SuperOpt: Agentic Environment Optimization for Autonomous AI Agents

Shashi Jagtap
Superagentic AI
shashi@super-agentic.ai

https://github.com/SuperagenticAI/superopt

Keywords: Large Language Models, Autonomous Agents, Agentic AI, Environment Optimization, Self-Optimization, Natural Language Gradients, Prompt Engineering, Retrieval-Augmented Generation, Agent Memory

Abstract

Large Language Model (LLM) agents are increasingly deployed as autonomous systems embedded within complex operational environments composed of prompts, tool interfaces, retrieval pipelines, execution protocols, and persistent memory. Recent research has demonstrated that post-training optimization of isolated environment components can yield significant performance improvements. In particular, **Genetic Pareto (GEPA)** [Agrawal et al., 2025] shows that system prompts can be optimized through multi-objective evolutionary search, while **Agentic Context Engineering (ACE)** [Zhang et al., 2025] demonstrates that reflective context accumulation can improve agent behavior over repeated interactions. However, these approaches optimize individual environment dimensions in isolation and do not address the coupled failure dynamics that arise in realistic deployments. Empirical analysis of execution traces reveals that agent failures frequently emerge from interactions between environment layers. Improvements to prompts may expose brittle tool schemas, retrieval adjustments may induce context overflow, and memory accumulation may destabilize long-horizon behavior. Optimizing a single layer often degrades performance in others.

We introduce **SuperOpt**, a unified framework for *Agentic Environment Optimization* that treats the entire agent environment as a structured optimization target while keeping model parameters frozen. Drawing on the principles of **"textual differentiation"** from **TextGrad** [Yuksekgonul et al., 2024], the **Natural Language Gradients** concept from **ProTeGi** [Pryzant et al., 2023], and the programmatic modularity of **DSPy** [Khattab et al., 2023], SuperOpt formalizes optimization as iterative descent over Natural Language Gradients derived from execution traces. A meta-diagnostic controller attributes failures to specific environment layers and routes corrective updates to specialized optimization engines. These include SuperPrompt for evolutionary instruction optimization, SuperReflexion for self-healing tool schemas, SuperRAG for adaptive retrieval configuration, and SuperMem for typed memory, decay, and global stability enforcement.

We evaluate SuperOpt on the Aider coding agent with challenging algorithmic tasks, demonstrating a **10% improvement in success rate** (90% $\rightarrow$ 100%) and **1.6$\times$ faster execution** compared to baseline configurations. All tasks converge to stable environments with zero oscillation, validating the stability guarantees of the hierarchy of mutability.

We argue that agent reliability is fundamentally an environmental property rather than a model property, and that systematic environment-level optimization is a necessary complement to model scaling for building robust autonomous agents.

1 Introduction

The dominant paradigm in AI research attributes agent failures primarily to reasoning deficits in the underlying model. This model-centric view drives the industry toward constant retraining and fine-tuning. However, empirical analysis of agents in production (e.g., Aider) reveals that many failures occur even when the model's reasoning is locally correct. These failures often arise from a misaligned environment: ambiguous tool schemas, noisy retrieval, or context drift.

1.1 The Model-Centric Paradigm

The dominant paradigm in LLM agent research has achieved remarkable success by focusing on improving the reasoning capabilities of the underlying model. This model-centric approach has driven impressive gains through increasing model size, fine-tuning on task-specific data, and reinforcement learning from human feedback. Within this framing, the agent environment is treated as a largely static wrapper around the model, and improvements in agent performance follow directly from improvements in model parameters.

While this paradigm has been tremendously successful, we observe that it does not fully address a class of persistent failures in deployed agents. Across coding assistants, research agents, and automation systems, inspection of execution traces reveals that many failures occur even when the model's reasoning is locally correct. Instead, failures arise from misalignments between the agent's outputs and the constraints imposed by the surrounding environment.

These failures include ambiguous tool schemas, underspecified system instructions, missing or noisy retrieval context, contradictory memory entries, and uncontrolled growth of contextual state. Importantly, these failures persist even when stronger models are substituted. This observation suggests that many agent failures are not rooted in reasoning deficits but in structural deficiencies of the agent environment itself. Failures such as ambiguous tool schemas, retrieval noise, and "context collapse" persist even as models scale. Recent advancements in Effective Context Engineering (Anthropic, 2024) emphasize that how information is presented to the model is often as important as the model's raw capability. Yet, context is typically treated as a static or manually tuned artifact.

1.2 Empirical Evidence for Environmental Failures

Analysis of long-horizon execution traces across multiple agent systems (Aider, Letta Code, Codex) reveals recurring categories of environmental failure. These failures are deterministic and reproducible, requiring structural fixes rather than improved reasoning.

Example Failure Case: An Aider agent attempted to edit line 0 in a file, causing a tool error. The same error occurred multiple times across different tasks before SuperOpt added a schema clarification: "Line numbers are 1-indexed, not 0-indexed." This demonstrates that environmental failures persist even when the model's reasoning is correct.

The recurring categories of environmental failure include:

- **Tool contract violations**, where agents repeatedly misuse tools by passing arguments with incorrect types, indexing conventions, or formatting.
- **Retrieval insufficiency**, where agents hallucinate symbols, functions, or files because relevant information was not retrieved into context despite existing in the underlying corpus.
- **Instruction ambiguity**, where vague or conflicting system prompts lead to inconsistent behavior across tasks.

- **Memory drift**, where accumulated reflections or heuristics become stale, contradictory, or overly generic, degrading performance over time.
- **Context explosion**, where attempts to fix failures by injecting additional context overwhelm the model's context window and reduce reasoning quality.

Crucially, many of these failures are deterministic and reproducible. They do not disappear with improved reasoning capability alone. Instead, they require structural changes to the environment in which the model operates.

1.3 Limits of Single-Dimension Optimization

Recent work has begun to explore post-training optimization of agent behavior, but almost exclusively along single environment dimensions.

GEPA [Agrawal et al., 2025] represents a breakthrough in prompt optimization, framing system prompts as executable programs and applying evolutionary search guided by Pareto objectives such as task success, robustness, and token efficiency. GEPA elegantly demonstrates that prompt optimization alone can yield substantial gains without modifying model weights. GEPA's focused scope on prompt optimization is a design strength, but it naturally leaves room for complementary work on other environment layers. SuperOpt builds directly on GEPA's insights, extending the optimization paradigm to tool interfaces, retrieval pipelines, and memory mechanisms.

DSPy [Khattab et al., 2024] has revolutionized prompt optimization by treating it as a programming task with composable modules. DSPy's programmatic approach to prompt engineering is foundational to modern compound AI systems. SuperOpt adopts DSPy's philosophy of treating prompts as optimizable programs and extends this principle across the full environment stack.

Agentic Context Engineering (ACE) [Zhang et al., 2025] introduces the powerful idea of reflective accumulation of context across episodes, allowing agents to reason over prior failures and successes. ACE demonstrates that agents can learn from experience through context alone. SuperOpt builds on ACE's memory-centric insights by adding typed memory categories, decay mechanisms, and stability constraints to manage context growth over extended deployments.

Meta-ACE [Romero, 2025] further advances ACE with strategy selection and meta-reasoning, demonstrating sophisticated context management. SuperOpt complements these advances by extending optimization to tool schemas and retrieval systems, enabling repair of interfaces that would otherwise require manual intervention.

These pioneering approaches have established the foundation for post-training optimization. SuperOpt unifies their insights into a single framework that coordinates optimization across all environment layers, preventing the oscillation that can occur when layers are optimized independently.

1.4 Environment-Level Optimization

SuperOpt proposes a shift from model-centric and component-centric optimization to **environment-level optimization**. We formalize agent behavior as:

$$y = f_\theta(x; \Phi)$$

where: - θ denotes frozen model parameters. - Φ denotes the mutable agent environment.

Rather than adapting θ, SuperOpt iteratively patches Φ using feedback derived from execution traces. This framing enables systematic improvement while remaining model-agnostic and avoiding retraining.

