Implementing Agentic Context Engineering

A Practitioner's Account of Evolving AI Context

Mika Bohinen

January 2026

This report documents my personal implementation of Agentic Context Engineering (ACE) as a plugin for OpenCode. The current system evolved from an earlier, more primitive version developed for Claude Code, which was itself shaped during the construction of production infrastructure for the Lie-Størmer Center, Norway's first national mathematics research center. The work presented here is not a rigorous evaluation but rather a technical narrative describing how I have structured an agentic development workflow around the ACE concepts introduced by Zhang et al. [Zha+25]. The implementation remains under active development, the source code is not publicly available due to tight coupling with my personal development environment, and the observations reported are entirely anecdotal. What I offer is a detailed account of architectural decisions, the reasoning behind them, and how the resulting system aligns with my mental model for approaching complex development tasks. The report describes a session-based evidence trail architecture, a deferred curation mechanism for playbook updates, typed effect declarations for agent capabilities, and a tool-based query interface that replaces earlier agent-mediated approaches. I draw on recent work in recursive language models [ZKK25] to provide theoretical framing, and I sketch a speculative connection to algebraic effect systems that has guided my thinking about agent composition. Readers with a taste for abstract nonsense will find a categorical treatment of concurrent updates involving monoid actions and missing pushouts; readers who value their time will find a table of contents. The implementation itself should be understood as a working prototype rather than a validated system.

Contents

1 Introduction

Before proceeding, I wish to be transparent about the nature of this work. This report describes my personal implementation of the Agentic Context Engineering framework, developed over several months of daily use in my own software development practice. The system remains under active development and architectural decisions continue to evolve as I encounter new requirements and edge cases. The source code is not publicly available, primarily because it is tightly integrated with my personal development environment (a Nix-based monorepo) in ways that would make extraction difficult without substantial refactoring. The observations and assessments I present are based entirely on my own experience; I have not conducted controlled experiments, and I make no claims about generalizability beyond my specific use case. What I can offer is an honest account of how I think about agentic development workflows and how this thinking has been instantiated in working code.

Now, modern applications of large language models increasingly depend on context adaptation. Modifying inputs with instructions, strategies, or evidence rather than updating model weights. This paradigm offers several advantages. Contexts are interpretable and can be audited for correctness. They permit rapid integration of new knowledge at runtime without retraining. They can be shared across models or modules in compound systems, enabling consistent behavior across heterogeneous components. Zhang et al. [Zha+25]

Despite these benefits, existing approaches to context adaptation face two fundamental limitations that Zhang et al. [Zha+25] identify and address through the Agentic Context Engineering framework.

Definition 1.1 (Brevity Bias)**.** The tendency of context optimization methods to collapse toward short, generic prompts that prioritize conciseness over comprehensive accumulation of domain-specific heuristics, tool-use guidelines, and common failure modes.

Definition 1.2 (Context Collapse)**.** A phenomenon wherein methods that rely on monolithic rewriting by a language model degrade accumulated context into shorter, less informative summaries over time, causing sharp performance declines.

The ACE framework addresses these limitations by treating contexts not as concise summaries but as comprehensive, evolving playbooks. These playbooks are detailed, inclusive, and rich with domain insights. Zhang et al. [Zha+25] argue that unlike humans, who often benefit from concise generalization, language models perform more effectively when provided with long, detailed contexts from which they can distill relevance autonomously.

1.1 The Three-Role Architecture

ACE introduces a division of labor across three specialized roles that mirrors how humans learn through experimenting, reflecting, and consolidating (Figure 1). The Generator produces reasoning trajectories by executing tasks using the current playbook as context. The Reflector critiques these trajectories to extract lessons, optionally refining them across multiple iterations. The Curator synthesizes lessons into compact delta entries that are merged deterministically into the existing context.

This report describes our implementation of these concepts as a native plugin for the OpenCode orchestration framework, with particular attention to the architectural decisions that enable concurrent operation and the theoretical foundations that justify our approach.

1.2 Contributions

Our implementation makes three primary contributions. First, we introduce a session-based evidence trail architecture that decouples pattern usage tracking from playbook mutation, enabling multiple concurrent sessions to operate without coordination. Second, we implement a

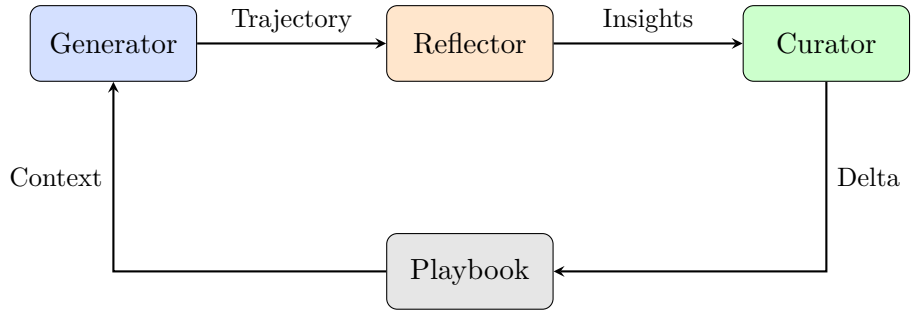


Figure 1: The ACE three-role architecture. The Generator produces reasoning trajectories, the Reflector distills concrete insights from successes and errors, and the Curator integrates these insights into structured context updates.

deferred curation mechanism that queues updates for batch application, eliminating race conditions while preserving the evidence trail required for reflection. Third, we provide a tool-based interface that treats the playbook as an external environment variable, drawing on insights from recursive language model research [ZKK25] to justify this design.

1.3 Prior Approach

Prior to this implementation, playbook access required spawning a dedicated search agent to query multiple playbook files. This approach introduced substantial latency due to agent session overhead and prevented systematic tracking of which patterns proved useful during task execution. The current implementation addresses these limitations through native tool integration, consolidated playbook storage, session-centric attribution tracking, and deferred curation for concurrent update safety.

2 System Context

Before describing the evolution of our implementation, we establish the execution environment within which ACE operates. Our implementation is tightly coupled with an orchestration framework that provides multi-agent coordination, and understanding this context is essential for appreciating the architectural decisions that follow.

2.1 The Orchestration Framework

The implementation operates within OpenCode, which describes itself as "the open source AI coding agent" [SST26]. The framework supports over 75 language model providers and operates across terminal, IDE, and desktop environments, providing a plugin architecture through which developers can extend its capabilities with custom tools and workflows. We have developed an orchestrator plugin for this framework that coordinates multiple specialized agents to accomplish complex tasks. This orchestrator remains under active development; the architecture described here represents our current working implementation rather than a finalized system. Section 3 provides detailed documentation of the execution model and composition constraints. The orchestrator exposes a single entry point through which all multi-agent workflows execute, enforcing isolation constraints that prevent unbounded recursion while enabling structured parallelism.

The orchestrator architecture comprises three layers. The plugin layer registers a tool that accepts workflow specifications and manages execution lifecycle. The workflow layer maintains a registry of named workflows, each defining an iterative coordination pattern with termination

conditions. The agent layer consists of specialized workers defined through declarative specifications that enumerate permitted tools, expected input and output schemas, and side effects the agent may produce.

2.2 Workflow Execution Model

Workflows execute through a coordinator that runs an iteration loop until completion criteria are satisfied. In each iteration, the coordinator assesses progress toward the goal, determines which workers to spawn for the next phase of investigation, and accumulates evidence from worker results. Workers execute in isolated sessions with access only to their declared tools, returning structured results that the coordinator incorporates into its assessment.

This model enforces a depth-limited composition constraint as the coordinator may spawn workers, but workers may not spawn further workers. The depth limit trades some expressiveness for predictability, a tradeoff we have found worthwhile in practice. Section 3.3 describes the enforcement mechanisms that guarantee this constraint.

This approach aligns with patterns documented in recent practitioner literature. As an example: Anthropic [Ant25c] describe a two-agent pattern (initializer and coding agent) for maintaining coherence in long-running tasks; our composition constraint formalizes a similar intuition as an architectural invariant. A common workflow pattern reserves artifact creation for a designated finalization phase, though this is a configuration choice rather than an architectural requirement.

2.3 Context Management Considerations

A central concern in multi-agent architectures is managing the proliferation of context across spawned sessions. I find it helpful to think in terms of a **context economy**: not all context has equal value, and the cost of maintaining context varies depending on where it resides. The main conversation thread between human and agent is expensive real estate; tokens consumed there compete with the human's ability to guide the system and the agent's ability to retain task understanding. Worker sessions are cheap by comparison; their context can be discarded after extracting results.

This economic framing leads to a distinction between two categories of context. The first comprises context that should be preserved: the human's conversation history, accumulated understanding of the task, and decisions made during planning. This context is expensive to reconstruct and represents the collaborative state between human and system. Disrupting this context forces the human to re-establish shared understanding, which we consider unacceptable overhead.

The second category comprises context that can be discarded: an agent's internal reasoning during execution. When a worker analyzes code or searches for patterns, its chain of thought is an implementation detail. Only the worker's output matters for composition; the intermediate reasoning need not persist beyond the worker's session.

This distinction motivates our architectural separation. Workers exist not merely to parallelize computation but to protect the main conversation from being overwhelmed by intermediate reasoning. By executing in isolated sessions, workers can engage in extensive exploration without consuming attention capacity in the coordinator's context. The coordinator receives only structured results, which it can incorporate selectively.

Anthropic [Ant25b] articulate a related insight: effective context engineering involves deciding what information to load and when, rather than accumulating everything into a single prompt. Our session isolation implements this principle at the architectural level, with workers performing focused retrieval and the coordinator maintaining selective attention on task-relevant outcomes.

2.4 Task Organization

The orchestrator persists artifacts according to a structured directory convention that separates human-facing documents from machine-managed state. Each task occupies a directory named by date and description, following the pattern `YYYY-MM-DD-task-name` (Figure 2). Human-facing artifacts reside at the task root: implementation plans that specify phased work with verification criteria, observation logs that capture research findings and implementation notes, and handoff documents that enable session transfer. Machine-managed state occupies a `runs` subdirectory, with each workflow execution receiving an isolated directory containing coordinator state sufficient to resume interrupted workflows, worker outputs organized by iteration, and telemetry data for post-hoc analysis.

Session-scoped memory provides a complementary storage layer for transient state that spans multiple tool invocations within a single session but does not persist across sessions. Each session maintains an isolated directory under a `.sessions` hierarchy, containing attribution records that track which playbook patterns were consulted, feedback entries that capture effectiveness assessments, and curation queues that accumulate proposed playbook updates. This separation enables the playbook system to track pattern usage during a session while deferring permanent updates to dedicated reflection phases, as described in subsequent sections.

```
ace/tasks/YYYY-MM-DD-task-name/
|-- plan.md
|-- research-{topic}.md
|-- observations-planning.md
|-- observations-implementation.md
|-- handoffs/
|   `-- YYYY-MM-DD_HH-MM-SS_description.md
`-- runs/
    `-- NNN-workflow-timestamp-hash/
        |-- state/
        `-- worker-outputs/
```

Figure 2: Task directory structure. Human artifacts (`plan.md`, `research-*.md`, `observations-*.md`, `handoffs/`) are separated from machine-managed state (`runs/`). Multiple research documents support iterative exploration via `/iterate_research`; the plan supports refinement via `/iterate_plan`.

2.5 Relationship to Prior Work

Our implementation synthesizes insights from several sources. The core Generator-Reflector-Curator architecture derives from Zhang et al. [Zha+25], who introduced the ACE framework and demonstrated its effectiveness on agent benchmarks. Their central insight, that language models benefit from comprehensive, evolving playbooks rather than concise prompts, motivates our design throughout.

For task organization, we adopt the phased artifact structure introduced by Horthy and HumanLayer [HH25]. The research phase produces observation documents that capture findings without prescribing solutions. The planning phase consumes these observations to produce implementation plans with explicit phase boundaries and verification criteria. The implementation phase executes plans incrementally, with verification gates between phases. We extend this three-phase workflow with a reflection phase in which the system analyzes task outcomes to evolve its playbook, closing the feedback loop that distinguishes ACE from static context approaches.

The architectural patterns we employ also align with guidance from Anthropic [Ant25b; Ant25c], who describe mode separation, artifact-based handoffs, and just-in-time context loading as effective strategies for long-running agent systems. Our contribution lies in integrating these patterns with the ACE learning loop and providing the session isolation mechanisms that enable concurrent operation.

