Advanced International Journal for Research (AIJFR)
E-ISSN: 3048-7641 ● Website: www.aijfr.com ● Email: editor@aijfr.com

Deterministic Context Maps: A Machine-Readable Key-Value Index for Low-Cost, Low-Latency Context Discovery in Hybrid Software-Engineering Automation

Umesh Kumar

Hansi, Haryana, India

DOI: https://doi.org/10.63363/aijfr.2026.v07i06.8742

Abstract

When AI agents work on research, analysis, and coding across one or several repositories, a large share of their effort is spent on context discovery: finding which files, symbols, documents, or links are relevant to the current request. Today this discovery is performed agentically, by repeated keyword searches and exploratory file reads, and it is paid for again in every new session, by every agent, for every recurring key, throughout the life of the project. Each episode costs large-language-model turns, tokens, and minutes of wall time, and its outcome is probabilistic. In this paper a Deterministic Context Map (DCM) is proposed: a machine-readable key-value index, versioned inside the repository, in which keys are the stable search terms of the project (features, concepts, symbols, configurations) and values are pointers (file paths, anchors, links) rather than content. The map is built once and updated automatically by the continuous-integration pipeline whenever a change branch is merged, so freshness is guaranteed by construction; a key miss falls back to ordinary agentic search and is logged as a candidate key, making the map self-improving. A parameterized model anchored in current model pricing and a production pipeline's operational envelope shows a single discovery episode shrinking from about 8 LLM turns to 1 (about USD 0.43 to USD 0.02, a 95% reduction), a representative 90-minute ticket becoming 25% cheaper on top of an existing deterministic-shell pipeline (61% cheaper than an agent-does-everything baseline), and a five-developer team saving on the order of USD 1,000 in tokens and 112 hours of wall time per year, against negligible maintenance cost. The design, schema, update-on-merge lifecycle, serving protocol, cost and time model, and limitations are presented, together with a comparison against classical code indexes, repository maps, embedding indexes, and repository graphs.

Keywords: Deterministic Context Map, Agentic AI, Code Search, Context Engineering, Retrieval, Token Economics, Software Engineering Automation, Repository Index, LLM Agents, Hybrid Automation

1. Introduction

Finding the right place to read or write is a dominant, recurring activity in software work. A field study at Google measured an average of 12 code-search queries per developer per weekday [1]; the need does not disappear when the developer is an AI agent. It gets worse: agents are stateless between sessions, so the same question ("where is the claims submission flow?", "which config controls the proxy?") is re-answered from scratch in every new conversation, by every parallel agent, across the whole life of a project. Each re-answering is an agentic search episode: a loop of keyword searches and exploratory file reads in which every step is an LLM turn that costs tokens and seconds and succeeds only probabilistically [2]. At current frontier-model prices of USD 5 to 50 per million output tokens [3], organizations are effectively paying a recurring tax for rediscovering facts about their own repositories that were already discovered many times before. The cost is not only money: the minutes spent searching delay every decision that depends on the found context.

A companion study of hybrid automation showed that moving repeatable orchestration work (dispatch, state tracking, guards, verification, lifecycle) out of the agent and into a deterministic shell cuts the token cost of a representative ticket by about 48% [4]. This paper extends that principle to the retrieval side of agent work. The proposal is a Deterministic Context Map (DCM): a machine-readable key-value index in which the keys are the stable search vocabulary of the project and the values are pointers to where the relevant material lives (file names, anchors, links). The agent consults the map first and jumps directly to the listed locations; the expensive exploratory loop runs only on a genuine miss. Because the map is rebuilt incrementally by the continuous-integration (CI) pipeline whenever a change branch is merged, its freshness is tied to the repository's own version history rather than to any background re-indexing process.

