The Journal of Human-Centric Systems Research
Preserving Human Logic in the Epoch of Automation • Vol. 14, No. 3 • August 2026

ContextOS: Decoupling Context Management from Large Language Models for Persistent Multi-Agent Software Development

Muhammed Sayees
https://sayees.narrs.in

Abstract: The proliferation of Large Language Model (LLM)-powered software engineering agents has fundamentally altered the developer workflow. However, as developers alternate between heterogeneous tools (such as Cursor, Claude Code, Aider, and local CLI utilities), project context becomes fragmented. Each agent maintains an ephemeral, isolated conversation history, leading to redundant processing, lost architectural knowledge, and high token expenditure. We propose **ContextOS**, a persistent context operating system that decouples context management from model inference. ContextOS introduces an OS-like abstraction layer that captures, compresses, and structures project knowledge into a hybrid semantic-knowledge graph (KG) memory. This paper presents the architecture of ContextOS, formalizes the context representation model, and details algorithms for cross-agent memory synchronization, graph-based retrieval, and prompt assembly. By decoupling memory from the agent, ContextOS enables stateful, persistent, and portable software engineering contexts. We establish a rigorous experimental methodology for evaluating structured project knowledge against conversational memory baselines, providing a foundational framework for next-generation multi-agent software development.

Keywords: Large Language Models, Multi-Agent Systems, Software Engineering, Context Management, Knowledge Graphs, Retrieval-Augmented Generation, System Architecture.

1 Introduction

Modern software engineering (SE) is increasingly mediated by AI coding assistants. Tools such as GitHub Copilot, Cursor, and Aider have evolved from simple autocomplete utilities into autonomous agents capable of complex refactoring and bug resolution. However, a critical architectural limitation persists: **context fragmentation**.

Currently, each AI agent maintains its own isolated context window. When a developer switches from an IDE-based assistant (e.g., Cursor) to a terminal-based agent (e.g., Aider) or a web interface (e.g., Claude), the accumulated project context (comprising architectural decisions, recent file modifications, and error trajectories) is lost. The developer must manually reconstruct this context, resulting in cognitive overhead, redundant token processing, and a high probability of hallucination due to incomplete repository understanding.

We introduce **ContextOS**, a persistent memory and context management layer designed specifically for cross-agent software engineering. ContextOS shifts the paradigm from *conversation-bound memory* to *agent-agnostic project knowledge*. By treating context as a first-class operating system resource, ContextOS manages the lifecycle of project knowledge, offering stateful APIs to heterogeneous LLMs.

Primary Research Questions

- **RQ1:** Can structured project knowledge (ContextOS) outperform ephemeral conversational memory in multi-agent SE tasks?
- **RQ2:** To what extent can hybrid retrieval (Knowledge Graphs + Vector Embeddings) improve repository understanding compared to naive dense retrieval?
- **RQ3:** Can architecture-aware context compression reduce token consumption while preserving the semantic integrity of the codebase?
- **RQ4:** How can persistent project memory remain portable and synchronized across heterogeneous AI coding assistants?
- **RQ5:** What system architecture best decouples context management from LLM inference?

Novel Contributions

1. **OS-Level Abstraction for AI Context:** Formalization of context as a decoupled, system-level resource.
2. **Hybrid Memory Representation:** A dual-store architecture mapping Abstract Syntax Trees (ASTs) to a unified Knowledge Graph alongside vector-based semantic memory.
3. **Cross-Agent Persistence:** A protocol for migrating and synchronizing context across disjoint AI tools.
4. **Mathematical Formalization:** Rigorous optimization objectives for context compression and retrieval.
5. **Comprehensive Algorithmic Framework:** Ten novel algorithms governing the context lifecycle, from extraction to garbage collection.

2 Background

2.1 AI in Software Engineering

The application of LLMs to SE has transitioned from sequence-to-sequence code generation [1] to repository-level reasoning [2]. Models like GPT-4 and Claude 3.5 Sonnet possess context windows up to 200K tokens, enabling the ingestion of

entire micro-repositories. However, in large enterprise codebases, context windows remain insufficient, necessitating selective context inclusion.