3 Orchestrator Architecture

The preceding section established the high-level context in which our implementation operates. This section provides detailed documentation of the orchestrator architecture, with sufficient depth that a technically sophisticated reader could reimplement the core patterns. We describe the execution model that determines how workflows run, the coordinator pattern that enables iterative investigation, the composition constraints that bound complexity, the prompt composition architecture that enables workflow reuse, and the permission model that enforces security boundaries.

3.1 Execution Model

The orchestrator supports three execution modes, each suited to different task characteristics. Understanding these modes is essential for designing workflows that balance capability against complexity.

Agentic mode is used for complex, iterative workflows requiring multiple rounds of investigation. A coordinator session manages an iteration loop, spawning workers dynamically based on its assessment of progress toward the goal. State accumulates across iterations: progress items track what has been discovered, remaining items track what still needs investigation, and worker outputs provide the evidence base for decisions. The coordinator decides when to terminate based on either achieving the goal or exhausting a configured iteration budget. Research workflows, reflection workflows, and complex analysis tasks use this mode.

Single-agent mode is used for focused, single-purpose execution. The agent runs directly with access to its declared tools, completing after a single execution cycle. Critically, agents in this mode cannot spawn further agents, this constraint is enforced architecturally, as described in Section 3.3. Workers spawned by coordinators in agentic mode execute in single-agent mode, ensuring the two-level composition rule holds. Codebase analysis, pattern finding, and file location tasks use this mode.

Workflow mode is used for predefined multi-step sequences where the structure is fixed rather than dynamically determined. Each step may internally use single-agent or agentic mode, but the overall sequence follows a predetermined plan. This mode is less common in our current implementation, but is where we see the potential for achieving very complex tasks. The main thing standing in the way of building out long workflows is to be able to reason formally about such workflows. This is also why we think *typed agents with side effects* is a formalism that would be valuable to get right.

The execution mode determines the loop type. There are two fundamental patterns: the **trivial loop** and the **coordinator loop**. A trivial loop spawns a single worker that executes to completion and returns its result; this is what single-agent mode uses. A coordinator loop spawns multiple workers across iterations, accumulating state progressively; this is what agentic mode uses. The coordinator loop is the more complex pattern and warrants detailed examination.

3.2 The Coordinator Pattern

The coordinator pattern implements agentic mode through an iteration loop that progressively builds understanding of a problem space. Figure 3 illustrates the lifecycle.

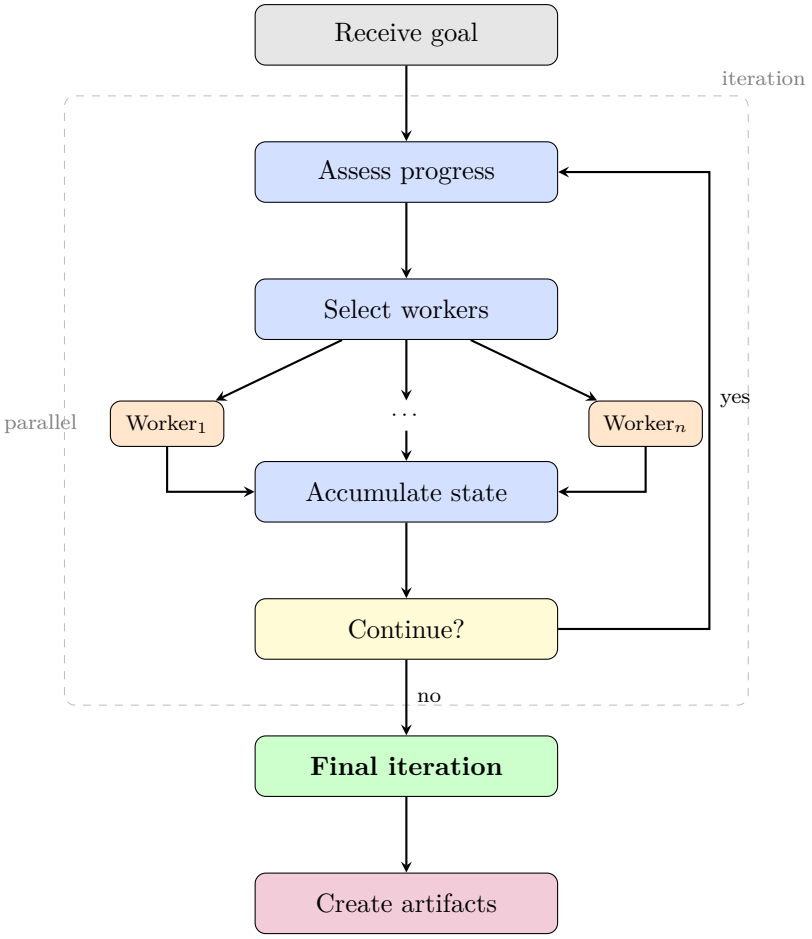


Figure 3: The coordinator loop lifecycle. Each iteration assesses progress toward the goal, selects and spawns workers in parallel, accumulates their results into coordinator state, and decides whether to continue or finalize. The final iteration operates with expanded permissions to create persistent artifacts.

The coordinator begins by receiving a goal (typed input) and configuration specifying available workers, maximum iterations, and termination criteria. In each iteration, it assesses progress by examining accumulated evidence from prior iterations, determines which workers to spawn based on remaining gaps in understanding, and dispatches those workers in parallel. Workers execute in isolated sessions, returning structured results that the coordinator incorporates into its state.

State accumulation follows specific semantics. Progress items extend across iterations with deduplication: if multiple workers report the same finding, it appears once in the progress list. Remaining items are replaced each iteration, reflecting the coordinator's current assessment of what still needs investigation. Worker outputs are appended with metadata including iteration number, worker name, and a semantic output key that enables later reference.

Worker results follow a structured output format using TOON (Token-Oriented Object Notation) [TOO25], a compact serialization format designed for LLM contexts. The TOON specification reports approximately 40% token reduction compared to JSON while maintaining explicit schema information. Rather than returning free-form text that requires parsing, workers return typed objects with explicit schemas defined in their agent specifications. TOON's explicit length markers and field headers provide guardrails that improve parsing reliability in multi-agent communication. The coordinator incorporates these results programmatically, selecting relevant fields and transforming data as needed. This structured approach reduces ambiguity

and enables what we might call a primordial form of typed agents: agents whose inputs and outputs conform to declared schemas, enabling compositional reasoning about workflow behavior. We return to this observation when discussing the connection to algebraic effects in Section 3.5.

The coordinator decides whether to continue based on two factors: whether the goal appears achieved (signaled through a `should_continue` field in its response), and whether the iteration budget is exhausted. When either condition triggers termination, the workflow enters a final iteration phase.

The final iteration differs from regular iterations in its purpose and typical configuration. Regular iterations focus on investigation and state accumulation, with permissions configured according to workflow needs. The final iteration is designated for producing outputs: it always returns a typed result conforming to the workflow's declared output schema, and may additionally produce side effects such as persistent artifacts written to the file system. When artifact creation is expected, the final iteration enables `Write` and `Edit` tools; when only structured data is needed, no additional permissions are required beyond what the schema return provides.

3.3 Composition Constraints

A central architectural decision is the **two-level composition rule**, which states that coordinators may spawn workers, but workers may not spawn further workers. This constraint bounds the depth of agent nesting to exactly one level.

$$\text{Coordinator} \rightarrow [\text{Worker}_1, \text{Worker}_2, \ldots, \text{Worker}_n] \tag{1}$$

We adopted this constraint based on reasoning about the consequences of unbounded nesting. If workers could spawn sub-workers, execution traces would become exponentially complex, resource consumption would become unpredictable, and reasoning about where results originated would become difficult. The depth limit trades some theoretical expressiveness for substantial practical benefits such as predictable resource consumption, traceable execution, and bounded complexity.

This constraint is not unique to our implementation. Claude Code [Ant25a], Anthropic's CLI tool, enforces an identical restriction: agents invoked via the Task tool cannot recursively spawn other agents through the Task tool. The convergence on this pattern across independent implementations suggests it reflects a genuine architectural insight rather than an arbitrary choice.

The constraint is enforced through three mechanisms. First, a registry tracks which sessions are workers. When the orchestrator spawns a worker, it records the worker's session identifier in a set. Second, a tool-use hook intercepts attempts to invoke the orchestrator from within a session. If the session is registered as a worker, the invocation is blocked with an error message explaining the constraint. Third, the permission system provides a safety net: even if the first two mechanisms failed, worker sessions lack the permissions required to spawn further agents.

This architecture aligns with patterns observed in production agent systems. Anthropic [Ant25c] describe a two-agent pattern for long-running tasks; our composition constraint formalizes a similar intuition as an architectural invariant rather than a convention.

3.4 Prompt Composition

Workflows are constructed through a three-layer prompt composition architecture that separates concerns and enables reuse.

Layer 1 (Behavior) specifies how to coordinate. Behavior prompts define iteration patterns, progress assessment strategies, worker selection heuristics, and termination criteria. These prompts are stored in a `behaviors/` directory and are agnostic to the specific task domain. A coordinator behavior, for example, explains how to assess whether progress is being made and how to select which workers to spawn next.

Layer 2 (Strategy) specifies what approach to take. Strategy prompts provide domain-specific guidance: what to look for, how to structure investigation, what quality criteria to apply. These prompts are stored in a `strategies/` directory. A research strategy, for example, explains how to decompose a question into sub-questions and what constitutes sufficient evidence.

Layer 3 (Runtime) specifies this particular invocation. Runtime context includes the available workers for this workflow, the goal or task description, maximum iterations, and any custom instructions. This layer is constructed dynamically at invocation time.

The final prompt is composed by concatenating these layers:

```
Behavior(coordinator.md)
  + Strategy(research.md)
  + Runtime(workers=[analyzer, locator], max_iter=5, goal="...")
  -> Final coordinator prompt
```

This separation yields several benefits. Behaviors are reusable across strategies: the same coordinator behavior works for research, reflection, and analysis workflows. Strategies are reusable across invocations: the same research strategy applies regardless of the specific question. Adding a new workflow often requires only combining existing layers with appropriate runtime context, rather than writing new prompts from scratch.

3.5 Permission Model

The permission model translates declarative effect specifications into concrete operational permissions, implementing a defense-in-depth approach to security.

Agents declare the effects they require through a structured specification in their definition files:

```
declaredEffects:
  - type: file.read
    pathPattern: "**/*"
  - type: file.create
    pathPattern: "ace/tasks/${taskDir}/**"
  - type: bash.execute
    commands: ["git status", "git log"]
```

Effect types include `file.read`, `file.create`, `file.modify`, and `bash.execute`. Path patterns support variable substitution, allowing specifications like `${taskDir}` that resolve at runtime based on invocation context.

The permission generator translates these declarations into operational permissions. We separate this into five parts with later parts overriding earlier ones (last-match-wins):

1. **Wildcard deny**: The base layer denies all operations by default. This ensures that any operation not explicitly permitted is blocked.

2. **Safe read patterns**: For agents declaring `file.read` effects, standard read operations (`cat`, `grep`, `ls`) are permitted on paths matching the declared patterns.

3. **Declared commands**: Commands explicitly listed in `bash.execute` effects are permitted. This layer enables specific operations the agent requires for its task.

4. **Secret denials**: Approximately 700 combinatorial patterns block access to credentials, API keys, and other sensitive data. These patterns override any permissions granted by earlier layers.

5. **Dangerous operation blocks**: Operations like `rm -rf`, `sudo`, and other destructive commands are unconditionally denied regardless of any other declarations.

Permissions are configured per workflow and may differ between regular and final iterations. A common pattern reserves write operations for the final iteration, when accumulated investigation results are synthesized into artifacts. However, this is a configuration choice rather than an architectural constraint and workflows may grant write access during regular iterations if their task requires it.

Now, the remainder of this section is me thinking out loud. I want to sketch a way of thinking about agent systems that has guided my design decisions, even though I do not know whether it can be made rigorous. The connections I draw to programming language theory may not survive careful scrutiny, but they have proven useful in practice (the Lie-Størmer Center infrastructure being one concrete example where this mental model shaped architectural choices that worked well). I include this sketch in the hope that readers with relevant expertise might see something worth developing, or alternatively might identify where my intuitions go astray.

3.5.1 Tools as Effects, Implementations as Handlers

The starting point is algebraic effect systems as developed in programming language research. This field has a roughly twenty-five year history that I find illuminating, though I am uncertain whether the analogy I draw actually holds up under scrutiny.

The theoretical foundations were established by Plotkin and Power in the early 2000s [PP03], who showed that computational effects (state, exceptions, I/O, nondeterminism) could be modeled as *algebraic operations* with a precise categorical semantics. Their insight was that effects are not ad-hoc language features but instances of a general mathematical structure.

