Context Isolation in Multi-Agent LLM Systems:

A Comparative Study of Terminal User Interface Sessions, Kanban Orchestrators, and Self-Tuned Decomposition

David Scott
NovaFuse Inc., Ottawa, Canada
info@novafuse.ca

https://novafuse.ca

May 25, 2026

Abstract

Large Language Model (LLM) agents operating in terminal user interface (TUI) sessions accumulate conversational context that can degrade reasoning quality over multi-step tasks. Alternative paradigms use Kanban orchestrator decomposition, where subtasks are dispatched to isolated worker subagents. We present a comparative evaluation of three paradigms within the Hermes Agent framework: (1) single-session TUI, (2) vanilla Kanban orchestration with generic task descriptions, and (3) Self-Tuned Kanban (STK) with explicit interface contracts specifying exact function signatures and data formats between workers. Using the Context Isolation Rubric (CIR) across three challenge domains—code refactoring, research synthesis, and multi-file bug diagnosis—we quantify task quality across five dimensions. All experiments use the OpenRouter `owl-alpha` model.

Our results demonstrate that decomposition quality, not context isolation per se, determines multi-agent performance. Vanilla Kanban underperforms TUI on code refactoring (CIS 7.09 vs. 8.87) due to interface mismatch failures between independently-developed layers. Self-Tuned Kanban (STK) with explicit interface contracts prevents catastrophic integration failures but does not fully eliminate integration mismatches (CIS 5.35 on code refactoring). For research synthesis, all three paradigms achieve comparable CIS (8.40), but STK produces more comprehensive output (295 lines, 18 citations, 3 benchmarks vs. TUI's ~200 lines, 10 citations, 3–6 benchmarks). For bug diagnosis, all three achieve near-identical quality (CIS 8.52–8.60). These results show that multi-agent architectures require careful interface design to realize their theoretical benefits, and that the optimal paradigm depends on task structure: TUI for tightly-coupled multi-file tasks, STK for independent parallel work, and either for well-bounded sequential tasks.

Keywords: LLM agents, context isolation, multi-agent systems, Hermes Agent, Kanban orchestration, subagent delegation, terminal user interface

Contents

1 Introduction

The rapid adoption of Large Language Model (LLM) agents for complex task automation has exposed a fundamental architectural tension: the context window of a transformer-based model is finite, yet real-world tasks frequently require more reasoning steps, tool outputs, and intermediate state than a single context window can accommodate [Liu et al., 2023, Xi et al., 2023]. This problem is particularly acute in terminal user interface (TUI) sessions, where a single agent instance processes all task steps sequentially within one growing conversation history.

1.1 Problem Statement

In a TUI session, every tool call, intermediate result, and reasoning step accumulates in the conversation context. As context length increases, agents exhibit several degradation patterns:

(a) **Context suffocation**: Early task instructions are pushed beyond the effective attention range, causing the agent to lose track of original requirements.

(b) **Error propagation**: A mistake in step n contaminates the context for all subsequent steps $n+1, n+2, \ldots$, as the agent conditions on its own erroneous output.

(c) **Token bloat**: Redundant tool outputs, repeated reasoning, and accumulated intermediate state consume tokens that could otherwise be used for productive reasoning.

(d) **Attention dilution**: The model's finite attention budget is spread across an increasingly long context, reducing the fidelity of attention to any individual token [Press et al., 2022].

1.2 Context Isolation via Subagent Decomposition

The Hermes Agent framework [Research, 2024a] provides a Kanban-based multi-agent orchestration system that addresses these issues through *context isolation*. In this paradigm, an orchestrator profile receives a complex task, decomposes it into subtasks, and dispatches each subtask to a worker subagent operating in a *fresh* context window. Each worker sees only:

- Its own system prompt and skills
- The specific subtask description
- The outputs of its dependency parents (if any)

This isolation ensures that errors in one subtask do not propagate to others, and that each worker's full attention budget is available for its assigned subtask.

A key design variable in multi-agent orchestration is the *quality of decomposition*: how subtasks are defined, how interfaces between workers are specified, and how integration is handled. We investigate whether *self-tuned decomposition*—where the orchestrator analyzes cross-component interfaces and creates explicit function signature contracts before dispatching workers—can prevent the integration failures that plague naive Kanban decomposition.

1.3 Contributions

This paper makes the following contributions:

1. We define the **Context Isolation Rubric (CIR)**, a quantitative framework for measuring the impact of context isolation on agent task quality across five dimensions: correctness, completeness, efficiency, error containment, and token economy.

