CONTEXT SYNCHRONIZATION FOR MULTI-AGENT WORKFLOWS

Ryan Baker

ABSTRACT

Large language model (LLM) agents have demonstrated remarkable capability as autonomous coding assistants, motivating workflows in which multiple agents operate concurrently on a shared codebase. However, existing frameworks treat each agent's context as private and static: agents reason from symbols, signatures, and dependencies observed at task start, unaware when concurrent edits have invalidated those assumptions. We term this failure mode *context drift*. We present *CodeCtx*, a shared context layer for multi-agent coding workflows. CodeCtx constructs a symbol-level dependency graph over a Python repository, records each agent's observations as explicit context dependencies, and propagates targeted invalidations when files change. When an edit lands, CodeCtx identifies changed symbols, traverses dependency edges to compute transitive impact, and reports precisely which agents hold stale context. This occurs without requiring a full context refresh for any agent. Our initial evaluation suggests CodeCtx reduces redundant context refreshes and improves cross-agent consistency under parallel editing. Source code available at `github.com/rybkr/codectx`.

1 INTRODUCTION

When multiple LLM agents edit a shared codebase in parallel, no single agent has a complete view of what its peers are doing. Each operates from a context snapshot populated at task assignment. This snapshot is a frozen picture of symbols, signatures, and dependencies that may be quietly invalidated before the agent acts on it. The failure is silent: an agent does not know that an interface it is programming against has changed, or that an invariant it relied on no longer holds. It continues generating code with full confidence, and the bug surfaces at integration time (or not at all), manifesting as a behavioral regression.

This is more subtle than a merge conflict, which existing tooling handles well. Merge conflicts are visible: they block progress and demand resolution. Context drift is invisible: no error is raised, no warning is issued, and no agent is interrupted. The divergence between what an agent believes and what the codebase has become accumulates silently, until it is too late to cheaply fix.

Current multi-agent frameworks do not address this. Systems such as Codex and Claude Code excel in single-agent settings precisely because one coherent context stream is maintained throughout. Multi-agent extensions built on these systems, which typically assign subtasks to parallel agents and merge outputs, inherit the assumption that context, once populated, remains valid. That assumption breaks immediately under concurrent editing.

We present *CodeCtx*, a shared context layer that makes agent beliefs explicit and keeps them current. CodeCtx reifies symbol observations as traceable dependencies, detects staleness when files change, and presents this information to affected agents. This allows mismatches to surface at implementation time rather than at build time.

We make the following contributions:

- We present CodeCtx, a shared context layer that tracks symbol-level observations and propagates dependency-aware invalidations across parallel agents.
- We evaluate CodeCtx on a real-world Python repository, demonstrating that it eliminates context-drift-induced test failures.

2 Related Work

2.1 LLM Agents for Code

Single-agent systems, such as Claude Code or Codex, can resolve real-world engineering tasks when given repository access and an execution toolbox. These systems are effective because a single agent maintains one coherent context stream. Moreover, many models have developed a *primacy-recency* effect within their context windows, meaning they learn to prioritize information near the beginning and end of their contexts. The challenge of context drift does not arise in a single-agent setting by construction. The assumptions that single-agent systems rely on — static context, sequential edits, and no concurrent writers — do not extend naturally to concurrent multi-agent workflows.

2.2 Multi-Agent LLM Frameworks

A growing body of work has explored decomposing complex tasks across multiple LLM agents in parallel. AutoGen (Wu et al., 2024) introduces a conversational multi-agent framework in which agents coordinate via structured message passing, but treats each agent's internal context as opaque and privately managed. CrewAI and LangGraph similarly provide orchestration primitives for role-based agent teams, focusing on task routing and output aggregation rather than explicit context consistency. Broader orchestration literature therefore optimizes scheduling and aggregation, while largely leaving shared epistemic state implicit.

A key distinction is between *orchestrated* and *synchronized* workflows. In an orchestrated workflow, agents are assigned fixed roles — such as *planner* or *architect* — and operate serially, each passing results to the next. Although multiple agents are nominally present, they do not run concurrently. A synchronized workflow, by contrast, deploys multiple independent agents simultaneously on parallel tasks. It is precisely this concurrent setting that existing frameworks leave underserved, and that CodeCtx is designed to support.