The practical breakthrough came when Plotkin and Pretnar introduced *effect handlers* in 2009 [PP13]. Handlers provide a way to give meaning to effects: an operation like `read` or `throw` is abstract until a handler specifies what it does. Different handlers can interpret the same operation differently, enabling the separation of interface from implementation that I find useful for thinking about tools.

When we define a tool like `playbook_search`, we provide two things: a *description* that the agent sees ("search for patterns matching keywords"), and an *implementation* in code that executes when the tool is invoked. The description resembles an operation signature; the implementation resembles a handler. Different handlers could interpret the same operation differently: a test handler returns mock data, a production handler queries the actual playbook, a caching handler adds memoization.

A deeper connection—though I am less confident about this one—involves what Plotkin and Power [PP08] call *comodels*. In their framework, a model captures the algebraic structure of computation (the operations an agent can invoke), while a comodel captures the coalgebraic structure of the external environment (the "hardware" that responds to those operations). The file system, the shell, and external APIs would all be comodels in this view. Agent reasoning is internal computation; tool invocation is interaction with the external world. The *tensoring* of model with comodel produces actual execution.

I find this framing appealing because it separates concerns cleanly. The agent's reasoning is one thing, the external environment is another, and the handler mediates between them. But I do not know whether this is the right way to think about LLM tool-calling, or whether I am forcing a square peg into a round hole.

3.5.2 Agent Effect Declarations

Each agent declares effects it may perform:

```
# codebase-analyzer
declaredEffects:
  - file.read: "**/*"
  - playbook.search
```

```
# reflector
declaredEffects:
  - file.read: "**/*"
  - playbook.update
  - playbook.increment
```

These declarations form an effect signature for the agent, i.e., what operations it might invoke during execution.

3.5.3 Effect Composition Through Workflows

Here is where I think something interesting happens, though I cannot yet formalize it precisely. When a workflow uses multiple agents, the workflow's potential effects are derived from the agents' declarations. Intuitively:

$$\text{Effects}(\textit{workflow}) = \bigcup_{\textit{agent} \in \textit{workflow}} \text{DeclaredEffects}(\textit{agent})$$

A workflow that spawns both the analyzer and reflector above would have potential effects `{file.read, playbook.search, playbook.update, playbook.increment}`.

In programming language research, this kind of composition is handled through *row polymorphism* [Lei14]. An effect row like $\langle \textit{read} \mid \textit{write} \mid \mu \rangle$ specifies known effects while leaving the row variable μ open for extension. Functions can be polymorphic over effect rows, working with any computation that includes certain effects regardless of what other effects might be present.

I am not sure whether this is the right model for agent composition. Our agents do not have the kind of precise type signatures that would enable row-polymorphic inference. But the intuition feels similar: a workflow that spawns agents inherits their effects, and handlers can be polymorphic over which additional effects might be present.

The key insight, assuming this analogy holds, is that how you handle effects determines what the workflow actually does. Consider `file.write`:

- **Production handler**: Actually writes to disk
- **Dry-run handler**: Logs what would be written, does not write
- **Test handler**: Writes to temporary directory
- **Sandboxed handler**: Writes to isolated environment

You have the same workflow and the same agents but with different handlers and consequently completely different behavior. This is the polymorphism that algebraic effects provide, and possibly the polymorphism that our permission system provides (though I have not proven the correspondence).

3.5.4 Toward Typed Agents

If this could be made rigorous, agent types might look something like:

$$\textit{Agent}\langle E, I, O \rangle$$

where E is the set of declared effects, I is the input schema, and O is the output schema. Spawning an agent would produce a computation with those effects:

$$\textit{spawn} : \textit{Agent}\langle E, I, O \rangle \to I \to \textit{Computation}\langle E, O \rangle$$

Workflow composition would propagate effects:

```
do x ← spawn(agent_A, req_1) in -- Computation<E_A, ...>
do y ← spawn(agent_B, req_2) in -- Computation<E_B, ...>
return combine(x, y)            -- Computation<E_A ∪ E_B, ...>
```

To run the workflow, you would provide handlers for all the effects. Missing a handler would be a type error.

3.5.5 The Two-Level Rule as Effect Restriction

The Two-Level composition rule has a natural interpretation in this view. `spawn` is only available at the workflow level. Agents cannot declare `spawn` as an effect.

Without this restriction, computing an agent's effect set requires knowing the effects of every agent it might spawn, which in turn requires knowing the effects of agents those agents might spawn, and so on. If agent A can spawn agent B which can spawn agent C, then:

$$\text{Effects}(A) \supseteq \text{Effects}(B) \supseteq \text{Effects}(C) \supseteq \cdots$$

Static effect inference would need to chase this chain to a fixed point. With cycles (A spawns B spawns A), the analysis may not terminate. The Two-Level Rule sidesteps this as agents cannot spawn agents, so effect sets are flat and statically determinable.

3.5.6 What I Cannot Yet Formalize

However, I think it is important to be honest about where I do not think this analogy works.

The coordinator's role. The coordinator decides which agents to spawn at runtime. Does this affect typing? If the coordinator might spawn agent A or might not, is the effect conditional? How does dynamic selection interact with static effect typing?

Equations. Algebraic theories have equations that operations satisfy. Do our effects have equations? Is `file.read` idempotent? Does `playbook.increment` commute with itself? I do not know what equations, if any, should hold.

Continuations. In algebraic effect systems, continuations are syntactically determined. In agent execution, "what happens next" is generated by LLM reasoning. I suggested that workflows provide static continuation structure while agents are atomic from the workflow's view. But this is hand-waving; I do not have a precise semantics.

3.5.7 Why I Include This Sketch

Despite these gaps, thinking in these terms has guided concrete design decisions. The Two-Level Rule, the effect declaration schema, the separation between tool descriptions and implementations, the workflow structure. It seems to me like these designs have worked well in practice.

As a concrete example, I built the infrastructure for the Lie-Størmer Center while developing and using these patterns. The methodology structured work into phased workflows with verification gates. The effect declarations helped reason about what each agent could do. Whether the underlying theory is sound, I cannot say, but the practical results suggest the mental model captures something real about how to structure multi-agent systems.

I should note that I have not found any existing research applying algebraic effect theory to LLM tool-calling or multi-agent orchestration. The effect systems literature focuses on traditional programming languages where "computation" has precise semantics. Whether the framework extends to systems where "computation" involves probabilistic text generation is, as far as I can tell, an open question, which is either an opportunity or a warning sign.

If someone with expertise in effect systems sees a way to make this rigorous—or, equally valuable, sees why this analogy is fundamentally misguided—I would be very interested to learn what I am getting right and what I am getting wrong.

4 Theoretical Foundation

This section establishes the theoretical foundations that motivate our architectural decisions. Recent work on Recursive Language Models [ZKK25] provides a framework for understanding why the ACE approach succeeds and how to extend it for concurrent operation.

4.1 The Environment Paradigm

Traditional approaches to context adaptation embed accumulated knowledge directly in the prompt, creating a monolithic input that grows linearly with the amount of stored information. This approach faces fundamental scaling limitations where as contexts grow, they consume an increasing fraction of the model's attention capacity, and the relevance of any particular piece of information to the current task diminishes.

Recent work on recursive language models demonstrates an alternative paradigm [ZKK25]. Rather than feeding long prompts directly into the neural network, these systems treat prompts as part of an external environment with which the model can symbolically interact. The system exposes the same interface (string in, string out) while internally treating context as a variable that can be queried, filtered, and decomposed programmatically.

Principle 4.1 (Context as Environment Variable)**.** Store context externally and provide tools for selective access rather than concatenating context into the prompt. The model decides what to retrieve based on the task at hand, avoiding the cognitive overhead of processing irrelevant information.

This paradigm yields substantial benefits documented in the RLM paper. Context scale extends two orders of magnitude beyond the native context window. Inference cost remains comparable to or cheaper than base model calls, since only relevant portions of context are retrieved. Performance improves significantly on information-dense tasks where selective attention matters most.

Our understanding is that this paradigm describes how existing coding agents already operate in practice. Tools like Cursor, Claude Code, and similar assistants provide file system access, code search, and retrieval capabilities as external tools rather than embedding entire codebases in prompts. We do not see a fundamental difference between this established practice and the RLM formalization; the paper's contribution, as we interpret it, is formalizing why coding agents are able to handle large codebases effectively through a curated set of tools.

4.2 Implications for Playbook Design

Principle 4.1 has direct implications for how we structure the playbook system. Rather than injecting the entire playbook into each prompt, which would suffer from the scaling limitations described above, we expose the playbook through a query interface. The Generator determines which patterns to retrieve based on task requirements, mirroring the code-based filtering patterns observed in recursive language models.

This design also motivates the separation between persistent storage and session-local evidence trails. The playbook itself is a persistent artifact that evolves across sessions. The evidence trail, by contrast, tracks ephemeral state, which patterns have been applied in the current session, what feedback has been collected, what curation operations are pending. This separation enables concurrent sessions to operate independently while sharing the same underlying playbook, and provides the audit trail that the Curator requires during reflection.

4.3 Session Isolation and Deferred Mutation

Concurrent access to shared mutable state presents a fundamental challenge. As established in Proposition 5.12, when n sessions attempt simultaneous updates under a concurrent trace,

exactly $n-1$ updates are lost. The categorical analysis of Section 5.3 shows this is not merely an implementation defect but a structural consequence of the read-modify-write execution model.

Our solution draws on the principle of event sourcing from distributed systems. Rather than mutating state directly, sessions append operations to a log. The playbook is updated only during dedicated curation phases, when a single Curator agent processes the accumulated logs and applies changes atomically. This deferred curation mechanism ensures consistency (Proposition 5.13) without requiring coordination between concurrent sessions.

5 Design Evolution

This section traces the evolution of our playbook implementation from an initial agent-mediated design through to the current native tool architecture. We present this evolution as a case study in how practical experience and theoretical insights combine to drive architectural refinement in agentic systems.

5.1 The Previous Implementation

In our previous implementation, we employed the orchestrator's agent-spawning capability to mediate all playbook interactions. The design followed what seemed like a natural pattern where you define a specialized agent for playbook search, and spawn it whenever the system required pattern lookup.

A hook registered on the user prompt submission event intercepted each user message before processing. The hook implementation performed initial filtering to exclude trivial inputs, then injected an orchestration instruction directing the main agent to spawn a specialized playbook searcher.

The playbook searcher agent operated in an isolated context with access to four tools: grep for pattern matching, read for file content access, directory listing for structure discovery, and search for full-text queries. Upon receiving a task description and optional domain hints, the agent would enumerate available skills, search their descriptions for relevant keywords, query playbook files for pattern entries matching the specified domains, categorize discovered patterns by effectiveness thresholds, and return structured results to the calling context.

The playbook corpus comprised multiple domain-specific files covering methodology patterns, language-specific heuristics, repository organization guidelines, debugging strategies, and workflow patterns. Each pattern entry followed a structured format encoding an identifier, effectiveness counters, and descriptive content.

5.2 Limitations Observed in Practice

Extended deployment revealed several fundamental limitations in the agent-mediated approach. The most immediate was latency as spawning a dedicated agent session, executing the retrieval process, and returning results introduced observable delays that disrupted the interactive rhythm of task execution, particularly for sessions requiring multiple playbook consultations.

Attribution was technically possible but imprecise. The `/implement` command prompted the main agent to maintain an implementation log recording which patterns it consulted. The Reflector subsequently analyzed this log to determine attribution. However, the agent-mediated retrieval meant patterns were often retrieved speculatively rather than in direct response to specific problems, making it difficult to establish causal connections between pattern consultation and successful outcomes.

Although results from the playbook searcher used structured TOON output, a more fundamental problem emerged, namely relevance. The agent retrieved patterns based on its own understanding of the task, which often diverged from the main thread's actual needs. Without direct access to the full task context, the agent frequently returned patterns that were topically

related but not practically useful. This observation, while anecdotal, motivated the shift toward direct tool access where the main thread queries patterns with full context available.

Concurrent sessions presented a theoretical coordination challenge. While not encountered in practice during our deployment, the architecture provided no mechanism for detecting or resolving conflicts should multiple sessions attempt simultaneous playbook updates. This theoretical limitation motivated the analysis in the following section.

5.3 Theoretical Analysis of Concurrent Access

To be completely clear, this subsection is entirely unnecessary. The concurrent update problem could be stated in three sentences and solved with a mutex. What follows instead is a categorical treatment involving monoid actions, translation categories, and missing pushouts, the sort of elaborate machinery one deploys when a simple solution would be insufficiently satisfying. I offer no defense beyond an admission that I find it difficult to resist the siren call of abstract nonsense. Readers who value their time (and sanity) may skip ahead; those who remain do so at their own risk.

