabilistic exploration and make navigation more predictable. Starting from a strong baseline, Codex from OpenAI, we systematically inject varying granularities of structural annotations and measure their effects on localization, trajectory behavior, and run-to-run stability. Our study identifies what we call the **deterministic anchoring effect**: static structure helps less by making agents "smarter" and more by making their navigation *disciplined and reproducible*. Three observations support this finding: ❶ **Anchoring works**: lightweight call/inheritance topology improves function-level localization (+2.2pp Func@5) and shortens trajectories (−1.6 interaction rounds); ❷ **Anchoring is scale-sensitive**: the optimal granularity and directionality depend on repository characteristics, where denser semantics show diminishing returns and hub-heavy projects benefit from inverse-only links that expose "who-calls-me" without forward edges; ❸ **Anchoring stabilizes**: tags raise link-following rate from 0.15–0.18 to 0.21–0.24, roughly halve run-to-run variance, and improve single-run reliability (Pass@1 +3.4 pp) on medium-scale repositories, at the cost of roughly 10% more input tokens. These observations suggest practical guidelines: default to lightweight topology on medium projects, prune forward edges in large repositories, and reserve dense tags for implicit-dependency cases. Code and traces are available in our repositories: https://anonymous.4open.science/r/CodeAnchor-Impl-8CA2.

ACM Reference Format:
Anonymous Author(s). 2018. How Much Static Structure Do Code Agents Need? A Study of Deterministic Anchoring. *ACM/IMS J. Data Sci.* 37, 4, Article 111 (August 2018), 23 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Large language models have transformed code assistance from single-completion tools into autonomous agents that can navigate, analyze, and modify entire repositories [14, 39]. Commercial agent systems such as Copilot Workspace, Claude Code, and Gemini Code Assist [7, 20, 21], along with open research agents like SWE-agent and SWE-Gym [42, 61], have converged on a shared architecture: *grep-first retrieval* paired with strong LLM reasoning. Given a natural language description of a bug or feature request, the agent iteratively issues keyword queries (`grep`/`rg`), inspects matching snippets, refines its hypotheses, and searches again. This design is attractive because it is fast, language-agnostic, robust to broken code, and requires no heavyweight indexing infrastructure. Empirically, these grep-first agents

ACM 2831-3194/2018/8-ART111
https://doi.org/XXXXXXX.XXXXXXX

are highly competitive and often outperform more complex graph-based approaches in end-to-end repair benchmarks [26, 32, 58, 61].

Yet despite this practical success, grep-first agents suffer from a fundamental *representation mismatch*. Grep returns lines ranked by simple lexical criteria and does not expose the *structural relationships* that organize real software: who calls whom, how configuration values propagate, which classes inherit from which base, or how data flows between modules. Whether a line is relevant to a bug depends not only on its keyword overlap with a query but also on where it sits in this structural topology. This mismatch has two critical consequences. First, agents must rediscover structural links through ad-hoc multi-hop queries, resulting in fragmented, locally myopic context. Second, LLM controllers are inherently stochastic: small perturbations in early searches cascade into qualitatively different navigation paths. When coupled with structure-blind retrieval, this produces *brittle trajectories*: two runs on the same task may visit different files, take different numbers of steps, and produce different patches, even when the final outcome is identical. Recent work on agent trajectories similarly highlights the sensitivity of tool-using agents to exploration decisions and early branching [51].

From a software engineering perspective, this *behavioral unpredictability* is a critical quality concern: practitioners need agent behavior to be inspectable and reproducible, not merely correct in aggregate [6, 9]. Our goal is therefore to understand the *marginal value of static structure* when added to already-strong grep-first agents. We start from a baseline that already outperforms graph-based alternatives by a wide margin (Section 3), ask what additional benefit lightweight structural annotations provide, and treat trajectory quality and stability as first-class outcomes alongside localization accuracy.

This goal leads to a key insight: improving grep-first agents does not require abandoning text retrieval or building a new graph-guided controller. Instead, we can *inject stable program structure directly into the agent's text view*. We call these structure facts *deterministic anchors*: repository-level relationships (e.g., who calls whom, inheritance links, configuration usage) that are fixed for a given code snapshot and therefore should not vary across runs. Exposing them inline reduces the need for the agent to rediscover links via ad-hoc search and can make exploration more disciplined under stochastic LLM control.

To explore this idea, we present *CodeAnchor*, a framework that augments grep-based code agents with *static-analysis-based structured comments*. *CodeAnchor* performs lightweight static analysis offline to extract structural relationships such as calls, inheritance, data flow, and configuration usage. It then injects these facts back into the codebase as compact "CodeAnchor tags," which are plain-text comments colocated with functions, classes, and configuration entries. Figure 1 shows an example of injecting tags into the code. At runtime, the agent continues to issue ordinary text searches over the repository, but search results now contain both raw code and nearby tags that surface repository-internal cross-file structure in place. Because tags are in-band text, they can influence both where the agent lands and how it navigates after opening a file. This design enables what we call *retrieval short-circuiting*: when the agent inspects an entity, the surrounding tags already reveal nearby structural links (e.g., callers/callees, imports, or configuration usage). The agent can follow these links directly instead of rediscovering them via additional searches, constraining exploration and making trajectories more consistent across stochastic runs.

We instantiate *CodeAnchor* on top of Codex, a programmable code agent, and evaluate it on SWE-bench Lite and SWE-bench Verified [26]. We ask:

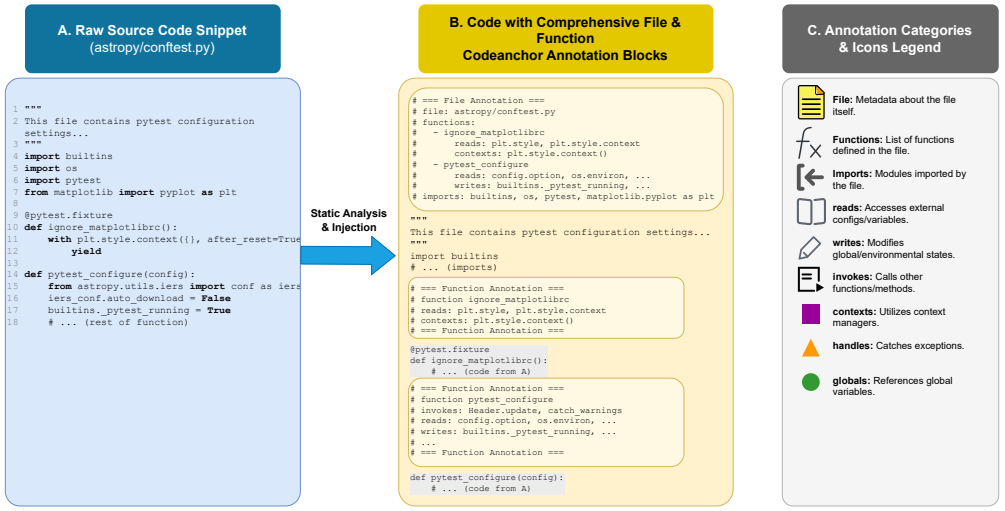

Fig. 1. The framework of *CodeAnchor*: Offline static analysis (PyCG call graphs + AST extractors) generates structured tags encoding call/inheritance/data-flow relationships, which are injected into source files as plain-text comments. At runtime, the grep-first agent retrieves code with embedded tags, enabling structure-guided navigation without changing the agent loop.

- **RQ1 (Does Topology Help?):** Does injecting basic topological structure (call/inheritance links) improve localization and interaction efficiency over pure keyword search?
- **RQ2 (Granularity and Directionality):** How do varying levels of semantic detail and link directionality affect agent behavior across different repository scales?
- **RQ3 (Behavioral Change & Stability):** How do structural cues reshape trajectories, and to what extent do deterministic anchors improve run-to-run stability?

Across these questions, our primary lens is *behavioral*: we treat trajectory quality and stability as first-class outcomes and interpret function-level localization gains as secondary evidence that injected structure is genuinely used rather than ignored. We release the implementation, tagging pipeline, and evaluation traces as open source at https://anonymous.4open.science/r/CodeAnchor-Impl-8CA2.

In summary, our study identifies what we call the **deterministic anchoring effect**: static structure helps less by making agents "smarter" and more by making their navigation *disciplined and reproducible*. Three observations support this finding: ❶ **Anchoring works**: lightweight call/inheritance topology improves function-level localization (+2.2pp Func@5) and shortens trajectories (−1.6 interaction rounds) (Section 5.2); ❷ **Anchoring is scale-sensitive**: optimal granularity and directionality depend on repository scale, with hub-heavy projects favoring inverse-only links (Section 5.3); ❸ **Anchoring stabilizes**: tags raise link-following rate from 0.15–0.18 to 0.21–0.24 and roughly halve run-to-run variance (Section 5.4). These gains cost ∼9.9% more input tokens on Lite, motivating practical guidelines: default to lightweight topology on medium projects, prune forward edges in large repositories, and reserve dense tags for implicit-dependency cases.