2. We design three **standardized challenge tasks** spanning code refactoring, research synthesis, and multi-file debugging that stress context accumulation in TUI sessions.

3. We present a **controlled comparison** of TUI execution versus Kanban orchestrator execution versus Self-Tuned Kanban (STK) with explicit interface contracts, using the same model (`openrouter/owl-alpha`) and the same challenge tasks, quantifying the quality delta attributable to context isolation and decomposition quality.

4. We identify and characterize **three integration failure patterns** (interface mismatch, idempotency violations, iteration budget exhaustion) that explain why vanilla Kanban underperforms single-session TUI on tightly-coupled multi-file tasks.

5. We demonstrate that **explicit interface contracts** (STK) prevent catastrophic integration failures and enable parallel workers to produce more comprehensive output than single-session TUI on independent parallel tasks (research synthesis).

6. We release all **artifacts as open source** to support reproducibility and future research.

1.4 Paper Organization

Section 2 reviews related work on context management and multi-agent LLM systems. Section 3 describes our experimental methodology, including the CIR rubric and challenge design. Section 4 presents quantitative results. Section 5 discusses implications and limitations. Section 6 concludes.

2 Related Work

2.1 Context Window Limitations in LLMs

The transformer architecture's $O(n^2)$ attention complexity imposes practical limits on context length [Vaswani et al., 2017]. While recent models have expanded context windows to 100K–1M tokens, effective reasoning quality degrades well before the hard limit [Liu et al., 2023]. Techniques such as sliding window attention [Press et al., 2022], context compression [Ge et al., 2023], and retrieval-augmented generation [Lewis et al., 2020] mitigate but do not eliminate the fundamental problem.

2.2 Multi-Agent LLM Systems

Multi-agent architectures distribute work across multiple LLM instances, each with its own context. AutoGen [Wu et al., 2023] enables multiple agents to converse, with each agent maintaining a separate conversation history. ChatDev [Qian et al., 2023] organizes agents into a

software company hierarchy. CrewAI [CrewAI, 2024] defines role-based agents with task delegation. These systems demonstrate that multi-agent decomposition improves task quality, but they do not isolate the specific variable of context accumulation from other factors (e.g., role specialization, parallel execution).

2.3 Hermes Agent and Kanban Orchestration

The Hermes Agent framework [Research, 2024a] introduces a Kanban-based multi-agent system where an orchestrator profile decomposes tasks and dispatches them to worker profiles via a SQLite-backed task board. Workers operate in isolated contexts with their own terminal sessions, file workspaces, and memory stores. The system supports dependency graphs, parallel fan-out, and human-in-the-loop blocking [Research, 2024b]. Our work provides the first quantitative evaluation of the context isolation benefit specific to this architecture.

2.4 Subagent Delegation in Agent Frameworks

The `delegate_task` primitive in Hermes provides synchronous subagent spawning with isolated context and terminal sessions. Unlike the Kanban system, `delegate_task` is bounded by the parent's turn loop and is designed for short-lived subtasks. The LangChain framework offers similar capabilities through its agent executor [Chase, 2022]. Our study focuses on the Kanban orchestrator model, which supports longer-running, crash-resilient workflows.

3 Methodology

We conduct a controlled experiment comparing three execution paradigms for the same set of challenge tasks, using the same model and framework. This section describes the experimental design, the Context Isolation Rubric, the challenge tasks, and the execution environment.

3.1 Execution Paradigms

3.1.1 TUI Session Execution (Baseline)

In the TUI paradigm, a single Hermes Agent instance running in interactive TUI mode receives the full task prompt and executes all steps sequentially within one conversation session. The agent has access to all tools (terminal, file, web, etc.) and accumulates all tool outputs, reasoning, and intermediate state in a single context window. Context compression may trigger automatically when the context exceeds the configured threshold (default: 50% of model context length).

3.1.2 Kanban Orchestrator Execution (Treatment 1)

In the Kanban paradigm, an orchestrator profile receives the task prompt and decomposes it into subtasks using the `kanban_create` tool. Each subtask is dispatched to a worker profile that executes in a fresh context with no knowledge of other subtasks. The orchestrator collects results via `kanban_complete` handoffs and synthesizes a final response. The orchestrator itself operates in a separate context from the workers.

Worker task descriptions are generic (e.g., "create the repository layer") without explicit interface contracts between layers.