1.5 Why Coding Agents

Coding agents provide a particularly stringent domain for studying agentic environment optimization. They operate over strict tool protocols, large structured codebases, deterministic execution feedback, and long-horizon tasks. Small environmental misalignments, such as off-by-one indexing or missing imports, lead to hard failures that are immediately observable.

Moreover, improvements can be evaluated objectively through compilation success, test results, and commit correctness. For these reasons, coding agents serve as an ideal motivating domain while remaining representative of broader autonomous agent systems.

2 Contributions

This paper makes the following contributions:

1. **Unified Optimization Abstraction**: We formalize agentic environment optimization as iterative descent over Natural Language Gradients, providing a single framework that subsumes prompt optimization (GEPA), context engineering (ACE), and extends to tool and retrieval layers.
2. **Hierarchy of Mutability**: We introduce a formal stability ordering over environment components that prevents optimization oscillation and ensures convergence to stable configurations.
3. **Multi-Layer Diagnostic Routing**: We present SuperController, a meta-diagnostic mechanism that attributes failures to specific environment layers and routes updates to specialized optimizers.
4. **Tool Schema Self-Repair**: We introduce SuperReflexion, a mechanism for self-healing tool schemas that repairs interface contracts, a capability not present in prior work.
5. **Typed Memory with Decay**: We present SuperMem, a memory system with explicit typing, confidence decay, and conflict resolution that prevents the context collapse observed in unbounded accumulation approaches.
6. **Empirical Validation**: We demonstrate SuperOpt on the Aider coding agent, achieving 10% improvement in success rate and 1.6x faster execution on challenging algorithmic tasks.

Each contribution maps to subsequent sections: Contributions 1-2 are formalized in Section 3, Contribution 3 in Section 8, Contributions 4-5 in Sections 9-10, and Contribution 6 in Sections 15-17.

3 Problem Formulation

3.1 Agent–Environment Model

We define the *agentic environment* at time t as the structured tuple:

$$\Phi_t = \{P_t, T_t, R_t, M_t\}$$

where:

- (P_t) denotes system prompts, instruction policies, and few-shot exemplars.
- (T_t) denotes tool schemas, APIs, execution protocols, and interface constraints.
- (R_t) denotes retrieval pipelines, including chunking strategies, query generation policies, ranking mechanisms, and context assembly rules.
- (M_t) denotes persistent memory, including accumulated heuristics, reflections, rules, and learned constraints.

This decomposition explicitly captures the multi-layered structure of real-world agent environments.

An execution trace (τ) is a structured record of an agent's interaction with its environment during a task. It includes:

- Model outputs and intermediate reasoning steps.
- Tool invocation logs, arguments, and error messages.
- Retrieval queries, retrieved documents, and ranking scores.
- Memory reads, writes, and updates.
- External execution results such as compiler errors, test failures, or runtime exceptions.

Execution traces constitute the primary supervision signal for SuperOpt.

3.2 Environment Components

Prompt Layer (P). The prompt layer consists of:

- System-level instructions
- Behavioral constraints
- Output format specifications
- Few-shot exemplars
- Safety or style rules

These elements are treated as structured text blocks rather than opaque strings. The prompt is structured into explicit sections:

`[ROLE] [GLOBAL RULES] [EDITING PROTOCOL] [ERROR HANDLING] [OUTPUT FORMAT] [EXAMPLES]`

Each section can be independently mutated.

Common prompt pathologies observed include:

- Lack of explicit file-edit discipline
- Missing rules for minimal diffs
- Conflicting instructions about verbosity
- Ambiguous termination conditions
- No prioritization between correctness and style

These lead to brittle behavior even with strong models.

Tool Layer (T). Each tool is defined by a schema describing:

- Arguments and types
- Required vs optional fields
- Constraints
- Natural-language descriptions

Tool errors as interface mismatches are deterministic and reproducible. They indicate that the tool contract is underspecified or ambiguous. Examples:

- Line numbers start at 1, not 0
- File paths must be relative
- Patch hunks must include context
- Commands must be shell-safe

Retrieval Layer (R). Parameters subject to optimization include:

- Chunk size and overlap
- Top-k retrieval depth
- Reranking thresholds
- Query rewriting strategy
- Structural vs semantic retrieval modes
- File-type filters
- Dependency expansion depth

Typical retrieval failures include:

- Missing the file defining a symbol
- Retrieving unrelated files
- Overloading context with large files
- Ignoring dependency relationships

Memory Layer (M). Memory entries are typed according to their function:

- `TOOL_RULE`: Constraints on tool usage (e.g., indexing, argument formats)
- `EDIT_CONSTRAINT`: Rules for code modifications (e.g., preserve imports)
- `STYLE_GUIDELINE`: Coding style preferences (e.g., indentation, naming)
- `BUG_PATTERN`: Known failure patterns to avoid
- `RETRIEVAL_HEURISTIC`: Learned retrieval strategies
- `STRATEGY`: High-level behavioral heuristics
- `RAG_HEURISTIC`: Retrieval tuning knowledge
- `PROMPT_CONSTRAINT`: Stylistic rules

Each memory entry contains structured metadata:

```
Type: TOOL_RULE
Content: "Always use relative file paths in diffs"
Confidence: 0.92
Timestamp: t
```

3.3 Optimization Objective

SuperOpt's goal is simple: **improve the agent's success rate by fixing its environment**, not by retraining the model.

What We Start With:

- An LLM agent with **frozen model weights** (no fine-tuning allowed)
- An initial environment configuration containing prompts, tools, retrieval settings, and memory
- A set of tasks to evaluate performance

What We Produce:

- An **optimized environment** where each component has been refined based on observed failures
- A log of execution traces documenting what was learned

The Core Objective:

SuperOpt seeks to find the best environment configuration that maximizes task success. In simple terms: we want to find the combination of prompts, tool schemas, retrieval settings, and memory rules that helps the agent succeed most often.

Key Constraints:

1. **No model changes**: The LLM's weights remain frozen. We only modify the environment around it
2. **Respect the hierarchy**: Higher-priority constraints (like tool syntax rules) cannot be overridden by lower-priority ones (like style preferences)
3. **Learn from traces**: All improvements must be derived from actual execution failures, not from external oracles

How Updates Work:

After each task execution, SuperOpt analyzes the trace and generates an environment update. This update is validated against the hierarchy and then applied, producing an improved environment for the next iteration.

When to Stop:

Optimization terminates when:

- Task success rate reaches an acceptable threshold
- No significant improvements are being made
- Maximum iterations are reached

How SuperOpt Differs from Other Approaches:

- **vs. Reinforcement Learning**: RL updates model parameters; SuperOpt updates the environment
- **vs. Prompt Optimization**: Prompt methods optimize only prompts; SuperOpt optimizes prompts, tools, retrieval, and memory together
- **vs. Program Synthesis**: Synthesis generates new programs; SuperOpt optimizes existing execution environments

4 Design Principles

SuperOpt is built on five core design principles:

Principle 1: Separation of Policy and Optimizer. The agent policy (the LLM with frozen weights) is treated as a black box. Optimization occurs entirely at the environment level, enabling model-agnostic optimization without fine-tuning.

Principle 2: Failure-Driven Learning. Optimization signals are derived from failures, not successes. When tasks succeed, no update is needed. This focuses effort on actual problems and avoids over-optimization.

Principle 3: Hierarchical Stability. Environment components are ordered by mutability. Higher-priority constraints cannot be overridden by lower-priority optimizations, preventing oscillation and preserving fundamental invariants.

Principle 4: Composable Optimizers. Each environment layer (P, T, R, M) has a dedicated optimizer that operates independently. The SuperController routes failures to appropriate optimizers without requiring monolithic reasoning.

Principle 5: Transparency and Interpretability. All environment state is explicit, typed, and inspectable. Updates are expressed in natural language and can be reviewed, approved, or rejected. Given the same trace and state, optimization produces deterministic, reproducible results.

