The Advantage of Micro-Modular Architecture May Be Further Amplified in Vibe Coding

Wu Junchen Dr.DS

2026-08-27

Contents

1 Introduction **2**

2 The Structure of the Problem **2**
2.1 The "blast radius" of a change is opaque . . . 3
2.2 Understanding of the code drifts . . . 3
2.3 No "did I get it right?" verification backstop . . . 3
2.4 The three compound into "getting worse over time" . . . 3

3 The Position: Deriving the Strategy from the Attribution **3**
3.1 Attribution → Strategy: three problems, three necessary mechanisms . . . 3
3.2 A key distinction: we mean "spec-like documentation," not "a real spec" . . . 4
3.3 Why "module-level + spec-like + verification," not "retrieval / indexing / static documentation" . . . 5
3.4 Why "moderately cohesive, dynamic," not "finer is better" . . . 5
3.5 Why "external-scaffold maintainability," not "rely on LLMs' natural modularity preference" . . . 6

4 Related Work: Overlap, Difference, and the Overlooked Empty Setting **6**
4.1 The overlapping parts (existing work) . . . 6
4.2 The most fundamental difference . . . 7
4.3 Counter-evidence we must confront . . . 7

5 Yanxi: An Instantiation (PoC) **8**
5.1 Lazy loading . . . 8
5.2 Spec-like freshness (aspect ②) . . . 8
5.3 Module boundary and verification gate (aspects ①③) . . . 8
5.4 Design and known limits . . . 8

6 Evidence (Balanced by "Attribution → Strategy") **8**
6.1 Supporting the attribution (firm) . . . 8
6.2 Supporting the strategy (partly firm, partly inferred) . . . 9
6.3 Counter-evidence (we admit) . . . 9
6.4 People gap . . . 9

7 Where We Are Most Likely Wrong (Self-Critique) **9**

8 Alternative Views **10**

9 Research Agenda (H1–H5) **10**

10 Conclusion 11

Abstract

We argue that, for creative projects that continuously evolve under Vibe Coding, **module-level spec-like documentation (maintained in real time by the Agent and kept fresh as the code evolves), together with atomic micro-modules and module-level verification**, may support an Agent in making repeated, safe changes better than existing retrieval, indexing, or static documentation. We do *not* claim that "modularity is beneficial in every setting"—evidence shows the granularity–performance relationship is non-monotonic, static decomposition can even be harmful, and LLMs are in fact "neutral" about modularity. We restrict our claim to the *specific, overlooked setting* of "creative projects + repeated Agent modification + module-level spec-like freshness." We use the concept-proof tool Yanxi (a salt-extraction tool) as a testable prototype and give five focused research hypotheses (H1–H5). The core value lies in identifying this overlooked setting and converging the evidence through an H1–H5 testable agenda.

1 Introduction

Vibe Coding is spreading rapidly. Its appeal is low friction: using fairly simple natural language to guide a model to generate and modify code. But it has a repeatedly verified weakness—**an Agent cannot reliably understand a continuously evolving codebase, leading to incomplete changes, missed modification points, and even breaking previously correct functionality** [1, 3, 4]. When you are pushing forward a very creative project, having your intentions repeatedly misinterpreted by the agent is absolutely maddening. Documentation is part of the problem, but I believe code structure also plays a role.

Our claim: We focus on an overlooked setting—**a creative, continuously evolving vibe-coding project, repeatedly modified by an Agent**. In this setting, **module-level spec-like documentation (maintained in real time by the Agent, kept fresh as the code evolves) + atomic micro-modules + module-level verification** may support the Agent in making safe, non-omitting changes better than existing "retrieval, indexing, static documentation."

We do not claim "modularity is better everywhere." What we really want to emphasize is exploring how to improve the Agent's ability to maintain a project from the level of project structure. We also think this claim may have longer-term significance: as vibe coding becomes widespread, software is no longer a frozen artifact after release, but **a continuously evolving object co-maintained by humans and Agents**—software "customization" may shift from **fixed plugin mechanisms** to **being continuously modifiable by non-expert users (highly customizable via vibe)**. In this state, "how to let an Agent safely change code" is no longer a tool-optimization problem but a **core infrastructure problem**—and the value of modularity will be amplified in this "continuous evolution" setting.