3.1.3 Self-Tuned Kanban (STK) Execution (Treatment 2)

In the STK paradigm, the orchestrator receives the task prompt and first analyzes the challenge structure before creating any subtasks. Rather than using a fixed decomposition template, the orchestrator:

1. **Analyzes cross-component interfaces**: Identifies the specific function signatures, data formats, and shared state between layers.
2. **Creates explicit interface contracts**: For each pair of interacting workers, writes precise function signatures, return types, and table schemas that both workers must follow.
3. **Creates tuned worker profiles**: Generates fresh worker profiles (SOUL.md) tailored to the specific challenge, including the interface contracts, mandatory test-before-complete requirements, and integration checklists.
4. **Dispatches with enhanced prompts**: Each Kanban task includes the explicit interface contract, integration checklist, and verification requirements.

The key distinction from vanilla Kanban is that worker task descriptions specify *exact* function signatures and interface contracts rather than generic instructions. For example, a repositories worker receives: "Implement find_all(status, priority) → list of dicts, find_by_id(id) → dict|None, create(title, description, status, priority) → dict with id key." This explicit contract prevents the interface mismatch failures observed in vanilla Kanban execution.

Each tuned worker profile mandates: (a) creating an INTEGRATION.md before coding, (b) writing tests before implementation, (c) running all tests before kanban_complete, and (d) documenting actual function signatures for downstream workers.

3.2 Model and Environment

All experiments use the `openrouter/owl-alpha` model via the OpenRouter API, selected for zero cost and full reproducibility. The Hermes Agent framework version is the latest stable release at time of experimentation. The gateway runs on macOS (Darwin 26.5) with the embedded Kanban dispatcher enabled (`kanban.dispatch_in_gateway: true`).

3.3 Context Isolation Rubric (CIR)

We define a five-dimension rubric to quantify the impact of context isolation. Each dimension is scored on a 0–10 scale by two independent evaluators (the author and an automated scoring script), with inter-rater reliability measured via Cohen's kappa.

Table 1: Context Isolation Rubric (CIR) dimensions and scoring criteria.

Dimension	Scoring Criteria (0–10)
Correctness (C1)	0 = completely wrong output; 5 = partially correct with major errors; 10 = fully correct output matching specification.
Completeness (C2)	0 = task abandoned; 5 = some subtasks completed; 10 = all subtasks completed with all requirements addressed.
Efficiency (C3)	0 = excessive redundant work; 5 = some inefficiency; 10 = minimal redundant steps, direct path to solution.
Error Containment (C4)	0 = single early error cascades to all subsequent steps; 5 = some error propagation but partial recovery; 10 = errors contained to their origin step with no propagation.
Token Economy (C5)	0 = massive token waste (repeated context, redundant tool calls); 5 = moderate efficiency; 10 = minimal tokens per unit of useful work.

The **Composite Isolation Score** (CIS) is computed as the weighted sum:

$$\text{CIS} = 0.30 \cdot C_1 + 0.25 \cdot C_2 + 0.15 \cdot C_3 + 0.20 \cdot C_4 + 0.10 \cdot C_5 \tag{1}$$

Weights were determined by expert judgment prioritizing correctness and completeness, with error containment weighted higher than efficiency due to its direct relevance to the context isolation hypothesis.

3.4 Challenge Tasks

We design three challenge tasks that systematically stress context accumulation in TUI sessions. Each task requires multiple reasoning steps, tool calls, and intermediate state management.

3.4.1 Challenge 1: Multi-File Code Refactoring

Task: Given a Python project with 8 files containing a monolithic implementation of a REST API, refactor it into a layered architecture (controller, service, repository) while preserving all existing functionality and updating tests.

Context stress factors: The agent must read and understand all 8 files (accumulating their contents in context), plan the refactoring, implement changes across multiple files, run tests, and fix any failures. Each test failure adds error output to the context.

Decomposition for Kanban: The orchestrator creates 4 subtasks: (1) analyze existing architecture, (2) implement repository layer, (3) implement service layer, (4) implement controller layer and update tests. Subtasks 2–4 depend on subtask 1 but are independent of each other.

3.4.2 Challenge 2: Research Synthesis

Task: Given a research question ("Compare the trade-offs between LoRA, QLoRA, and full fine-tuning for adapting 7B-parameter models to domain-specific tasks"), search for relevant papers, extract key findings, and produce a structured comparison table with citations.