2 Background

This section introduces the grep-first agent architecture that serves as the foundation for our study.

Grep-first code agents. Modern code agents [14, 39, 61] follow a tool-using pattern: given an issue description, the agent iteratively issues keyword queries (`grep`/`rg`), inspects matching snippets, refines its hypothesis, and searches again. Commercial assistants (Copilot Workspace, Claude Code, Gemini Code Assist) [7, 20, 21] predominantly adopt this architecture for its simplicity, language-agnosticism, and robustness to incomplete code. The LLM is responsible for precision: filtering irrelevant matches, reasoning about code semantics, and deciding where to navigate next.

The connectivity gap. While grep-based retrieval excels at high-recall keyword matching, it treats the codebase as an unstructured collection of text. Structural relationships that organize real software, such as who calls whom, which classes inherit from which base, and how configuration values propagate, remain implicit. Agents must rediscover these links through ad-hoc multi-hop queries, resulting in fragmented context and unpredictable navigation paths. This motivates our investigation: can lightweight structural annotations bridge this connectivity gap without abandoning the simplicity of grep-first retrieval?

3 Motivation and Problem Analysis

For the agents we study, the core interaction pattern is a retrieve-then-read loop: the agent issues text queries against the repository, opens matching snippets, updates its hypothesis, and repeats. Tools such as `grep` or BM25 [45] treat the codebase as unstructured text and rank by lexical relevance. With modern reasoning models, this has become a hard-to-beat baseline: grep supplies broad candidates, the LLM supplies precision. What this view still lacks is *connectivity*. Hybrid retrieval (e.g., BM25 + embedding reranking [30]) improves matching but still returns independent text spans without making call, inheritance, or configuration/data-flow links explicit. Yet whether a location is relevant to a bug depends on how it sits in the program's structure—which configuration values reach it, which functions it calls or overrides, and which downstream components it affects.

Consider a configuration-driven bug as a running example (shown in Figure 2). Changing a single timeout in a YAML file may indirectly affect a database connection pool, a circuit breaker, and a load balancer, each implemented in different modules. An engineer investigating such an incident usually follows a structural chain: identify the configuration, find all its uses, understand how the value is transformed, and reason about the downstream components. A structure-blind agent, in contrast, must rediscover each link via additional searches. The result is a fragmented, locally myopic view: the agent may see strong textual evidence at a call site yet never reach the internal helper where the actual fix belongs. This perspective closely mirrors classic change impact analysis workflows that trace dependency chains across modules [8, 12, 22, 28, 31].

This structural mismatch interacts with the stochastic nature of LLM controllers. An agent run is not a single response but a *trajectory* that interleaves tool calls and model invocations. Small differences in early search queries or intermediate summaries can send the agent down different branches of the structural chain. Our preliminary observations (later formalized in RQ3) indicate that, on the same SWE-bench Lite issue and with the same agent configuration, repeated runs can visit different sets of files, take different numbers of search steps, and produce different patches, even when the final pass/fail outcome is the same. For practitioners, this means that behavior is difficult to predict or reproduce, and failures are hard to debug because different runs often fail for different structural reasons [6, 51]. In real deployments, developers only run the agent once. However, the trajectory variance leads to the situation that a bug found in one run may be missed in the next. Engineers cannot

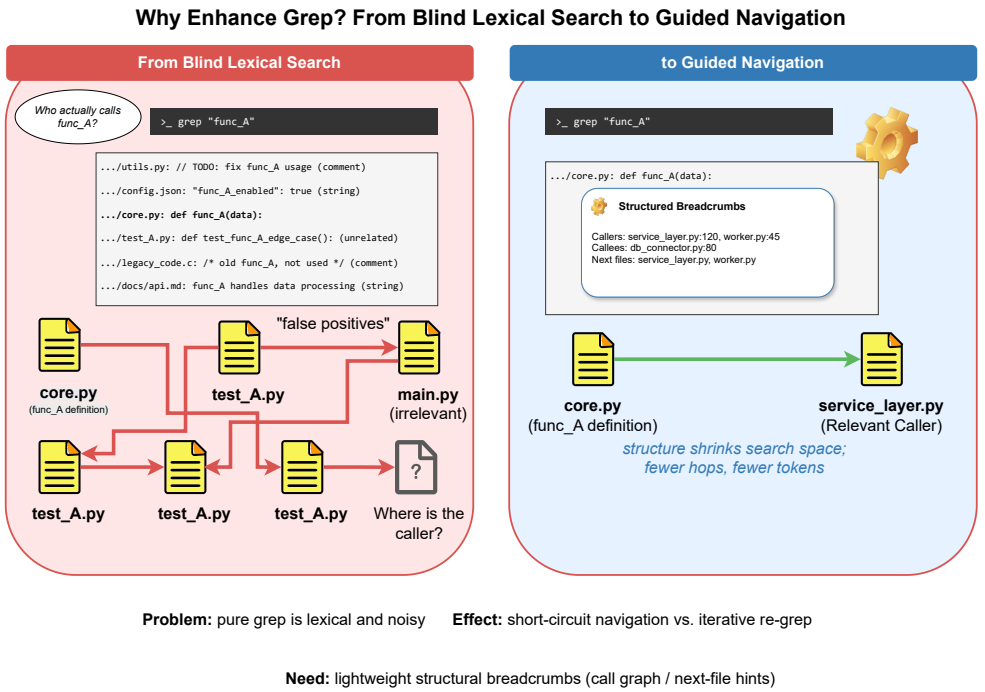

Fig. 2. Motivating example: pure grep-style keyword search is lexical and noisy, while inline CodeAnchor tags surface structured breadcrumbs (e.g., callers/callees and next-file hints) to guide navigation.

tell whether the failure reflects a systematic weakness or just stochastic noise. If agents can run stable, the failures may be more interpretable: when two failed runs follow the same path, the missing step is plain and can be addressed; when every run fails differently, each failure is a fresh investigation. Therefore, in the rest of this section, we treat fragility and run-to-run variance as symptoms of the underlying mismatch between text-oriented retrieval and structure-oriented code, and ask how much explicit structure is needed to control them.

Our goal is not only to make code agents smarter but also to make their behavior less random. Source code contains rich *deterministic structure*—call graphs, inheritance hierarchies, data-flow and configuration usage, module-level dependencies—that does not change across runs for a fixed repository. Surfacing such facts inline lets them serve as *deterministic anchors* for an otherwise stochastic process. This leads to *CodeAnchor*: run lightweight static analysis offline, inject the resulting structural facts back into the codebase as compact structured comments (CodeAnchor tags), and let the agent continue using its familiar grep-based retrieval over both raw code and these annotations. When the agent opens a function, the surrounding tags already reveal its callers, call targets, and upstream configuration, short-circuiting the multi-hop searches that would otherwise be needed. We keep the underlying agent loop, tools, and model fixed, and study this design on SWE-bench Lite and Verified [26].

A natural alternative is to expose structure via an explicit MCP (Model Context Protocol) tool (e.g., LSP "find references" [36]). However, in a pilot on 20 SWE-bench tasks with an optional call-graph tool, we observed low tool-use rates: the agent typically relied on

plain `grep` instead. This motivates our "passive injection" stance: surface structural context inline, right next to the code the agent opens, so that following structure does not require an explicit tool-usage decision. We also choose to strengthen a grep-first loop rather than build a new graph-guided controller because our goal is to study the marginal impact of static structure on architectures that practitioners already deploy. To validate this focus, we benchmarked LocAgent [16] on SWE-bench Lite under matched conditions: the graph agent trails grep-first by 23.7pp (59.5% vs. 83.2% Func@5: top-5 predictions cover all ground-truth functions; see Section 5 for full definitions of File@k and Func@k, Table 1).

Table 1. Preliminary SWE-bench Lite head-to-head (matched model/limits, N=274).

Agent	File@1	File@3	File@5	Func@5	Func@10
LocAgent (graph-based)	0.726	0.850	0.861	0.595	0.664
Codex (grep-based)	0.912	0.967	0.967	0.832	0.836

Analysis reveals clear grep-based failure modes: *function-level failures* where the agent stops at call sites without reaching private implementations. Textual evidence finds the right file, but disambiguating similarly named functions requires explicit knowledge of who calls whom. These observations motivate our design: keep grep-first tools fixed, and inject just enough static structure to repair multi-hop and disambiguation failures.

Based on this analysis, the next section presents *CodeAnchor*, our concrete instantiation of deterministic anchors for grep-based code agents.

4 Approach

We now describe *CodeAnchor*, our instantiation of static structure-as-anchors for grep-based code agents. We first provide an overview of the system, then detail how we build CodeAnchor tags via lightweight static analysis, and finally explain how these tags integrate with an existing Codex-style agent. To make the design concrete, we also present a pseudocode description of the offline tag generation pipeline.