The rest of this paper is organized as follows. Section 2 attributes the problem; Section 3 derives the strategy from the attribution, explaining why it differs from retrieval/indexing approaches, why it is "moderately cohesive" rather than "finer is better," and why maintainability depends on "external scaffolding"; Section 4 surveys overlaps with and differences from prior work (including micro-skills, memory granularity, and spec-forwarding); Section 5 instantiates the claim as Yanxi; Section 6 presents evidence in balance; Section 7 attacks ourselves; Section 8 responds to objections; Section 9 gives a testable agenda; Section 10 concludes.

2 The Structure of the Problem

To argue for the strategy, we must first attribute the problem clearly. When an Agent repeatedly modifies a continuously evolving vibe project, failure concentrates in three separable, *mutually*

compounding aspects.

2.1 The "blast radius" of a change is opaque

An Agent changes one place but does not know which other modules it affects—either missing downstream modules that should also be updated, or unintentionally breaking unrelated functionality. Failure-attribution research on code Agents points out: the most error-prone part is "understanding and strategy," not "locating the relevant files" [3]. That is, **the problem is not "finding the file," but "judging where a change propagates."**

An intuitive explanation: the Agent faces a huge code network **without explicit boundaries**. It can see a function, a file, but not "if I change this, what else must change." This is like a relationship **without a contract**—without a computable structure of "what this unit does, what it depends on, who depends on it," the blast radius is forever fuzzy.

2.2 Understanding of the code drifts

A module not touched for many rounds of iteration: the Agent's understanding of it (what it does, interface, dependencies) is already outdated. Documentation fails as code evolves: in the Linux kernel alone there are **869 comment references left unsynchronized by code evolution** [8]; LLM-generated specs are often "incomplete, unhelpful, and factually incorrect" [9], even superficially credible yet internally contradictory [10].

Once understanding drifts, the Agent next **modifies based on an outdated model**—a more hidden and more dangerous failure than "not finding the file": it appears to be acting, but is acting within a stale cognitive frame.

2.3 No "did I get it right?" verification backstop

Even if the first two aspects fail, a mechanism that quickly verifies "does this change break anything?" could catch most regressions. But vibe coding precisely lacks this—non-programmers, even when told AI often errs, still miss key defects [11]; and the "perception–action gap" shows perception of risk is similar across groups but **verification ability diverges sharply with experience** [12].

2.4 The three compound into "getting worse over time"

- opaque blast radius ⇒ changes easily miss/break things;
- drift in understanding ⇒ changes are based on wrong premises;
- no verification ⇒ errors are let through.

The three are not parallel but **compounding**. This explains why a creative project becomes harder to maintain the further it goes—not because of one wrong change, but because these three structural failures recur.

3 The Position: Deriving the Strategy from the Attribution

3.1 Attribution → Strategy: three problems, three necessary mechanisms

Following the attribution, **each aspect corresponds to a mechanism that is necessary (not optional)**:

Necessity argument: these three are not "icing on the cake" but **mechanisms that map one-to-one to eliminate the three failures**. Only with clear boundaries can the blast radius

Attribution	Necessary mechanism	Why
opaque blast radius	clear, verifiable module boundaries	makes "where a change affects" computable and checkable
drift in understanding	spec-like documentation maintained in real time and kept fresh	keeps "what the code is now" readable, not a stale snapshot
no verification	module-level verification (dev cycle)	makes "did I get it right?" checkable and blocks regressions

Table 1: Attribution–strategy correspondence

be computed; only with fresh spec-like documentation does understanding not drift; only with a verification gate are errors not silently let through. Missing any one leaves that failure class structurally unpreventable.

Note that **"module-level verification" execution depends on an external test toolchain (e.g. pytest); Yanxi provides the evidence gate and state gate**—this is a deliberate PoC boundary; the claim is not weakened by the tool not itself running tests.