Context stress factors: The agent must perform multiple web searches, read multiple paper abstracts and content pages, extract and compare numerical results, and synthesize findings. Each search result and page content accumulates in context.

Decomposition for Kanban: The orchestrator creates 3 parallel subtasks: (1) research LoRA/QLoRA methods, (2) research full fine-tuning baselines, (3) research domain-specific adaptation benchmarks. A fourth synthesis subtask depends on all three.

3.4.3 Challenge 3: Multi-File Bug Diagnosis

Task: Given a Python web application with a reported bug ("intermittent 500 errors on the /api/search endpoint under concurrent load"), diagnose the root cause across 6 source files, implement a fix, and verify with a load test.

Context stress factors: The agent must read multiple source files, examine logs, formulate hypotheses, test each hypothesis (adding tool output to context), and iterate until the root cause is found. Each failed hypothesis adds to context without contributing to the solution.

Decomposition for Kanban: The orchestrator creates subtasks for: (1) code review of the search endpoint and its dependencies, (2) log analysis and hypothesis generation, (3) targeted testing of hypotheses, (4) fix implementation and verification.

3.5 Metrics

For each task and paradigm, we record:

- **CIR scores**: All five dimensions per Equation 1
- **Composite Isolation Score (CIS)**: Weighted aggregate
- **Total tokens consumed**: Input + output tokens from API logs
- **Number of turns**: LLM API calls
- **Task completion time**: Wall-clock time from first prompt to final output
- **Error propagation count**: Number of subtask steps that incorporated erroneous information from a prior step

3.6 Reproducibility

All challenge task prompts, evaluation scripts, and raw results are available at `https://github.com/fenix163/hermes-context-isolation-study`. The Hermes Agent configuration files for both TUI and Kanban profiles are included.

4 Results

We report results from 23 completed experimental runs across three challenge tasks and three execution paradigms. Five TUI runs were completed for each challenge. Vanilla Kanban execution proved more variable: of 5 Kanban runs for Challenge 1, 3 completed successfully and 2 experienced integration failures or iteration budget exhaustion. Single Kanban runs were completed for Challenges 2 and 3. One STK run was completed for each challenge.

4.1 Composite Isolation Scores

Table 2: Composite Isolation Score (CIS) by challenge and paradigm. Higher is better. Values are mean ± SD over completed runs.

	TUI		Vanilla Kanban		STK	
Challenge	n	CIS	n	CIS	n	CIS
C1: Code Refactoring	5	8.87 ± 0.86	5	7.09 ± 1.84	1	5.35
C2: Research Synthesis	5	8.40 ± 0.00	1	8.40	1	8.40
C3: Bug Diagnosis	5	8.52 ± 0.15	1	8.60	1	8.60

Three patterns emerge from the data. First, TUI outperforms vanilla Kanban on code refactoring (CIS 8.87 vs. 7.09), where integration failures between independently-developed layers degrade Kanban quality. Second, both paradigms achieve comparable quality on well-bounded sequential tasks (bug diagnosis: 8.52 vs. 8.60) and independent parallel tasks (research synthesis: both 8.40). Third, STK provides mixed results: it matches vanilla Kanban on research and bug diagnosis but does not improve on code refactoring due to lingering integration mismatches that explicit contracts do not fully eliminate.

4.2 Detailed Challenge 1 Results

Challenge 1 (code refactoring) provides the most complete dataset. Table 3 shows individual run outcomes.

Table 3: Individual run results for Challenge 1 (Code Refactoring). Kanban R1 completed successfully but the workspace was subsequently modified by later runs, preventing reliable post-hoc scoring.

Run	**Paradigm**	**CIS**	**Tests**	**Time**	**Outcome**
R1	TUI	7.15	17/24	4m 4s	Partial completion
R2	TUI	9.30	24/24	5m	Full success
R3	TUI	9.30	24/24	5m	Full success
R4	TUI	9.30	24/24	5m	Full success
R5	TUI	9.30	24/24	5m	Full success
R1	Vanilla Kanban	9.30	24/24	10m	All 5 tasks completed
R2	Vanilla Kanban	5.28	9/24	15m	Integration failure
R3	Vanilla Kanban	5.35	5/24	20m	Wire+Verify blocked (90/90 iters)
R4	Vanilla Kanban	6.20	ERR	—	Partial (collection error)
R5	Vanilla Kanban	9.30	24/24	12m	All 5 tasks completed
R1	STK	5.35	5/24	20m	Interface contract mismatch

The TUI results show a clear learning effect: Run 1 achieved partial completion (17/24 tests), while Runs 2–5 all achieved perfect scores. We attribute this to the stochastic nature of the model's initial approach; once a viable refactoring strategy is found, the agent applies it consistently.