4.1 System Overview

CodeAnchor has two components: an *offline static analysis pipeline* that extracts structural relationships from the codebase and encodes them as structured comments (CodeAnchor tags), and an *agent integration layer* that lets a grep-driven agent consume these tags without changing its high-level control loop. The offline pipeline runs once per repository snapshot (or incrementally after code changes) and is *task-agnostic*: it depends only on repository contents and does not use the natural-language issue description or any ground-truth localization/patch metadata. It parses source files, identifies key entities (functions, classes, configuration entries, files), and computes relationships such as callers, callees, inheritance, imports, containment, data dependencies, and configuration usage. These relationships are attached to entities as synthetic comments in a compact, machine-parsable format.

At runtime, the agent continues to use plain text search (`rg`) over the working tree. Because CodeAnchor tags are inserted directly into source files, search results now include both code and nearby tags, which the agent reads as ordinary text in its context window. No new tools or APIs are required beyond grep and file I/O. In particular, we do not add a separate embedding index or an explicit reranking stage: retrieval remains the same grep-first primitive, and only the repository view changes.

4.2 CodeAnchor Tags: Static-Analysis-Based Structured Comments

CodeAnchor tags are structured comment blocks colocated with code entities. They follow a simple schema:

```
# === Function Annotation ===
# function _get_locale_dirs
# used by: fetch, lang_stats, update_catalogs
# === Function Annotation ===
def _get_locale_dirs(resources, include_core=True):
```

The exact syntax varies across languages (e.g., `//` for Java/C++, `#` for Python), but the semantics are uniform: each tag block has a stable identifier for the entity it annotates (`function`, `class`, or `file`), fields encode structural relationships (`usedby`, `invokes`, `inherits`, `imports`), and values are normalized references to other entities. Because tags are structured comments with uniform delimiters (`===Annotation===`), they are trivially identifiable and removable via regular-expression passes, ensuring that repositories can cleanly revert to untagged code if needed.

We distinguish two tiers of relationships that correspond to the two configurations in our experiments. In the `Anchor-Topo` configuration, we surface only four LocAgent-style structural relations [16]: *contains*, *imports*, *invokes*, and *inherits*. These appear in tags as fields such as `PARENT`/`CHILDREN` (for containment), `IMPORTS`/`IMPORTED_BY` (for module imports), `CALLS`/`CALLED_BY` (for function calls), and `BASE`/`DERIVED` (for class inheritance). This lightweight topology gives the agent an explicit call graph and class hierarchy, plus the file/module skeleton, without attempting to encode richer semantics.

In the `Anchor-Dense` configuration we add further relations on top of this core: configuration-driven edges such as `CONFIG_USAGE`, `ENV_VAR`, and `CONST_REF` that link config entries and constants to the code that consumes them; simple data-flow edges such as `DATA_DEP` and `IO_DEP` that sketch where values and I/O effects travel [2, 24, 52, 57]; and a small number of domain-specific links (e.g., test-to-code mappings and plugin registrations) that help in repositories with heavy plugin or test harness use. These extra fields are exactly what distinguishes `Anchor-Dense` from `Anchor-Topo` in RQ2: both share the same four structural backbones, but only the full variant carries configuration and data-flow hints.

Finally, we vary directionality to study how much "forward" navigation signal is useful. The `Anchor-Inv` configuration keeps only inverse dependencies such as `CALLED_BY` (dropping `CALLS`). This is not suggested as a universal improvement over the policy; it isolates one type of tension that arises with hub-heavy projects when links to functions with high degrees repeat the name of the function throughout many call sites, allowing that hub to eclipse others in keyword search. Inverse-only tags keep backward navigability at this expense. In principle, intermediate policies are possible (e.g., degree caps or hub filtering on forward links), but we focus on these clean variants to isolate behavioral effects under a fixed grep-first loop.

Scalability and Language Support. The analysis is language-pluggable: the tag schema and agent integration are language-agnostic, while relationship extractors can be implemented with language-specific tooling. For established languages such as Java, JavaScript, C++, and Go, mature static-analysis frameworks (e.g., WALA, Soot, CodeQL, Semgrep, or LLVM-based tools) can provide call graphs and dataflow information [17, 29, 37, 44, 54]. For newer or less-supported languages, tree-sitter [53] enables rapid development of lightweight AST-based extractors that capture containment, imports, and simple call patterns without

requiring a full compiler frontend. For repositories where deep interprocedural analysis is infeasible, *CodeAnchor* still provides value via the four structural relations alone, which is precisely the setting captured by our `Anchor-Topo` runs, in line with classic guidance on scalable static analysis [18].

4.3 Offline Tag Generation

The offline pipeline constructs CodeAnchor tags for a given repository snapshot by: (1) parsing all source files into an intermediate representation, (2) identifying entities (functions, classes, config entries, files), (3) for each entity, inferring call relations, data flow, and domain links using language-specific frontends, (4) normalizing relationships to stable identifiers, (5) rendering tags as language-specific comments, and (6) inserting tags near definition sites. This workflow mirrors classic dependency extraction used in impact analysis and regression testing [10, 40].

Two terms above need clarifying. A *stable identifier* is how we name a code entity inside a tag; for a function, we use its file path with its dotted name (e.g., `django/db/backends/base/schema.py:BaseDatabaseSchemaEditor._delete_composed_index`), so tags survive whitespace, reformatting, or moves within a file, while renames and cross-file moves regenerate the tag. *Domain links* are the extra relationships added by `Anchor-Dense` on top of call/inheritance: `CONFIG_USAGE` for code reading Django/Flask settings, `CONST_REF` for code using module-level constants, `IO_DEP` for functions touching files, network, or databases, and `TEST_REF` for test files matched to production modules; all four come from AST pattern matches rather than interprocedural analysis, keeping the pipeline fast and free of spurious edges. Tag generation is a one-time cost paid per repository snapshot and reused across all queries. On four Python repositories, static graph construction (AST walk plus call/inheritance/import extraction) takes **6.8 s** (`pytest-7432`, 73k LOC), **21.1 s** (`sklearn-15512`, 247k), **58.4 s** (`astropy-12907`, 341k), and **133.4 s** (`django-13658`, 367k LOC); cost scales near-linearly with LOC and graph size. Incremental regeneration after code changes touches only the affected files.

Python Instantiation. Our prototype targets Python using PyCG [48] for call graphs and AST passes for containment/imports/inheritance (`Anchor-Topo` config). For `Anchor-Dense`, we add lightweight intraprocedural extractors for config usage and data/I/O hints. All analysis prioritizes precision over completeness: PyCG and our syntactic extractors emit only verifiable relationships, avoiding spurious edges from dynamic dispatch, reflection, or monkey patching, which are well-known sources of unsoundness in static analysis [11]. We accept incomplete coverage for lightweight, CI-deployable pipelines; results (Section 5) show that the captured relationships suffice for most localization tasks.

4.4 Agent Integration

We integrate *CodeAnchor* with a Codex-style agent exposing text search, file open, patch application, and test execution. Crucially, we do *not* modify the agent's outer control loop; tags are treated as part of the codebase. The agent issues keyword searches (hitting both code and tags), opens files and reads nearby tags as hints, and follows explicit links or issues new searches. This realizes *retrieval short-circuiting*: tags provide deterministic local structure, reducing search calls and constraining plausible next steps.

We compare four retrieval views: Baseline (raw grep), `Anchor-Topo` (bidirectional call/inheritance), `Anchor-Dense` (adds data-flow/config edges), and `Anchor-Inv` (inverse-only).

The localization prompt is purely task-oriented and makes no reference to tags; the model treats them as ordinary comments.

5 Evaluation

We evaluate *CodeAnchor* by integrating it into a Codex-style programmable code agent and running it on SWE-bench Lite and SWE-bench Verified [26], which are benchmarks of real GitHub issues paired with tests. This agent architecture matches the grep-first design used by many modern industrial assistants and is already highly competitive: on SWE-bench Lite, for example, our raw baseline achieves Func@5 ≈83.2%. Our evaluation is structured around the three research questions introduced in Section 1:

- **RQ1: Does Topology Help?** - Does injecting basic topological structure (call and inheritance links) improve localization accuracy and interaction efficiency over pure keyword search?
- **RQ2: Granularity and Directionality** - How do varying levels of semantic detail (basic topology vs. dense annotations) and link directionality (bidirectional vs. inverse-only) affect agent behavior across different repository scales?
- **RQ3: Behavioral Change & Stability** - How does structural injection alter the agent's navigation trajectory, and to what extent can deterministic anchors mitigate the stochastic nature of LLM navigation across repeated runs?

5.1 Experimental Setup

Model and agent. We use the `Codex` coding agent with OpenAI's GPT-5.1-codex model (thinking effort set to high). The agent exposes many bash commands such as Search (`rg`), Open, ApplyPatch, and RunTests; we focus on localization and disable patching/tests to isolate navigation effects.