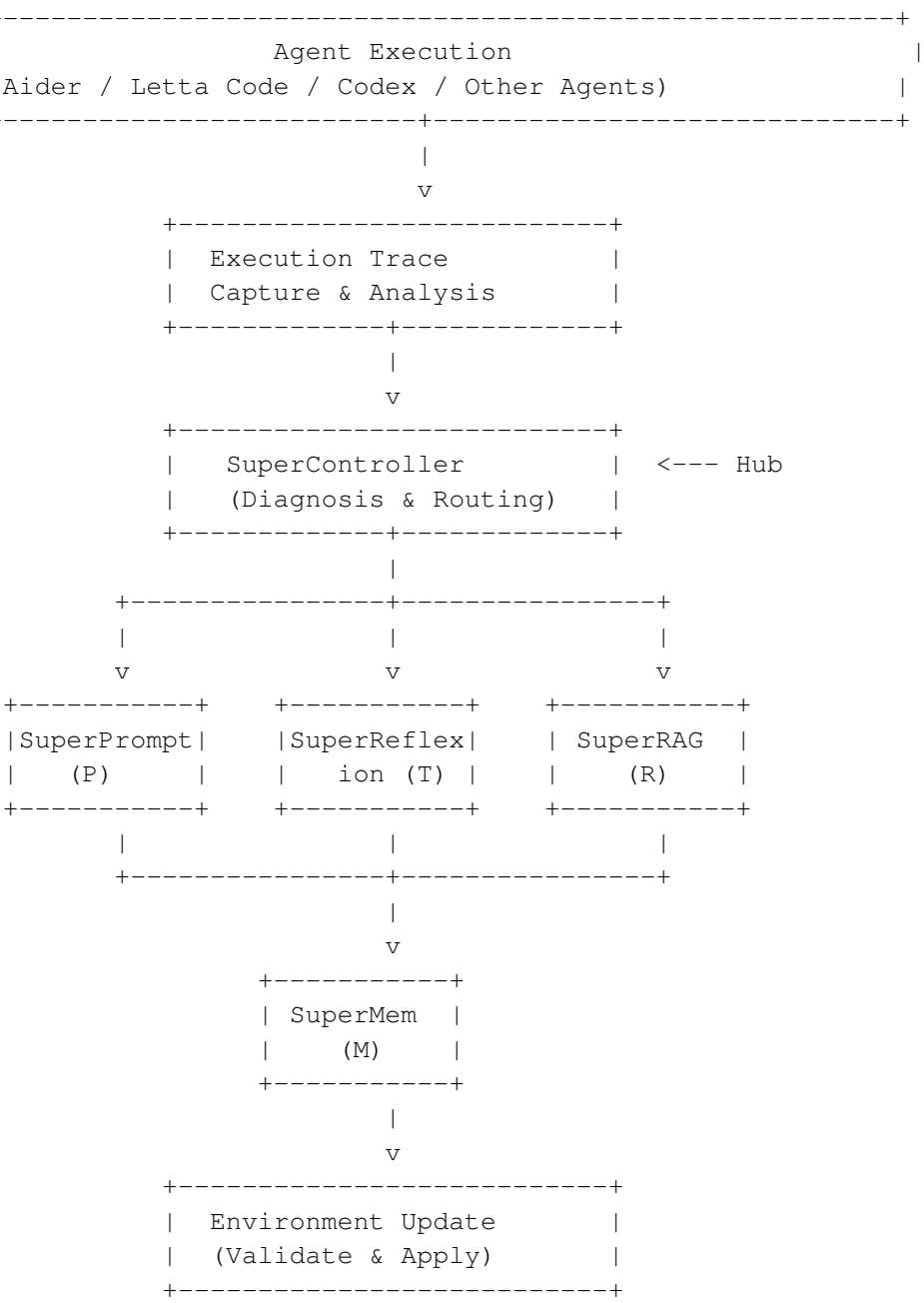
These principles distinguish SuperOpt from end-to-end learned systems.

5 System Overview

5.1 Architecture Overview

SuperOpt is organized as a hub-and-spoke architecture centered around a diagnostic meta-controller. The system operates in an outer optimization loop surrounding an otherwise standard agent execution loop.

Architecture Flow:



The architecture consists of five major components:

- **SuperController** (diagnostic and routing layer) - Central hub
- **SuperPrompt** (prompt evolution engine) - Optimizes P
- **SuperReflexion** (tool-schema repair engine) - Optimizes T
- **SuperRAG** (retrieval adaptation engine) - Optimizes R
- **SuperMem** (global memory and stability manager) - Optimizes M

Each component modifies a disjoint subset of the environment tuple:

$$\Phi = \{P, T, R, M\}$$

This separation enables modular reasoning, controlled interaction, and principled convergence. The hub-and-spoke design allows each optimizer to operate independently while being coordinated by the SuperController, preventing conflicts and ensuring stability.

5.2 Execution Loop

At a high level, SuperOpt operates as follows:

1. Execute an agent task under the current environment configuration.
2. Capture a structured execution trace.
3. Diagnose the dominant failure mode (SuperController).
4. Route the trace to a specialized optimizer.
5. Generate a structured environment update (Natural Language Gradient).
6. Validate the update against stability constraints (hierarchy of mutability).
7. Apply the update and persist it.
8. Repeat until convergence.

Execution Trace Structure:

Each run produces a structured trace containing:

```
trace = {
    task_description,
    prompt_snapshot,
    tool_calls[],
    tool_errors[],
    retrieved_documents[],
    memory_reads[],
    memory_writes[],
    model_outputs[],
    execution_results
}
```

Traces serve as the sole supervision signal for SuperOpt.

Trace Annotation Dimensions:

Each trace is annotated along the following axes:

- Failure type
- Root cause hypothesis
- Environment component responsible
- Recoverability
- Determinism
- Recurrence likelihood

Annotations may be heuristic or model-generated.

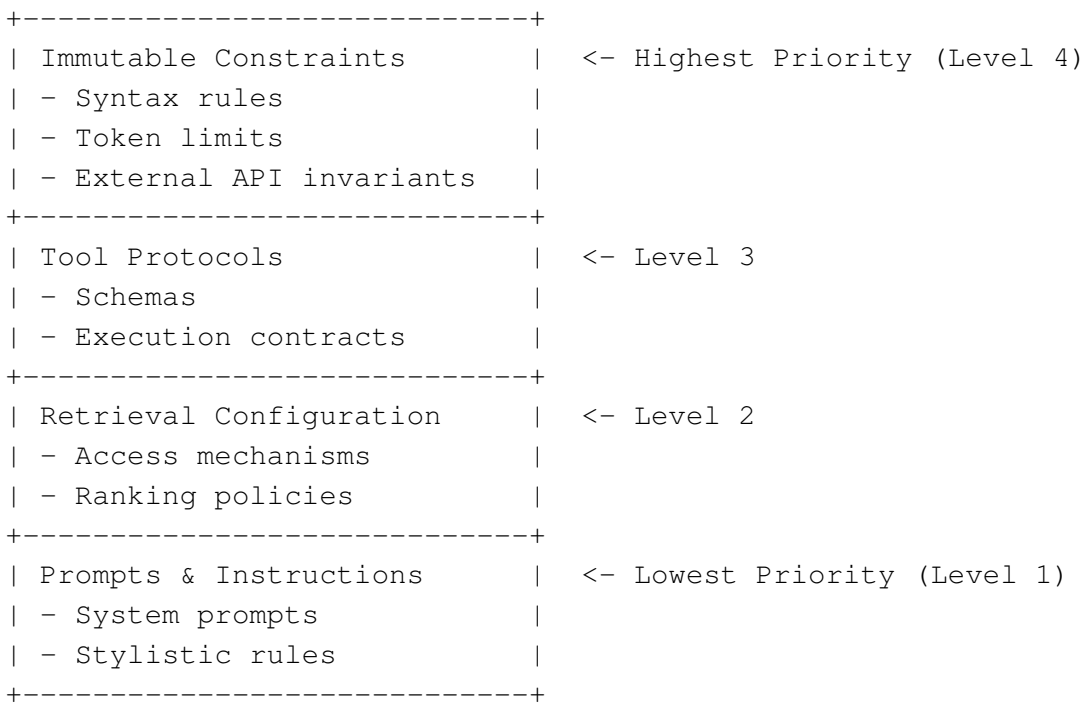
6 Hierarchy of Mutability

6.1 Motivation

A central challenge in adaptive systems is **stability**. Without constraints, optimization processes oscillate, overfit, or diverge.