3.2 A key distinction: we mean "spec-like documentation," not "a real spec"

Before going further, we must draw a line running through the paper: what we propose is *not* the traditional formal requirements spec written **by a human**, but a **spec-like, recording-natured, near-spec Agent note maintained and kept fresh by the Agent** (spec-like: a structured description maintained by the Agent, not a formal requirements spec).

- **A real spec (human-written)**: the spec-kit [7], "spec-first" approach—first have a human write a formal spec, then drive the Agent to implement it. Its burden is on the human and it is one-time.
- **Spec-like documentation (Agent-written)**: what we mean—**the Agent precipitates its own understanding into a structured note**, kept fresh in real time as the code evolves. Its burden is on the Agent and it is continuous.

This distinction matters. We do *not* oppose "spec-forwarding"—a human writing a formal spec first is **desirable, a good starting point**; but it is often "write once, feed the Agent once." **What is more central and primary is that the Agent *internalizes* this spec (or requirement) into its own spec-like documentation and keeps it fresh as it evolves**—because what truly determines whether the Agent can reliably modify code later is not "a spec that was fed in" but "the continuously fresh self-understanding in the Agent's head."

As such, **spec-like documentation is inherently "unreliable"**—it is Agent-written. But the literature shows **the Agent does not "write documentation badly"**: LLM-generated comment quality is **overall not worse than human** (about 58.8% equivalent, 27.7% better) [34], and module-level TDD verification has been shown to improve LLMs [33]; the real risk is **drift and hallucination**—naive LLM documentation can have a **false-positive rate above 90%** [35], and LLM updating existing documentation often hallucinates [36]. So the conclusion is not "the Agent writes poorly," but "**it can write it but it is unreliable and rots with evolution, hence must be supported by freshness + verification mechanisms**". This in fact strengthens our claim: **spec-like documentation cannot rely on "write once," it must rely on "Agent real-time maintenance + freshness mechanism + module-level verification."**

3.3 Why "module-level + spec-like + verification," not "retrieval / indexing / static documentation"

This is the most fundamental divide from existing tools and needs to be clearer. Retrieval (repo map [6], RAG), cognitive memory (like Mnemosyne), static documentation (AGENTS.md [5]), spec-first (spec-kit [7])—all answer "**how to feed information to the Agent**" (feeding relevant fragments at some moment). But against the attribution:

- They **cannot solve the blast radius**: the retrieved fragment is **boundary-less**—it helps you "see relevant code," but not "if I change this, where does it propagate."
- They **cannot solve drift**: retrieval indexes, static docs, one-shot snapshots **expire** and do not stay fresh with evolution—the essence of drift.
- Their **verification granularity is too coarse**: of course they can be paired with mature verification tools (CI, test suites, static analysis), but those are **project-level / large-module-level** coarse-grained—slow to run, hard to locate "exactly which piece was changed"; our difference is **module-level verification**: *with each change, do our best to ensure no error is introduced*—link-verifying the affected downstream via dependency semantics (**cascading verification**), rather than only testing the changed module; regressions locate quickly.

In other words: **retrieval/indexing is "read-time supply," aimed at "how to feed the code to the Agent"; we address "at coding time, shaping the code into a computable, fresh, verifiable object."** They, too, can be paired with verification tools, but those are mostly **coarse-grained** project-level; our difference is **module-level (cascading) verification**—with each change, do our best to ensure no error, with module boundaries and spec-like documentation keeping the blast radius controllable and regressions locating quickly.

This distinction is echoed by recent evidence♦ (cross-domain analogy: we reference evidence from memory, skill, and tool-design to support "code structuring" direction; marked throughout with ♦ as cross-domain analogy): treating "tools/interfaces given to the Agent" as **contracts** (strict data model for in/out + explicit description), only tuning tool descriptions made Claude Sonnet 4.5 reach SOTA on SWE-bench Verified [29]; organizing knowledge/context into **manageable segments** (system-prompt segmentation, sub-Agents returning only 1–2k condensed results) is the core of "context engineering" [30]. These point to the same thing: **structured, bounded, traceable "contract/documentation" is more effective than flat information**. And this is evidence for our "module + spec-like + verification."