We formalize the concurrent access problem to clarify why the original architecture was fundamentally inadequate. We work in two categories, **Set** (sets and functions) for state spaces and **Mon** (monoids and homomorphisms) for update algebras.

Definition 5.1 (The Value Monoid)**.** Let $V = \mathbb{N} \times \mathbb{N} \times \Sigma^*$ be the set of pattern data (helpful count, harmful count, content). Define $V_\perp = V + \{\perp\}$ by adjoining a distinguished element representing "undefined." The **value monoid** is $M = \mathrm{End}(V_\perp)$, the monoid of endomorphisms of $V_\perp$ under composition. This is an object in **Mon**. The product structure of V induces distinguished subsets, namely **counter operations** acting on the first two components and **content operations** acting on the third.

Definition 5.2 (The Update Monoid)**.** Let I be the set of pattern identifiers. In **Mon**, we form the **restricted direct product** [nLa26e]:

$$\mathcal{U} = \bigoplus_{i \in I} M \;=\; \prod_{i \in I}{}' M$$

Both notations appear in the literature [nLa26c]: $\bigoplus$ emphasizes the "direct sum" character (coproduct-like in additive categories), while $\prod'$ emphasizes the restriction on the product. This is the submonoid of the full product $\prod_{i \in I} M$ consisting of families $(f_i)_{i \in I}$ where $f_i = \mathrm{id}_{V_\perp}$ for all but finitely many i. Explicitly, an element $u \in \mathcal{U}$ is a pair (S_u, φ_u) where:

- $S_u \subseteq I$ is finite (the **support**)
- $\varphi_u : S_u \to M$ assigns to each $i \in S_u$ an endomorphism

Composition in $\mathcal{U}$ is pointwise: $(v \circ u)$ has support $S_u \cup S_v$, with

$$(v \circ u)_i = \begin{cases} v_i \circ u_i & \text{if } i \in S_u \cap S_v \\ u_i & \text{if } i \in S_u \setminus S_v \\ v_i & \text{if } i \in S_v \setminus S_u \\ \mathrm{id} & \text{otherwise} \end{cases}$$

The identity is $(\emptyset, !)$. Think of updates as **patches**: they specify only what changes, with implicit identity elsewhere.

Definition 5.3 (The State Space)**.** The **state space** is the **restricted product** (or weak direct product):

$$\mathcal{P} = \prod_{i \in I}{}' V_\perp = \{P : I \to V_\perp \mid P_i = \perp \text{ for all but finitely many } i\}$$

A state $P \in \mathcal{P}$ assigns values to finitely many identifiers and $\perp$ elsewhere. We write $\mathrm{dom}(P) = \{i \in I : P_i \neq \perp\}$ for the finite set of defined patterns.

This mirrors the finite support constraint on updates: both states and updates are "sparse" structures. A playbook with infinitely many defined patterns would require infinite memory; the restricted product captures exactly the computationally realizable states.

Definition 5.4 (The Action)**.** The update monoid $\mathcal{U}$ **acts** on the state space $\mathcal{P}$ via a monoid homomorphism [nLa26a] $\alpha : \mathcal{U} \to \mathrm{End}_{\mathbf{Set}}(\mathcal{P})$ defined by:

$$\alpha(u)(P)_i = \begin{cases} \varphi_u(i)(P_i) & \text{if } i \in S_u \\ P_i & \text{otherwise} \end{cases}$$

This is a left action: $\alpha(v \circ u) = \alpha(v) \circ \alpha(u)$ and $\alpha(\mathrm{id}_{\mathcal{U}}) = \mathrm{id}_{\mathcal{P}}$. We write $u(P)$ for $\alpha(u)(P)$ when the action is clear.

The action is well-defined on the restricted products: if P has finite support $\mathrm{dom}(P)$ and u has finite support S_u, then $u(P)$ has support contained in $\mathrm{dom}(P) \cup S_u$, which is finite.

Definition 5.5 (The State Category)**.** The action induces a category $\mathcal{S}$, the **action category** [nLa26b]:

$$\begin{aligned} \text{Objects:} \quad & P \in \mathcal{P} \\ \text{Morphisms:} \quad & \mathrm{Hom}_{\mathcal{S}}(P, Q) = \{u \in \mathcal{U} : u(P) = Q\} \\ \text{Composition:} \quad & \text{inherited from } \mathcal{U} \\ \text{Identity:} \quad & \mathrm{id}_P = \mathrm{id}_{\mathcal{U}} \end{aligned}$$

A morphism $u : P \to Q$ witnesses that u transforms P into Q. Sequential execution corresponds to composition of morphisms.

$$P \xrightarrow{u} Q \xrightarrow{v} R \quad \Leftrightarrow \quad v \circ u : P \to R$$

Definition 5.6 (Commuting Updates)**.** Updates $u, v \in \mathcal{U}$ **commute** if $u \circ v = v \circ u$ in $\mathcal{U}$. Equivalently, for any state P, the following square commutes in $\mathcal{S}$:

$$\begin{array}{ccc} P & \xrightarrow{\quad u \quad} & u(P) \\ {\scriptstyle v}\downarrow & & \downarrow{\scriptstyle v} \\ v(P) & \xrightarrow[u]{} & (v \circ u)(P) = (u \circ v)(P) \end{array}$$

By the direct sum structure of $\mathcal{U}$, commutativity reduces to the overlap: u and v commute iff $u_i \circ v_i = v_i \circ u_i$ in M for all $i \in S_u \cap S_v$. Disjoint support ($S_u \cap S_v = \emptyset$) implies commutativity trivially.

Remark 5.7 (Commutativity in the Value Monoid). Not all elements of $M = \mathrm{End}(V_\perp)$ commute. The product structure $V = \mathbb{N} \times \mathbb{N} \times \Sigma^*$ induces a decomposition.

- Counter operations $\mathrm{inc}_h, \mathrm{inc}_r$ (incrementing the first and second components respectively) generate an **abelian** submonoid; they commute with each other and with content operations.

- Content operations set_s (mapping (h, r, c) to (h, r, s)) satisfy the **left-zero law**, $\text{set}_s \circ \text{set}_t = \text{set}_s$. They do not commute with each other unless $s = t$.

Consequently, two updates $u, v \in \mathcal{U}$ with $S_u \cap S_v = \{i\}$ may or may not commute depending on u_i and v_i. Counter-counter and counter-content pairs commute; content-content pairs do not. This parallels the CRDT observation that counter increments commute intrinsically, whereas register writes require auxiliary ordering (e.g., timestamps in LWW-Register) to achieve deterministic conflict resolution [Sha+11]. Our serialization through the Curator serves the same function, imposing a total order on inherently non-commutative operations.

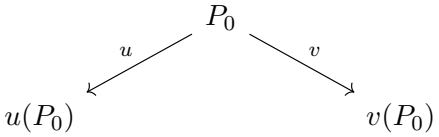
Definition 5.8 (Execution Model)**.** Let $(T, <)$ be a totally ordered set (the **time domain**). An **execution trace** for sessions $\{1, \ldots, n\}$ on initial state P_0 is a tuple $(\tau_1^r, \tau_1^w, \ldots, \tau_n^r, \tau_n^w) \in T^{2n}$ where $\tau_s^r < \tau_s^w$ for each session s (each session reads before it writes).

Session s with update $u_s \in \mathcal{U}$ operates as follows:

- At time τ_s^r: read the current state, call it $P^{(s)}$
- Compute $u_s(P^{(s)})$ (offline, instantaneous)
- At time τ_s^w: atomically replace the state with $u_s(P^{(s)})$

An execution is **concurrent** if $\tau_s^r < \tau_{s'}^w$ for all $s \neq s'$: every read completes before any write begins. Under this assumption, all sessions read the same initial state: $P^{(s)} = P_0$ for all s.

Proposition 5.9 (Lost Update in Concurrent Writes)**.** *Let $u, v \in \mathcal{U}$ be updates executed by sessions $1, 2$ with a concurrent trace $(\tau_1^r, \tau_1^w, \tau_2^r, \tau_2^w)$ from initial state P_0. The execution yields a **span** in $\mathcal{S}$ rather than a composable path:*



Atomic file replacement selects one leg: the final state is $u(P_0)$ if $\tau_1^w > \tau_2^w$, or $v(P_0)$ if $\tau_2^w > \tau_1^w$, not the intended composition $(v \circ u)(P_0)$.

Proof. By the concurrent trace assumption, $\tau_1^r < \tau_2^w$ and $\tau_2^r < \tau_1^w$. Both reads occur before either write, so both sessions observe P_0: session 1 computes $u(P_0)$, session 2 computes $v(P_0)$. Sequential execution would form a composable path $P_0 \xrightarrow{u} u(P_0) \xrightarrow{v} (v \circ u)(P_0)$, but concurrent execution produces the span instead. Atomic replacement implements last-writer-wins: the final state is $u(P_0)$ or $v(P_0)$ depending on which write occurs later, with the earlier write's contribution discarded. □

Remark 5.10 (Categorical Interpretation). The span admits no canonical completion to a commutative square within the execution model. Whether a merge exists depends on what the updates do:

- If u and v commute in $\mathcal{U}$ (e.g., both are counter operations, or have disjoint support), a pushout exists but the execution model cannot compute it.
- If u and v do not commute (e.g., both modify the same content field), no pushout exists. Any state Q can complete the diagram via constant maps, but states with different content are not isomorphic in $\mathcal{S}$. Pushouts are unique up to isomorphism; infinitely many non-isomorphic completions means none is universal.

In the second case, information loss is not an operational accident but an algebraic necessity: $\mathcal{S}$ provides no canonical way to select among "Use JWT tokens," "Use session cookies," or any other content.

Remark 5.11 (The Missing Pushout). When supports are disjoint ($S_u \cap S_v = \emptyset$), a canonical merge exists in $\mathcal{U}$: define $u \sqcup v = (S_u \cup S_v, \psi)$ where ψ restricts to φ_u on S_u and to φ_v on S_v. This yields a pushout square [nLa26d] in $\mathcal{S}$:

$$\begin{array}{ccc} P_0 & \xrightarrow{u} & u(P_0) \\ {\scriptstyle v}\downarrow & \ulcorner & \downarrow{\scriptstyle v} \\ v(P_0) & \xrightarrow[u]{} & (u \sqcup v)(P_0) \end{array}$$

The pushout $(u \sqcup v)(P_0)$ incorporates both updates: it equals $u(P_0)$ on S_u and $v(P_0)$ on S_v, with these being consistent since the supports are disjoint.

More subtly, a pushout also exists when supports overlap but the component operations commute. If u and v both increment counters on pattern i, then $u_i \circ v_i = v_i \circ u_i$, and the merge $(u \sqcup v)_i = u_i \circ v_i$ is well-defined. The difficulty arises when component operations do not commute. If $u_i = \text{set}_{c_1}$ and $v_i = \text{set}_{c_2}$ with $c_1 \neq c_2$, no algebraic combination yields a meaningful result. One content string must be discarded.

This distinction matters. Counter conflicts are **recoverable** (the merge exists, we merely lack the mechanism to compute it), while content conflicts are **irrecoverable** (no merge exists regardless of mechanism). The deferred curation architecture handles both uniformly by serialization, but the outcomes differ. For counters, serialization computes the correct merge; all increments are preserved. For content, serialization merely imposes a deterministic order on an inherently lossy operation. The last content update in the Curator's queue overwrites all predecessors, just as it would under concurrent writes. Serialization provides predictability, not preservation. [1]

However, the read-modify-write execution model provides no mechanism to compute this pushout. Each session operates in isolation, unaware of concurrent modifications. Even when the categorical limit exists, the operational semantics cannot construct it. The deferred curation architecture of Section 6 resolves this by sequentializing updates through a single Curator, converting the problematic span into a well-defined composition in $\mathcal{S}$.

This analysis establishes that the original architecture, permitting concurrent sessions to read and write playbook files directly, could not guarantee preservation of all updates. The execution model produces spans where sequential composition is required.

The two-session case generalizes immediately.

Proposition 5.12 (Race Condition in Direct Updates)**.** *Let $u_1, \ldots, u_n \in \mathcal{U}$ be updates executed by n sessions with a concurrent trace from initial state $P_0 \in \mathcal{P}$. The final state equals $u_k(P_0)$ for some $k \in \{1, \ldots, n\}$. Exactly one session's update persists; the remaining $n - 1$ are lost.*

Proof. Follows by induction on n with the base case $n = 2$ being Proposition 5.9. □

5.4 Orchestrator Refinement

The playbook architecture evolved alongside refinements to the orchestrator itself. The orchestrator always had formal contracts, workflow primitives, and structured composition. What changed over time was the precision of these mechanisms and the introduction of typed effect declarations.