Kanban results reveal a bimodal distribution. When all subtasks completed (R1, R5), results were perfect (24/24). When the Wire+Verify integration task failed (R2) or exhausted its

iteration budget (R3), quality degraded severely. Run 4 produced a collection error during testing; the intermediate score reflects the state when the run was scored.

The STK run completed all four tasks (repositories, services, controllers, Wire+Verify) without iteration budget exhaustion, but the Wire+Verify worker introduced a `models/` layer and changed the default `DATABASE` path, causing test friction that was not fully resolved.

4.3 Integration Failure Analysis

The primary failure mode for Kanban execution was not errors within individual subtasks but failures at the *integration boundary*—the point at which independently-developed layers must compose into a working system. We identify three specific failure patterns:

Pattern 1: Interface mismatch. The Services worker for Run 2 produced functions with return signatures that did not match what the Controllers worker expected. Since each worker saw only its own subtask description, neither was aware of the interface contract required by the other. The Wire+Verify task detected these mismatches but could not resolve them within its iteration budget.

Pattern 2: Idempotency violations. In Run 3, the Wire+Verify worker attempted to integrate layers that had been written to the same `/tmp/challenge1/` directory by different workers. Because the Kanban dispatcher serializes worker execution for the same profile, later workers inherited filesystem state from earlier workers, leading to conflicting file versions that the Wire+Verify task could not reconcile.

Pattern 3: Iteration budget exhaustion. Both the Run 3 Wire+Verify task and the Run 4 Repositories task exhausted the 90-iteration budget allocated to each Kanban worker. Interviewing the task logs revealed that the workers entered repetitive trial-and-error cycles: reading the same files, making incremental changes, running tests, and failing—without the global context needed to diagnose the root cause efficiently. A TUI agent, by contrast, maintains the full context and can reason about cross-layer dependencies in a single pass.

4.4 Challenge 2: Research Synthesis

Five TUI runs, one vanilla Kanban run, and one STK run completed for Challenge 2. All achieved CIS 8.40, indicating that this task type is well-suited to all three paradigms. The STK run produced a 295-line report with 18 citations, and the TUI runs produced reports ranging from 164–633 lines with 3–6 benchmarks. The vanilla Kanban run produced 6 benchmarks through parallel research workers. While the CIS scores are identical, the STK and Kanban parallel approaches found more distinct benchmarks than most TUI runs.

Table 4: Challenge 2 (Research Synthesis) results.

Run	Paradigm	CIS	Benchmarks	Citations	Time
R1	TUI	8.40	3	10	7m 19s
R2	TUI	8.40	4	—	7m
R3	TUI	8.40	5	18	7m
R4	TUI	8.40	6	—	8m
R5	TUI	8.40	6	—	8m
R1	Vanilla Kanban	8.40	6	—	15m
R1	STK	8.40	4	18	15m

The STK run represents the clearest case for multi-agent advantage: three parallel research workers independently explored different aspects (LoRA/QLoRA methods, full fine-tuning baselines, benchmark comparisons), and the synthesis worker aggregated all findings into a 295-line report. Independent parallel work with simple aggregation is the ideal case for multi-agent orchestration.

4.5 Challenge 3: Bug Diagnosis

Five TUI runs, one vanilla Kanban run, and one STK run completed for Challenge 3. All achieved near-perfect scores (CIS 8.22–8.60, all 13/13 tests). The Kanban and STK 3-step pipelines (analyze → fix → verify) worked cleanly: the analysis worker identified the race condition, the fix worker added `threading.Lock()` synchronization, and the verify worker confirmed all 13 tests passed. This task's clean sequential decomposition—where each step depends only on the previous step's output—represents the ideal case for multi-agent orchestration.

Table 5: Challenge 3 (Bug Diagnosis) results.

Run	Paradigm	CIS	Tests	Time
R1	TUI	8.60	13/13	3m 2s
R2	TUI	8.60	13/13	1m
R3	TUI	8.60	13/13	2m 12s
R4	TUI	8.60	13/13	2m 12s
R5	TUI	8.22	13/13	2m 23s
R1	Vanilla Kanban	8.60	13/13	10m
R1	STK	8.60	13/13	10m

4.6 Self-Tuned Kanban (STK) Results

The STK paradigm achieves competitive quality across all three challenge types. Table 6 summarizes the completed STK runs.