3.4 Why "moderately cohesive, dynamic," not "finer is better"

A natural objection: "you say modularize, so is finer always better?" **No.** Three pieces of evidence point to "moderate":

- granularity–performance is **non-monotonic**—file-level localization is the dominant factor (15–17× over a no-file baseline), line-level expansion often degrades from noise amplification, and about 6–10 relevant files is most often successful [15];
- **static/fixed decomposition is even worse than monolithic**, only re-running failed subtasks at runtime helps [16];
- LLMs are **neutral** about modularity—perplexity shows almost no preference for modular vs. non-modular, because pretraining saw both [14].

So our module boundary is **cohesive (same responsibility) + dynamically adjustable** (grow, merge, split, as evolution changes); granularity is "**good enough, maintainable**." We

treat it as a **direction**, not a tunable metric—a key constraint of the whole position and the aspect we admit needs the most empirical support (H1).

This "not finer but intermediate granularity + atomic verifiable + layered on-demand" judgment has adjacent evidence♦ (cross-domain analogy): aligning memory granularity to functionally decomposed subtasks (rather than whole instances) improved SWE-bench Verified Pass@1 by +**4.7pp** [23]; in memory distillation, **intermediate granularity (subtask) memory gave the largest gain**, and the finest function-level memory was fetched reactively only on error [24]; changing skills from "flat" to "progressive disclosure (short root file + fetch resources on demand)" raised resource touches from 1.18 to 3.85, effective adoption events from 1.33 to 3.92, and added 17 passed trials (+4.1%) [25]. These point to a "sweet spot": **too coarse mismatches, too fine dilutes; the optimum is "intermediate granularity + atomic verifiable + on-demand/layered loading."**

3.5 Why "external-scaffold maintainability," not "rely on LLMs' natural modularity preference"

Finally a grounding point. Since LLMs do not naturally prefer modularity [14], the goal of "let the Agent write more modular code" likely fails—**you cannot count on the Agent to spontaneously write maintainable structure**. What we mean is not "let the Agent love modularity," but:

> **Generating hard-to-maintain code is worse than not generating it** (here "hard-to-maintain" means changes cause regression rate or repair cost to rise). Since "maintainability" cannot be guaranteed by the LLM's natural preference, turn it into a **structured, fresh, verifiable guarantee**—i.e. "module boundary + spec-like documentation + verification gate." Let maintainability become something the tool **underpins**, rather than depending on the model's inclination.

This judgment has adjacent evidence♦ (cross-domain analogy, from skill/knowledge compression): splitting skills/knowledge and removing non-essential content (compress descriptions 48%, body 39%) *improved* functional quality +**2.8%** (less-is-more, 0.965 retention across 5 models) [26]; conversely, **self-generated skills can hurt performance** [27]. This shows "maintainable/usable" guarantee cannot rest on the Agent's "natural tendency," but on **deliberate minimization, structurization, and validation**.

4 Related Work: Overlap, Difference, and the Overlooked Empty Setting

In several directions our claim **overlaps** with existing work—and this **is actually good**: it shows the direction we point to is not fantasy but has a consensus basis. What we want to highlight is not "replacing" these works, but **focusing, on that consensus, on one overlooked setting** (module-level spec-like freshness + module-level fine-grained verification for continuously evolving projects).

4.1 The overlapping parts (existing work)

- **spec-first / spec-driven**: spec-kit [7], Vibe Reasoning [17], Constitutional SDD (writing non-negotiable principles into a machine-readable "constitution"; bank microservice case cut safety defects by **73%**) [20]. Difference: these are mostly human-written *formal* specs and mostly one-time generated.