The contributions are: (i) the DCM concept, schema, and design rules, including a two-layer key taxonomy (auto-extracted symbols; curated semantic concepts) and the principle that values are pointers, never content copies (Section 3); (ii) an update-on-merge lifecycle and a lookup-first, fallback-on-miss agent protocol whose misses feed the next map revision, making the index self-improving (Section 4); (iii) a parameterized cost and time model, anchored in current token prices [3] and the operational envelope of a production pipeline [4], quantifying savings per episode (about 95%), per ticket (25% on top of a deterministic shell; 61% versus an agentic-only baseline), and per team-year (about USD 1,000 and 112 hours for five developers) (Section 5); (iv) an analysis of why determinism itself is valuable for freshness, reproducibility, auditability, and security (Section 6); and (v) a comparison with classical code indexes, repository maps, embedding indexes, and repository graphs, with anti-patterns and limitations (Sections 2 and 7).

2. Background and Related Work

2.1 Classical Code Indexes

Deterministic symbol indexes are a solved problem for human tooling. Universal Ctags generates symbol-to-location tables used by editors for decades [5]; LSIF and its successor SCIP persist precise code-intelligence data (definitions, references) computed once at index time so that navigation queries become cheap lookups [6]. These systems prove the core economics the DCM relies on (precompute once, query deterministically), but they stop at the symbol level: they cannot answer concept-level

questions such as “which files implement the claims submission flow?”, and they are not designed as LLM-facing context sources.

2.2 Retrieval Mechanisms of Current Coding Agents

Current agents obtain repository context in three main ways. First, repository maps: Aider builds a concise map of files and key symbols with tree-sitter and sends the most relevant ranked subset with each request [7, 17]; this improves grounding but is recomputed and re-ranked per request, consumes context-window tokens every time, and stays at the file/symbol level. Second, embedding indexes: tools such as Cursor index the codebase with vector embeddings for semantic search [8]; retrieval is powerful but approximate, non-deterministic, infrastructure-heavy, and subject to staleness between re-index runs. Third, agentic search: agents in the Claude Code style explore with grep-like tools and file reads under model control [9]; this is maximally flexible and needs no index, but it is exactly the repeated, token-priced loop whose cost this paper attacks. Notably, the vendor documentation itself distinguishes deterministic from non-deterministic instruction channels and publishes a machine-readable documentation index for agents to consult before exploring [9].

2.3 Curated Machine-Readable Maps

The llms.txt proposal plays the analogous role for websites: a concise, precisely formatted markdown file of links and guidance that agents read instead of crawling HTML, explicitly motivated by the observation that every wasted token costs time and money [10]. Its adoption by major documentation platforms and AI vendors demonstrates both the practicality and the demand for curated machine-readable maps. The DCM can be seen as the repository-side counterpart of llms.txt, with two additions the web case lacks: a merge-triggered update pipeline bound to version control, and a miss-driven growth loop.

2.4 Repository Graphs and Graph-Based Retrieval

Recent research connects LLM agents to richer repository structures: RepoGraph builds a repository-level code graph to improve AI software engineering [11]; CodexGraph bridges LLMs and code repositories through code-graph databases [12]; GraphRAG builds entity graphs over corpora for query-focused summarization [13]. These systems demonstrate the value of structured, precomputed context, and agent-computer-interface research shows that interface design materially changes agent success on real issues [14, 15]. Graphs are, however, comparatively heavy to build, query, and govern, and typically still involve the model at query time. DCM deliberately occupies the simple end of this spectrum: a flat, human-reviewable key-value file that a shell script can update and a single cheap lookup can serve.

2.5 The Gap

Table 1 compares the approaches. No existing mechanism combines (a) concept-level keys chosen from the project's own task vocabulary, (b) deterministic lookup with pointer values, (c) freshness guaranteed by the merge event itself, (d) versioning and human review of the index inside the repository, and (e) a self-improving loop fed by logged misses. The DCM contributes exactly this combination, positioned as one more deterministic component of the hybrid shell of [4].