Prompt. All configurations share the same localization prompt template, which is purely task-oriented and makes no reference to CodeAnchor tags or any tag-following policy. Thus, the only experimental variable is whether tags are present in the retrieved code context. The prompt instructs the agent to localize files, classes, and functions for a given GitHub issue and return a JSON response with ranked candidates sorted by confidence.

Benchmark configuration. We use SWE-bench Lite and Verified [26], and for SWE-bench Lite, we use the same 274 instances following the setting of LocAgent. For the stability analysis in RQ3, we construct a 50-instance subset per dataset with k=10 repeats, prioritizing boundary cases where Baseline and `Anchor-Topo` disagree, then filling with uniform random samples.

Configurations. We compare four retrieval views of the same agent: a Baseline over raw code, an `Anchor-Topo` version with forward and backward call/inheritance tags, an `Anchor-Dense` variant that also injects data-flow and configuration edges, and an inverse-only `Anchor-Inv` variant that retains only "who-calls-me" links (no forward call edges). RQ1 focuses on Baseline vs. `Anchor-Topo` to establish whether structure helps at all; RQ2 compares the three structural variants (`Anchor-Topo`, `Anchor-Dense`, `Anchor-Inv`) to explore granularity and directionality effects; RQ3 examines behavioral changes primarily through Baseline vs. `Anchor-Topo` to isolate stability effects under a fixed topology choice.

Metrics. We evaluate localization performance using File@k and Func@k. A task is considered successful if and only if the top-k retrieved entities (files or functions) encompass

the entire set of ground-truth targets for that task; the metric reports the percentage of such successful tasks across the dataset. On *Lite* (274 instances), every instance has one ground-truth file. 84.0% of them have a single ground-truth function (230/274); 11.3% have two, and the remaining 4.7% have three or more. *Verified* (500 instances) is harder: 85.8% have one ground-truth file and 14.2% have two or more; 64.2% have a single ground-truth function and 35.8% have two or more. Func@k at larger k therefore tests whether the top-k predictions cover all ground-truth functions, which is most demanding on multi-function tasks.

We also report Rounds, defined as the average number of tool calls per task, to measure interaction efficiency. A *navigation transition* is any tool call that opens, searches, or inspects a code entity; Hops counts all such transitions summed across tasks. Link Following Rate (LFR) measures trajectory guidance: for each navigation step $t \to t+1$, we classify the transition as *structural* if the opened entity at $t+1$ was mentioned in a CodeAnchor tag visible at t, *lexical* if it appeared in non-tag text at t, or *exploratory* otherwise. LFR is the fraction of steps that are either structural or lexical (i.e., non-exploratory). Struct and Lex denote the structural-only and lexical-only fractions, respectively. We quantify uncertainty via nonparametric bootstrap, and report stability as mean ± std across 10 runs.

5.2 RQ1: Does Topology Help?

RQ1: *Does injecting basic topological structure (call and inheritance links) improve localization accuracy and interaction efficiency over pure keyword search?*

Before exploring richer annotations or alternative configurations, we first establish whether structural topology provides measurable benefit at all. We compare Baseline (raw grep over untagged code) against `Anchor-Topo` (bidirectional call/inheritance tags) on SWE-bench Lite (274 tasks) and Verified (500 tasks). Table 2 summarizes results.

Table 2. RQ1: Baseline vs. `Anchor-Topo` on SWE-bench Lite and Verified.

	SWE-bench Lite					SWE-bench Verified				
Config	File@1	File@3	Func@5	Func@10	Rounds	File@1	File@3	Func@5	Func@10	Rounds
Baseline	**0.9124**	0.9672	0.8321	0.8358	35.3	0.8520	0.8680	0.6187	0.6227	42.4
`Anchor-Topo`	0.8978	**0.9745**	**0.8540**	**0.8577**	**33.7**	0.8480	0.8620	**0.6308**	**0.6369**	**40.9**
Δ	−1.5pp	+0.7pp	**+2.2pp**	**+2.2pp**	**−1.6**	−0.4pp	−0.6pp	**+1.2pp**	**+1.4pp**	**−1.5**

Localization Accuracy. On Lite, `Anchor-Topo` improves function-level recall by +2.2pp (Func@5: 0.8540 vs. 0.8321; Func@10: 0.8577 vs. 0.8358), with the Func@5 improvement reaching statistical significance (McNemar p=0.041). On Verified, improvements are consistent though smaller (+1.2pp Func@5, +1.4pp Func@10; McNemar p=0.023 for Func@10). The baseline already achieves 83.2% Func@5 on Lite, leaving limited headroom; the observed gains therefore reflect improvements in the regime of an already strong agent.

Interaction Efficiency. Tags also reduce trajectory length: −1.6 rounds on Lite (33.7 vs. 35.3) and −1.5 rounds on Verified (40.9 vs. 42.4). Shorter trajectories suggest that explicit structural links help agents reach targets more directly, reducing exploratory tool calls.

Structural Detour. We observe a notable pattern: `Anchor-Topo` slightly lowers File@1 on Lite (−1.5pp) even while improving Func@10. This reflects what we term a *structural detour*: agents with tags may first visit hub or helper files surfaced by structural annotations, then

navigate to the ground-truth function within the top few opens. Trace analysis shows that a substantial fraction (34% of Verified tasks exhibiting File@1 regression) still reach the target function within 3 opens despite initially visiting a structural neighbor. In essence, agents trade an immediate "lucky" lexical hit for a grounded structural path, a trade-off that pays off in final function identification.

Downstream Impact: Localization → Repair. To test whether localization gains propagate to repair, we isolate the 80 SWE-bench Verified instances where Baseline and `Anchor-Topo` disagree on the top-5 function set, then run a controlled repair experiment on this differential subset: the repair agent is held fixed (GPT-5.1-codex), and only the top-5 localized functions vary. `Anchor-Topo` resolves 48/80 against Baseline's 38/80 (60.0% vs. 47.5%, +12.5 pp), with a strict *superset property*: every Baseline-repaired instance is also repaired by `Anchor-Topo`, plus 10 additional cases (zero Baseline-exclusive repairs). This is a conditional effect $\mathbb{E}[\Delta\text{Fix} \mid \Delta\text{Loc} > 0]$; projected to all 500 Verified instances, the 10 extra repairs correspond to ∼+2 pp overall, consistent with the +1.2 pp Func@5 gain.

Insight 1:
- Structural topology reliably improves function-level recall (**+2.2pp Func@5** on Lite, **+1.2pp** on Verified) over a strong grep baseline.
- Tags shorten trajectories (**−1.6 rounds** on Lite, **−1.5 rounds** on Verified), indicating more direct navigation.
- Structural detours may lower early file hits but ultimately improve function identification.
- Localization gains propagate to repair on the differential subset: **+12.5pp** repair success with a strict superset property; projected to full Verified, ∼+2 pp overall.

5.3 RQ2: Granularity and Directionality

RQ2: *How do varying levels of semantic detail (basic topology vs. dense annotations) and link directionality (bidirectional vs. inverse-only) affect agent behavior across different repository scales?*

Having established that structural injection benefits localization (RQ1), we now investigate how different structural configurations affect agent behavior. We explore two dimensions: (1) **granularity**, whether denser semantics (data-flow, configuration edges) beyond basic topology provide additional benefit or introduce overhead; and (2) **directionality**, how bidirectional vs. inverse-only links interact with repository characteristics.

We compare three structural configurations: `Anchor-Topo` (bidirectional call/inheritance), `Anchor-Dense` (adds data-flow and config edges), and `Anchor-Inv` (inverse-only, i.e., "who-calls-me" links without forward edges). Table 3 summarizes results.

In RQ2, we combine `call` and `inheritance` into one "basic topology" type. For instance, call edges have roughly 8–20× than inheritance edges in our Python repositories; an inheritance-only variant would be dominated by a few class-hierarchy cases while failing to have enough power under our α=0.05 design to detect differences. Granularity and Directionality are then the dimensions where effect sizes stay large enough to discriminate, which is what RQ2 studies.

Effect of Granularity: Topo vs. Dense. Moving from `Anchor-Topo` to `Anchor-Dense` produces different effects across datasets. On Lite, function accuracy remains unchanged (Func@5/10 both 0.8540) while rounds increase by +4.9 and input-token usage grows by +18.8% (530k vs. 446k). On Verified, `Anchor-Dense` marginally improves Func@10 (0.6389 vs.

Table 3. RQ2: Effect of granularity and directionality on localization and efficiency. Tokens are mean per-instance input tokens; Baseline shown for reference.

	SWE-bench Lite					SWE-bench Verified				
Config	Func@5	Func@10	Rounds	Input tok.	Δ vs. Base	Func@5	Func@10	Rounds	Input tok.	Δ vs. Base
Baseline	0.8321	0.8358	35.3	406k	—	0.6187	0.6227	42.4	567k	—
`Anchor-Topo`	**0.8540**	**0.8577**	**33.7**	446k	+9.9%	0.6308	0.6369	**40.9**	618k	+9.0%
`Anchor-Dense`	**0.8540**	**0.8577**	38.6	530k	+30.5%	0.6288	**0.6389**	41.6	647k	+14.1%
`Anchor-Inv`	0.8242	0.8278	55.0	566k	+39.4%	**0.6329**	**0.6389**	41.7	620k	+9.3%

0.6369) but slightly lowers Func@5 (0.6288 vs. 0.6308). These results indicate a **saturation effect**: most benefit comes from basic topology, and adding dense data/config edges yields diminishing returns while increasing context overhead.