- **modularity + docs-as-blueprint + verification**: Repo0 (generating a repo from scratch) [18], RustPrint (C→Rust migration) [19]. Difference: oriented to "from scratch" or "specific migration," not "multi-round iteration on an existing project."
- **micro-skill / Agent Skills granularization**: Agent Skills are the formal paradigm of "composable instruction+code+resource package, loaded on demand" [21]; empirical work (GitSkills found 3.8M SKILL.md) shows huge real-world practice [22]. Difference: skill granularization targets "capability/tools," not "understanding and freshness of a project's internal code modules."
- **memory/document granularization**: subtask-level memory [23], multi-granularity memory pyramid [32]. Difference: targets "cross-conversation memory/retrieval," not "a single project's code-structure freshness."
- **fine-grained contract → verifiability**: line-by-line fine-grained reference identifiers buy **automatic hallucination detection** (TDR 86.4%, GLM 88.0%, others 0%, FPR 0%) [28]. This gives quantitative support for "atomized + independently verifiable."

4.2 The most fundamental difference

Existing work mostly tests "**from scratch**" "**specific migration**" or "**capability/memory**"; our setting is: **a creative project, grown by an Agent from a prototype, iterated many rounds, needing to resist decay**. That setting's controlled empirical evidence is **currently missing**—it is what we consider most distinctive and most worth testing.

4.3 Counter-evidence we must confront

Counter-evidence	Conclusion	Implication for us
Revisiting Modularity [14]	perplexity shows LLM "neutral" to modularity; modularity not core to generation performance; human vs. LLM view differ	only refutes "modularity makes generation better," not "modularity makes an existing project maintainable and re-modifiable"
Fault Localization [15]	61 granularities, non-monotonic; file dominant, line harmful	do not claim "finer better"; moderate, cohesive
Runtime-Structured [16]	static decomposition worse than monolithic	admit fixed boundaries are harmful; hence need dynamic boundary adjustment
Skill-Use [37]	best config score only 0.613 on 79 real skills; trigger and compliance are independent bottlenecks	having module-level spec is not enough; must ensure Agent reliably retrieves and strictly complies
Progressive Disclosure [38]	"progressive disclosure buys context, not intelligence"	granularization gains mainly context-saving/precision; do not overpromise "smarter"

Table 2: Counter-evidence we must confront

5 Yanxi: An Instantiation (PoC)

To show feasibility, we built a concept-proof tool **Yanxi (a salt-extraction tool)**—a zero-dependency, single-file MCP server that manages atomic micro-modules and their **spec-like** documentation (explain) via 14 tools; repository: https://github.com/The-Milky-Way-traveller/yanxi-single.

5.1 Lazy loading

`module_discover()` returns a project map in about 200 tokens → module summaries → `module_read()` reads on demand (explain / summary / interface / full), enough is enough.

5.2 Spec-like freshness (aspect ②)

`AutoDegrade` (source mtime changed → auto-degrade + counter), `module_read`'s `ExplainStale`, discover's threshold warning (explain/depend behind), `CheckStableGate` (source newer than last pass → refuse stable). It makes "understanding freshness" an auto-detectable, refluxable mechanism—this is how "spec-like freshness" rather than "stale snapshot" is realized.

This module-level organization with spec-like documentation also **naturally suits multi-Agent collaboration**: each module is an isolatable context unit, Agents focus on their own, sharing spec-like docs as a common contract, making integration and conflicts more controllable.

5.3 Module boundary and verification gate (aspects ①③)

State machine `wip`→`testing`→`stable`→`frozen`→`deprecated`→`migrated`→`archived`; `module_validate` records test evidence and auto-marks broken on failure; `CheckStableGate` is a hard gate for stable (first_fail + latest_pass + downstream + mtime freshness); `module_merge/extract` support **dynamic boundary adjustment**.

5.4 Design and known limits

1. Yanxi does not run tests, only records evidence (design decision)—"verification" means "evidence gate," not replacing tests; there is a "spec-gaming" space.
2. "Homogeneity" is "advisory record," not enforced schema—consistency by convention, not machine check.
3. Not worth it for small projects (below 3 modules, write code directly is faster).
4. Not yet self-hosted (Yanxi hasn't managed itself).