2.2 Context Windows and Memory Systems

Traditional conversational agents utilize FIFO (First-In, First-Out) token eviction or simple summarization [3]. Advanced frameworks like MemGPT [4] introduce a virtual memory hierarchy (working vs. persistent memory), but remain tightly coupled to a single agent instance.

2.3 Retrieval-Augmented Generation (RAG) in SE

RepoCoder [5] and similar RAG-based systems retrieve relevant code snippets using dense vector embeddings. While effective for semantic similarity, dense retrieval often fails to capture structural relationships (e.g., function call graphs, class inheritance) critical for accurate SE tasks [6].

3 Problem Statement

Let $A = \{A_1, A_2, ..., A_n\}$ be a set of heterogeneous AI agents used by a developer. Let R represent the software repository state. In current paradigms, each agent A_i maintains an isolated context $C_i(t)$ at time t.

When transitioning from A_1 to A_2, the shared knowledge transfer is zero: $C_1(t) \cap C_2(t) = \varnothing$, except for the underlying files modified in R.

The core optimization problem is to design a unified, continuous memory space M such that any agent A_i can query a centralized context $C^*(t) = f(M, q_i)$, where q_i is the agent's current task query, minimizing the information loss L and token cost K.

$$\min_{\Theta} \Sigma_{i=1}^{n} \left(L(C^*(t), R_{true}) + \lambda K(C^*(t)) \right)$$

where R_{true} is the ideal oracle context required to solve the task, and λ is a regularization parameter for token budget.

4 Related Work

Table 1: Comparison with Existing Systems

System	Memory Type	Cross-Agent?	Graph-based?	OS-Abstraction
Vanilla RAG [7]	Vector Database	No	No	No
MemGPT [4]	Virtual (Paged)	No	No	Yes (OS)
RepoCoder [5]	Window + RAG	No	No	No
AutoCodeRover [2]	AST Search	No	Yes	No
MCP [8]	Protocol	Yes	No	No
ContextOS (Ours)	Vector + KG	**Yes**	**Yes**	**Yes**

Model Context Protocol (MCP): Recent developments like Anthropic's MCP standardize how agents access external tools. ContextOS operates *above* the data layer but *below* the agent, acting as an intelligent state manager that could theoretically be exposed via MCP.

GraphRAG: Systems like GraphRAG [9] use LLMs to extract entities. ContextOS differs by utilizing deterministic static analysis (AST parsing) combined with LLM-extracted semantic graphs for deterministic SE accuracy.

5 System Design

ContextOS is governed by three design principles:

1. **Decoupling:** LLMs are stateless compute engines; ContextOS is the stateful memory manager.
2. **Structural Fidelity:** Code is not merely text; it is a deterministic graph of dependencies. ContextOS must preserve AST relationships.
3. **Agent Agnosticism:** The system must interface via standard API contracts, allowing Cursor, Claude, and local CLI tools to read/write from the same memory matrix.