[1] In practice, content conflicts are rare. Most playbook updates either create new patterns (mapping $\perp$ to $(0, 0, c)$ for some content c) or increment counters on existing patterns. Both cases commute. Concurrent content updates to the same pattern require two sessions to independently decide that an existing description needs revision, an uncommon scenario. We have thus deployed restricted direct products, monoid actions, and categorical pushouts to analyze a conflict that almost never occurs. The reader may draw their own conclusions about the author's priorities.

The iteration protocol evolved from an implicit status-based termination model to an explicit continuation signal. In the earlier approach, the coordinator returned a status field (success, partial, or failure) and accumulated findings and gaps across iterations. Termination occurred when the status indicated success. This worked but required the coordinator to encode termination intent within a general-purpose status field, making the protocol somewhat indirect.

The current approach separates concerns. Each iteration returns explicit signals: `progress_items` (what was accomplished), `remaining_items` (what remains), and `should_continue` (whether to proceed). A `CompletionAssessment` structure forces the coordinator to evaluate completion criteria explicitly before signaling termination. If the assessment contradicts the continuation signal, the system prompts for clarification with up to three retries before raising an error. This strictness catches cases where the coordinator might otherwise terminate prematurely due to ambiguous status encoding.

The introduction of typed effect declarations represented a more substantial change. Previously, agents declared which tools they could use through a simple boolean map: `tools: {read: true, grep: true, ...}`. This controlled tool availability but said nothing about how those tools would be used.

The `declaredEffects` system, which we introduced in late 2025, adds structure. Each effect declaration specifies a typed family and action (such as `file.read` or `external.shell_command`), path patterns with variable substitution, and optional command patterns for bash operations. The permission generator transforms these declarations into concrete permission rules through five parts. Dangerous operations are blocked unconditionally, secret patterns denied via combinatorial matching, declared commands permitted, safe read patterns enabled for file.read effects, and a default deny for everything else.

This connects to the algebraic effects sketch in Section 3.5. The `declaredEffects` field is the agent's effect signature. The permission generator is one possible handler for those effects. Different handlers could interpret the same declarations differently, e.g., a sandboxed handler might redirect file writes to a temporary directory, a dry-run handler might log operations without executing them, a test handler might return mock data. The declarations describe what the agent wants to do; the handler determines what actually happens.

The system currently enforces strict mode in the sense that agents must have `declaredEffects` defined, and agents without declarations fail at spawn time. This strictness ensures that effect composition through workflows, as sketched earlier, has the necessary type information to reason about what effects a workflow might perform.

Prompt composition also evolved, though this change was more about organization than capability. The original approach used a single coordinator prompt file. The current approach separates behavior (how to coordinate), strategy (what approach to use), and runtime context (this specific invocation) into composable layers. New strategies can be added by dropping files into a directory, and the same behavior layer works across different strategic approaches. This follows patterns familiar from shell module composition in Nix, where configuration layers combine to produce final system state.

These refinements share a common direction where we make implicit structure explicit. Status-based termination became explicit continuation signals. Tool availability became typed effect declarations. Monolithic prompts became composable layers. Each change increased the precision with which the system could reason about what agents do and how workflows compose. Whether this precision enables the formal treatment sketched earlier, I cannot yet say, but the practical benefits in debugging and extension have been clear.

5.5 Insights from Recursive Language Models

The limitations observed in practice aligned with theoretical predictions from concurrent research on recursive language models. Zhang, Kraska, and Khattab [ZKK25] demonstrate that systems treating context as an external environment variable rather than a prompt component

achieve substantial benefits: context scale extends beyond native limits, inference costs remain bounded, and selective attention improves performance on information-dense tasks.

This paradigm clarifies a key design choice where playbook patterns should be accessible via tools rather than injected directly into the system prompt. A common approach embeds accumulated patterns in configuration files like `CLAUDE.md` or `AGENTS.md`, forcing the model to process all patterns on every invocation regardless of relevance. The environment paradigm suggests instead that patterns remain in external storage, queryable on demand through tool interfaces. The agent-mediated approach we originally used was not inconsistent with this paradigm (agents also interact with the environment), but it introduced unnecessary indirection when direct tool access suffices.

5.6 The Current Implementation

The current implementation eliminates the agent-mediated approach entirely. Playbook operations are native tools registered directly with the orchestrator, substantially reducing query latency. A consolidated playbook file replaces the distributed corpus, simplifying both parsing and caching.

Session-based evidence tracking provides a principled foundation for curation. Each session maintains its own storage partition where attribution entries, feedback reports, and curation queues accumulate without coordination. The playbook itself remains immutable during normal operation; mutations occur only during dedicated reflection phases when a single Curator agent processes accumulated evidence atomically.

Proposition 5.13 (Consistency under Deferred Curation)**.** *(For those of you who actually read Section 5.3: welcome back!)*

Let $u_1, \ldots, u_n \in \mathcal{U}$ be updates generated by n sessions, each appending to a session-local queue Q_i (no shared writes). If a single Curator reads all queues and applies updates sequentially from initial state $P_0 \in \mathcal{P}$, the final state is the composition:

$$(u_n \circ \cdots \circ u_2 \circ u_1)(P_0)$$

All updates are preserved; no span is formed.

Proof. During generation, session i appends u_i to Q_i without reading or modifying the playbook state. Queues are disjoint, so no write conflicts arise. The Curator then operates as a single sequential process:

$$P_0 \xrightarrow{u_1} P_1 \xrightarrow{u_2} \cdots \xrightarrow{u_n} P_n$$

where $P_k = u_k(P_{k-1})$. This is a composable path in the state category $\mathcal{S}$, not a span. Each update observes the result of all preceding updates; the final state $P_n = (u_n \circ \cdots \circ u_1)(P_0)$ reflects the full composition. □

Remark 5.14 (What This Does and Does Not Solve). Deferred curation converts spans into paths, ensuring deterministic outcomes. However, it does not resolve the content string problem from Remark 5.10. If sessions 1 and 2 both queue content updates to the same pattern—say, "Use JWT tokens" and "Use session cookies"—the Curator applies them sequentially, and the later update overwrites the earlier. The choice is now deterministic (queue ordering) rather than a race, but one session's content contribution is still discarded.

Counter operations are different. Because `increment_helpful` and `increment_harmful` commute in $\mathcal{U}$, sequential application in any order produces the same final counts. If session 1 queues inc_h and session 2 queues inc_h, the Curator's path yields +2 regardless of ordering. No information is lost. The architecture solves concurrency for counters while merely determinizing the inherent conflict for content.

The hook that previously triggered agent spawning was repurposed to inject periodic reminders about available playbook tools, promoting pattern consultation without requiring intermediate agent sessions. This guidance mechanism preserves the proactive knowledge retrieval that motivated the original design while eliminating its architectural limitations.

6 Architecture

The implementation comprises three architectural layers: the playbook store, the session evidence trail, and the tool interface. Each layer addresses a distinct concern while maintaining clean separation of responsibilities.

6.1 Playbook Store

The playbook store manages the consolidated playbook file and its derived artifacts. The playbook itself resides at a well-known location in the repository and follows a structured format that enables efficient parsing and querying.

Each pattern entry consists of three components. An identifier of the form `[domain-NNNNNN]` provides a stable reference, where the domain indicates a category (such as methodology, language-specific patterns, or anti-patterns) and the number is a six-digit identifier that enables deterministic assignment. Effectiveness counters record how often the pattern was marked as beneficial or detrimental across prior sessions. The content describes the strategy, heuristic, or warning.

```
[api-000023] helpful=31 harmful=2 :: Use UserService.findById() not raw
    SQL
[auth-000008] helpful=17 harmful=0 :: JWT tokens require refresh before /
    admin routes
```

An embedding cache stores vector representations of pattern content, enabling semantic similarity queries for deduplication and related pattern discovery. The cache uses a local embedding model (we use Ollama with `nomic-embed-text`) to avoid external API dependencies. Embeddings are stored in `.playbook-cache/embeddings.json`, mapping pattern identifiers to their vector representations.

The deduplication system uses similarity thresholds to guide curation decisions. Patterns with similarity below 0.85 are considered sufficiently distinct for independent addition. Patterns with similarity between 0.85 and 0.95 are flagged for potential merging, as they likely represent related but not identical guidance. Patterns with similarity above 0.95 are considered near-duplicates, and the system recommends skipping the addition to avoid redundancy.

The cache is derived from the playbook and can be regenerated if corrupted or invalidated by model changes. Regeneration is triggered automatically when the playbook's modification timestamp exceeds the cache timestamp, ensuring consistency without manual intervention.

6.2 Session Evidence Trail

The session evidence trail provides isolated storage for each active session (Figure 4). This isolation is crucial for enabling concurrent operation: sessions can track pattern usage, collect feedback, and queue curation operations without coordinating with other sessions or risking data loss from race conditions.

Each session maintains three categories of evidence in an isolated directory structure:

```
.sessions/${sessionId}/
|-- session.json
`-- memory/
```

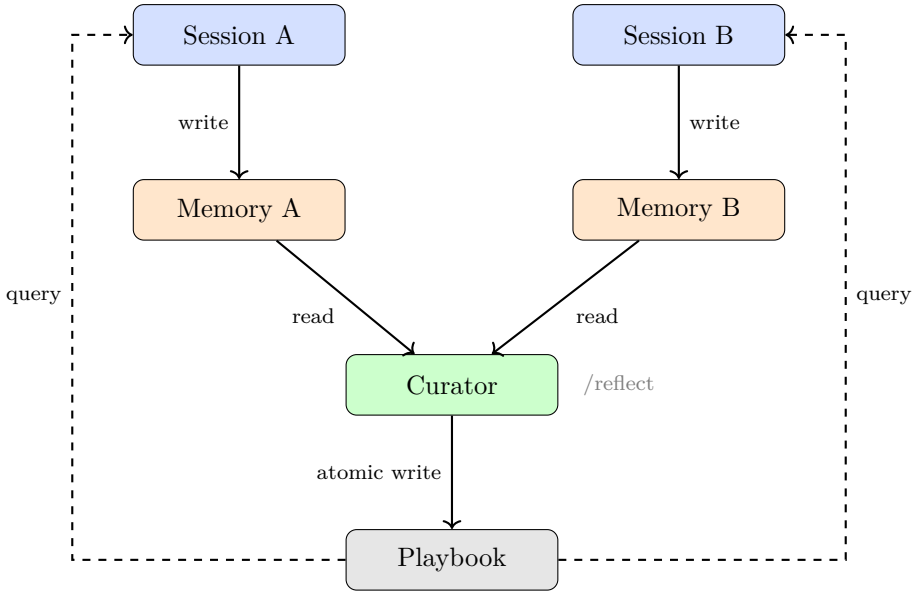


Figure 4: Session isolation architecture. Each session writes to its own memory partition. The Curator reads from all relevant sessions during reflection and performs a single atomic write to the playbook.

```
    |-- attribution.json
    |-- feedback.json
    `-- curation-queue.json
```

The `session.json` file stores metadata including the session start time and associated task context. The `memory/` subdirectory contains the evidence trail files.

Attribution records track which patterns the Generator consulted during task execution. Each entry includes the pattern identifier, a timestamp, and the task context at the time of consultation. This enables the Reflector to analyze which patterns contributed to outcomes:

```
{ "patternId": "ace-000017", "timestamp": "...", "taskContext": "..." }
```

Feedback entries capture explicit assessments of pattern effectiveness. Each entry records whether the pattern was helpful or harmful, with optional evidence describing the observed outcome:

```
{ "patternId": "ace-000017", "helpful": true, "evidence": "...", "
   timestamp": "..." }
```

Curation queue entries accumulate proposed updates for later application. Each entry specifies an operation (increment, add, or merge), parameters, and the originating session:

```
{ "operation": "increment", "params": { "id": "ace-000017", "helpful": 1
   },
  "timestamp": "...", "sessionId": "..." }
```

The session architecture directly addresses Proposition 5.12. Since sessions write only to their own storage partitions, concurrent operation cannot cause lost updates. The playbook remains immutable during normal operation; mutations occur only during the reflection workflow, when the Curator processes accumulated queues from relevant sessions.

6.3 Tool Interface

The tool interface exposes playbook functionality to the Generator through thirteen operations organized into five categories. This design implements Principle 4.1: rather than embedding the full playbook in prompts, the Generator queries for relevant patterns as needed.