Qualitatively, `Anchor-Dense` rescues a small set of long-tail implicit-dependency tasks: 3/274 Lite instances and 15/500 Verified instances involving multi-hop value propagation (e.g., Django encoding paths, Matplotlib collection state, SymPy shape propagation). However, full traces also reveal a trade-off: dense tags can shift attention toward high-degree helper functions, changing salience ordering and causing agents to explore utility modules before reaching the actual bug site.

Effect of Directionality: Bidirectional vs. Inverse-Only. Directionality effects are **scale-dependent**. On Lite (medium-scale repositories, mean 35k LOC), removing forward edges (`Anchor-Inv`) degrades performance: Func@5 drops to 0.8242 (−3pp vs. `Anchor-Topo`) and rounds inflate to 55.0 (+21.3 vs. `Anchor-Topo`). Without forward links, agents lose explicit "where-to-go-next" guidance and resort to more expensive keyword exploration.

On Verified (larger repositories, mean 120k LOC, 23% hub nodes), the pattern differs. `Anchor-Inv` matches or exceeds other variants on function metrics (Func@5: 0.6329, Func@10: 0.6389) with an input-token footprint comparable to `Anchor-Topo` (620k vs. 618k). In hub-heavy projects, forward links from high-degree nodes generate repetitive tag text that can overexpose structurally central helpers in search results. Inverse-only tags surface "who depends on me" relationships without this amplification effect.

Insight 2:
- **Granularity saturation**: Dense data-/config-flow tags show diminishing returns (**+4.9 rounds** on Lite) beyond basic topology, though they rescue **3/274 Lite, 15/500 Verified** implicit-dependency cases.
- **Scale-dependent directionality**: Medium-scale repos benefit from bidirectional links; large hub-heavy repos show better results with inverse-only view (**+0.2pp Func@5** on Verified, at input-token parity with `Anchor-Topo`).
- **Practical implication**: Structural configuration should adapt to repository characteristics: lightweight topology for most cases, inverse-only for large projects, dense tags for implicit-flow debugging.

5.4 RQ3: Behavioral Change & Stability

RQ3: *How does structural injection alter the agent's navigation trajectory, and to what extent can deterministic anchors mitigate the stochastic nature of LLM navigation across repeated runs?*

Beyond aggregate accuracy, we look at how agents actually use structural information during navigation, and whether static structure reduces the variance that comes with

LLM-based navigation. The latter matters as much as average accuracy for production deployment.

Trajectory Behavior. We measure Link Following Rate (LFR): the fraction of navigation steps that follow an entity mentioned in a previous tag. Results from instrumented traces appear in Table 4.

On Lite, structural tags raise LFR substantially over Baseline. `Anchor-Topo` reaches the highest overall and structural LFR (0.236 and 0.163), with `Anchor-Dense` close behind (0.212 / 0.158). `Anchor-Inv` drops to 0.186 overall and 0.143 structural, only slightly above Baseline's 0.178—without forward edges, the agent falls back to lexical hopping. This matches RQ2's finding that medium-scale repositories benefit from bidirectional links.

On Verified, all tagged configurations raise LFR (0.147 → 0.174–0.212). `Anchor-Topo` and `Anchor-Dense` lead (0.209 / 0.212), and `Anchor-Inv` sits between them (0.174, structural 0.096). High LFR is not always good on hub-heavy repositories: dense tags pull the agent into utility modules whose names show up everywhere but which have nothing to do with the bug—a *structural distraction*, where the structural cue and the actual target point in different directions. `Anchor-Inv` avoids this because it has fewer links to chase in the first place.

To check whether tags actually *guide* the agent or just get followed mechanically, we compute the *effective tag rate*: out of all hops the agent takes by following a tag, the fraction that lands on a ground-truth entity. This rate is 27.1% on Lite and 26.6% on Verified, and at the instance level, 55.2% (Lite) and 58.7% (Verified) of tasks contain at least one such hop. Tags are not merely followed; they reach the right target often enough to explain the accuracy gains in RQ1 and RQ2.

Figure 3 backs this up visually: tagged configurations sustain higher target-hit probability across early steps on both datasets.

Table 4. Trajectory guidance metrics (RQ3). Hops = total navigation transitions across all tasks. Struct/Lex are structural-/lexical-link following rates; LFR is overall link-following rate.

	SWE-bench Lite					SWE-bench Verified			
Config	Hops	Struct	Lex	LFR	Config	Hops	Struct	Lex	LFR
Baseline	1132	0.000	0.178	0.178	Baseline	2426	0.000	0.147	0.147
`Anchor-Topo`	1059	**0.163**	0.162	**0.236**	`Anchor-Topo`	2521	**0.153**	0.144	0.209
`Anchor-Dense`	1200	0.158	0.154	0.212	`Anchor-Dense`	2483	0.149	0.151	**0.212**
`Anchor-Inv`	586	0.143	0.154	0.186	`Anchor-Inv`	2411	0.096	0.140	0.174

Run-to-Run Stability. We run each configuration k=10 times on 50-task subsets per dataset, giving 500 observations per config-dataset pair (post-hoc power analysis confirms 80% power at α=0.05 for medium effect sizes $d \geq 0.4$). Table 5 summarizes the results. On Lite, `Anchor-Topo` improves function metrics (Func@5: 0.7720 vs. 0.7400; Func@10: 0.7760 vs. 0.7400) and roughly halves run-level variance (0.00050 / 0.00062 vs. 0.00184). Mean per-task variance drops from 0.0726 to 0.0626, meaning fewer high-variance instances. Paired analysis gives Func@10 $\Delta = +0.036$ (95% CI [0.008, 0.068]; Wilcoxon p=0.0625, r_{rb}=0.78).

On Verified, tags improve function means (Func@5: 0.458 vs. 0.430; Func@10: 0.468 vs. 0.422) with smaller standard deviations. Mean per-task variance drops (0.0832 vs. 0.0916). Paired comparisons give Func@10 $\Delta = +0.046$ (p=0.023, r_{rb}=0.92). The 50-task subset

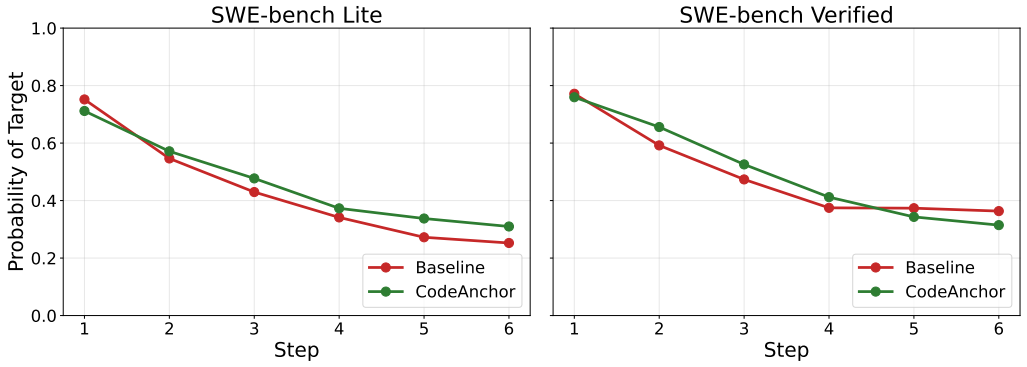

Fig. 3. Target-file hit probability over steps (RQ3). CodeAnchor tags sustain higher early hit rates on both datasets.

Table 5. Stability over k=10 runs (mean ± std). Right block: single-run Pass@k [15] on Func@5 across 50 Lite tasks.

	Lite		Verified		Lite Pass@k (Func@5)			
Config	Func@5	Func@10	Func@5	Func@10	Pass@1	Pass@3	Pass@5	Pass@10
Baseline	0.740 ± .043	0.740 ± .043	0.430 ± .036	0.422 ± .040	0.742	0.850	0.878	0.900
CodeAnchor	**0.772 ± .022**	**0.776 ± .025**	**0.458 ± .030**	**0.468 ± .018**	**0.776**	**0.873**	**0.891**	0.900

matches the full 274-task distribution (KS test $p = 0.609$); `Anchor-Topo` trajectories also show 7.2% lower cross-task diversity (mean edit distance 8.51 vs. 9.17).

Developers typically run the agent once, so we also report the unbiased Pass@k [15] on Lite Func@5 (Table 5, right block). *CodeAnchor* is ahead at every $k < 10$ (Pass@1 +3.4 pp, Pass@3 +2.3 pp), so the gain is stable on a single run, not only on average.