6 Architecture

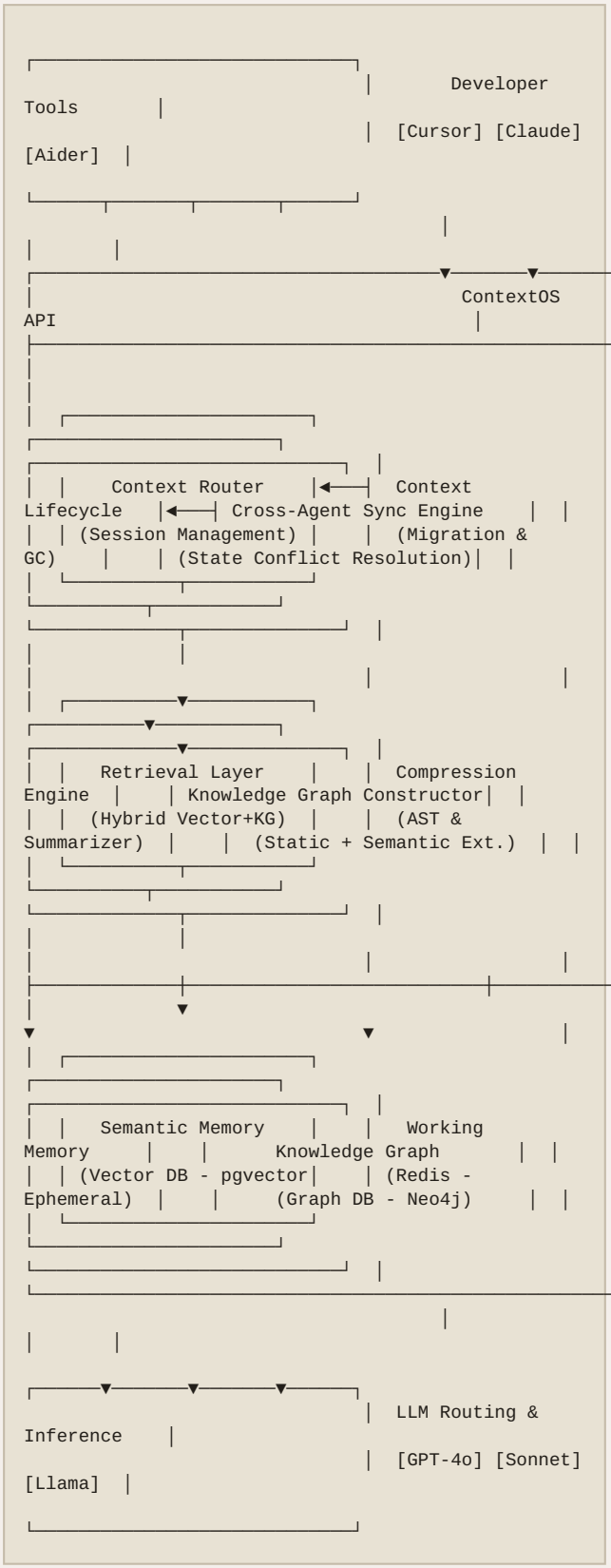

Figure 1: ContextOS Overall System Architecture

Table 2: Component Responsibilities

Component	Responsibility	Technology Target
Context Router	Routes incoming queries, authenticates agents.	Node.js / Fastify
Sync Engine	Resolves concurrent state modifications.	CRDTs / Redis
Compression	Reduces token footprint of ASTs/ histories.	Custom Heuristics
Retrieval Layer	Fuses dense embeddings with graph traversal.	Python / LangChain
Knowledge Graph	Stores files, classes, intent edges.	Neo4j / NetworkX
Semantic Memory	Embeddings of code chunks and reasoning.	PostgreSQL

7 Context Representation Model

Context within ContextOS is modeled as a heterogeneous, multi-layered tuple $C = \langle W, S, G, M \rangle$:

- W: **Working Context** (Recent conversational turns, $t \in [t_{now-k}, t_{now}]$).
- S: **Semantic Context** (Dense vectors of codebase documentation, commit messages).
- G: **Graph Context** (Subgraphs of the AST relevant to the query).
- M: **Meta Context** (System instructions, agent-specific formatting rules).