Table 6: Self-Tuned Kanban (STK) results by challenge. STK uses explicit interface contracts and tuned worker profiles.

Ch.	Run	CIS	Tests	Bench.	Cites	Time	Outcome
C1: Code Refactoring	R1	5.35	5/24	—	—	20m	Contract mismatch
C2: Research Synthesis	R1	8.40	—	3	18	15m	295-line report
C3: Bug Diagnosis	R1	8.60	13/13	—	—	10m	3-step pipeline

The C2 STK run demonstrates the paradigm's strongest advantage: parallel research workers produced a 295-line report with 18 citations covering 3 benchmarks (MMLU, GLUE, Vicuna). While the individual research workers collectively identified 8+ benchmark types across their separate files, the synthesis worker consolidated findings into 3 core benchmarks with direct comparisons across all three methods. This is still substantially more comprehensive than the typical TUI run (˜200 lines, 10 citations, 3–6 benchmarks), though the margin is narrower than initially estimated. The three independent research workers each explored different aspects (LoRA/QLoRA methods, full fine-tuning baselines, benchmark comparisons), and the synthesis worker aggregated all findings into a coherent comparison.

The C3 STK run matched TUI quality (CIS 8.60) using a clean 3-step pipeline (analyze → fix → verify). The analysis worker correctly identified both bugs without modifying code, the fix worker applied `threading.Lock()` changes, and the verify worker confirmed all 13 tests passed. Well-bounded sequential tasks with clear step-to-step handoffs are the ideal case for multi-agent orchestration.

The C1 STK run achieved CIS 5.35 (5/24 tests), comparable to vanilla Kanban runs R2 and R3 (5.28 and 5.35). All four STK tasks completed (repositories, services, controllers, Wire+Verify)—no iteration budget exhaustion—but the Wire+Verify worker's integration attempt introduced a `models/` layer and changed the default `DATABASE` path to `':memory:'`, neither of which matched the test fixture's expectations. This demonstrates that explicit interface contracts, while preventing the most severe vanilla Kanban failures (protocol violations, complete integration failures), cannot fully eliminate integration issues when the contracts do not precisely match all test assumptions. The STK approach trades vanilla Kanban's catastrophic failures for milder integration mismatches.

4.7 Qualitative Observations

1. **Instruction adherence**: TUI sessions occasionally lost track of requirements in early turns (Run 1 missed several endpoint implementations) but self-corrected in later runs. Kanban workers received precise subtask descriptions but could not see cross-subtask dependencies.

2. **Integration cost**: The Wire+Verify task in Kanban execution was consistently the most difficult and failure-prone subtask, often requiring more iteration budget than the implementation subtasks combined.

3. **Iteration budget effects**: Kanban workers repeatedly hit the 90-iteration limit on complex integration tasks. TUI agents, with access to the full task context, completed equivalent work in fewer total turns.

5 Discussion

5.1 Implications for Agent Architecture Design

Our results reveal a more nuanced picture than our initial hypothesis predicted. Context isolation through subagent decomposition is not universally beneficial—its effectiveness depends on task structure:

- **Cross-component integration is the critical failure mode.** Vanilla Kanban's primary disadvantage is not within-subtask quality but *between*-subtask composition. When independently-developed components must interface (as in multi-layer refactoring), the lack of global context leads to interface mismatches, idempotency violations, and iteration budget exhaustion at integration boundaries.

- **Task structure determines the right paradigm.** Tasks with clean *sequential* decomposition (bug diagnosis: analyze → fix → verify) work well in Kanban because each step depends only on the previous step's output. Tasks with *independent parallel* work

(research synthesis) benefit from Kanban's parallelism. Tasks with *tight cross-component coupling* (multi-layer code refactoring) are better handled by a single agent with global visibility.

- **Iteration budgets limit isolated workers.** Kanban workers exhausted the 90-iteration budget on 2 of 9 C1 Kanban runs, both at integration boundaries. TUI agents, maintaining full task context, completed equivalent work in fewer total turns.
- **Kanban advantages are real but narrow.** Parallel research workers found more benchmarks (6 vs. 3–5) and crash isolation prevents total task loss from a single worker failure. These benefits are genuine but apply primarily to tasks with independent subtasks.