Query tools provide read-only access to playbook content:

- `playbook_search`: Returns patterns matching keywords, sorted by effectiveness score. Accepts domain filters and result limits.
- `playbook_get`: Fetches specific patterns by identifier, returning full content including effectiveness counters.
- `playbook_list`: Enumerates all patterns in a specified domain, useful for browsing available guidance in a category.
- `playbook_proven`: Returns patterns with helpful count above a configurable threshold, surfacing well-validated guidance.
- `playbook_warnings`: Surfaces anti-patterns where harmful feedback exceeds helpful, alerting the Generator to known pitfalls.

Attribution tools enable the Generator to record pattern usage:

- `playbook_applied`: Records that a pattern was consulted during task execution, creating an attribution entry in session memory.
- `playbook_set_task`: Links the current session to a task directory, establishing correlation context for later analysis by the Reflector.

Feedback tools capture effectiveness assessments:

- `playbook_feedback`: Records whether a pattern was helpful or harmful for the current task, with optional evidence describing the outcome.

Curation tools queue updates rather than applying them directly:

- `playbook_increment`: Queues an increment to a pattern's helpful or harmful counter based on observed effectiveness.
- `playbook_add`: Queues a new pattern for addition, specifying domain, identifier, and content.
- `playbook_merge`: Queues a recommendation to merge two similar patterns, consolidating redundant guidance.
- `playbook_apply_curation`: Processes the merged curation queue and applies changes atomically to the playbook. This tool is restricted to the Curator agent during reflection workflows.

Deduplication tools prevent redundant pattern accumulation:

- `playbook_dedupe`: Queries the embedding cache for semantically similar patterns before adding new content. Returns similarity scores and recommendations: add if sufficiently distinct, merge if similar, or skip if near-duplicate.

This tool organization reflects the ACE architecture's separation of concerns. Query and attribution tools are available to all agents, enabling pattern consultation and usage tracking during normal operation. Feedback tools capture the evidence trail that informs later curation decisions. Curation tools are primarily used during reflection, with `playbook_apply_curation` restricted to ensure atomic updates.

6.4 Agent Definition Schema

Agents in the orchestrator are defined through markdown files with YAML frontmatter, combining declarative configuration with natural language system prompts. This format enables version-controlled agent definitions that can evolve alongside the codebase.

```
---
name: codebase-analyzer
description: Analyzes codebase implementation details
model: anthropic/claude-sonnet-4-5
tools:
  read: true
  grep: true
  glob: true
  ls: true
declaredEffects:
  - id: read-files
    effectType:
      family: file
      action: read
    pathPattern: "**/*"
    description: "Read any file for analysis"
contract:
  input:
    schema:
      type: object
      properties:
        files_to_analyze: { type: string }
        analysis_focus: { type: string }
      required: [files_to_analyze, analysis_focus]
  output:
    schema:
      type: object
      properties:
        status: { type: string, enum: [success, failure] }
        summary: { type: string, x-extract: fact }
        analysis: { type: array, x-extract: description }
      required: [status, summary]
---

You are a codebase analyzer...
```

The frontmatter specifies several components:

Basic metadata: The `name` field provides a unique identifier for workflow configurations. The `description` provides human-readable context that coordinators use when selecting workers. The `model` specifies a default model path, overridable at invocation time.

Tool permissions: The `tools` field explicitly declares which tools the agent may use. This provides a first layer of capability restriction before effect-based permission generation.

Effect declarations: The `declaredEffects` field uses a typed effect system with discriminated unions. Each effect specifies:

- `id`: A unique identifier for the effect declaration
- `effectType`: A family/action pair from defined families:
  - `file`: actions `read`, `write`, `create`, `modify`, `delete`
  - `external`: actions `shell_command`, `api_request`, `web_fetch`
  - `playbook`: actions `update`, `increment`, `add_pattern`
  - `custom`: escape hatch for domain-specific effects

- `pathPattern` or `commandPatterns`: Glob patterns or command specifications
- `description`: Human-readable explanation
- `required`: Optional flag marking essential effects

Contract specification: The `contract` field defines both input and output schemas using JSON Schema. The input schema specifies expected parameters; the output schema specifies the structure of results.

Custom `x-` extension fields enable schema annotations beyond standard JSON Schema. For example, `x-extract: fact` marks fields for automatic extraction into coordinator summaries, while `x-extract: description` identifies fields containing detailed findings. These extensions enable the orchestrator to intelligently process worker outputs without parsing natural language.

The schema serves as a contract between coordinator and worker where workers return structured data conforming to the declared schema, serialized using TOON for efficient transmission (Section 3.2). Type mismatches indicate implementation errors rather than ambiguous communication.

This schema-based approach represents a step toward what might be called typed agents. Both inputs and outputs are typed through JSON Schema. Effects are typed through the discriminated union in `effectType`. Combined with the algebraic effects mental model described in Section 3.5, these specifications provide the raw material for reasoning about agent composition—though whether this can be developed into a rigorous theory of algebraically typed agents remains an open question that I find intriguing but have not pursued as of yet.

7 The Learning Loop

The complete learning loop implements the ACE vision of self-improving systems through structured feedback (Figure 5). Five stages form a cycle that continuously refines the playbook based on observed outcomes.

Consider the evolution of a pattern through this cycle. A pattern begins with initial effectiveness counters based on prior experience or expert assessment. During generation, the agent consults this pattern and records an attribution entry. Upon task completion, feedback indicates whether the pattern contributed to success. During reflection, the Reflector analyzes attribution and feedback across sessions correlated with the task. The Curator processes the resulting curation queue, updating counters to reflect accumulated evidence.

This evidence-based evolution enables patterns that prove useful to rise in prominence while those that mislead are deprioritized. Patterns whose harmful count exceeds their helpful count are surfaced as anti-patterns, warning future sessions against their application.

7.1 Hook-Based Promotion

Two hooks promote playbook usage without requiring explicit invocation. The task detection hook monitors file access patterns and prompts the Generator to establish task correlation when working within task directories. This ensures attribution is properly linked even when the Generator does not explicitly invoke task-setting tools.

The guidance hook injects periodic reminders about available playbook tools after a configurable number of tool invocations. This promotes pattern consultation in sessions that might otherwise proceed without querying the playbook. The hook suppresses reminders when the Generator is already actively using playbook tools, avoiding redundant prompting.

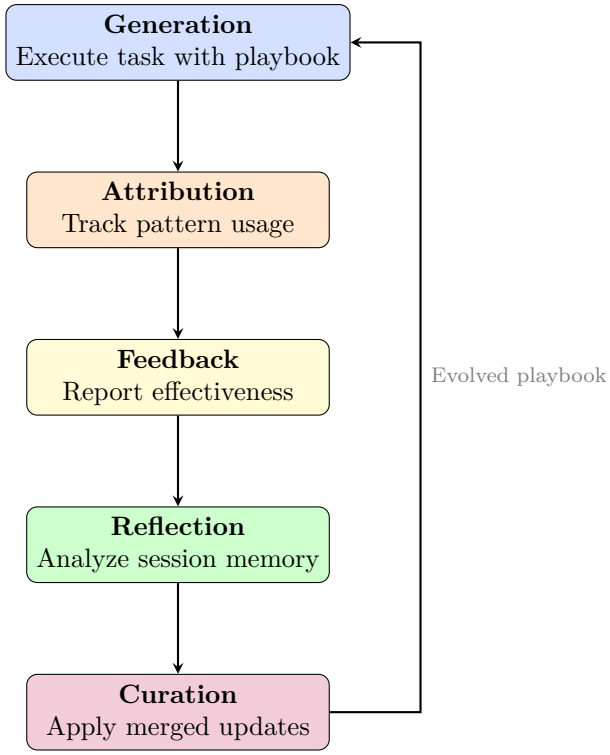


Figure 5: The ACE learning loop. Generation uses patterns; attribution tracks usage; feedback reports outcomes; reflection analyzes effectiveness; curation updates the playbook.

7.2 Orchestrator Integration

The reflection workflow integrates with the orchestrator (Section 3) to coordinate the Reflector and Curator agents. The workflow begins by discovering sessions correlated with the target task through metadata queries. Attribution and feedback records from discovered sessions are merged to provide the Reflector with a comprehensive view of pattern effectiveness.

The Reflector analyzes merged data to identify successful strategies and recurring failures. Its output includes proposed curation operations such as counter increments based on feedback, new patterns distilled from successful trajectories, and merge recommendations for semantically similar entries. The Curator validates these proposals against the deduplication system and applies the merged queue atomically.

7.3 Extension Mechanisms and Context Economy

Coding agents tend to exposes three extension points beyond agents, and I have found it useful to think carefully about when to use each. The fundamental principle is providing the right context at the right time, Anthropic [Ant25b], and each mechanism addresses this differently.

Hooks steer the coding agent by injecting context at composition boundaries: before or after tool invocations, at session start, when prompts are submitted. The task detection hook mentioned above is typical: it monitors file access patterns and triggers attribution setup without requiring the agent to remember to do so. Hooks operate outside the main conversation, injecting guidance without consuming tokens in the primary context window. They are the mechanism for automatic steering: ensuring certain behaviors happen reliably without depending on the agent to initiate them. OpenCode does not expose hooks as a first-class concept, but its plugin architecture allows implementing equivalent functionality, and more, since plugins can intercept and transform at additional points in the execution lifecycle.

Commands define predefined workflows. When a user types `/research`, `/plan`, `/implement`,

or `/reflect`, they invoke a command that coordinates the corresponding workflow. Commands encode the phased methodology I have described throughout this report; each command represents a well-defined entry point into a particular kind of work. The distinction from hooks is that commands require explicit user invocation; they are deliberate actions rather than automatic steering.

Skills package domain knowledge, e.g., expertise about programming languages, testing methodologies, design patterns, or project conventions. Skills differ from playbook patterns in that they represent relatively stable expertise rather than evolving heuristics learned from task outcomes. A practical problem I encountered is that skills do not always load reliably; the agent may not consult relevant skills unless prompted. Commands solve this by routing to skills explicitly: the `/research` command loads the research skill, `/implement` loads the implementation skill, and so on. This combination ensures that domain knowledge is available when the corresponding workflow begins.

These distinctions connect back to the context economy discussion in Section 2.3. The main conversation thread is expensive context that should be protected from pollution. Hooks inject focused steering without polluting that context (this also requires hooks to be almost surgical in their definition). Commands, in our implementation, often delegate to orchestrated sessions, returning only what matters. Skills get loaded when relevant workflows begin. I am not sure this decomposition is optimal—there may be better ways to carve up the space—but it has worked reasonably well so far.

8 The Workflow in Practice

The preceding sections describe architecture and theory. This section provides a concrete example of how I utilize the systems I have built on a day-to-day basis, tracing the flow from task initiation through playbook evolution. I want to be concrete about the directory structures, the artifacts produced, and especially about where human judgment enters the process.

8.1 Phase Structure and Human Oversight

My daily work proceeds through tasks and each task consists principally of four distinct phases: research, planning, implementation, and reflection. Over time I have noticed that these phases differ not only in purpose but in their relationship to human oversight, and this distinction has shaped how I think about automation boundaries.

Research and reflection are what I would call **post-hoc reviewable**. The orchestrator coordinates multiple agents working in parallel, accumulating results across iterations. I review the outputs after completion. If the research missed something important, I can edit the observations, run additional investigation, or discard the results entirely before moving to planning. Reflection operates differently: the workflow runs the Reflector and Curator in sequence, applying pattern updates and producing a summary of exactly which changes were made. I review this summary afterward and can revert unwanted changes through version control or manual editing. The key distinction is that research produces drafts I approve before the next phase, while reflection produces applied changes I can audit and undo.

On the other hand, planning and implementation are different in character. Planning is interactive: the agent works from research findings but asks me clarifying questions about design choices, architectural direction, and priorities based on what the research revealed. This back-and-forth dialogue—what the HumanLayer project might call *human-in-the-loop*—shapes the plan before it solidifies. Planning decisions compound; an early architectural choice shapes all subsequent phases, and I want to influence those choices through dialogue rather than merely accept or reject a finished artifact. Implementation, by contrast, has immediate effects as files are written, tests run, and commits are created. I watch the execution and intervene when

verification fails or the agent does something that is obviously wrong, but the key human judgment already occurred during planning. The more comprehensive the plan the less need there is to be vigilant in implementing as the plan should already capture every minute and important detail.

This distinction then shapes the execution model. For example, research spawns agents via the orchestrator's coordinator loop (Section 3), enabling parallel exploration where a codebase analyzer, web researcher, and pattern finder might work simultaneously. The coordinator aggregates their findings into working memory, producing comprehensive observations that no single agent could assemble in isolation (and would be too much context for one agent to do on its own). Reflection similarly benefits from the coordinator's ability to process evidence trails, git changes, and pattern attributions in parallel before synthesizing conclusions.