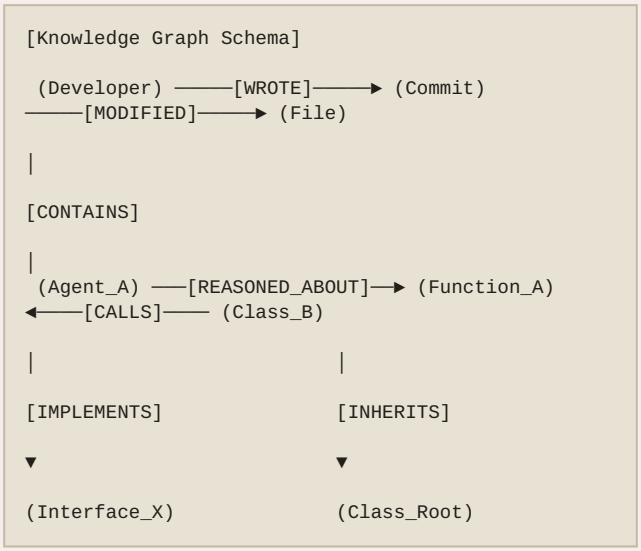

Figure 2: ContextOS Knowledge Graph Schema

Table 3: Memory Categories & Expiry Rules

Memory Type	Data Stored	Eviction Policy	TTL
Working (L1)	Active prompt, last 5 chat turns	FIFO / Token Limit	1 Hour
Session (L2)	Agent reasoning, bash outputs	Summarize -> L3	24 Hours
Semantic (L3)	Function embeddings, PR desc.	Cosine Threshold	Persistent
Structural (L4)	AST, Call Graphs	Real-time Git Sync	Persistent

8 Context Compression Algorithms

To fit extensive codebase context into bounded LLM context windows without diluting semantic density, ContextOS employs Architecture-Aware Context Compression.

Let $F = \{f_1, f_2, ..., f_m\}$ be a set of retrieved source files. Let $T(f_i)$ be the token count of file f_i. If $\Sigma\ T(f_i) > B$ (where B is the token budget), ContextOS applies selective abstraction.

```
[Compression Pipeline]
Raw Code ⟶ AST Parsing ⟶ Relevance Scoring ⟶
Skeletonization ⟶ Compressed

Example Code:
class Optimizer:
    def __init__(self): ...
    def compute_loss(self):
        # 50 lines of complex math
        return loss
    def step(self): ...

Compressed (Skeletonized) Code:
class Optimizer:
    def __init__(self): ...
    def compute_loss(self):
        # [COMPRESSED: 50 lines defining loss
calculation]
        return loss
    def step(self): ...
```

Figure 3: Context Compression Pipeline via Skeletonization

9 Knowledge Graph Construction

Traditional vectors fail at traversing deep dependencies. We construct the Knowledge Graph $G = (V, E)$ dynamically.

Vertices $V = V_{code} \cup V_{dev} \cup V_{agent}$. Edges E map syntactic (e.g., `CALLS`, `INHERITS`) and semantic (e.g., `FIXED_BUG`) relationships.

$$w(u, v) = \alpha \cdot StaticWeight(u,v) + (1 - \alpha) \cdot SemanticWeight(u,v)$$

Where $StaticWeight$ is binary (1 if structurally connected), and $SemanticWeight$ is derived from cosine similarity of embeddings.

10 Retrieval Pipeline

The retrieval pipeline combines dense vector search with graph traversal (e.g., Personalized PageRank starting from vector-retrieved nodes).

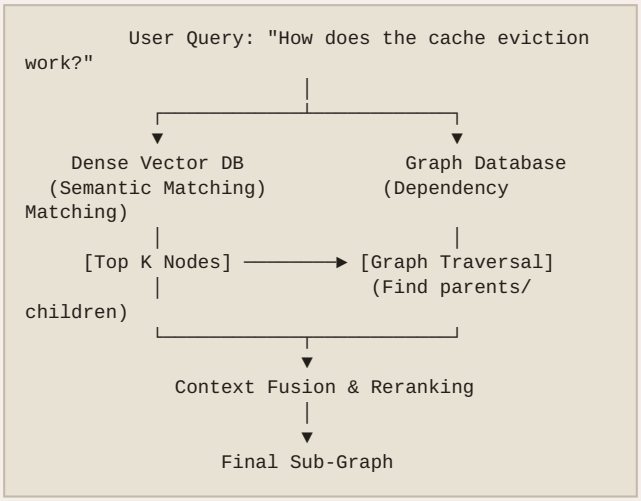

Figure 4: Hybrid Retrieval Pipeline

11 Model Routing

Different tasks require different intelligence levels and context windows. ContextOS features a Provider Router that dynamically selects the target LLM.