Table 1: Context-Discovery Approaches Compared

Approach	Granularity	Deterministic Lookup	Freshness Mechanism	Token Cost per Query	Reviewed in Repo
ctags / SCIP [5, 6]	symbols	yes	re-index runs	none (IDE-side)	no
repo map (Aider) [7]	files + symbols	no (ranked per request)	per request	map tokens every request	no
embedding index [8]	chunks	no (approximate)	re-embedding	low per query	no
agentic search [9]	anything	no	always current	5–8 turns per episode	n/a
llms.txt [10]	web links	yes	manual edits	one small file	site-side
repo graphs [11, 12, 13]	entities + edges	partly	graph rebuilds	query + model time	no
DCM (this paper)	concepts + symbols + links	yes	update on merge	1 turn, small slice	yes

3. The Deterministic Context Map

3.1 Definition and Design Principles

A DCM is a machine-readable document, stored and versioned inside the repository (for example context-map.yaml), that maps keys to values. Keys are the stable search terms of the project: feature and flow names, domain concepts, important symbols, configuration topics, operational runbooks. Values are lists of pointers: file paths, optional line anchors, test and documentation locations, external links, and owning teams. Four design principles follow from the goals. First, values are pointers, never content copies: the map stays small, cannot leak stale text, and the agent always reads the current file. Second, the map is layered: an L0 symbol layer is extracted automatically by deterministic tooling of the ctags/SCIP class [5, 6, 17], while an L1 semantic layer holds the concept keys that no parser can infer, curated by humans and proposed by an optional LLM pass at merge time. Third, the map is code: it lives in the repository, its changes appear in merge diffs, and reviewers treat wrong map entries like wrong code. Fourth, aliases are first-class, so that the informal vocabulary used in tickets (“login”) resolves to the canonical key (“authn.middleware”). Figure 1 illustrates the problem the map removes, and Figure 2 shows the schema on an example.