6 Evidence (Balanced by "Attribution → Strategy")

As a position paper, we do not claim completed controlled experiments; we present evidence in balance, distinguishing "firm" from "inferred."

6.1 Supporting the attribution (firm)

Most error-prone in "understanding and strategy" not localization [3]; "confidently wrong / silent failure" dominant [4]; long-context unreliable [1, 2]; docs/comments drift with evolution (869 unsynchronized [8], spec factually wrong [9, 10]).

6.2 Supporting the strategy (partly firm, partly inferred)

- **structured, bounded context beats flat** (adjacent evidence, incl. cross-domain): chunk-level beats file-level by +6%, +16% over no-context [31]; fine-grained contract → verifiability [28]; tools/knowledge as contract or segmented [29, 30].
- **intermediate granularity + atomic verifiable + layered on-demand is the sweet spot**: subtask memory +4.7pp [23]; intermediate granularity most contributes [24]; progressive disclosure +4.1% pass [25]; less-is-more +2.8% [26].
- **module-level verification / dev cycle**: module/class-level TDD verification improves LLMs [33].
- **freshness people** (inferred, to be tested by H4/H2): non-programmers miss defects [11]; perception–action gap [12].

6.3 Counter-evidence (we admit)

See Table 2 in Section 4. Here only the core sentence: **modularity is not core to generation performance** [14], but its "why" (LLM neutral, human vs. LLM view differ) **precisely shows "maintainability needs external scaffolding"**—the grounding point of Section 3.5.

6.4 People gap

Vibe coding does not remove the need for experience; it redistributes skills into "context management, quick code evaluation, when to switch AI/manual" [13], and the perception–action gap is large [12]. The middle group lacks exactly "context management + verification"—which is what our claim aims to **take over**.

7 Where We Are Most Likely Wrong (Self-Critique)

Critique 1: physical boundary ≠ no omission. Clear boundaries may make the Agent think "only this module" and ignore cross-module coupling; finer modules = more interfaces = larger blast radius. **Response:** so boundaries cannot rely only on "physical division"; must pair with **dependency semantics** and **module-level verification** to cover the blast radius. Boundary is the start of "help locate + activate verification," not a guarantee of "no omission"—the most empirically needed aspect (H1).

Critique 2: the Agent maintains spec-like docs, but Agent-written specs are unreliable (circular). Freshness with what is unreliable—contradiction? **Response:** precisely why freshness must rely on **mechanism** (AutoDegrade + ExplainStale + gate), not "write once." And nuance: LLM comment quality is **overall not worse than human** [34]; drift stems from "maintenance rots," not "LLM-specific defect"—so **whoever maintains it needs freshness mechanism**, which strengthens our claim.

Critique 3: the middle group is precisely the least likely to configure MCP / run validate. Tool barrier vs. target group seem contradictory. **Response:** precisely because they can't/won't do verification, they more need **the tool to do it for them** (delegate verification, automate freshness). But "whether the middle group will adopt this flow" is unverified (H3); if it fails, the applicable group shifts up to "semi-professional."

> Other boundaries (granularity not quantified, weak direct evidence, cognitive analogy unverified, Yanxi not self-hosted, validate not running tests) are addressed in Section 10.

8 Alternative Views

- **"repo map: I can give boundaries without splitting"**—repo map is a "read-time" temporary boundary, one-shot, expires; we propose "write-time" fixation + freshness + verification.
- **"RAG / cognitive memory: we also on-demand, also remember"**—they record "what happened" (semantic approximation, boundary-less, drifts, no verification); we record "what the code is now / where the change propagates / has it been verified" (structure, freshness, verification, bounded).
- **"long context keeps improving"**—full-repo strict pass still $< 40\%$ [1]; the main failure is understanding/strategy [3]; we want not "short" but "controllable boundary + freshness."
- **"small modules have higher defect density" (Basili [39])**—1984 human data; LLM failure modes differ; we claim "moderate granularity + independently verifiable," never unlimited splitting.
- **"RCT conflict: AI doesn't always improve efficiency"**—all only test professional developers and results vary by task type; the middle group is a blank (H3).
- **"norms kill flow and emergent creativity"**—agree over-norms kill exploration; propose "gradual norms" (first vibe-explore, then solidify on demand), guard rails can be introduced asynchronously.