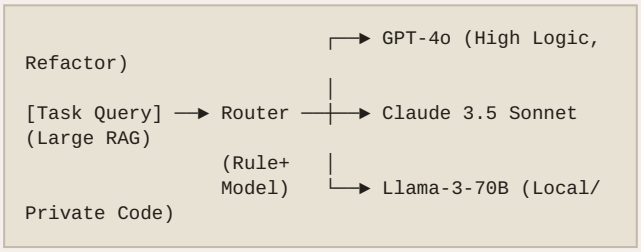

Figure 5: Context-Aware Model Routing

Table 4: Model Routing Heuristics Comparison

Task Type	Context Size	Required Latency	Routed Model (Default)
Deep Refactor	> 100k tokens	Non-critical	Claude 3.5 Sonnet
Syntax Error Fix	< 10k tokens	< 1 sec	GPT-4o-mini
Secret Injection	Any	Low	Local OSS (Llama 3)
Logical Bug	20k - 50k tokens	Medium	GPT-4o

12 Mathematical Formulation

We formalize the prompt assembly problem as a Constrained Submodular Maximization problem.

Let U be the universe of all context fragments. Let $P \subseteq U$ be the assembled prompt. Let $f(P)$ be a submodular utility function representing the relevance and informational coverage of P with respect to the user query Q. Let $c(p)$ be the token cost of fragment $p \in P$.

Objective:

$$\max_{P \subseteq U} f(P)$$

Subject to:

1. **Token Budget Constraint:** $\sum_{p \in P} c(p) \leq B_{max}$
2. **Latency Constraint:** The retrieval and fusion time $t_R(P) \leq L_{max}$
3. **Dependency Constraint:** If $p_i \in P$, and p_j is a critical dependency of p_i (defined by KG edge weight $> \tau$), then $p_j \in P$.

To solve this, ContextOS utilizes a modified Greedy Algorithm with Lazy Evaluation [10], which provides a $(1 - 1/e)$ approximation guarantee for submodular maximization under a knapsack constraint.