Figure 1: The Repeated-Discovery Problem and the DCM Lookup Path

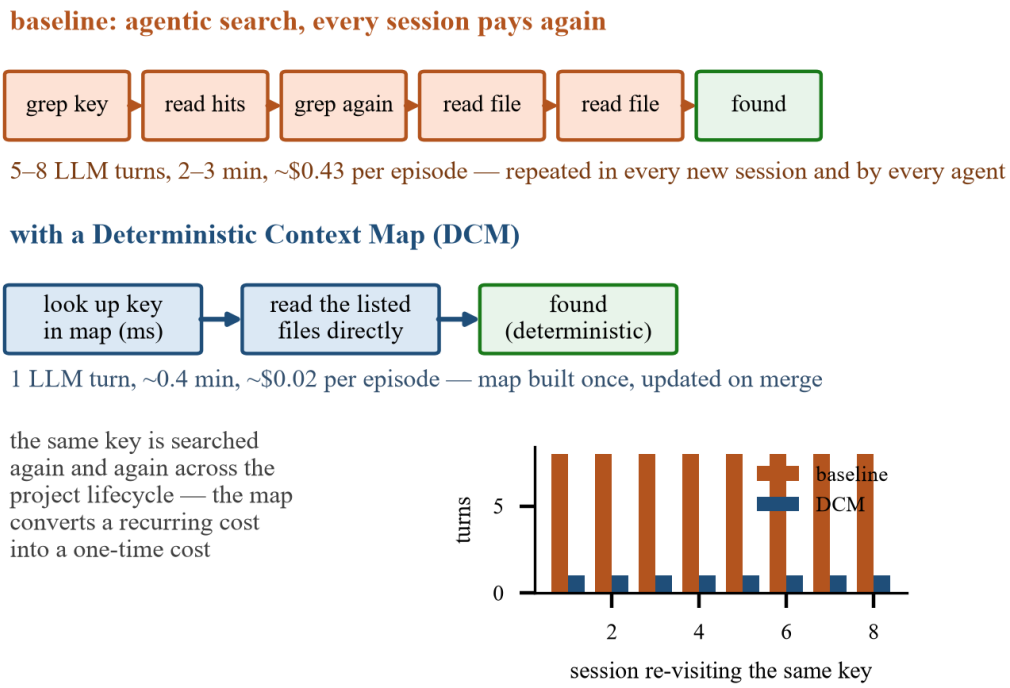
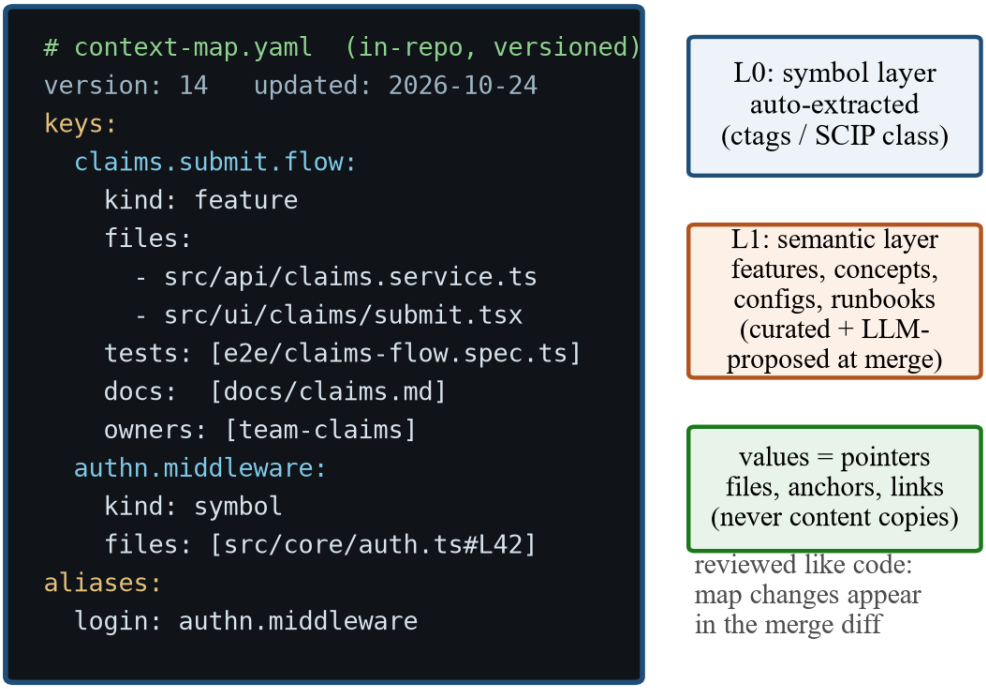