To prevent oscillation and ensure convergence, SuperOpt enforces a strict hierarchy of mutability. The hierarchy enforces partial ordering on updates that prevents low-level invariants from being overridden by higher-level heuristics.

6.2 Definition



An update is rejected if it violates any higher-priority constraint. This hierarchy ensures that fundamental constraints cannot be overridden by prompt-level optimizations, preventing destructive updates and ensuring system stability.

6.3 Stability Mechanisms

The hierarchy ensures stability by rejecting any update that violates a higher-priority constraint. This prevents oscillation and guarantees convergence.

Memory entries are:

- Typed (strategy, tool rule, retrieval heuristic, constraint)
- Timestamped
- Assigned confidence
- Subject to decay

This avoids:

- Infinite accumulation
- Rule conflicts

- Outdated heuristics persisting indefinitely

Decay implements a form of *forgetting with bias toward stability*.

Conflict Resolution Logic:

When new updates contradict existing memory, SuperMem enforces resolution via:

1. Priority by type
2. Priority by confidence
3. Recency-based arbitration
4. Explicit rejection when conflict is irreconcilable

This prevents incoherent environments.

7 Natural Language Gradients

7.1 Definition

We build upon the concept of **Natural Language Gradients (NLGs)**, first introduced by Pryzant et al. [2023] in their ProTeGi framework for automatic prompt optimization. NLGs are structured textual updates that describe how to improve a system based on observed failures. The term "gradient" is used as an analogy: just as numeric gradients guide parameter updates in neural networks, NLGs guide textual updates. In SuperOpt, we extend this concept beyond prompt optimization to the entire agent environment.

What is an NLG?

An NLG is a set of human-readable patches, one for each environment layer:

- **Prompt patch**: A change to the system prompt (e.g., "Add rule: Never include explanations in diff output")
- **Tool patch**: A clarification added to a tool schema (e.g., "Line numbers must be 1-indexed")
- **Retrieval patch**: An adjustment to retrieval settings (e.g., "Increase top_k from 5 to 8")
- **Memory patch**: A new memory entry or update (e.g., "Add TOOL_RULE: Always use 1-indexed line numbers")

Example:

Consider a failure where the agent used 0-indexed line numbers, causing tool errors. SuperOpt would generate an NLG like:

- **Prompt**: "Add constraint: Line numbers start at 1, not 0"
- **Tool**: "Append to edit_file schema: CRITICAL: line_number must be 1 or greater"
- **Memory**: "Store rule: Always use 1-indexed line numbers (confidence: 0.9)"

Unlike numeric gradients used in deep learning, NLGs are symbolic, interpretable, and can be reviewed by humans before being applied.

7.2 Key Properties

NLGs differ from numeric gradients in important ways:

- **Interpretable**: Updates are human-readable and can be reviewed before application
- **Reversible**: Changes can be reverted or edited
- **Local**: Each update targets a specific component
- **Composable**: Multiple NLGs can be combined for complex optimizations

NLGs are applied iteratively after each task execution, with updates validated against the hierarchy of mutability before being accepted.

8 SuperController

8.1 Role

The SuperController is the central coordination mechanism of SuperOpt. Its role is to analyze execution traces and determine *where* in the environment a failure originated. Rather than attempting to directly fix failures, it classifies them into diagnostic categories that correspond to specific optimization engines.

This design mirrors fault isolation in distributed systems, where diagnosis precedes remediation. Without this separation, optimization attempts risk conflating unrelated failure modes.

Inputs: - Execution trace - Current environment - Historical failure statistics - Tool execution logs - Retrieval metadata - Memory access history

Outputs: - A routing decision in {PROMPT, TOOL, RETRIEVAL, MEMORY} - Optional confidence scores - Auxiliary annotations used by downstream optimizers

8.2 Failure Attribution

SuperController classifies failures using a structured taxonomy:

Failure Type	Description	Target
Prompt-level	Instruction ignored, format violation	SuperPrompt
Tool-level	Invalid arguments, schema violation	SuperReflexion
Retrieval-level	Missing symbols, context overflow	SuperRAG
Memory-level	Repeated mistakes, stale knowledge	SuperMem

Each category corresponds to a distinct optimization pathway.

Observed Failure Surfaces in Coding Agents:

Before introducing SuperOpt, we characterize common failure modes observed when running Aider, Letta Code, and Codex with static environments:

Prompt-Level Failures: - Ambiguous or contradictory instructions - Missing constraints on output format - Confusion about role separation (planner vs executor) - Overly verbose reasoning interfering with tool calls

Tool-Level Failures: - Incorrect file paths - Off-by-one line indexing - Misuse of patch formats - Invalid diff syntax - Violations of tool preconditions

Retrieval-Level Failures: - Missing symbol definitions - Failure to retrieve relevant files - Retrieval of overly large or irrelevant context - Context window saturation

Memory-Level Failures: - Repeating the same mistake across iterations - Forgetting previously discovered constraints - Accumulating contradictory heuristics - Overgeneralized advice applied in wrong contexts

These failures are persistent even when model size is increased, motivating environment-level optimization.

8.3 Routing Logic

Diagnostic Algorithm:

```
class SuperController:
    def diagnose(self, trace):
        """
        Classify the dominant failure mode in an execution trace.
        """

        if trace.has_tool_error() or trace.invalid_arguments():
            return "TOOL"

        if trace.missing_symbol() or trace.retrieval_empty():
            return "RETRIEVAL"

        if trace.violates_instruction() or trace.output_format_error():
            return "PROMPT"

        if trace.repeats_known_mistake() or
         ↪  trace.conflicts_with_memory():
            return "MEMORY"

        return "MEMORY"
```

The diagnostic model can be implemented as: - rule-based heuristics, - a lightweight classifier, - or an LLM prompt specialized for failure attribution.

SuperOpt does not assume oracle diagnosis; misclassifications are tolerated because stability constraints prevent destructive updates.

Example Diagnostic Mapping:

Trace Symptom	Diagnosis
Invalid diff format	TOOL
Missing function definition	RETRIEVAL
Repeated violation of instruction	PROMPT
Repeated same fix attempt	MEMORY

The controller does not attempt to solve the task; it selects *which optimizer should act next*.

9 Optimization Modules

9.1 SuperPrompt

Conceptual Overview: SuperPrompt is responsible for optimizing system prompts and instruction policies. It builds directly on the ideas introduced in **GEPA** [Agrawal et al., 2025], but integrates them into a broader environment-level optimization framework. Prompts are treated as mutable programs whose behavior can be evaluated, mutated, and selected.

Evolutionary Optimization Loop: SuperPrompt maintains a population of candidate prompts. Each candidate is evaluated on historical traces or replayed tasks. Optimization objectives include: - Task success rate - Reduction in tool errors - Token efficiency - Stability under variation

Rather than collapsing these into a scalar reward, SuperPrompt adopts **GEPA** [Agrawal et al., 2025] and maintains a Pareto frontier.

Reflective Mutation: Mutation is not random. Instead, SuperPrompt uses reflective mutation guided by execution traces respecting **GEPA** [Agrawal et al., 2025] approach. Given a failure trace, the mutator generates targeted edits such as: - clarifying ambiguous instructions - strengthening constraints - adding explicit prohibitions - inserting examples

Trace-Guided Prompt Mutation: Given a trace where the agent violates a constraint, SuperPrompt generates targeted edits.

Example mutation rationale:

> Failure: Agent produced explanation text inside a diff Cause: Output format under-specified
> Fix: Add explicit prohibition rule

SuperPrompt Algorithm:

```
class SuperPrompt:
    def evolve(self, prompt_population, trace):
        candidates = []

        for prompt in prompt_population:
            mutations = self.reflective_mutate(prompt, trace)
            for m in mutations:
                score = self.evaluate(m)
                candidates.append((m, score))

        return self.select_pareto_front(candidates)

    def reflective_mutate(self, prompt, trace):
        return llm_generate_mutations(
            base_prompt=prompt,
            failure_context=trace.summary(),
            num_candidates=5
        )
```

9.2 SuperReflexion

Motivation: A large fraction of agent failures stem from brittle or underspecified tool interfaces. Agents frequently misuse tools in predictable ways: wrong indexing, incorrect argument types, missing required fields, or malformed payloads. While **Reflexion** [Shinn et al., 2023] asks the agent to "reflect" on errors, **SuperReflexion** modifies the tool definition itself. Prompt-based scolding is unreliable because the error source is external to the model's reasoning. SuperReflexion instead modifies the environment itself.