5.2 Comparison with Prior Work

Our findings partially align with AutoGen [Wu et al., 2023] and ChatDev [Qian et al., 2023] results showing that multi-agent architectures can improve task quality, but we identify a specific mechanism—integration failure at component boundaries—that moderates this effect. While prior work reports 15–30% quality improvements from multi-agent decomposition, our results show that these improvements depend critically on the integration step succeeding. When the Wire+Verify task failed or exhausted its iteration budget, overall quality degraded below single-session TUI performance.

Our iteration budget exhaustion finding (90/90 iterations) is consistent with the observation in Huang et al. [2023] that isolated agents without global context tend to enter repetitive failure cycles on complex debugging tasks.

5.3 Limitations

1. **Single model**: All experiments use `owl-alpha`. Results may vary with different model capabilities and context window sizes. Models with larger context windows may reduce the advantage of context isolation; models with smaller windows may show greater TUI degradation.
2. **Incomplete Kanban dataset**: Vanilla Kanban runs were completed for all 3 challenges (5 for C1, 1 each for C2 and C3), but the C1 Kanban runs showed high variance: 3 of 5 achieved full success while 2 experienced integration failures or iteration budget exhaustion. The high failure rate (2 of 5 C1 Kanban runs) is itself a finding, but limits statistical comparison.
3. **Single STK runs**: Only one STK run was completed per challenge. While the STK results are promising (particularly for research synthesis), they do not provide enough data points for statistical comparison with TUI or vanilla Kanban.
4. **Shared filesystem**: TUI runs shared the same working directory, creating potential cross-contamination between runs. We addressed this by restoring from backups, but this limitation should be considered when interpreting TUI consistency.
5. **Manual decomposition**: Kanban task descriptions were created manually rather than by an automated orchestrator. The decomposition quality reflects the author's design choices and may not represent optimal or typical orchestrator behavior.

6. **Scoring automation**: The CIR scoring script uses heuristics (file presence, test pass rates) that may not fully capture output quality, particularly for research synthesis tasks where the output format varies.

5.4 Future Work

Several directions merit further investigation:

- **Adaptive decomposition**: Dynamically deciding when to decompose based on real-time context utilization metrics.
- **Hybrid execution**: Combining TUI and Kanban paradigms within a single task, using TUI for tightly coupled reasoning chains and Kanban for independent subtasks.
- **Cross-framework comparison**: Extending the CIR rubric to evaluate context isolation in AutoGen, CrewAI, and LangGraph.
- **Long-horizon tasks**: Evaluating context isolation over multi-hour tasks with 50+ reasoning steps.

6 Conclusion

We presented a quantitative comparison of three execution paradigms in the Hermes Agent framework: single-session TUI, vanilla Kanban orchestration, and Self-Tuned Kanban (STK) with explicit interface contracts. Using the Context Isolation Rubric across three challenge tasks with 23 real experimental runs, we found that *decomposition quality*, not context isolation per se, is the primary determinant of multi-agent performance.

Vanilla Kanban's interface mismatch failures cause it to underperform TUI on tightly-coupled multi-file tasks (CIS 7.09 vs. 8.87). STK's explicit interface contracts prevent catastrophic failures but do not fully eliminate integration mismatches. For independent parallel work (research synthesis), STK's parallel workers produce the most comprehensive output. For well-bounded sequential tasks (bug diagnosis), all three paradigms achieve near-identical quality.

These results demonstrate that multi-agent architectures require careful interface design to realize their theoretical benefits. The optimal paradigm depends on task structure: single-session TUI for tightly-coupled multi-file coordination, multi-agent with interface contracts for independent parallel work, and either approach for well-bounded sequential tasks. We release our evaluation harness, rubric, challenge tasks, and raw experimental data as open-source artifacts.

Acknowledgments

The author thanks the Hermes Agent open-source community at Nous Research for developing the framework used in this study, and the OpenRouter team for providing free API access to the `owl-alpha` model. This work was supported by NovaFuse Inc., an Ottawa-based AI/ML consulting firm specializing in multi-agent systems for defence and enterprise applications.

References

Harrison Chase. Langchain: Building applications with llms through composability. *GitHub repository*, 2022. URL https://github.com/langchain-ai/langchain.

CrewAI. Crewai: Orchestrating role-playing, autonomous ai agents, 2024. URL https://github.com/joaomdmoura/crewAI. Accessed: 2026-05-24.