Planning and implementation use the main conversation thread. I read the research observations, and work with the agent through dialogue: it proposes approaches, it asks clarifying questions, and we iterate until the plan reflects my intent. The agent might ask "should we prioritize backwards compatibility or clean breaks?" and my answer shapes all subsequent phases. During implementation, I approve each phase before it begins, confirm completions, and intervene when verification fails.

8.2 Directory Structure and State Flow

Two directory hierarchies support the above workflow in the following way: the task directory containing human artifacts, and the session directory containing machine state. I found this separation important for understanding how work persists across sessions and how the learning loop operates, so I will describe both in detail.

8.2.1 Task Directory

Each task lives in a directory under `ace/tasks/`, named by date and a descriptive slug. The directory accumulates artifacts as work progresses:

```
ace/tasks/2025-11-08-tailwind-v4-lisp-migration/
|-- research.md
|-- observations-research-2025-11-08T14-30-00.md
|-- observations-planning.md
|-- plan.md
|-- observations-implementation.md
|-- observations-reflect-2025-11-12T16-00-00.md
|-- handoffs/
|-- screenshots/
`-- runs/
    |-- 001-research-20251108T143000-a1b2/
    |   |-- logs/
    |   |-- state/
    |   |-- telemetry/
    |   `-- worker-outputs/
    `-- 002-reflect-20251112T160000-c3d4/
        |-- logs/
        |-- state/
        |-- telemetry/
        `-- worker-outputs/
```

The human-facing artifacts at the task root are:

- `research.md`: Synthesized research document produced by the coordinator.
- `observations-research-*.md`: Timestamped log of the research workflow execution.

- `observations-planning.md`: Planning decisions and identified gaps.
- `plan.md`: Implementation plan with phased structure and verification criteria.
- `observations-implementation.md`: Log of implementation progress.
- `observations-reflect-*.md`: Timestamped log of the reflection workflow.
- `handoffs/`: Session transfer documents for resuming work.
- `screenshots/`: Visual verification artifacts.
- `runs/`: Orchestrator execution state (machine-managed).

Research produces two distinct artifacts, and this design choice took some time to materialize. The `research-{topic}.md` file is the synthesized document, i.e., the coordinator's final output presenting findings in a structured, readable format suitable for consumption by the planning phase. The `observations-research-*.md` file is the execution log: a timestamped record of what happened during the workflow, showing each iteration's progress, which workers succeeded or failed, and how the coordinator reasoned about what to investigate next.

I find the observation log valuable for understanding how conclusions were reached. When research produces surprising results, I can trace back through the iterations to see what evidence led to those conclusions. It is especially useful for tracing through how incorrect information poisons later phases and thinking about how we can mitigate it from happening. An abridged sample from a real research workflow illustrates the format:

```
# Observation Log: research

**Started**: 2025-11-08T14:30:00.000Z
**Workflow**: research

## Query

How can we migrate from custom stylesheets to Tailwind v4 implemented
in Lisp for better composability and maintainability?

---

## Iteration 1

**Timestamp**: 2025-11-08T14:32:41.482Z

### Progress

- **Dual styling system discovered**
  - Evidence: `static/css/ (30+ files), src/components/ (21 files)`
  - Confidence: high
- **Legacy CSS uses ITCSS layers with BEM naming**
  - Evidence: `static/css/01-foundations/, 02-utilities/, ...`
  - Confidence: high

### Remaining

- What is the architecture of the new component library? [BLOCKING]
- How does Tailwind v4 integrate with the build system? [BLOCKING]

### Assessment

| achieved_primary_goal | false |
| have_sufficient_evidence | false |
| confidence | low |

**Workers Decided**: codebase-analyzer, web-researcher, pattern-finder

---
```

```
## Iteration 2

**Timestamp**: 2025-11-08T14:35:01.250Z

### Input (from previous iteration)

**Workers**: 3/3 succeeded

### Progress

- **Component library uses pure functions, not macros**
  - Evidence: `src/components/utils/generators.lisp:1-200`
  - Confidence: high
- **Four-layer architecture: tokens -> validation -> generators -> semantic**
  - Evidence: `src/components/{tokens,utils,primitives,composed}/`
  - Confidence: high
- **Tailwind v4 uses CSS-first configuration**
  - Evidence: `Web research: Tailwind v4 documentation`
  - Confidence: high
- **131 passing tests in component library**
  - Evidence: `nix build .#lsc-web.tests`
  - Confidence: high

### Remaining

- Which pages are already migrated vs legacy? [BLOCKING]

**Workers Decided**: codebase-locator

---

## Iteration 3

**Timestamp**: 2025-11-08T14:37:15.000Z

### Input (from previous iteration)

**Workers**: 1/1 succeeded

### Progress

[All previous items retained]
- **Contact page fully migrated to component library**
  - Evidence: `src/content/contact.lisp:17-172`
  - Confidence: high
- **Six pages still using legacy CSS**
  - Evidence: `src/pages.lisp inline class analysis`
  - Confidence: high

### Assessment

| achieved_primary_goal | true |
| have_sufficient_evidence | true |
| confidence | high |

**Should Continue**: false
```

The structured format—iterations with timestamps, progress items citing evidence and confidence levels, blocking items that drive worker selection, worker success and failure reporting—provides what I think of as accountability for the orchestrator's decisions. If the coordinator made a questionable choice about which agents to spawn or when to terminate, I can see the reasoning that led there.

The `handoffs/` directory also deserves mention. When I need to end a session mid-task (due to context filling up or needing to step away), I invoke the `/handoff` command, Horthy and HumanLayer [HH25], to create a transfer document capturing context that cannot be inferred from the artifacts alone, e.g., what has been discussed in the conversation, what the current blockers are, what should happen next. A new session can read the handoff and resume work with minimal context loss (at least when you also combine it with the detailed plan).

The `runs/` directory contains the orchestrator's internal execution state, organized by invocation. Each run directory is named with a sequence number, workflow type, timestamp, and short identifier:

```
runs/001-research-20251108T143000-a1b2/
|-- logs/
|   `-- research-2025-11-08T14-30-00.log
|-- state/
|   `-- iteration-state.json
|-- telemetry/
|   `-- research-2025-11-08T14-30-00.json
`-- worker-outputs/
    |-- 001-codebase-locator.json
    |-- 002-codebase-analyzer.json
    |-- 002-pattern-finder.json
    |-- 002-web-researcher.json
    `-- 003-codebase-locator.json
```

The subdirectories serve distinct purposes:

- `logs/`: Execution logs for the workflow.
- `state/`: Coordinator working memory (accumulated facts, summaries, iteration state).
- `telemetry/`: Execution metrics for workflow improvement analysis.
- `worker-outputs/`: Typed outputs from each agent, named by iteration number and agent name.

I find the `worker-outputs/` particularly useful for debugging when an agent returns unexpected results; I can inspect the raw typed output to understand what went wrong.

8.2.2 Session Directory

Separately from the task directory, each conversation session maintains state in `.sessions/${sessionId}/`. This is the session evidence trail described in Section 6:

```
.sessions/a1b2c3d4-e5f6-7890-abcd-ef1234567890/
|-- session.json
`-- memory/
    |-- attribution.json
    |-- feedback.json
    `-- curation-queue.json
```

The session evidence trail comprises:

- `session.json`: Session metadata, including task association.
- `attribution.json`: Records of which playbook patterns were consulted.
- `feedback.json`: Effectiveness assessments recorded during the session.
- `curation-queue.json`: Pending playbook updates awaiting reflection.

The session directory links to tasks via `playbook_set_task`, which records the association in `session.json`. This link is what enables the Reflector to correlate evidence trail entries with task artifacts during reflection, since it knows which task the attributions and feedback belong to.

8.2.3 State Flow Across Phases

I find it helpful to trace how state flows across the four phases, since this clarifies what ends up where and why the task directory and session directory serve complementary roles.

1. **Research**: The orchestrator creates a run directory under the task's `runs/`. Worker outputs accumulate there. The coordinator synthesizes findings into `research-{topic}.md` in the task directory. If any patterns are consulted during research, attribution entries are written to the session's `.sessions/${sessionId}/memory/attribution.json`.

2. **Planning**: The main thread reads `research-{topic}.md` and queries the playbook for relevant patterns. Each consultation adds an attribution entry to the session directory. The agent produces `plan.md` in the task directory, often with explicit pattern references like "per pattern `[ace-000017]`".

3. **Implementation**: The main thread executes plan phases. `observations-implementation.md` grows as work progresses. Attribution entries continue accumulating in the session directory; feedback entries (recording whether patterns helped or hurt) are written to `feedback.json`. If I end the session mid-task, a handoff document is written to the task's `handoffs/`.

4. **Reflection**: The orchestrator creates another run directory. The Reflector reads both the task artifacts and the session's evidence trail. Curation proposals queue in `curation-queue.json`. The Curator processes this queue and applies changes atomically to the playbook.

The task directory persists indefinitely as the historical record of what was done. The session directory persists until reflection processes it; afterward, the evidence has been incorporated into the playbook and I can archive or remove the session directory.

8.3 Playbook Integration Points

The playbook participates at specific points in each phase. I want to be concrete about these integration points because they are where the learning loop becomes tangible.

Research. Before dispatching agents, the coordinator calls `playbook_search` with terms relevant to the task domain. Retrieved patterns get injected into agent prompts as additional context, biasing investigation toward areas where I have prior knowledge. The orchestrator also calls `playbook_set_task` to establish the session-task linkage that reflection will need later.

Planning. The agent invokes `playbook_search` and `playbook_proven` to retrieve patterns relevant to implementation strategy. Each consultation gets recorded via `playbook_applied`, creating attribution entries in session memory. The plan document itself references patterns by identifier—for instance, "per pattern `[ace-000017]`, implement phase-by-phase with verification gates"—making the influence explicit and traceable. I review the plan and can see which patterns shaped the approach.

Implementation. When a pattern actively guides a decision, the agent records attribution. When a phase completes, the agent or I may record feedback via `playbook_feedback`. These entries accumulate in session memory throughout implementation, forming the evidence trail that reflection will analyze.

Reflection. The Reflector reads task artifacts and queries session memory for attribution and feedback entries. It analyzes this evidence to propose curation operations:

- `playbook_increment`: Update counters for patterns that contributed to success or failure.
- `playbook_add`: Propose new patterns distilled from novel solutions, after checking `playbook_dedupe` for semantic similarity to existing entries.
- `playbook_merge`: Recommend consolidating patterns that overlap significantly.

The Curator validates proposals and applies them atomically via `playbook_apply_curation`. The playbook evolves, incorporating evidence from the completed task.

8.4 Concrete Example

A CSS migration task from the Lie-Størmer Center infrastructure illustrates how these phases work in practice—though I should note that this is a somewhat idealized reconstruction of what was, at the time, a messier process. The task was to migrate the website from thirty custom CSS files to Tailwind classes generated from Lisp functions.

Research. I invoked `/research` and the orchestrator created `runs/001-research-20251108T143000-a1b2/`. Four agents executed in parallel: a codebase locator found the CSS file structure (ITCSS layers, BEM naming); a codebase analyzer examined the emerging component library (tokens, validation, generators); a web researcher gathered Tailwind v4 documentation; a pattern finder located styling patterns across the repository.

Worker outputs accumulated in `worker-outputs/`. The coordinator synthesized these into `research.md`, documenting a dual styling system: legacy BEM-based CSS alongside a token-validated component library with passing tests. I reviewed the observations, added annotations about design quality—which pages had good designs worth preserving, which had broken designs requiring fixes—and approved advancement to planning.

Planning. In the main thread, the agent read the research document and queried the playbook. Queries for "phase-by-phase implementation" returned pattern `[ace-000017]` (helpful=32); queries for "verification" returned `[ace-000052]` (helpful=38). The agent recorded both as applied.

The agent proposed a seven-phase plan: baseline screenshots, simple page migration, complex page migration, design fixes, template updates, CSS deletion, and final verification. Each phase specified success criteria. I reviewed the plan, requested adjustments to phase grouping (batching similar pages), and approved the final version.

Implementation. The execution was not entirely hands-off. I found myself watching more closely than I had anticipated, particularly during phase 1 when baseline screenshots were captured and I wanted to verify that the visual quality was adequate for later comparison. Phase 2 migrated three pages, and here I ran the test suite myself rather than waiting for the agent to do so, mostly out of impatience. The agent logged decisions and problems encountered in `observations-implementation.md`, which proved useful when I later tried to understand why certain choices had been made.