Insight 3:
- Tags shift agents from pure keyword hopping toward structure-following walks, raising LFR from **0.15–0.18** to **0.21–0.24**.
- On Lite, higher structural LFR correlates with fewer rounds and better localization.
- On Lite, tags raise function accuracy (**+3.6pp Func@10**) and **roughly halve variance** (std **0.0223 vs. 0.0429**).
- On Verified, tags improve function metrics (**+4.6pp Func@10**) with smaller standard deviations.
- Deterministic anchors help most on medium-scale repositories; stability benefits weaken on large, hub-heavy projects.

5.5 Case Studies: How Tags Reshape Agent Behavior

We present three case studies from SWE-bench Lite demonstrating qualitatively different behavioral improvements: entity discovery, function recall gains, and tool-call efficiency.

5.5.1 Case 1: Cross-Module Exception Discovery (django__django-11620). URL converters raising `Http404` crash Django's debug view with 500 instead of technical 404 page. Tags enabled discovering 8 entities vs. 3 for baseline, with 28.6% fewer searches. Both find

technical_404_response. Baseline searches "resolve" (83 matches), explores URLResolver.resolve, drifts into base.py. Annotated agent sees used by: response_for_exception tag, directly navigates to exception.py, discovers crash handler, follows CALLS to RoutePattern.match, with no blind guessing. Tags provide *deterministic anchors* at navigation forks where keyword search yields ambiguous results. Token cost: input 222k→443k (+99.4%), output 12.2k→13.4k; the extra context buys more structural exploration in this hub-heavy Django case.

5.5.2 Case 2: Dramatic Function Recall Gain (matplotlib__matplotlib-26020). ImageGrid axes show incorrect tick labels with label_mode="L"; fix needs Grid.__init__ and Grid.set_label_mode. This case shows the most dramatic improvement: functions located 1 → 6, with 26.7% fewer tool calls. Baseline lands on _tick_only (private helper), searches "Grid __init__" (47 matches), misses correct class. Tags show parent:Grid and used by: Grid.__init__, Grid.set_label_mode, enabling direct upward traversal without ambiguous class searches. Demonstrates *retrieval short-circuiting* for class hierarchies. Token cost: input 252k→207k (−17.8%), output 8.8k→12.4k; tags let the agent converge faster and spend fewer input tokens overall, despite the extra annotation text.

5.5.3 Case 3: Tool Call Efficiency (sympy__sympy-13915). SymPy expression simplification fails canonical form. This case shows the most dramatic tool-call reduction (34 → 20), with 30.8% fewer searches and a 3× increase in discovered entities. Baseline issues 13 searches to navigate expression tree. Tags expose transformation graph (Mul ↔ Add ↔ Pow) and base dependencies (Basic), enabling discovery of sympy/core/basic.py (which baseline never visits) without exhaustive keyword enumeration. Token cost: input 1.23M→1.03M (−16.5%), output 20.7k→15.0k; structural short-circuiting cuts both input and output tokens, since the agent skips expensive keyword exploration.

Synthesis. Tags have recurring behavioral effects that emerge across the three cases. Annotated agent finds 2–6× as many relevant entities with fewer tool calls, since structural traversal replaces broad keyword search. Second, inverse navigation via CALLED_BY and PARENT do most of the work: forward call-graph walks are easy for any agent, but locating the *caller* or *enclosing class* from a leaf function is precisely where lexical searches return ambiguous results. Third, the explicit links replace keyword enumeration which results in a 27–31% decrease of search volume. Fourth, the file set visited by agent changes: the baseline explores lexically nearby files (e.g., base.py), while the tagged agent follow structural edges towards semantically correct ones (e.g., exception.py). Together, these are what we call **deterministic anchoring**: when lexical search returns a handful of plausible paths forward, tags guide the agent toward the fundamentally correct branch. We use this structural guidance, but these cases are from medium-scale repositories (10–50k LOC); as Section 6 shows, in hub-heavy repositories it can instead inundate the agents with low-value hints, which is why we introduce the Anchor-Inv variant.

6 Discussion

We distill three observations from our study—*anchoring works*, *anchoring is scale-sensitive*, *anchoring stabilizes*—and discuss their practical implications. Detailed quantitative evidence for each observation is reported in Sections 5.2–5.4; here we focus on mechanisms and deployment guidance.

6.1 Why Lightweight Topology Suffices

Call and inheritance edges supply the *connectivity skeleton* of a repository: they answer "who calls whom" and "what inherits from what," enabling multi-hop navigation without exhaustive keyword searches. Topology is also more *robust* than richer semantics: call graphs can be extracted with high precision even from incomplete code (conservative points-to analysis [48]), whereas data-flow analysis requires whole-program assumptions that break in real repositories with dynamic features, missing dependencies, or broken builds.

Why explicit structure outperforms implicit embeddings. A natural question is why explicit structural tags help when modern LLMs already encode rich code semantics in their embeddings. We hypothesize two complementary mechanisms. First, *grounding*: explicit tags provide verifiable facts ("function X calls Y") that the model can reference directly, reducing hallucination risk compared to relying on implicit similarity judgments. Second, *locality*: embedding-based retrieval operates at the chunk level and cannot express multi-hop relationships; a function's callers may be scattered across distant files with no lexical overlap, invisible to embedding similarity but directly accessible via tags. This explains why *CodeAnchor* complements rather than replaces semantic search: tags provide the structural backbone that embeddings cannot capture, while embeddings handle fuzzy matching that rigid structure misses.

6.2 Scale-Sensitive Effects: Granularity and Directionality

Granularity saturation. Adding richer semantics in the `Anchor-Dense` configuration (data-flow, configuration, I/O, and test-to-code edges) leaves Lite function accuracy unchanged (Func@5/10 both 0.8540, identical to `Anchor-Topo`) while consuming +4.9 rounds and +18.8% more input tokens (530k vs. 446k); on Verified, Dense rescues only 15/500 implicit-dependency tasks and slightly lowers Func@5 (0.6288 vs. 0.6308). Most of the localization benefit thus comes from basic topology, motivating a *topology-first* policy: default to the four structural relations, and escalate to dense semantics only when traces show repeated failed attempts to bridge implicit dependencies.

Directionality and the hub problem. On hub-heavy repositories, the distraction is not a reasoning failure but a *retrieval distribution shift*. When high-degree helper functions accumulate dense forward-link annotations (e.g., "`CALLS`: foo.py:bar, baz.py:qux, ... [20 more]"), the tag text dominates lexical matching and pushes structurally central hubs to the top of search results. The agent follows these prominent links and spends multiple rounds in utility modules and re-exports before returning to the bug site, if at all. Inverse-only tags address this by surfacing "who depends on me" without amplifying outbound noise; the trade-off is a weaker forward signal for sharper retrieval.

Practical deployment heuristic. Based on our experiments, we propose a simple two-stage policy: (1) compute average call out-degree and hub percentage (nodes with degree > 10) from the repository callgraph; (2) if avg_out_degree < 5, use `Anchor-Topo`; if > 8, use `Anchor-Inv`; otherwise `Anchor-Topo` with optional degree capping. Lite (mean out-degree 4.2, 8% hubs) favors `Anchor-Topo`; Verified (mean 9.7, 23% hubs) favors `Anchor-Inv`. A coarser proxy is LOC: <50k favors full topology, >100k favors inverse-only.

6.3 Behavioral Stabilization as a First-Class Outcome

Nondeterministic agent behavior is a critical quality concern for deployment. When a developer reviews a localization or debugs a failure, they must understand *why* the agent

chose a path, not just whether it succeeded. Trajectories that vary qualitatively across runs frustrate systematic analysis: the same bug description can lead to different file visits and failure modes, making it difficult to identify whether a problem is task-specific or systemic. By providing deterministic anchors, tags constrain the search space and make agent behavior more reproducible and inspectable [6, 9]. Our Pass@k results (Section 5.4) put numbers on this: *CodeAnchor* wins +3.4 pp at Pass@1, which matches how localization agents are actually deployed.

Stability gains are weaker on Verified (per-task variance 0.0832 vs. 0.0916, Wilcoxon $p \approx 0.11$), suggesting that hub-heavy projects introduce additional variance sources (hub distraction, longer exploration horizons) that topology alone cannot fully constrain. Stability benefits are thus most reliable on medium-scale projects (10–50k LOC).

6.4 Implications for Practitioners

Structural augmentation is most valuable when: (1) the base agent already performs well (¿80% recall); (2) tasks involve cross-file dependencies or inheritance hierarchies; (3) trajectory stability matters as much as accuracy. It offers limited value when the base agent is weak (¡60% recall) or bugs are single-file, keyword-dense issues. Cost-wise, Section 4.3 shows Anchor-Topo adds ∼10% input tokens over Baseline on average; case studies show this is uneven: 27% of instances spend fewer input tokens under *CodeAnchor*. By halving run-to-run variance, deterministic anchors reduce the need for "best-of-N" sampling, potentially lowering the *total* cost to achieve a reliable outcome despite higher per-prompt usage.