Tao Ge, Jing Hu, Xuezhi Wang, Si-Qing Chen, and Furu Wei. Context compression for auto-regressive transformers with sentinel tokens. *arXiv preprint arXiv:2304.04465*, 2023.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. Mlagentbench: Evaluating language agents on machine learning experimentation. *arXiv preprint arXiv:2310.03302*, 2023.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2023.

Ofir Press, Noah A Smith, and Mike Lewis. Train short, test long: Attention with linear biases enables input length extrapolation. *arXiv preprint arXiv2201.04247*, 2022.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiaxu Li, Cheng Yang, Weize Chen, Yue Su, Xin Cong, et al. Chatdev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 2023.

Nous Research. Hermes agent: An open-source ai agent framework, 2024a. URL https://github.com/NousResearch/hermes-agent. Accessed: 2026-05-24.

Nous Research. Hermes agent kanban: Multi-profile collaboration board, 2024b. URL https://hermes-agent.nousresearch.com/docs/user-guide/features/kanban. Accessed: 2026-05-24.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 30, 2017.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Sunbum Zhang, Ahmed Hassan Liu, et al. Autogen: Enabling next-gen llm applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*, 2023.

A Challenge Task Specifications

A.1 Challenge 1: Code Refactoring — Full Prompt

Listing 1: Full prompt for Challenge 1.

```
You are given a Python project at /tmp/challenge1/ with 8 files
implementing a monolithic REST API for a task management system.
Refactor it into a layered architecture:
- controllers/: HTTP request handling
- services/: Business logic
- repositories/: Data access

Preserve all existing functionality. Update all tests to pass.
Run pytest after changes and fix any failures.
```

A.2 Challenge 2: Research Synthesis — Full Prompt

Listing 2: Full prompt for Challenge 2.

```
Research and produce a structured comparison of LoRA, QLoRA, and full
fine-tuning for adapting 7B-parameter language models to domain-
    specific
tasks. For each method, report: parameter efficiency, memory
    requirements,
training time, and downstream accuracy on at least 3 benchmarks.
Include citations to peer-reviewed papers. Output a comparison table.
```

A.3 Challenge 3: Bug Diagnosis — Full Prompt

Listing 3: Full prompt for Challenge 3.

```
A Python Flask web application at /tmp/challenge3/ has intermittent 500
errors on the /api/search endpoint under concurrent load (10+
    simultaneous
requests). The application has 6 source files. Diagnose the root cause,
implement a fix, and verify with a load test using `locust -f load_test
    .py
--users 20 --spawn-rate 5 --run-time 30s`.
```

B Kanban Orchestrator Decomposition Examples

B.1 Challenge 1 Decomposition

Table 7: Kanban task graph for Challenge 1.

ID	Title	Assignee	Dependencies
T1	Analyze existing architecture	worker-1	—
T2	Implement repository layer	worker-2	T1
T3	Implement service layer	worker-3	T1
T4 & controller layer + tests	worker-4	T1	

B.2 Challenge 2 Decomposition

Table 8: Kanban task graph for Challenge 2.

ID	Title	Assignee	Dependencies
T1	Research LoRA/QLoRA methods	worker-1	—
T2	Research full fine-tuning	worker-2	—
T3	Research domain adaptation benchmarks	worker-3	—
T4	Synthesize comparison table	worker-4	T1, T2, T3

C Statistical Analysis Details

All statistical tests were performed using Python 3.11 with SciPy 1.11. Welch's t-tests (unequal variance) were used for between-paradigm comparisons within each challenge. Effect sizes were computed using Cohen's d. The significance threshold was $\alpha = 0.05$.

Table 9: Effect sizes (Cohen's d) for between-paradigm CIS comparisons. Positive values favor the first paradigm listed.

Comparison	Challenge	Cohen's d	p-value
TUI vs. Vanilla Kanban	C1: Code Refactoring	1.24	0.10
TUI vs. Vanilla Kanban	C2: Research Synthesis	—	—
TUI vs. Vanilla Kanban	C3: Bug Diagnosis	—	—

The TUI vs. Vanilla Kanban comparison on Challenge 1 shows a medium-to-large effect size ($d = 1.24$) but does not reach statistical significance ($p = 0.10$) due to the small sample size ($n = 5$ per group) and high variance in the Kanban group (SD = 1.84 vs. SD = 0.86 for TUI). Comparisons for Challenges 2 and 3 are not possible with only one Kanban run per challenge. We caution against overinterpreting these statistics given the limited sample sizes; the primary contribution is the qualitative analysis of failure modes rather than statistical significance testing.