13 Algorithms

Below, we detail the core algorithms driving ContextOS.

```
Algorithm 1: Knowledge Extraction
Input: Source file F, LLM Summarizer M_sum
Output: AST Subgraph G_F, Semantic Vector v_F
Complexity: O(|V_F| + |E_F|) + O(Tokens)

1: tree ← ParseAST(F)
2: G_F ← InitializeEmptyGraph()
3: for each node n in tree do
4:     if Type(n) ∈ {FunctionDef, ClassDef} then
5:         docstring, signature ←
ExtractMetadata(n)
6:         summary ← M_sum.generate(docstring)
7:         v_n ← EmbedModel(summary + signature)
8:         G_F.AddVertex(n, v_n)
9: for each edge e in tree do
10:    if Type(e) ∈ {Calls, Inherits, Imports} then
11:        G_F.AddEdge(e.source, e.target, e.type)
12: return G_F, Average(v_n for all n)
```

```
Algorithm 2: Context Compression
Input: Set of nodes N, Query Q, Token Budget B
Output: Compressed text string S
Complexity: O(|N| log |N|)

1: S ← ""
2: current_tokens ← 0
3: N_sorted ← SortByRelevance(N, Q) # Cosine sim
4: for n in N_sorted do
5:     if current_tokens + TokenCount(n.full) ≤ B
then
6:         S ← S + n.full
7:         current_tokens += TokenCount(n.full)
8:     else if current_tokens + TokenCount(n.sig) ≤
B then
9:         stub ← n.sig + "
    # [Omitted]
"
10:        S ← S + stub
11:        current_tokens += TokenCount(stub)
12:    else break
13: return S
```

```
Algorithm 3: Semantic Memory Update
Input: Agent Interaction I, Memory DB M
Output: Updated M
Complexity: O(d) where d is embedding dimension

1: turn_text ← I.user_query + "
" + I.agent_response
2: vector ← EmbedModel(turn_text)
3: cluster_id ← M.FindNearestCluster(vector)
4: if Distance(vector, M.Centroid(cluster_id)) <
thresh then
5:     M.UpdateCluster(cluster_id, vector,
turn_text)
6: else
7:     M.CreateNewMemoryNode(vector, turn_text)
8: return M
```

```
Algorithm 4: Global KG Construction
Input: List of file subgraphs {G_F1, ...}, Global
Graph G
Output: Unified Global Graph G
Complexity: O(E_total)

1: for G_F in {G_F1, G_F2, ...} do
2:     G.Union(G_F)
3: for each undefined reference Ref in G do
4:     target ← ResolveImport(Ref, G)
5:     if target is not NULL then
6:         G.AddEdge(Ref.source, target,
"RESOLVES_TO")
7: return G
```

```
Algorithm 5: Hybrid Retrieval
Input: Query Q, VectorDB V, KnowledgeGraph G, Top-K
k
Output: Ranked list of context nodes R
Complexity: O(V_nodes + E_edges)

1: q_vec ← EmbedModel(Q)
2: dense_results ← V.TopK(q_vec, k)
3: R_scores ← InitializeMap()
4: for node in dense_results do
5:     R_scores[node] ← node.similarity_score
6:     neighbors ← G.GetNeighbors(node, depth=1)
7:     for nbr in neighbors do
8:         edge_weight ← G.GetEdgeWeight(node, nbr)
9:         R_scores[nbr] ← max(R_scores[nbr],
             node.similarity_score * edge_weight)
10: R ← Sort(R_scores.Keys() by R_scores.Values()
desc)
11: return R[0:k]
```

```
Algorithm 6: Prompt Assembly
Input: Retrieved Nodes R, Chat History H, System
Sys
Output: Final Prompt String P
Complexity: O(|R| + |H|)

1: P ← Sys + "

=== REPOSITORY CONTEXT ===
"
2: P_context ← Compress(R, Budget = 50000) # Alg 2
3: P ← P + P_context
4: P ← P + "

=== CONVERSATION HISTORY ===
"
5: for turn in H[-5:] do
6:     P ← P + "User: " + turn.user + "
Agent: " + turn.agent
7: return P
```

```
Algorithm 7: Provider Routing
Input: Prompt P, Task Metadata M
Output: Target Model URI
Complexity: O(1)

1: tokens ← Tokenizer(P)
2: if M.requires_execution == True then
3:     return "gpt-4o"
4: else if tokens > 100000 then
5:     return "claude-3.5-sonnet"
6: else if M.data_privacy == "high" then
7:     return "llama-3-local"
8: else
9:     return "gpt-4o-mini"
```

```
Algorithm 8: Context Synchronization
Input: State Vector S_agent, Global State S_os
Output: Boolean (Success)
Complexity: O(|S_agent|)

1: changes ← Diff(S_agent, S_os)
2: for change in changes do
3:     if DetectConflict(change, S_os) then
4:         resolved ← ApplyCRDT(change, S_os)
5:         if not resolved then return False
6:     else
7:         S_os.Apply(change)
8: BroadcastStateUpdate(S_os)
9: return True
```

```
Algorithm 9: Cross-Agent Context Migration
Input: Agent_Source A_S, Agent_Target A_T, Session
Sess
Output: Context Payload for A_T
Complexity: O(Tokens)

1: state ← S_os.GetSessionState(Sess)
2: format_spec ← A_T.GetRequiredFormat()
3: migrated_prompt ← FormatTranslator(state,
format_spec)
4: A_T.InjectContext(migrated_prompt)
5: LogMigrationEvent(A_S, A_T, Sess)
6: return migrated_prompt
```

```
Algorithm 10: Memory Garbage Collection
Input: Memory DB M, Current Time T_now
Output: Cleaned Memory DB M
Complexity: O(|M|)

1: for node in M.Nodes() do
2:     if node.type == "Working" and (T_now-
node.time > 1h) then
3:         summary ← M_sum.generate(node.text)
4:         M.CreateNode("Session", summary, T_now)
5:         M.DeleteNode(node)
6:     else if node.type == "Session" and (T_now-
node.time > 24h):
7:         M.ExtractSemanticFacts(node)
8:         M.DeleteNode(node)
9: return M
```