2.3 Consistency Models in Distributed Systems

The problem of multi-agent context drift resembles consistent-view maintenance in distributed systems. Classical models such as linearizability (Herlihy & Wing, 1990) and eventual consistency (Vogels, 2009) characterize guarantees for concurrent processes reading and writing shared state. Conflict-free replicated data types (CRDTs) (Shapiro et al., 2011) provide mergeable structures with convergence guarantees, but are not directly applicable to natural-language-heavy, tool-augmented LLM reasoning. Still, the principles of dependency tracking and targeted invalidation from this literature inspire CodeCtx's design: agent beliefs must remain actionable despite the concurrent evolution of a shared codebase.

3 Methodology

3.1 Problem Formalization

We model a multi-agent workflow as a set of n agents $\mathcal{A} = \{a_1, \ldots, a_n\}$ working on a shared codebase $\mathcal{C}$. Each agent a_i maintains a local context $\mathcal{K}_i$ built from a snapshot or by repository exploration. A *belief* is a tuple (a_i, f, v) indicating that agent a_i has observed fact f with value v — for example, that function `read_img` accepts two arguments. Beliefs cover signatures, dependencies, and symbol bodies.

At update time t, a change set $\Delta\mathcal{C}_t$ alters symbols and dependencies in $\mathcal{C}$. The goal is to compute three sets:

- changed symbols $\mathcal{S}_\Delta$, whose interface or implementation changed;
- impacted symbols $\mathcal{I}_\Delta$, which depend on members of $\mathcal{S}_\Delta$;
- affected agents $\mathcal{A}_\Delta \subseteq \mathcal{A}$, whose beliefs intersect $\mathcal{I}_\Delta$.

We seek precision in these sets while minimizing recomputation cost: imprecision causes agents to act on stale beliefs, while excessive overhead undermines the practical viability of the system.

3.2 Symbol Graph and Belief Representation

CodeCtx constructs a directed multigraph $\mathcal{G} = (V, E)$. Vertices V are repository symbols extracted from source: modules, classes, functions, and variables. Edges E encode semantic relations between symbols:

- `IMPORTS`: one module imports a symbol from another.
- `CALLS`: a function or module invokes an external function.
- `REFERENCES_TYPE`: a declaration references a user-defined type.
- `CONTAINS`: a symbol is nested within another.

Each symbol stores two hashes:

1. `interface_hash`: hash for public shape (name, parameters, annotations, base types).
2. `body_hash`: hash over full declaration span for internal implementation changes.

Agents observe symbols through queries; each observation stores the symbol id, source, and hashes. This creates an explicit network of traceable dependencies from agents to code facts.

3.3 Symbol Extraction and Graph Construction

Current support is Python-only via tree-sitter parsing. For each `.py` file, the system:

1. Parses module symbols and declaration spans.
2. Extracts import bindings.
3. Extracts calls and type references.
4. Resolves references through local and import namespaces with fallback matching for ambiguous or unresolved targets.
5. Adds parent-child containment edges to preserve nesting structure.

Source file discovery walks the repository while honoring nested `.gitignore` rules and excludes non-trackable paths. The resulting graph is then available for transitive dependency operations.

3.4 Serving Interface

The system presents both an HTTP service via FastAPI and an MCP tool server. Both modes share the same core service and provide:

- **Agent lifecycle:** agent registration, heartbeat, and removal.
- **Context queries:** task-relevant symbol retrieval, symbol details, and file-to-symbol listing.
- **Subgraph retrieval:** bounded-depth neighborhood expansion.
- **Update operations:** file-level override ingestion and invalidation reporting.

3.5 Invalidation and Stale-Context Detection

The system applies two-stage invalidation:

- **Change localization.** File overrides are applied, a fresh symbol graph is built, and changed symbols are identified by comparing old and new interface/body hashes and symbol presence.
- **Impact propagation.** Given changed symbols $\mathcal{S}_\Delta$, the system traverses reverse dependencies for edges in call/import/type-reference space to compute $\mathcal{I}_\Delta$.

For every registered agent, staleness is computed by comparing stored observation hashes against current symbol hashes.