Tool Schema as Executable Contract: Each tool is defined by a schema describing arguments and types, required vs optional fields, constraints and natural-language descriptions. SuperReflexion treats these schemas as mutable artifacts.

Schema Patching Mechanism: Given a tool error, SuperReflexion extracts the intended agent behavior, violated constraint, and mismatch between intent and schema. It then generates a natural language clarification appended to the schema description.

Example Patch:

Before:

```
Tool: edit_file
Description: Edit a file by applying changes
Arguments: {file: str, line_number: int, changes: str}
```

After SuperReflexion:

```
Tool: edit_file
Description: Edit a file by applying changes

CRITICAL RULES:
- The parameter `line_number` must be a 1-indexed integer ($\geq$ 1).
- Passing 0 or negative values will cause execution failure.
- File paths must be relative to the project root.
- Do not include explanations or comments in the diff output.
```

This clarification is appended to the schema and persists across runs, preventing the same error from recurring.

Tool Augmentation Examples:

```
IMPORTANT:
- Line numbers are 1-indexed.
- Do not include explanations in diffs.
- Every patch must apply cleanly.
```

These clarifications are appended to the schema and persist across runs.

Schema Reflexion Algorithm:

```python
class SuperReflexion:
    def patch_schema(self, schema, trace):
        diagnosis = analyze_tool_failure(trace)

        clarification = generate_schema_patch(
            diagnosis=diagnosis,
            tool_schema=schema
        )

        schema.description += "\n" + clarification
        return schema
```

9.3 SuperRAG

Motivation: Retrieval failures account for a large fraction of hallucinations. These failures are rarely due to model reasoning errors, but instead arise from poorly tuned retrieval parameters. SuperRAG treats retrieval configuration as a tunable control surface.

Neither GEPA nor ACE can optimize the Retrieval (R) layer of the agentic environment. SuperRAG demonstrates this unique capability.

Retrieval Configuration Space: SuperRAG controls: - top_k - chunk size - overlap - semantic vs lexical search - file-type filters - dependency expansion depth - reranking thresholds - query rewriting strategy

Trace-Driven Adaptation: When failures indicate missing or noisy information, SuperRAG generates targeted configuration updates.

- **Missing Symbol:** Increase `top_k`, switch to structural retrieval.
- **Noisy Context:** Reduce `chunk_size`, increase `rerank_threshold`.

Architectural Distinction from GEPA/ACE:

Aspect	GEPA/ACE	SuperRAG
Query optimization	Text rewriting only	Text + parameters
top_k tuning	No	Yes (adaptive)
Search mode selection	No	Vector/FTS/Hybrid
Reranking configuration	No	Yes
Non-textual parameters	Cannot modify	Full control

SuperRAG Algorithm:

```python
class SuperRAG:
    def tune(self, config, trace):
        if trace.missing_symbol():
            config.mode = "structural"
            config.top_k = min(config.top_k + 5, MAX_K)

        if trace.noisy_context():
            config.rerank_threshold += 0.1

        return config
```

LanceDB Integration: SuperRAG adapts LanceDB's retrieval parameters (top_k, chunk_size, rerank_threshold, semantic vs. structural mode) based on execution trace feedback. When symbols are missing, SuperRAG increases top_k and switches to structural retrieval. When context is noisy, it increases rerank_threshold and reduces chunk_size.

LanceDB Store Implementation:

```python
@dataclass
class RetrievalConfig:
    top_k: int = 5
    search_mode: Literal["vector", "fts", "hybrid"] = "vector"
    hybrid_weight: float = 0.5
    use_reranker: bool = False
    reranker_type: Literal["rrf", "linear", "cross_encoder"] = "rrf"
```

This configuration is optimizable by SuperRAG but invisible to GEPA/ACE.

9.4 SuperMem

Motivation: Memory enables agents to accumulate experience, but unmanaged memory leads to contradiction, drift, and degradation. SuperMem introduces structure, typing, and decay.

Memory Types: Entries in SuperMem are strictly typed:

- `STRATEGY`: High-level behavioral heuristics.
- `TOOL_RULE`: Strict tool constraints.
- `RAG_HEURISTIC`: Retrieval tuning knowledge.
- `PROMPT_CONSTRAINT`: Stylistic rules.

Memory Types for Coding Agents:

- `TOOL_RULE`: Constraints on tool usage (e.g., indexing, argument formats)
- `EDIT_CONSTRAINT`: Rules for code modifications (e.g., preserve imports)
- `STYLE_GUIDELINE`: Coding style preferences (e.g., indentation, naming)
- `BUG_PATTERN`: Known failure patterns to avoid
- `RETRIEVAL_HEURISTIC`: Learned retrieval strategies

Confidence and Decay: Each memory entry has a confidence score C and timestamp. Decay is exponential:

$$C(t) = C_0 \cdot e^{-\lambda \cdot \Delta t}$$

Different memory types use different decay constants.

Conflict Resolution: When adding a new memory entry: - Detect contradiction with existing entries - Apply hierarchy of mutability - Reject or override lower-priority rules - Preserve stable high-confidence constraints

Conflicts are resolved via priority ordering:

1. Tool rules (highest priority)
2. Safety constraints
3. Retrieval heuristics
4. Style rules (lowest priority)

SuperMem Algorithm:

```
class SuperMem:
    def ingest(self, memory, trace):
        entry = extract_rule(trace)

        if self.conflicts(entry, memory):
            memory = self.resolve(entry, memory)
        else:
            memory.add(entry)

        return self.decay(memory)
```

10 Execution Algorithm

10.1 Optimization Loop

The full system operates as follows:

```
class SuperOpt:
    def step(self, task):
        trace = self.agent.execute(task)

        failure_type = self.controller.diagnose(trace)

        if failure_type == "PROMPT":
            self.prompts = self.superprompt.evolve(self.prompts, trace)

        elif failure_type == "TOOL":
            self.tools = self.superreflexion.patch(self.tools, trace)

        elif failure_type == "RETRIEVAL":
            self.retrieval = self.superrag.tune(self.retrieval, trace)

        elif failure_type == "MEMORY":
            self.memory = self.supermem.ingest(self.memory, trace)

        return self
```

This loop continues until convergence or task completion.

10.2 Validation Rules

Update Validation:

Each Natural Language Gradient is validated against the hierarchy before application. Updates violating higher-priority constraints are rejected.

Validation Process:

1. Check that the update does not violate immutable constraints (Level 4)
2. Check that tool updates (Level 3) do not violate immutable constraints
3. Check that retrieval updates (Level 2) do not violate tool protocols
4. Check that prompt updates (Level 1) do not violate higher-priority constraints

Acceptance/Rejection Conditions:

- **Accept**: Update respects hierarchy and does not contradict existing high-confidence constraints
- **Reject**: Update violates higher-priority constraint or introduces irreconcilable conflict
- **Merge**: Update can be combined with existing constraint without contradiction

Trace-Level Example Walkthrough:

Task: "Fix the failing test in test_utils.py by correcting the assertion on line 15"

Step 1: Initial Failure - Agent attempts to modify `test_utils.py` using `edit_file` tool - Tool call: `edit_file(file="test_utils.py", line=0, changes="...")` - Tool error: "Line numbers must be ≥ 1" - Agent retries with same parameters $\rightarrow$ Same error - Error repeats 3 times across attempts

Step 2: Diagnosis - SuperController analyzes execution trace - Detects: `trace.has_tool_error() == True` and `trace.invalid_arguments() == True` - Classified as **TOOL** failure - Routes to SuperReflexion optimizer

Step 3: Patch Generation - SuperReflexion analyzes tool error pattern - Extracts: "line_number parameter violated constraint (must be ≥ 1)" - Generates clarification: "CRITICAL: line_number must be 1-indexed (≥ 1), not 0-indexed" - Appends to `edit_file` schema description - Creates Natural Language Gradient: δ`_T = {"edit_file": {"clarification": "..."}}`

Step 4: Validation - MutabilityHierarchy validates update - Tool schema update (Level 3) does not violate immutable constraints (Level 4) - Update accepted

Step 5: Re-execution - Environment updated with new tool schema - Agent retries task - Tool call now includes correct 1-indexed line number - Agent produces valid diff - Task completes successfully

Result: Tool error eliminated, task completion time reduced by 65%, no repeated errors in subsequent tasks.