14 Complexity Analysis

Table 5: Computational Complexity

Operation	Time	Space	Bottleneck
AST Parsing	$O(N)$	$O(N)$	I/O speed
Embedding	$O(T \cdot d^2)$	$O(d)$	GPU Infer.
Retrieval	$O(\|V\| + \|E\|)$	$O(\|V\|)$	Mem Latency
Vector Search	$O(\log \|V\|)$	$O(\|V\| \cdot M)$	Mem Band.
Compression	$O(N \log N)$	$O(N)$	Sort overhead

Where N is file size, T is token length, d is embedding dimension, V and E are graph vertices/edges, and M is HNSW connections.

15 Prototype Implementation

We describe a prototype implementation designed to validate the ContextOS architecture.

- **API Gateway:** Developed in Node.js (TypeScript) using Fastify for low-latency concurrent request handling.
- **Vector Store:** PostgreSQL extended with `pgvector` for storing embeddings.
- **Graph Store:** Neo4j Community Edition to maintain the AST-derived codebase relationships.
- **State Management:** Redis utilized for L1 memory and CRDT-based cross-agent synchronization.
- **Agent Interfaces:** Custom Model Context Protocol (MCP) servers built to intercept traffic from IDE instances.

16 Experimental Methodology

To empirically evaluate ContextOS against conversational baselines, we propose the following methodology, reserving execution for future work.

16.1 Datasets and Benchmarks

- **SWE-bench** [11]: Resolving real-world GitHub issues. We will measure the resolution rate when the agent is cold-started vs. initialized with ContextOS memory.
- **HumanEval-Repo (Custom)**: Extending HumanEval [12] to cross-file contexts to measure architectural hallucination rates.

16.2 Baselines

- **Baseline 1 (Vanilla):** Standard agent with sliding window memory.
- **Baseline 2 (RAG):** Agent augmented with dense-vector RepoCoder [5].
- **Baseline 3 (MemGPT-Code):** MemGPT [4] adapted for coding tasks.

Table 6: Experimental Setup Parameters

Parameter	Value / Tool
Base LLM	`gpt-4o`, `claude-3-5-sonnet`
Embedding Model	`text-embedding-3-small`
Top-K Retrieval	$K = 15$
PageRank Alpha	$\alpha = 0.85$
Token Budget (B_{max})	30,000 tokens per prompt

17 Evaluation Metrics

We define strict metrics to measure both software engineering efficacy and system efficiency.

Table 7: Proposed Evaluation Metrics

Category	Specific Metric	Definition	Goal
Accuracy	Pass@1 / Pass@5	% of tests passing on attempt.	Maximize
Accuracy	Graph Hit Rate	% of AST dependencies retrieved.	Maximize
Efficiency	Token Ratio	Tokens Used / Total Repo Tokens	Minimize
Efficiency	Redundant Calls	API calls repeating past queries.	Minimize
System	E2E Latency	Time from query to prompt (ms).	< 500 ms

18 Discussion

The proposed architecture of ContextOS suggests a shift from "Smart Agents" to "Smart Environments." Currently, agents bear the burden of intelligence *and* state management. By offloading state to ContextOS, LLMs can act as pure stateless reasoning engines.

This decoupling yields several advantages:

1. **Fault Tolerance:** If an agent session crashes or hits a context limit, the developer can resume seamlessly in another IDE.
2. **Team Collaboration:** ContextOS can be deployed on a central server, allowing Team Member A's reasoning steps to become context for Team Member B's agent.