A key practical advantage is tolerance for static analysis unsoundness. Python static analysis is notoriously difficult due to dynamic features, yet our results show strong gains even with conservative tools (PyCG). LLMs appear to utilize tags as *soft hints* rather than strict constraints: unlike a symbolic solver that breaks on a missing edge, an LLM agent uses available tags to bias its probabilistic search while falling back to lexical reasoning when structure is missing. This makes "imperfect structure" a viable and robust product feature for neural code agents.

Taken together, these three observations—*anchoring works*, *anchoring is scale-sensitive*, *anchoring stabilizes*—offer architectural principles for integrating static structure into grep-first code agents; Section 7 scopes the dimensions (agent architecture, language, fault depth) over which this transferability remains to be empirically validated.

7 Threats to Validity and Limitations

Internal Validity. Our implementation uses a specific agent loop and static analysis pipeline. We mitigate bias by fixing agent, prompt, model, and tools across configurations, varying only tag presence. The tagging pipeline is task-agnostic and does not use any ground-truth information.

External Validity. We evaluate on SWE-bench Lite and Verified with a single Codex-style agent. While these benchmarks stress cross-file reasoning, they don't cover all project types. Results reflect dominant grep-first retrieval rather than universal guarantees. Benchmark choice may also interact with dataset quality and evaluation protocols; recent analyses discuss these concerns for SWE-bench-style settings [4, 64]. Empirical validation is limited to Python; while our tag schema is language-agnostic and mature tools exist for Java/JavaScript/C++, multi-language effectiveness remains to be validated [66].

Static Analysis Unsoundness. CodeAnchor relies on conservative static analysis (PyCG [48] + AST extractors) for fast CI-friendly generation, prioritizing precision over recall [19, 46].

Consequently, the injected structure is *unsound*: it captures verified relationships but misses dynamic features (`getattr`, reflection, monkey patching). We quantified this limitation: fewer than 3% of ground-truth functions explicitly use dynamic dispatching constructs, and our extractors capture 94.2% of statically-visible imports and class hierarchies. Crucially, agents are resilient to this unsoundness: they utilize tags as probabilistic cues rather than absolute maps, benefiting from partial structure without being derailed by missing edges.

Evaluation Scope. Tags are intentionally searchable, coupling retrieval and navigation effects. While this precludes isolating post-landing value, our behavioral metrics (RQ3: LFR, trajectory patterns) demonstrate genuine navigational changes beyond first-hit effects. We evaluate localization only (not end-to-end repair) to isolate representation effects. Stability analysis uses a 50-task subset with k=10 runs; we validated representativeness via the Kolmogorov-Smirnov test ($p = 0.609 > 0.05$), confirming that findings generalize beyond sampled tasks.

Scope Dimensions Not Empirically Covered. Three caveats bound our claims. First, we evaluate only one grep-first agent (Codex). Tags are pure in-band text, so we expect that the effect will fix to the SOTA agent such as Claude Code, Cursor, and so on, but we have not tested this. Second, we only ran Python. The tag schema and agent integration are language-agnostic, and mature static-analysis tools exist for Java, C++, JavaScript, and Go that can populate the same tag fields, but effect sizes may still differ across languages. Third, SWE-bench Lite and Verified mostly contain 1–2 hop faults. We suspect deeper faults (multi-module refactoring, deep inheritance resolution) would benefit even more from tags, since tags offer guidance to the agent for multi-hop paths, while lexical search has to rebuild them one hop at a time, but confirming this requires a benchmark that sorts tasks by structural depth. These three are the main directions we leave for future work.

8 Related Work

LLM-based code agents. Tool-using agents [49, 62] have been adapted for code tasks, with systems like SWE-agent [61] and SWE-Gym [42] achieving strong results on repository-level benchmarks; related work explores alternative action spaces [56] and systematic evaluation [59, 64]. We complement these efforts by studying structural annotations within the dominant grep-first paradigm.

Graph-guided approaches. LocAgent [16] and RANGER [50] build explicit code graphs with specialized navigation operators, and RepoGraph [41] extends this line with repository-level graphs exposing call, inheritance, and data-dependency edges through dedicated APIs. Memory-augmented methods [63, 65] maintain persistent context across navigation steps. These approaches and *CodeAnchor* sit at different points: graph-guided agents externalize structure into a separate index that the agent queries through custom operators, while *CodeAnchor* embeds the same structural facts in-band as comments and lets the existing grep loop find them. The two designs trade expressiveness for accessibility. External graphs can encode richer queries (transitive closure, typed edges, path constraints); in-band tags need no new APIs and reuse the retrieval primitives that strong grep-first agents are already tuned for. Our Section 3 pilot (Table 1) suggests that, at current model strength, this accessibility advantage matters; we therefore treat *CodeAnchor* as complementary to graph-based retrieval, not a replacement, and see combining the two (e.g., tags as a fast-path with graph queries as fallback) as a natural extension.

Retrieval-augmented code agents. Beyond plain grep, another line of work strengthens retrieval with neural rankers, hybrid sparse-dense search, or embedding-based context

selection [3, 30, 67]. These methods improve lexical matching but still return independent snippets without explicit structural links. *CodeAnchor* is orthogonal: tags carry the structural edges that retrieval would otherwise have to reconstruct, and a tagged repository can be used with either plain grep or a neural retriever. We hold retrieval constant (plain grep) to isolate the marginal impact of static structure.

Fault localization and static analysis for LLMs. Traditional localization spans spectrum-based [1, 27], mutation-based [38, 43], and IR-based [47, 55, 68] approaches, with learning-based [33, 35] and LLM-based [60] extensions. Recent LLM-analysis pipelines include STALL+ [34], CodeT [13], and others [25], typically requiring specialized infrastructure. We instead augment text search with static structure encoded as comments.

Code representation for LLMs. Approaches include serialized ASTs [5], graph neural networks [23], structured prompting [67], and repository summaries (RepoMap [3]). Our plain-text tag injection is deliberately simpler, embedding structural facts as comments requiring no model or agent modifications.

9 Conclusion

We empirically studied how much static structure already-strong grep-first code agents need. Injecting varying granularities of structural annotations as plain-text comments, we identified the **deterministic anchoring effect**: static structure helps less by making agents smarter and more by making their navigation *disciplined and reproducible*. Concretely, lightweight topology improves localization (+2.2 pp Func@5, −1.6 rounds); optimal granularity/directionality depend on scale, with hub-heavy projects favoring inverse-only links; and tags roughly halve run-to-run variance on medium-scale repositories. Practical guidance: default to lightweight topology, prune forward edges in large hub-heavy repos, and reserve dense tags for implicit-dependency cases. Generalization to end-to-end repair, other agent architectures, and other languages remains open for future work.

Data Availability

Our code and results are available in https://anonymous.4open.science/r/CodeAnchor-Impl-8CA2.

References

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART-MUTATION)*. IEEE, 89–98.

[2] Hiralal Agrawal and Joseph R Horgan. 1990. Dynamic program slicing. *ACM SIGPlan Notices* 25, 6, 246–256.

[3] Aider AI. 2024. Aider RepoMap: Using a map of your codebase. https://aider.chat/docs/repomap.html. Accessed: 2024.

[4] Reem Aleithan. 2025. Revisiting SWE-Bench: On the Importance of Data Quality for LLM-Based Code Models. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 235–236. doi:10.1109/icse-companion66252.2025.00075

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. In *Proceedings of the ACM on Programming Languages*, Vol. 3. 1–29.

[6] Florian Angermeir, Maximilian Amougou, Mark Kreitz, Andreas Bauer, Matthias Linhuber, Davide Fucci, Daniel Mendez, Tony Gorschek, et al. 2025. Reflections on the Reproducibility of Commercial LLM Performance in Empirical Software Engineering Studies.

[7] Anthropic. 2025. Claude Code. https://code.claude.com/docs Agentic coding assistant with repository search and file inspection.

[8] Robert S Arnold. 1996. *Software change impact analysis*. IEEE Computer Society Press.

[9] Sebastian Baltes, Florian Angermeir, Chetan Arora, Marvin Muñoz Barón, Chunyang Chen, Lukas Böhme, Fabio Calefato, Neil Ernst, Davide Falessi, Brian Fitzgerald, Davide Fucci, Marcos Kalinowski, Stefano Lambiase, Daniel Russo, Mircea Lungu, Lutz Prechelt, Paul Ralph, Rijnard van Tonder, Christoph Treude, and Stefan Wagner. 2025. Guidelines for Empirical Studies in Software Engineering involving Large Language Models. arXiv:2508.15503 [cs.SE] Living resource available at https://llm-guidelines.org/.

[10] David Binkley. 1998. The application of program slicing to regression testing. *Information and software technology* 40, 11-12 (1998), 583–594.

[11] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*. 241–250.