9 Research Agenda (H1–H5)

H1 (granularity ↔ maintenance correctness; core; three-arm comparison) Under the same "continuously evolving project + several rounds of Agent modification," compare **over-fine / moderately cohesive / monolithic**, measuring omission, unintended-breakage, correctness. "Moderate" uses a rough interval: module interfaces $\leq$ N, fan-out $\leq$ M, size about 100–300 lines; report each arm's granularity distribution. Expect **moderately cohesive** best.

H2 (verification gate ↔ regression) module-level TDD/freeze gate vs. bare change; measure regressions.

H3 (people heterogeneity) middle group vs. professional developers, gain difference; also measure "whether the middle group will adopt this flow."

H4 (freshness ↔ misinterpretation; most priority) after N rounds of change, "real-time module-level spec-like docs" vs. "static AGENTS.md / one-shot repo map"; measure Agent misinterpretation of new requirements.

H5 (spec-like reading / compliance) measure whether the Agent actively reads and complies with explain—with/without "forced reach" (e.g. must read relevant module explain before writing); because Table 2 already hints "having spec is not enough."

H1 and H4 are the most distinctive and most able to test "will [14] break us"—they directly test "in that setting, does moderate granularity + spec-like freshness help," rather than broadly claiming "modularity is beneficial."

10 Conclusion

We argue: for creative, continuously evolving vibe projects, "module-level spec-like documentation (real-time, evolution-fresh) + atomic micro-modules + module-level verification" may support the Agent in repeatedly, safely changing code better than retrieval, indexing, or static docs. We gave a complete derivation from "attribution → strategy" (opaque blast radius → module boundary; drift → spec-like freshness; no verification → module-level verification gate), explained why "module-level + spec-like + verification" rather than retrieval/indexing, why moderately cohesive rather than finer-is-better, and why maintainability needs "external scaffolding" (generating hard-to-maintain code is worse than not generating). The concept-proof tool Yanxi realizes one layer. We leave the H1–H5 agenda; H1 and H4 are most worth prioritizing.

We admit several limitations: no controlled experiments, granularity not quantified, cognitive analogy not strictly verified, Yanxi is only a PoC and validate does not run tests. **Even if specific hypotheses are ultimately weakened by counter-evidence, we claim "in continuously evolving projects, using module-level spec-like freshness + module-level verification to support the Agent's repeated code changes" is a direction worth taking seriously.**

Acknowledgements

I sincerely want to express gratitude. As a high-school student, I never imagined finishing a fairly formal paper this summer. This project began as a simple idea—making vibe coding easier. In the process of researching, discussing, and advancing again and again with Agents, I gradually realized: we really could make something. Yanxi was realized as a concept proof; systematic empirical work is left to H1–H5; although crude, this journey enriched me a lot. Thanks to this era, to aixiv (an AI co-creation preprint platform) for giving a high-school student a venue to publish, and to DeepSeek and every Agent who walked with me along the way, for making this summer a little different.

References

[1] Hong, Ascoli, & Choi. 2026. *ReCUBE: Evaluating Repository-Level Context Utilization in Code Generation.* arXiv:2603.25770.

[2] Liu et al. 2023. *Lost in the Middle: How Language Models Use Long Contexts.* arXiv:2307.03172.

[3] Jiang, Huang, Wang, Chen, Liu, & Briand. 2026. *Characterizing the Failure Modes of LLMs in Resolving Real-World GitHub Issues.* arXiv:2605.12270.

[4] Mehta. 2026. *Confident and Wrong: Silent Semantic Failures in Coding Agents.* arXiv:2603.25764.