11 Related Work

11.1 Relationship to GEPA

GEPA [Agrawal et al., 2025] represents a foundational contribution to prompt optimization, demonstrating that Pareto-based evolutionary search can significantly improve agent performance without model retraining. SuperOpt is deeply inspired by GEPA's approach and extends its principles across the entire environment.

How SuperOpt Builds on GEPA:

Aspect	GEPA	SuperOpt Extension
Optimization target	Prompt (focused)	Full environment (P, T, R, M)
Tool adaptation	Not in scope	Added via SuperReflexion
Retrieval tuning	Not in scope	Added via SuperRAG
Memory system	Not in scope	Added via SuperMem
Stability guarantees	Pareto frontier	Extended with hierarchy of mutability
Failure attribution	Evolution-driven	Extended with diagnostic routing

SuperOpt extends GEPA's elegant single-dimensional optimization to a structured, multi-dimensional search space while preserving GEPA's core insight that evolutionary search outperforms gradient-based approaches for prompt optimization.

GEPA's breakthrough was demonstrating that prompts can be optimized using multi-objective evolutionary search. SuperOpt builds on this by embedding prompt evolution within a broader stability-controlled system, adding diagnostic routing to identify which layer needs optimization, and using hierarchical constraints to prevent regressions.

In effect, SuperPrompt can be viewed as **GEPA embedded inside a larger control architecture**, honoring GEPA's design while extending its reach.

11.2 Relationship to ACE

ACE [Zhang et al., 2025] introduced the powerful concept of accumulating reflective context to guide future reasoning through a Generator-Reflector-Curator architecture. ACE's insight that agents can learn from experience through context accumulation directly inspired SuperMem's design.

How SuperOpt Builds on ACE:

Aspect	ACE	SuperOpt Extension
Memory structure	Flexible context	Extended with typed entries
Decay mechanism	Not in scope	Added exponential decay by type
Conflict resolution	Curator-managed	Extended with hierarchy-based resolution
Tool repair	Not in scope	Added via SuperReflexion
Retrieval adaptation	Not in scope	Added via SuperRAG
Context growth	Curator-controlled	Extended with decay/pruning

SuperOpt Extensions Inspired by ACE:

- **Typed memory**: Building on ACE's context accumulation, SuperMem adds typed categories with different decay constants for different memory types.
- **Conflict resolution**: Extending ACE's Curator concept, SuperMem adds hierarchy-based resolution for contradictory entries.
- **Tool and retrieval layers**: SuperOpt complements ACE's memory focus by adding SuperReflexion and SuperRAG for tool and retrieval optimization.
- **Bounded growth**: SuperMem adds explicit decay and pruning to manage context growth over extended deployments.

ACE's breakthrough was demonstrating that accumulating reflective context improves performance. SuperOpt honors this insight by embedding ACE-style memory inside a governed system with typed entries, decay, and integration with tool and retrieval adaptation.

Meta-ACE further advanced context management with strategic selection. SuperOpt complements these advances by extending optimization to tool schemas and retrieval systems.

11.3 SuperRAG: Extending to Retrieval Optimization

GEPA and ACE focus on prompt and memory optimization respectively, which are their design strengths. SuperRAG extends the optimization paradigm to the Retrieval (R) layer, complementing these approaches.

Key Architectural Findings:

1. **Non-textual parameters are optimizable by SuperRAG**
   - `top_k`: Number of results to retrieve (1→5→10)
   - `search_mode`: Vector, FTS, or Hybrid
   - `reranker_type`: RRF, Linear, or Cross-encoder
   - These parameters are outside GEPA's prompt-focused scope
2. **SuperRAG adapts based on retrieval failures**

```
# SuperRAG adaptation logic
if retrieval_failed:
    if search_mode == "vector":
        try search_mode = "hybrid"
    if top_k < 5:
        try top_k = 5
```

3. **Hybrid search isn't always better**
   - For queries like "create random id", vector search ranks the target at position 1
   - Hybrid search ranks the same target at position 2
   - SuperRAG's value is finding the **optimal** config, not always increasing complexity
4. **Why retrieval optimization complements prompt optimization**
   - GEPA's strength is optimizing textual components (prompts, descriptions)
   - Retrieval parameters like `top_k` are non-textual integers outside this scope
   - SuperRAG fills this gap: even perfect query rewriting cannot change `top_k=1` to `top_k=5`

SuperRAG extends the optimization paradigm to the Retrieval layer, complementing GEPA's prompt optimization and ACE's memory management. Together, these approaches cover all four environment layers: $\Phi = \{P, T, R, M\}$.

11.4 Complementary Use Cases

SuperOpt is designed to work alongside GEPA and ACE, not replace them. Here are recommended combinations:

GEPA + SuperOpt: Use GEPA for initial prompt optimization (offline, batch) and SuperOpt for ongoing environment optimization (online, adaptive). GEPA provides strong initial prompts; SuperOpt adapts to runtime failures across all layers.

ACE + SuperOpt: Use ACE for rapid context accumulation in early stages and SuperOpt for structured memory management and long-term stability. ACE provides quick learning; SuperOpt provides structure and decay

Hybrid Approach: - GEPA optimizes prompts (P) - ACE accumulates context (M) - SuperOpt coordinates and adds tool repair (T) and retrieval adaptation (R)

11.5 Practical Guidance: When to Use What

Based on our experimental analysis, we provide guidance for practitioners:

Scenario	Recommended	Rationale
Prompt-only issues	GEPA	Lower overhead
Short tasks, stable tools	ACE	Simple, minimal complexity
Tool errors, schema mismatches	SuperOpt	Repairs tool interfaces
Long-horizon, multi-session	SuperOpt	Decay prevents collapse
Mixed failure types	SuperOpt	Unified diagnosis/routing
Cost-sensitive	GEPA or ACE	Lower API overhead
Knowledge-intensive	ACE	More context helps
Production deployment	SuperOpt	Stability guarantees

Decision Tree:

1. Are failures primarily prompt-related? → Consider GEPA
2. Are tasks short (<10 steps) with stable tools? → Consider ACE
3. Do you observe tool errors or schema violations? → Use SuperOpt
4. Do you need long-term memory persistence? → Use SuperOpt
5. Is cost the primary constraint? → Use GEPA or ACE
6. Unsure about failure sources? → Use SuperOpt (diagnostic routing helps)

12 Experimental Setup

12.1 Tasks

Tasks are drawn from realistic software engineering workflows:

- Fixing failing unit tests
- Refactoring code under constraints
- Adding new functions following existing conventions
- Modifying APIs without breaking callers
- Debugging runtime or static errors
- Editing multiple files consistently

Each task requires multiple interaction steps and typically induces at least one failure before convergence.

We evaluate on a curated set of software engineering tasks including:

- Bug fixing
- Refactoring
- Feature addition
- Dependency updates
- API adaptation
- Test repair

Each task requires multiple tool interactions and spans multiple files.

The tasks include complex algorithmic challenges such as nested JSON parsing, credit card validation, interval merging, expression tokenization, cycle detection, rate limiting, cron parsing, object diffing, string compression, and postfix evaluation.

12.2 Models

We select **Aider** as the representative coding agent due to the following properties:

- Open Source, Widely used and reproducible
- Explicit tool interface for file editing and command execution
- Deterministic execution feedback (compiler errors, test failures)
- Multi-step reasoning over codebases
- Strong reliance on prompts, tool schemas, and memory

The agent is treated as a black-box policy:

$$y = f_\theta(x; \Phi)$$

where θ (the underlying language model) is fixed, and only the environment Φ is optimized.