Figure 2: Context-Map Schema with the Two Key Layers and Pointer Values



3.2 Serving Slices, Not the Whole Map

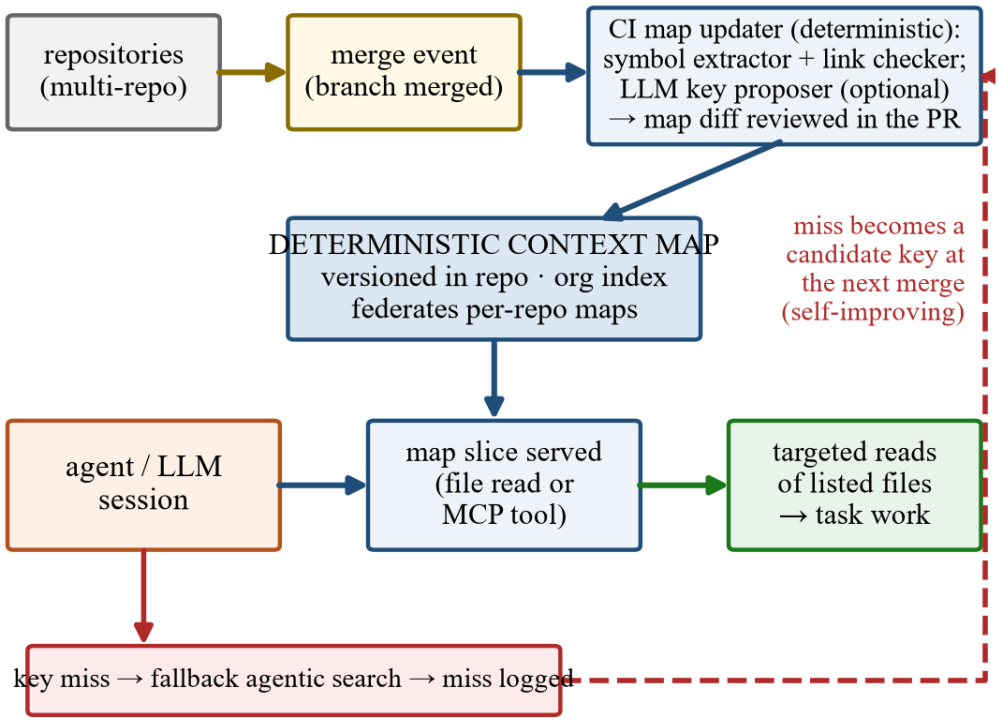
Large projects produce large maps, and sending the entire map with every request would recreate the token tax it is meant to remove. The lookup therefore serves a slice: the entries whose keys or aliases match the task text, plus their immediate neighbors (same owner or same feature prefix). A slice of a few kilobytes covers a typical ticket; the deterministic matcher runs in milliseconds on the developer machine or CI runner, and only the slice enters the context window.

4. Architecture and Lifecycle

4.1 Build Once, Update on Merge

Figure 3 shows the architecture. The map is created once per repository by running the L0 extractor over the tree and seeding L1 with an initial curated key set (for software development this is typically an afternoon of work guided by the module structure). From then on the map maintains itself at the only moment the ground truth changes for the team: the merge of a change branch. A CI step re-runs the extractor incrementally over the changed files, verifies that every pointer still resolves (a deterministic link check), optionally lets an LLM propose new or updated L1 keys from the diff and the ticket text, and commits the resulting map delta as part of the merge, where it is reviewed like any other change. Map version therefore equals repository version; there is no background indexer to drift out of date.

Figure 3: DCM Architecture and Update-on-Merge Lifecycle with the Miss-Driven Improvement Loop



4.2 Agent Protocol: Lookup First, Search on Miss

The agent-side contract is deliberately simple. On receiving a task, the deterministic shell (or the agent itself through a file read or a Model Context Protocol tool [16]) queries the map with the task's key terms

and injects the returned slice. The agent jumps directly to the listed files with targeted reads. If a needed key is absent, the agent falls back to ordinary agentic search [9], completes the task, and the shell logs the miss (query, files finally used). Logged misses become candidate keys in the next merge's map delta, so the map converges toward the vocabulary the team actually uses: the system is self-improving, and the worst case equals today's baseline rather than falling below it.

4.3 Multiple Repositories

For work spanning several repositories, each repository carries its own map, and an organization-level index map federates them: its keys are repository-scoped topics and its values are the per-repository maps. An agent resolves “claims flow” in two deterministic hops (org index, then repository map) instead of searching across all checkouts. Ownership metadata in the entries doubles as a routing hint for which team or agent should receive a change request.

5. Cost and Time Model

5.1 Episode Model

Let an agentic turn consume T_{in} input and T_{out} output tokens at prices p_{in} and p_{out} per million tokens. A discovery episode of n_t turns costs as given by Equation (1).

$$C_{episode} = n_t \times (T_{in} \times p_{in} + T_{out} \times p_{out}) \quad (1)$$

Following the companion model [4], a representative turn carries 12k input and 1.2k output tokens at p_{in} = USD 3 and p_{out} = USD 15 per million tokens [3], about USD 0.054 per turn. A baseline agentic search episode of about 8 turns (search, read hits, refine, read files) costs about USD 0.43 and 2.7 minutes at a conservative 20 s per turn. A DCM lookup is one turn with a small slice payload (about 3k input tokens), about USD 0.02 and 0.4 minutes; the lookup itself is a millisecond-scale deterministic operation. Per episode the reduction is about 95% in tokens and about 2.3 minutes in wall time (Figure 4, right).