[12] Markus Borg, Emil Alégroth, and Per Runeson. 2017. Software engineers' information seeking behavior in change impact analysis-an interview study. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 12–22.

[13] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397* (2022).

[14] Mark Chen. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[15] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021).

[16] Zhaoling Chen, Robert Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. 2025. LocAgent: Graph-Guided LLM Agents for Code Localization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Vienna, Austria, 8697–8727. doi:10.18653/v1/2025.acl-long.426

[17] WALA Contributors. 2024. WALA: T. J. Watson Libraries for Analysis. https://github.com/wala/WALA. Accessed: 2026.

[18] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages (POPL)*. ACM, 238–252. doi:10.1145/512950.512973

[19] Michael D Ernst. 2003. Static and dynamic analysis: synergy and duality. *WODA 2003: ICSE Workshop on Dynamic Analysis* (2003), 24–27.

[20] GitHub. 2024. GitHub Copilot Workspace. https://github.blog/news-insights/product-news/github-copilot-workspace/ Agentic workflow for repository-level tasks.

[21] Google. 2024. Gemini Code Assist. https://docs.cloud.google.com/gemini/docs/codeassist/overview Repository-aware coding assistant for IDEs and CI workflows.

[22] Alex Gyori, Shuvendu K Lahiri, and Nimrod Partush. 2017. Refining interprocedural change-impact analysis using equivalence relations. In *Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis*. 318–328.

[23] Vincent J Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2019. Global relational models of source code. In *International conference on learning representations*.

[24] Susan Horwitz, Thomas W. Reps, and David W. Binkley. 1990. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems* 12, 1 (1990), 26–60. doi:10.1145/77606.77608

[25] 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 Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–79.

[26] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues?. In *Proceedings of the 12th International Conference on Learning Representations (ICLR)*. https://www.swebench.com/ arXiv:2310.06770.

[27] James A Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*. ACM, 467–477.

[28] Maria Kretsou, Elvira-Maria Arvanitou, Apostolos Ampatzoglou, Ignatios Deligiannis, and Vassilis C Gerogiannis. 2021. Change impact analysis: A systematic mapping study. *Journal of systems and software* 174 (2021), 110892.

[29] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *2nd IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, 75–88. doi:10.1109/CGO.2004.1281665

[30] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems* 33, 9459–9474.

[31] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. 2013. A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability* 23, 8 (2013), 613–646.

[32] Fengjie Li, Jiajun Jiang, Jiajun Sun, and Hongyu Zhang. 2025. Hybrid Automated Program Repair by Combining Large Language Models and Program Analysis. *ACM Transactions on Software Engineering and Methodology* 34, 7, Article 202 (August 2025), 28 pages. doi:10.1145/3715004

[33] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. DeepFL: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 169–180.

[34] Junwei Liu, Yixuan Chen, Mingwei Liu, Xin Peng, and Yiling Lou. 2024. STALL+: Boosting LLM-based Repository-level Code Completion with Static Analysis. *arXiv preprint arXiv:2406.10018* (2024). arXiv:2406.10018 [cs.SE] https://arxiv.org/abs/2406.10018

[35] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 664–676.

[36] Microsoft. 2016. Language Server Protocol. https://microsoft.github.io/language-server-protocol/. Accessed: 2026.

[37] Microsoft. 2019. CodeQL: The libraries and queries that power security researchers around the world. https://github.com/github/codeql

[38] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 153–162.

[39] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774

[40] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. 2003. Leveraging field data for impact analysis and regression testing. *ACM SIGSOFT Software Engineering Notes* 28, 5, 128–137.

[41] Siru Ouyang, Wenhao Yun, Yu Zeng, Zonghai Yang, Meng Zhang, and Jiawei Han. 2025. RepoGraph: Enhancing AI Software Engineering with Repository-level Code Graph. *arXiv preprint arXiv:2410.14684* (2025).

[42] Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2025. Training Software Engineering Agents and Verifiers with SWE-Gym. In *Forty-second International Conference on Machine Learning (ICML)*. OpenReview.net. arXiv:2412.21139 [cs.SE] https://openreview.net/forum?id=Cq1BNvHx74

[43] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: Mutation-based fault localization. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 605–628.

[44] r2c. 2020. Semgrep: Lightweight static analysis for many languages. https://semgrep.dev/

[45] Stephen E. Robertson and Hugo Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Foundations and Trends in Information Retrieval* 3, 4 (2009), 333–389. doi:10.1561/1500000019

[46] Barbara G Ryder. 2003. Dimensions of precision in reference analysis of object-oriented programming languages. In *International Conference on Compiler Construction*. Springer, 126–137.

[47] Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. 2013. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, Ewen Denney, Tevfik Bultan, and Andreas Zeller (Eds.). IEEE, 345–355. doi:10.1109/ASE.2013.6693093

[48] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. 2021. PyCG: Practical Call Graph Generation in Python. In *43rd IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 1646–1657. doi:10.1109/ICSE43902.2021.00146 arXiv:2103.00587.

[49] Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. In *Advances in Neural Information Processing Systems 36 (NeurIPS)*. http://papers.nips.cc/paper_files/paper/2023/hash/d842425e4bf79ba039352da0f658a906-Abstract-Conference.html arXiv:2302.04761.

[50] Pratik Shah, Rajat Ghosh, Aryan Singhal, and Debojyoti Dutta. 2025. RANGER – Repository-Level Agent for Graph-Enhanced Retrieval. *arXiv preprint arXiv:2509.25257* (2025). arXiv:2509.25257 [cs.SE] https://arxiv.org/abs/2509.25257

[51] Yifan Song, Da Yin, Xiang Yue, Jie Huang, Sujian Li, and Bill Yuchen Lin. 2024. Trial and Error: Exploration-Based Trajectory Optimization for LLM Agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Bangkok, Thailand, 7584–7600. https://aclanthology.org/2024.acl-long.409

[52] Frank Tip. 1994. A survey of program slicing techniques. (1994).

[53] Tree-sitter Contributors. 2018. Tree-sitter: An incremental parsing system for programming tools. https://github.com/tree-sitter/tree-sitter. Accessed: 2024.

[54] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. 214–224.

[55] Shaowei Wang and David Lo. 2015. Amalgam+: Composing rich information sources for accurate bug localization. In *Journal of Software: Evolution and Process*, Vol. 27. Wiley, 921–942.

[56] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable code actions elicit better llm agents. In *Forty-first International Conference on Machine Learning*.

[57] Mark Weiser. 1984. Program Slicing . *IEEE Transactions on Software Engineering* 10, 04 (July 1984), 352–357. doi:10.1109/TSE.1984.5010248

[58] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 959–971. doi:10.1145/3540250.3549101 arXiv:2207.08281.

[59] Frank F. Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Z. Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, Mingyang Yang, Hao Yang Lu, Amaad Martin, Zhe Su, Leander Maben, Raj Mehta, Wayne Chi, Lawrence Jang, Yiqing Xie, Shuyan Zhou, and Graham Neubig. 2024. TheAgentCompany: Benchmarking LLM Agents on Consequential Real World Tasks. arXiv:2412.14161 [cs.AI]

[60] Aidan Z.H. Yang, Claire Le Goues, Ruben Martins, and Vincent J. Hellendoorn. 2024. Large Language Models for Test-Free Fault Localization. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. 165–176. doi:10.1145/3597503.3623342

[61] 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. In *Advances in Neural Information Processing Systems 38 (NeurIPS)*. http://papers.nips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html arXiv:2405.15793.

[62] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*.

[63] Inseok Yeo, Duksan Ryu, and Jongmoon Baik. 2025. Improving LLM-Based Fault Localization with External Memory and Project Context. arXiv:2506.03585 [cs.SE]

[64] Boxi Yu, Yuxuan Zhu, Pinjia He, and Daniel Kang. 2025. UTBoost: Rigorous Evaluation of Coding Agents on SWE-Bench. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Vienna, Austria, 3762–3774. doi:10.18653/v1/2025.acl-long.189

[65] Zhongming Yu, Hejia Zhang, Yujie Zhao, Hanxian Huang, Matrix Yao, Ke Ding, and Jishen Zhao. 2025. OrcaLoca: An LLM Agent Framework for Software Issue Localization. arXiv:2502.00350 [cs.SE] To appear at ICML 2025.

[66] Daoguang Zan, Zhirong Huang, Ailun Yu, Shaoxin Lin, Yifan Shi, Wei Liu, Dong Chen, Zongshuai Qi, Hao Yu, Lei Yu, Dezhi Ran, Muhan Zeng, Bo Shen, Guangtai Liang, Bei Guan, Pengjie Huang, Tao

Xie, and Yongji Wang. 2024. SWE-bench-java: A GitHub Issue Resolving Benchmark for Java. *arXiv preprint arXiv:2408.14354* (2024).

[67] 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. *arXiv preprint arXiv:2303.12570* (2023).

[68] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE Computer Society, 14–24. doi:10.1109/ICSE.2012.6227210