[5] AGENTS.md. https://agents.md/.

[6] RepoGraph, 2024. arXiv:2410.14684; Aider *Repo Map*.

[7] GitHub Spec Kit. https://github.com/github/spec-kit.

[8] Sun et al. 2026. *ReCite / We Must Have Missed This Comment.* ASE 2026. arXiv:2608.03734.

[9] Yang et al. 2025. *DocAgent.* ACL 2025. arXiv:2504.08725.

[10] Mattukat et al. 2026. *Can ChatGPT Generate Realistic Synthetic System Requirement Specifications?* ENASE 2026. arXiv:2603.09335.

[11] Virk & Liu. 2025. *Non-programmers Assessing AI-Generated Code.* VL/HCC 2025. arXiv:2508.06484.

[12] Fawzy, Tahir, Blincoe. 2026. *From Prompting to Verification.* arXiv:2605.24521.

[13] Sarkar & Drosos. 2025. *Vibe Coding.* arXiv:2506.23253.

[14] Kang, Seo, & Kim. 2024. *Revisiting the Impact of Pursuing Modularity for Code Generation.* EMNLP 2024 Findings. arXiv:2407.11406.

[15] Sepidband, Pham, & Hemmati. 2026. *On the Role of Fault Localization Context for LLM-Based Program Repair.* arXiv:2604.05481.

[16] Asthana et al. 2026. *Runtime-Structured Task Decomposition for Agentic Coding Systems.* arXiv:2605.15425.

[17] Mitchell & Shaaban. 2025. *Position: Vibe Coding Needs Vibe Reasoning.* arXiv:2511.00202.

[18] Repo0, 2026. arXiv:2608.19854.

[19] RustPrint, 2026. arXiv:2605.14634.

[20] Marri. 2026. *Constitutional Spec-Driven Development.* arXiv:2602.02584.

[21] Xu & Yan. 2026. *Agent Skills for LLMs.* arXiv:2602.12430.

[22] Destefanis et al. 2026. *GitSkills: A Dataset of Agent Skills on GitHub.* arXiv:2608.10906.

[23] Shen et al. 2026. *Structurally Aligned Subtask-Level Memory for SWE Agents.* arXiv:2602.21611.

[24] Kim, Kim, & Hwang. 2026. *Agent Memory Distillation.* arXiv:2608.07169.

[25] Chen et al. 2026. *SkillJuror: Measuring How Agent Skill Organization Changes Runtime Behavior.* arXiv:2606.11543.

[26] Gao et al. 2026. *SkillReducer: Optimizing LLM Agent Skills for Token Efficiency.* arXiv:2603.29919.

[27] Jiang et al. 2026. *SoK: Agentic Skills – Beyond Tool Use.* arXiv:2602.20867.

[28] Panda. 2026. *Citation Discipline in Spec-Driven Development.* arXiv:2606.30689.

[29] Anthropic. 2025. *Writing effective tools for AI agents.*

[30] Anthropic. 2025. *Effective context engineering for AI agents.*

[31] Yusuf, Caumartin, & Costa. 2025. *Beyond More Context: Granularity and Order Drive Code Completion Quality.* ASE 2025. arXiv:2510.06606.

[32] Xu et al. 2026. *From Passive Retrieval to Active Memory Navigation.* arXiv:2607.05794.

[33] 2026. *Scaling TDD Functions to Classes.* arXiv:2602.03557.

[34] *On the Quality of AI-Generated Source Code Comments: A Comprehensive Evaluation.* arXiv:2408.14007.

[35] 2025. *DocPrism.* arXiv:2511.00215.

[36] 2026. *DocSync.* arXiv:2605.02163.

[37] Han et al. 2026. *Skill-Use: Can LLMs Actually Use Skills in Agentic Harnesses?* arXiv:2608.04828.

[38] He et al. 2026. *Is Progressive Disclosure All You Need for Long-Context Agents?* arXiv:2607.17598.

[39] Basili & Perricone. 1984. *Software Errors and Complexity.* CACM.