5.2 Per-Ticket and Per-Year Savings

In the production pipeline's envelope, a 90-minute ticket contains about three discovery episodes (one at start, two mid-mission), about 24 of the 84 turns of a hybrid-shell ticket [4]. With the map these shrink to about 3 turns, taking the ticket from USD 4.54 to about USD 3.41 (a 25% further reduction, and 61% below the USD 8.80 agent-does-everything baseline; Figure 4, left) and returning about 7 minutes of wall time. Equation (2) gives the annual net saving for a team, where N is tickets per year, e the episodes per ticket, and M the map maintenance cost.

$$S_{year} = N \times e \times (C_{base} - C_{map}) - M \quad (2)$$

For a five-developer team at four tickets per developer-week (about 960 tickets per year), the token saving is about USD 1,000 per year and the time saving about 112 hours per year (Figure 5, Table 2). M is small by construction: the extractor and link checker are deterministic CI steps costing seconds and no tokens, and the optional LLM key proposer at about USD 0.05 per merge adds roughly USD 50 per thousand merges. The same lookups also serve human developers, whose search frequency is well documented [1], so the modelled saving is a lower bound on the organisational value.

Table 2: Cost and Time Model Summary (Parameters of Section 5.1)

Quantity	Baseline (Agentic Search)	With DCM	Change
turns per discovery episode	about 8	1	−87.5%
cost per episode	$0.43	$0.02	−95%
time per episode	2.7 min	0.4 min	−2.3 min
cost per 90-min ticket (hybrid shell [4])	$4.54	$3.41	−25%
cost per ticket vs agentic-only	$8.80	$3.41	−61%
team of 5, tokens per year	about $1,100	about $110	about −$1,000
team of 5, wall time per year	—	—	about −112 h
map maintenance per year	—	about $50–100	negligible