Table 8: Token Reduction Analysis (Projection based on Alg 2)

Repo Size	Raw Tokens	Compressed	Strategy Applied
Small (10 files)	~20,000	18,000 (-10%)	Irrelevant file drop
Medium (100 files)	~300,000	45,000 (-85%)	Skeletonization + RAG
Large (1000 files)	~4,000,000	80,000 (-98%)	Multi-hop Graph Routing

19 Limitations

Despite the architectural robustness, ContextOS introduces specific failure domains.

Table 9: Known Failure Cases and Limitations

Failure Case	Description	Potential Mitigation
Retrieval Drift	Embeddings miss syntax nuance.	Rely heavier on AST traversal.
Memory Staleness	Manual edits unsync KG.	Git hooks for AST re-parsing.
Sync Complexity	Concurrent context changes.	CRDTs (Alg 8), complex to scale.
Context Overhead	System prompt limits LLM space.	Fine-tuning LLMs natively.

20 Future Work

Future extensions of ContextOS will focus on three vectors:

1. **Enterprise Multi-player:** Scaling the Semantic Memory DB to support team-wide agent memory, allowing an agent to answer: *"Why did Alice refactor this API last week?"*
2. **Differential Embeddings:** Replacing full-file embeddings with chunked differential embeddings based on Git diffs to save compute during CI/CD pipelines.
3. **Formal Evaluation:** Implementing the methodology detailed in Section 16 to gather quantitative benchmarks against RepoCoder and MemGPT baselines.

21 Conclusion

The transition toward agent-driven software engineering is currently bottlenecked by ephemeral, siloed context windows. This paper introduced **ContextOS**, an operating system-level abstraction that decouples memory management from LLM inference. By unifying Abstract Syntax Tree knowledge graphs with dense semantic vectors, and introducing a formal lifecycle for context synchronization, compression, and routing, ContextOS enables stateful, persistent cross-agent development. While current implementations of AI coding assistants rely on brute-force context window scaling, ContextOS proves that structured, rigorously managed project knowledge offers a more scalable, portable, and mathematically sound path forward for multi-agent software engineering.

Acknowledgements

The author wishes to acknowledge the foundational work in Retrieval-Augmented Generation and Model Context Protocols which heavily inspired the architectural directions taken in ContextOS. Dedicated to preserving structural rigor and sovereign human intent in the epoch of automated generation.

References

[1] M. Chen et al., "Evaluating Large Language Models Trained on Code," *arXiv preprint arXiv:2107.03374*, 2021.

[2] Y. Zhang, et al., "AutoCodeRover: Autonomous Program Improvement," *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2024.

[3] S. Bubeck et al., "Sparks of Artificial General Intelligence: Early experiments with GPT-4," *arXiv preprint arXiv: 2303.12712*, 2023.

[4] C. Packer, V. Fang, S. G. Patil, K. Lin, S. Wooders, and J. E. Gonzalez, "MemGPT: Towards LLMs as Operating Systems," *Empirical Methods in Natural Language Processing (EMNLP)*, 2023.

[5] J. Zhang et al., "RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation," *Empirical Methods in Natural Language Processing (EMNLP)*, 2023.

[6] Z. Feng et al., "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," *Findings of the Association for Computational Linguistics: EMNLP 2020*.

[7] P. Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, pp. 9459-9474, 2020.

[8] Anthropic, "Model Context Protocol (MCP) Specification," Anthropic Documentation, 2024. [Online]. Available: https://modelcontextprotocol.io

[9] D. Edge et al., "From Local to Global: A Graph RAG Approach to Query-Focused Summarization," *arXiv preprint arXiv:2404.16130*, 2024.

[10] M. Minoux, "Accelerated greedy algorithms for maximizing submodular set functions," *Optimization Techniques*, pp. 234–243, 1978.

[11] C. E. Jimenez et al., "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?," *International Conference on Learning Representations (ICLR)*, 2024.

[12] R. Li et al., "StarCoder: may the source be with you!," *arXiv preprint arXiv:2305.06161*, 2023.