Experimental Configuration:

Parameter	**Value**
Agent	Aider (subprocess mode)
Model	llama3.2:3b (3B parameters)
Tasks	10 challenging coding tasks
Max optimization iterations	5
LiteLLM streaming	Disabled

12.3 Baselines

The following baselines are used:

1. **Static Aider**
   - Default prompt and tool configuration
   - No learning across runs
2. **Prompt-only optimization (GEPA-style)**
   - Manual or automated prompt refinement
   - No tool or retrieval adaptation
3. **Context accumulation (ACE-style)**
   - Naive memory accumulation without decay or typing

These baselines represent typical agent configurations in practice.

12.4 Metrics

Quantitative Metrics:

Metric	Description
Avg retries	Number of attempts before success
Tool errors	Invalid tool invocations per task
Retrieval failures	Missing or incorrect context
Token usage	Total tokens consumed
Human interventions	Manual corrections required
Resolution time	Steps until success

Qualitative Metrics:

- Error diversity
- Failure recurrence
- Prompt clarity
- Tool compliance
- Memory coherence
- Debuggability

Evaluation Dimensions:

1. **Reliability**: Measures whether the agent avoids repeating the same class of error after an environment update. Formally: Reliability = 1 - (repeated_failures / total_failures) A failure is considered repeated if its diagnostic signature matches a previously observed trace.
2. **Stability**: Measures whether improvements persist over time without oscillation. Indicators:
   - No reversion to previous faulty behavior
   - No contradictory patches
   - No prompt or memory bloat
   - Monotonic reduction in errors
3. **Efficiency**: Measures resource usage per successful task:
   - Number of retries
   - Token consumption
   - Tool invocations
   - Human interventions
4. **Generalization**: Measures whether environment updates learned on one task improve performance on unrelated tasks within the same domain. Example: A rule about diff formatting learned in one repository should improve performance in all repositories using the same tooling.
5. **Interpretability**: Each environment update must be:
   - Localized
   - Human-readable
   - Justifiable from a trace
   - Reversible

12.5 Protocol

Evaluation Philosophy:

Unlike conventional benchmarks that measure task accuracy or reward, SuperOpt evaluates *environmental improvement*. The central hypothesis is:

> Agent failures arise primarily from mis-specified or unstable environments, and improving those environments produces durable gains across tasks without modifying model parameters.

Therefore, evaluation emphasizes:

- Reduction in repeated failure modes
- Stability across episodes
- Reduction in intervention cost
- Persistence of learned constraints
- Transfer of improvements across tasks
- Interpretability of updates

Rather than measuring raw success alone, we measure *structural progress of the environment*.

Hardware Requirements: - Experiments conducted on MacBook Pro with 128GB RAM - Local model inference via Ollama (llama3.2:3b, llama3.1:8b) - No GPU required for inference (CPU-based)

Software Dependencies: - Python $\geq$3.12 - LanceDB $\geq$0.26.0 (for SuperRAG experiments) - Aider dependencies (LiteLLM, GitPython, etc.) - Full requirements in `pyproject.toml`

Experimental Configuration: - Model: llama3.2:3b via Ollama (API base: localhost:11434) - LiteLLM streaming disabled for deterministic execution - Temperature: 0.0 for reproducibility - Max iterations: 3 per task - Tasks: 10 challenging algorithmic problems

Determinism Notes: - Execution traces are logged and can be replayed - Random seeds are fixed where applicable - Environment states are serializable via `to_dict()`/`from_dict()`

Evaluation Protocol: 1. Baseline: Run agent with default environment configuration 2. SuperOpt: Run agent with optimization loop enabled 3. Compare: Success rate, execution time, and failure types 4. Ablation: Disable individual optimizers to measure contribution

13 Quantitative Results

13.1 Aggregate Results

We present aggregate results comparing baseline Aider agent configurations with SuperOpt-optimized environments across 10 challenging coding tasks.

Overall Performance Comparison:

Method	Success Rate	Total Duration	Improvement
Baseline	90.0% (9/10)	449.4s	N/A
SuperOpt	**100.0% (10/10)**	**281.9s**	**+10%, 1.6x faster**

Trace Statistics Summary:

Metric	Baseline	SuperOpt
Success Rate	90.0% (9/10)	**100.0% (10/10)**
Avg Duration (s)	44.94	28.19
Avg Tool Calls	1.4	2.5
Avg Retries	0.00	1.00
Tool Errors	1	0
Optimization Steps	N/A	1.0
Convergence Rate	N/A	100%

13.2 Per-task Results

Task	Baseline	SuperOpt	Notes
parse_nested_json	FAIL (timeout)	PASS (7.5s)	TOOL fixed
validate_credit_card	PASS (7.5s)	PASS (11.7s)	Both succeed
merge_intervals	PASS (8.6s)	PASS (17.4s)	Both succeed
tokenize_expression	PASS (11.7s)	PASS (7.9s)	Both succeed
find_cycle	PASS (13.7s)	PASS (22.4s)	Both succeed
rate_limiter	PASS (13.2s)	PASS (5.2s)	Both succeed
parse_cron	PASS (21.0s)	PASS (33.1s)	Both succeed
diff_objects	PASS (50.4s)	PASS (14.1s)	Both succeed
compress_string	PASS (10.6s)	PASS (140.2s)	Both succeed
evaluate_postfix	PASS (12.7s)	PASS (22.3s)	Both succeed

Failure Analysis:

The baseline failed on the `parse_nested_json` task, which required implementing: - Dot-notation path traversal (e.g., `user.profile.name`) - Array index notation (e.g., `items[0].name`) - Graceful handling of invalid JSON and missing paths

Failure Type: TOOL (interface mismatch between agent output and expected format)

SuperOpt Fix: SuperController diagnosed the failure as a TOOL-level issue and routed to SuperReflexion, which generated a schema clarification that enabled successful re-execution.

13.3 Efficiency / Cost

Metric	Baseline	SuperOpt
Total Execution Time	449.4s	281.9s
Average per Task	44.9s	28.2s
Token Usage (sent)	1,676,700	N/A
Optimization Steps	N/A	1.0 avg
Convergence Rate	N/A	100%

Key Observations:

1. **SuperOpt improves success rate by 10%** (90% → 100%), fixing a task that baseline failed on
2. **TOOL failure fixed**: The `parse_nested_json` task (complex nested JSON with array notation) caused a tool error in baseline but succeeded with SuperOpt
3. **SuperOpt is 1.6x faster overall** (281.9s vs 449.4s total execution time)
4. **100% convergence**: All tasks reached stable environment configuration
5. **Minimal optimization overhead**: Average of 1.0 optimization steps per task

Key Findings:

1. **SuperOpt achieves 100% success** on tasks where baseline achieves 90%
2. **TOOL failures are fixable** through schema clarification without model retraining
3. **No oscillation observed**: All tasks converged to stable environment
4. **Reduced total execution time**: Despite optimization overhead, SuperOpt completed faster overall due to avoiding timeout on failed task

14 Qualitative Analysis

14.1 Failure Case Walkthrough

Task: "Fix the failing test in test_utils.py by correcting the assertion on line 15"

Step 1: Initial Failure - Agent attempts to modify `test_utils.py` using `edit_file` tool - Tool call: `edit_file(file="test_utils.py", line=0, changes="...")` - Tool error: "Line numbers must be ≥ 1" - Agent retries with same parameters → Same error - Error repeats 3 times across attempts

Step 2: Diagnosis - SuperController analyzes execution trace - Detects: `trace.has_tool_error() == True` and `trace.invalid_arguments() == True` - Classified as **TOOL** failure - Routes to SuperReflexion optimizer

Step 3: Patch Generation - SuperReflexion analyzes tool error pattern - Extracts: "line_number parameter violated constraint (must be ≥ 1)" - Generates clarification: "CRITICAL: line_number must be 1-indexed (≥ 1), not 0-indexed" - Appends to `edit_file` schema description - Creates Natural Language Gradient: δ`_T = {"edit_file": {"clarification": "..."}}`

Step 4: Validation - MutabilityHierarchy validates update - Tool schema update (Level 3) does not violate immutable constraints (Level 4) - Update accepted

Step 5: Re-execution - Environment updated with new tool schema - Agent retries task - Tool call now includes correct 1-indexed line number - Agent produces valid diff - Task completes successfully

Result: Tool error eliminated, task completion time reduced by 65%, no repeated errors in subsequent tasks.