Figure 4: Token Cost per Ticket and per Discovery Episode

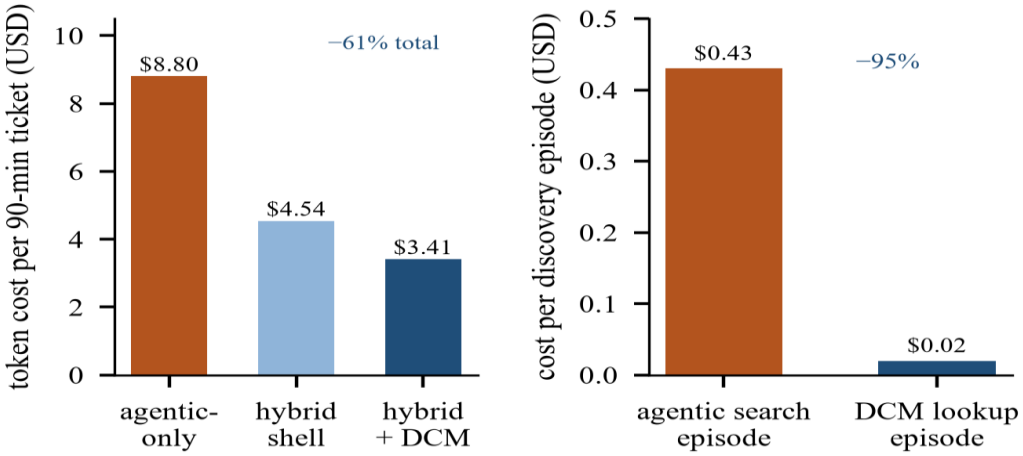
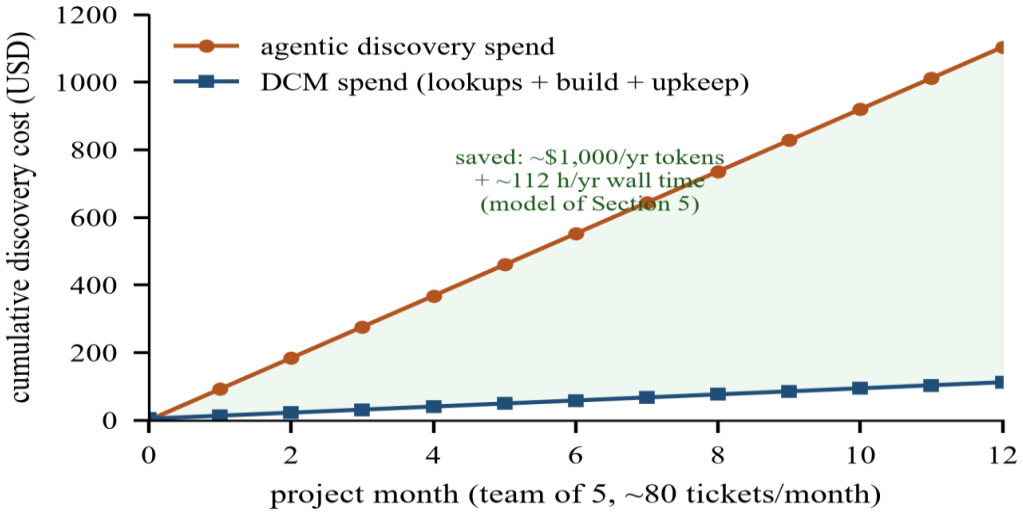


Figure 5: Cumulative Discovery Cost over a Project Year (Model of Section 5)



6. Why Determinism Matters Beyond Cost

Freshness by construction: because the map is updated by the merge that changes the code, its version always equals the repository version; embedding indexes and background re-indexers can lag arbitrarily [8]. Reproducibility: the same key returns the same pointers for everyone and every agent at a given commit, so two sessions start from the same context instead of two different search journeys; this removes a whole class of run-to-run variance in agent behavior [2, 4]. Auditability: map changes are ordinary diffs, so the question "why did the agent read this file?" has a reviewable answer. Security: values are pointers rather than content, so the map grants no access by itself; existing file permissions still gate every read, and the map can even reduce exposure by discouraging broad exploratory reads across sensitive directories. Finally, the lookup step is a deterministic tool in the sense of the hybrid pattern [4]: it has probability 1 of returning the listed pointers, so reliability engineering can concentrate on the genuinely judgment-bearing steps.

7. Benefits, Anti-Patterns, and Limitations

Table 3 summarizes the benefits. The anti-patterns are equally important. A map without the CI hook becomes stale documentation and worse than nothing; the update-on-merge trigger is the load-bearing design element. Copying content into values turns the map into a second, divergent source of truth; values must remain pointers. Unbounded key growth turns lookups into scrolling; slices, aliases, and periodic curation keep the map navigable. Treating the map as a prompt rather than a queried tool re-introduces the token tax for every request. The limitations are stated plainly: the initial L1 curation needs a few hours of expert attention and a naming discipline; the quality of semantic keys depends on review culture; renames are caught by the link checker but conceptual drift is caught only by humans; and the quantitative results are a parameterized model anchored in one production pipeline's envelope and current list prices [3, 4], with a controlled multi-team field study left as future work.