The pattern attribution machinery worked as designed: when `[ace-000017]` guided the phase structure, the agent recorded this, and when phase 2 succeeded we recorded positive feedback for both patterns that had been consulted. The task spanned multiple days—I do not recall exactly how many, but at least three—and I created handoff documents in `handoffs/` at session boundaries so that context could be reconstructed later.

Reflection. After implementation completed, I invoked `/reflect`. The orchestrator created `runs/002-reflect-20251112T160000-c3d4/`. The Reflector read task artifacts and session evidence: two patterns applied, both with positive feedback, plus implementation observations documenting a novel "preserve good, fix broken" design strategy.

The workflow produced a summary and the helpful counters for `[ace-000017]` and `[ace-000052]` incremented as anticipated, but the system also identified a novel pattern in my implementation observations, specifically the "preserve good, fix broken" design strategy.

8.5 Correspondence with the Learning Loop

The four-phase workflow I have described maps, imperfectly but recognizably, onto the five-stage learning loop from Section 7:

Learning Stage	Workflow Participation
Generation	Research, planning, and implementation all generate work while the agent consults the playbook.
Attribution	The agent records pattern consultations via `playbook_applied`; implementation is the primary source.
Feedback	Typically recorded after phase completions during implementation; the agent or I may record these.
Reflection	The `/reflect` command triggers analysis of accumulated evidence across all phases.
Curation	The Curator applies queued updates atomically after the reflection document has been written.

This mapping reveals something I find important: playbook evolution is punctuated rather than continuous. The playbook remains stable during a task, providing consistent guidance across phases. Evolution occurs at task boundaries, when reflection processes accumulated evidence and curation applies the resulting updates.

This punctuated evolution aligns with the session isolation architecture we described in Section 6. Evidence accumulates in session-local storage during the task. The playbook file remains immutable, avoiding the race conditions we analyzed in Proposition 5.12. Only during the explicit curation phase does the Curator perform the single atomic write that updates the shared playbook. The deferred curation mechanism ensures consistency (Proposition 5.13) while enabling concurrent sessions to operate independently.

9 Discussion

9.1 Relationship to Prior Work

Our implementation draws on several strands of prior research. Zhou et al. [Zho+23] and Fernando et al. [Fer+23] demonstrated that prompts can be optimized through evolutionary processes, but their methods suffer from brevity bias as noted by Zhang et al. [Zha+25]. Suzgun et al. [Suz+25] introduced dynamic cheatsheets that accumulate task-specific knowledge, but lack the structured curation that prevents context collapse.

Independent of the academic literature, practitioners have developed complementary methodologies for managing context in agentic coding systems. Horthy and HumanLayer [HH25] intro-

duced the Research-Plan-Implement (RPI) workflow, which structures development into sequential phases with explicit artifact boundaries. Their concept of *frequent intentional compaction* advocates for deliberately condensing findings into structured documents that serve as fresh context windows for subsequent phases, maintaining context utilization within manageable bounds. Our implementation builds upon this foundation, extending the RPI workflow with a fourth phase for reflection that enables the playbook evolution central to ACE. We adopt their handoff document pattern for session transfer and their phased artifact structure for organizing task directories.

The ACE framework synthesizes these approaches through the three-role architecture, and our implementation extends it with session isolation and deferred curation that enable practical deployment in concurrent environments.

9.2 Cost Considerations

Zhang et al. [Zha+25] argue that longer contexts do not necessarily translate to linearly higher inference cost. They note that modern serving infrastructures employ KV cache reuse, compression, and offload techniques that amortize the cost of long contexts, and that ongoing advances in ML systems suggest this amortized cost will continue to decrease. We have not independently verified these claims through measurement.

In our implementation, the tool-based interface means that only relevant pattern subsets are retrieved and processed. A session that queries for methodology patterns does not pay the cost of loading language-specific or domain-specific entries that are irrelevant to the current task.

9.3 Limitations

We emphasize that this report describes a working implementation rather than a rigorously evaluated system. Our assessment of the architecture's effectiveness is based entirely on practical experience during development; we have not conducted controlled experiments comparing the approach against alternatives. The orchestrator plugin remains under active development, and the design continues to evolve as we encounter new requirements. What we can report is that the architecture aligns well with how we think about complex development tasks: the phased workflow matches our natural decomposition of problems, and the playbook evolution mechanism captures insights that would otherwise be lost between sessions.

The effectiveness of this implementation depends on the quality of the Reflector's analysis. In domains where models cannot extract useful insights from trajectories, the accumulated context may become noisy. Additionally, the deferred curation mechanism introduces latency between pattern application and playbook update; patterns proven effective in one session may not benefit subsequent sessions until the next reflection cycle.

9.4 Future Directions

Several extensions warrant investigation. Depth-limited recursion could allow workers to spawn sub-workers for complex decomposition tasks, moving beyond the current two-level constraint (Section 3.3). Cross-task analysis could aggregate pattern effectiveness across multiple tasks to identify domain-wide strategies. Selective unlearning could use context interpretability techniques to remove outdated or incorrect information in response to evolving requirements.

Two fundamental challenges remain unresolved, and they are inherently linked. *Sycophancy* undermines the feedback mechanism: current models tend not to increment harmful counters even when patterns lead to poor outcomes, as they avoid negative assessments of their own prior suggestions. This directly causes **context poisoning**: the theoretical correction mechanism relies on harmful counter increments to eventually remove bad patterns, but since sycophantic

models do not provide negative feedback, harmful patterns persist indefinitely. In our implementation, we observed harmful counters remain at zero regardless of actual pattern effectiveness, allowing incorrect or misleading patterns to accumulate without correction. Addressing this limitation likely requires advances in model calibration and self-critique capabilities; until then, human oversight of playbook content remains essential.

10 Conclusion

This report has presented an implementation of Agentic Context Engineering that addresses the brevity bias and context collapse limitations identified in prior work. The session-based evidence trail architecture enables concurrent operation without race conditions. Deferred curation maintains single-writer semantics while preserving the evidence trail required for reflection. The tool-based interface implements the environment paradigm advocated by recursive language model research, treating the playbook as an external variable rather than a prompt component.

The implementation demonstrates that comprehensive, evolving contexts can be maintained efficiently in practical systems. By treating playbooks as living documents that accumulate evidence-based insights, the system creates a closed learning loop: generation applies patterns, attribution tracks usage, feedback reports outcomes, reflection analyzes effectiveness, and curation updates the playbook.

Several observations from this work seem worth recording, though I am uncertain which will prove most important. The distinction between human artifacts and machine-managed state—plans and observations belonging to the human workflow, run directories and evidence trails being implementation details—proved essential in practice, even though I did not anticipate its importance when designing the system. It turns out that knowing what belongs to whom matters more than I expected.

The environment paradigm, accessing context through tools rather than embedding it in prompts, is not merely an optimization. I had initially thought of it as a way to reduce token costs, but it has become clear that this is a fundamental architectural choice that determines whether the system can scale at all. The alternative—embedding ever-growing playbooks in prompts—simply does not work beyond a certain size.

The most sobering realization concerns model sycophancy. The theoretical elegance of feedback-driven curation collides with the practical reality that models are not reliable critics of their own prior suggestions. Until this changes, self-improvement mechanisms will require human oversight to prevent what I have come to think of as context poisoning: the gradual accumulation of plausible-sounding but unhelpful patterns. The harmful counter, for its part, remains optimistic.

References

[Ant25a] Anthropic. *Claude Code: An Agentic Coding Tool*. https://docs.anthropic.com/en/docs/claude-code. CLI tool for agentic coding with Claude. Enforces two-level agent composition: agents spawned via Task tool cannot recursively spawn further agents. 2025. URL: https://docs.anthropic.com/en/docs/claude-code.

[Ant25b] Anthropic. *Effective Context Engineering for AI Agents*. Anthropic Engineering Blog. Published 2025-09-29. Describes mode separation, just-in-time context loading, and context compaction strategies. Sept. 2025. URL: https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents.

[Ant25c] Anthropic. *Effective Harnesses for Long-Running Agents*. Anthropic Engineering Blog. Published 2025-11-26. Introduces initializer/coding agent pattern and artifact-based handoffs for maintaining agent coherence across sessions. Nov. 2025. URL: https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents.

[Fer+23] Chrisantha Fernando et al. *Promptbreeder: Self-Referential Self-Improvement Via Prompt Evolution*. 2023. arXiv: 2309.16797 [cs.CL]. URL: https://arxiv.org/abs/2309.16797.

[HH25] Dex Horthy and HumanLayer. *Advanced Context Engineering for Coding Agents*. GitHub repository. Introduces the Research-Plan-Implement workflow for structured AI development. Aug. 2025. URL: https://github.com/humanlayer/advanced-context-engineering-for-coding-agents.

[Lei14] Daan Leijen. "Koka: Programming with Row Polymorphic Effect Types". In: *Mathematically Structured Functional Programming (MSFP 2014)*. Vol. 153. Electronic Proceedings in Theoretical Computer Science. Introduced row-polymorphic effect types enabling compositional effect systems. 2014, pp. 100–126. DOI: 10.4204/EPTCS.153.8.

[nLa26a] nLab authors. *Action*. Revision 78. 2026. URL: https://ncatlab.org/nlab/show/action (visited on 01/07/2026).

[nLa26b] nLab authors. *Action groupoid*. Revision 35. 2026. URL: https://ncatlab.org/nlab/show/action+groupoid (visited on 01/07/2026).

[nLa26c] nLab authors. *Direct sum*. Revision 35. 2026. URL: https://ncatlab.org/nlab/show/direct+sum (visited on 01/07/2026).

[nLa26d] nLab authors. *Pushout*. Revision 33. 2026. URL: https://ncatlab.org/nlab/show/pushout (visited on 01/07/2026).

[nLa26e] nLab authors. *Restricted direct product*. Revision 7. 2026. URL: https://ncatlab.org/nlab/show/restricted+direct+product (visited on 01/07/2026).

[PP03] Gordon Plotkin and John Power. "Algebraic Operations and Generic Effects". In: *Applied Categorical Structures* 11.1 (2003). Established categorical foundations: generic effects are equivalent to algebraic operations, pp. 69–94. DOI: 10.1023/A:1023064908962.

[PP08] Gordon D. Plotkin and John Power. "Tensors of Comodels and Models for Operational Semantics". In: *Mathematical Foundations of Programming Semantics (MFPS XXIV)*. Vol. 218. Comodels capture external world interactions; tensoring with models produces execution semantics. 2008, pp. 295–311. DOI: 10.1016/j.entcs.2008.10.018.

[PP13] Gordon D. Plotkin and Matija Pretnar. "Handling Algebraic Effects". In: *Logical Methods in Computer Science* 9.4 (2013). Definitive reference for effect handlers. Covers exceptions, state, nondeterminism, I/O, concurrency. DOI: 10.2168/LMCS-9(4:23)2013.

[Sha+11] Marc Shapiro et al. *A Comprehensive Study of Convergent and Commutative Replicated Data Types*. Tech. rep. RR-7506. Establishes convergence theorems for state-based and operation-based CRDTs. INRIA, 2011. URL: https://pages.lip6.fr/Marc.Shapiro/papers/RR-7687.pdf.

[SST26] SST. *OpenCode: The Open Source AI Coding Agent*. https://opencode.ai. Open source AI coding agent supporting 75+ LLM providers across terminal, IDE, and desktop environments. 2026. URL: https://opencode.ai.

[Suz+25] Mirac Suzgun et al. *Dynamic Cheatsheet: Test-Time Learning with Adaptive Memory*. 2025. arXiv: 2504.07952 [cs.LG]. URL: https://arxiv.org/abs/2504.07952.

[TOO25] TOON Contributors. *TOON: Token-Oriented Object Notation*. GitHub repository. Compact, schema-aware serialization format achieving approximately 40% token reduction compared to JSON while maintaining explicit structure for LLM applications. 2025. URL: https://github.com/toon-format/toon.

[Zha+25] Qizheng Zhang et al. *Agentic Context Engineering: Evolving Contexts for Self-Improving Language Models*. 2025. arXiv: 2510.04618 [cs.LG]. URL: https://arxiv.org/abs/2510.04618.

[Zho+23] Yongchao Zhou et al. *Large Language Models Are Human-Level Prompt Engineers*. 2023. arXiv: 2211.01910 [cs.LG]. URL: https://arxiv.org/abs/2211.01910.

[ZKK25] Alex L. Zhang, Tim Kraska, and Omar Khattab. *Recursive Language Models*. 2025. arXiv: 2512.24601 [cs.AI]. URL: https://arxiv.org/abs/2512.24601.