Affected symbols generally fall into three categories:

- missing symbols (deleted/moved),
- altered interfaces (interface hash drift),
- altered implementations (body hash drift with stable interface).

Affected agents and stale symbols are returned for orchestration decisions.

4 EXPERIMENT

4.1 SETUP

We evaluate CodeCtx on the `pallets/click` repository, a real-world open-source Python project of approximately 15,000 lines of code. We selected this repository for its moderate complexity: it is large enough to have non-trivial inter-module dependencies, but small enough for agents to operate on meaningfully overlapping regions within a single task.

For each trial, two parallel agents were assigned tasks requiring overlapping edits to the completion subsystem, simulating the overlapping-edit scenario that most stresses context consistency. We ran three trials with CodeCtx active and three trials without (baseline), for six total runs. The baseline condition uses no shared context layer; each agent operates from a static snapshot populated at task start and receives no staleness notifications.

> **Agent A:** Add richer shell completion display using completion item metadata.
>
> **Agent B:** Change the completion item representation to include `display`, `insert_text`, and `help` fields.

We measure two outcomes: task correctness, assessed as pass/fail against the repository's existing test suite, and total token consumption per agent across the full task trajectory. The model used for all runs was `gpt-5.3-codex-spark`, with reasoning set to `medium`.

4.2 RESULTS

With CodeCtx, all tasks passed the test suite across all three trials. Without CodeCtx, results were consistently degraded: one trial produced 5 test failures, one produced a compilation error preventing the test suite from running at all, and one produced 2 test failures. No baseline trial achieved a fully clean result. This suggests that context drift manifests reliably, and in varied forms.

Table 1 reports token counts per agent under each condition.

Condition	Trial 1	Trial 2	Trial 3	Median
Agent A w/ CodeCtx	59,310	143,351	108,656	108,656
Agent A w/o CodeCtx	60,948	100,691	56,788	60,948
Agent B w/ CodeCtx	35,574	33,191	40,638	35,574
Agent B w/o CodeCtx	27,578	38,129	53,791	38,129

Table 1: Token consumption per agent across trials, with median.

CodeCtx conditions show higher token usage in Agent A, with slightly lower token usage in Agent B.

5 CONCLUSION

We presented *CodeCtx*, a shared context layer for multi-agent coding workflows that addresses context drift — the silent failure mode that arises when parallel agents act on stale beliefs about a concurrently evolving codebase. CodeCtx makes agent observations explicit as symbol-level dependencies, detects staleness through hash-based change localization, and propagates targeted invalidations to only the agents whose beliefs are affected.

Our experiments on the `pallets/click` repository demonstrate that context drift is a reliable source of correctness failures in overlapping-edit scenarios: baseline agents without CodeCtx produced test failures or compilation errors in every trial, while agents operating with CodeCtx passed all tests across all trials. Token overhead was observed for the agent most actively receiving invalidations, reflecting the real cost of synchronization — a cost we argue is worthwhile given the correctness guarantees it provides.

Contribution This work presents a system prototype that operationalizes belief-aware context management for multi-agent LLM workflows. The core contribution is a concrete architecture, symbol graph construction, dependency-tracked observation recording, and commit-time invalidation propagation, that can be integrated with existing agent orchestration frameworks via HTTP or MCP. CodeCtx demonstrates that treating agent context as a shared, update-reactive data product, rather than a static private snapshot, is both practically feasible and measurably beneficial.

Limitations and Future Work The current implementation is Python-only, and hash-based invalidation is syntactic rather than semantic — behaviorally equivalent rewrites may trigger unnecessary notifications. The benchmark is small and synthetic; broader evaluation across larger repositories and more diverse task types remains future work. We also intend to explore more token-efficient reconciliation strategies, such as injecting only changed symbol signatures rather than re-querying full context.

REFERENCES

Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, 2011.

Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. Autogen: Enabling next-gen llm applications via multi-agent conversation. In *ICLR Workshop on LLM Agents*, 2024.

ACKNOWLEDGEMENT OF LLM USE

This paper was written with the assistance of Claude (Anthropic) for drafting and editing prose sections. All technical content, system design, implementation, and experimental results are the author's own work.