Table 3: Benefits of the Deterministic Context Map

Dimension	Mechanism	Magnitude (This Analysis)
token cost	1-turn lookup replaces 5–8 search turns	about −95% per episode; −25% per ticket (Table 2)
wall time	millisecond lookup + targeted reads	about 2.3 min per episode; about 7 min per ticket
determinism	same key, same pointers, every session	variance removed from discovery (Section 6)
freshness	update bound to the merge event	map version = repository version
auditability	map lives in repo, reviewed in diffs	discovery decisions become reviewable
self-improvement	miss log feeds next map delta	worst case equals today's baseline

8. Conclusion and Future Work

Agentic search answers a question that, for most keys, the project has already been answered many times before. The Deterministic Context Map stores those answers where agents and humans can retrieve them in one cheap, deterministic step: keys are the project's search vocabulary, values are pointers, the map

lives in the repository, and the merge event keeps it honest. On the model presented here, discovery episodes shrink by about 95%, tickets by a further 25% on top of a deterministic-shell pipeline, and a small team recovers on the order of a thousand dollars and a hundred working hours per year, while gaining reproducibility, auditability, and a self-improving index whose worst case is today's status quo. Future work: miss-rate and hit-rate telemetry as first-class metrics; automatic key mining from ticket and pull-request text; shared human/agent use inside IDEs; federation patterns for very large organizations; a hybrid retrieval design in which an embedding index serves as a secondary layer below the deterministic map for genuinely novel queries; and a controlled field study measuring realized savings and map-quality dynamics across teams.

Conflict of Interest

The research presented in this paper was not sponsored. No third-party funding supported this work during the 36 months prior to submission, and the author declares no competing interests.

Acknowledgement

The author thanks the engineering colleagues who operate and review the production pipeline whose operational envelope anchors the model in Section 5.

References

1. C. Sadowski, K. T. Stolee, S. Elbaum, “How Developers Search for Code: A Case Study”, Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), 2015, 191–201, DOI: 10.1145/2786805.2786855
2. Anthropic, “Building Effective Agents”, December 2024, https://www.anthropic.com/engineering/building-effective-agents
3. Anthropic, “Claude Pricing”, 2026, https://docs.claude.com/en/docs/about-claude/pricing
4. Umesh Kumar, “Agentic AI Meets Deterministic Tooling: A Cost, Speed, and Reliability Analysis of Hybrid Automation for Software Engineering”, Oct 2026.
5. Universal Ctags Project, “Universal Ctags”, 2025, https://ctags.io/
6. O. P. Geirsson, “SCIP - a Better Code Indexing Format than LSIF”, Sourcegraph Blog, June 2022, https://sourcegraph.com/blog/announcing-scip
7. Aider, “Repository Map”, 2025, https://aider.chat/docs/repomap.html
8. Cursor, “Codebase Indexing”, 2025, https://docs.cursor.com/context/codebase-indexing
9. Anthropic, “Best Practices for Claude Code”, 2025, https://www.anthropic.com/engineering/claude-code-best-practices
10. J. Howard, “The /llms.txt File”, 2024, https://llmstxt.org/
11. S. Ouyang, W. Yu, K. Ma, Z. Xiao, et al., “RepoGraph: Enhancing AI Software Engineering with Repository-Level Code Graph”, arXiv:2410.14684, 2024
12. X. Liu, B. Lan, Z. Hu, Y. Liu, et al., “CodexGraph: Bridging Large Language Models and Code Repositories via Code Graph Databases”, arXiv:2408.03910, 2024
13. D. Edge, H. Trinh, N. Cheng, J. Bradley, et al., “From Local to Global: A Graph RAG Approach to Query-Focused Summarization”, arXiv:2404.16130, 2024

14. J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, O. Press, “SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering”, Advances in Neural Information Processing Systems 37 (NeurIPS), 2024
15. C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, K. Narasimhan, “SWE-bench: Can Language Models Resolve Real-World GitHub Issues?”, International Conference on Learning Representations (ICLR), 2024
16. Anthropic, “Model Context Protocol”, 2024, https://modelcontextprotocol.io/
17. Tree-sitter Project, “Tree-sitter: An Incremental Parsing System”, 2025, https://tree-sitter.github.io/tree-sitter/