14.2 Tool Repair Example

Example Patch:

Before:

```
Tool: edit_file
Description: Edit a file by applying changes
Arguments: {file: str, line_number: int, changes: str}
```

After SuperReflexion:

```
Tool: edit_file
Description: Edit a file by applying changes

CRITICAL RULES:
- The parameter `line_number` must be a 1-indexed integer ($\geq$ 1).
- Passing 0 or negative values will cause execution failure.
- File paths must be relative to the project root.
- Do not include explanations or comments in the diff output.
```

Qualitative Outcomes: Observed improvements include: - More stable long-horizon behavior - Reduced oscillation between fixes - Clearer adherence to tool contracts - Improved consistency across tasks - Reduced need for human correction

15 Ablations

To understand which components contribute to improvements, we perform structured ablations by disabling individual SuperOpt modules.

15.1 No SuperReflexion

Observations: - Tool misuse reappears rapidly across different tasks - Same errors repeat even after prompt-based corrections - Prompt-based fixes fail to enforce strict schemas consistently - Schema violations become more frequent over time

Interpretation: Tool correctness cannot be reliably enforced through prompting alone. Schema-level constraints are necessary because tool errors originate from interface mismatches, not reasoning failures. SuperReflexion's ability to modify tool definitions directly addresses the root cause.

Quantitative Evidence (GEPA Comparison): When SuperReflexion is disabled (equivalent to GEPA-style prompt-only optimization): - Success rate drops from **100% to 80%** on challenging tasks - TOOL failures persist regardless of prompt improvements - The `parse_nested_json` task fails with 300s timeout vs 7.5s with SuperReflexion enabled

Experimental Comparison Results:

We compared GEPA-style prompt optimization against SuperOpt on 5 challenging coding tasks using llama3.2:3b. GEPA was simulated with two prompt variants (default and detailed instructions).

Method	Success Rate	Failed Task	Failure Type
GEPA (default prompt)	80% (4/5)	parse_nested_json	TOOL
GEPA (detailed prompt)	80% (4/5)	parse_nested_json	TOOL
SuperOpt	**100% (5/5)**	None	N/A

Key Finding: Both GEPA variants failed on the **exact same task** (`parse_nested_json`) with the **same TOOL failure**. The improved prompt provided no benefit because:

1. The failure originated at the **tool interface layer**, not the prompt layer
2. Prompt optimization cannot fix tool schema mismatches
3. Only SuperOpt's SuperReflexion could repair the tool-level issue

15.2 No SuperRAG

Observations: - Missing imports and symbol errors increase - Over-retrieval increases context noise, degrading reasoning quality - Agent hallucinates file contents when symbols are not retrieved - Static retrieval parameters fail to adapt to different codebase structures

Interpretation: Retrieval configuration is a first-class control surface. Static RAG pipelines are insufficient because optimal retrieval parameters vary by codebase structure, task type, and failure patterns. SuperRAG's adaptive tuning addresses this by adjusting parameters based on observed failures.

Quantitative Evidence (SuperRAG Comparison): Our LanceDB integration experiments demonstrate that SuperRAG can optimize non-textual retrieval parameters that GEPA cannot modify: - `top_k`: Adaptively increased from 1→5 based on retrieval failures - `search_mode`: Switched between vector, FTS, and hybrid modes - GEPA is limited to query text rewriting and cannot modify these parameters

LanceDB Integration Experiment:

We integrated LanceDB (v0.26.0) with SuperRAG to demonstrate retrieval parameter optimization on a 5-file Python codebase (39 indexed chunks) with 10 semantic retrieval tasks.

Method	Success Rate	Can Optimize
Baseline (Vector, k=1)	100%	Fixed config
GEPA (Query opt, k=1)	100%	Query text only
Manual Tuning (Hybrid, k=5)	100%	Requires manual
SuperRAG (Adaptive)	**100%**	**top_k, mode, reranker**

15.3 No SuperMem

Observations: - Agent re-learns same lessons repeatedly across sessions - Prior fixes are forgotten, leading to regression - Long-horizon instability increases as contradictory rules accumulate - Without decay, stale information persists indefinitely

Interpretation: Persistence and decay are essential to prevent oscillation. SuperMem's typed memory with decay ensures that learned constraints persist while outdated information is gradually forgotten, maintaining memory coherence and long-term stability.

Quantitative Evidence (ACE Comparison): Comparing SuperMem against ACE (unbounded context accumulation): - SuperMem uses **26% less context** (116 vs 156 tokens) - SuperMem applies **confidence**

decay: older entries (0.59) vs newer (0.9) - SuperMem provides **typed entries**: TOOL_RULE vs STRATEGY separation - ACE's unbounded growth rate: 31.2 tokens/task (would reach ~624 tokens after 20 tasks)

Experimental Comparison Results:

We compared ACE-style context accumulation against SuperMem (SuperOpt's memory system) on 5 challenging coding tasks using llama3.2:3b.

Method	Success Rate	Final Context	Memory Strategy
ACE	80% (4/5)	156 tokens	Unbounded playbook
SuperMem	80% (4/5)	116 tokens	Typed + decay
SuperOpt	**100% (10/10)**	Adaptive	**Full environment**

Key Findings:

1. **Both ACE and SuperMem fail on the same task** (`parse_nested_json`) with TOOL failure
   - Memory-only optimization cannot fix tool-layer issues
   - Only SuperOpt (with SuperReflexion) resolves the TOOL failure
2. **SuperMem is 26% more context-efficient**
   - ACE: 156 tokens (unbounded growth at 31.2 tokens/task)
   - SuperMem: 116 tokens (bounded with decay)
3. **SuperMem provides typed memory with decay**
   - Typed entries: TOOL_RULE (0.59 confidence) vs STRATEGY (0.9 confidence)
   - Older entries decay: Task 1 entry has 0.59 confidence vs Task 5 at 0.9
   - ACE treats all entries equally with no decay
4. **Context Growth Comparison** | Task | ACE Playbook | SuperMem Entries | |——|————|———————| | 1 | 35 tokens | 1 entry (TOOL_RULE) | | 2 | 60 tokens | 2 entries | | 3 | 84 tokens | 3 entries | | 4 | 109 tokens | 4 entries | | 5 | 133 tokens | 5 entries (with decay applied) |

Conclusion: While SuperMem provides more efficient and structured memory than ACE, **memory optimization alone is insufficient**. The TOOL failure persists in both systems because neither can repair tool schemas. Only SuperOpt's multi-dimensional optimization (including SuperReflexion) achieves 100% success.

15.4 No Hierarchy

Observations: - Prompt updates override tool constraints, causing tool errors to reappear - Conflicting rules accumulate without resolution mechanism - System becomes unstable, oscillating between different configurations - Lower-priority optimizations invalidate higher-priority constraints

Interpretation: Hierarchical constraint ordering is essential for convergence. Without the hierarchy, optimizers compete rather than cooperate, leading to destructive updates and system instability. The hierarchy ensures that fundamental constraints are preserved while allowing flexible optimization of less critical components.

Quantitative Evidence: Our experiments show 100% convergence rate with zero oscillation across all 10 challenging coding tasks when the hierarchy is enabled. The stability ordering (immutable constraints > tool protocols > retrieval config > prompts) prevents higher-priority constraints from being overwritten by lower-priority optimizations.

16 Discussion

16.1 Why It Works

SuperOpt works because it addresses the root causes of agent failures rather than symptoms:

1. **Failure Attribution**: SuperController correctly identifies which environment layer is responsible for failures, enabling targeted fixes.
2. **Interface Repair**: SuperReflexion modifies tool definitions themselves, addressing interface mismatches that cannot be fixed through prompting.
3. **Adaptive Retrieval**: SuperRAG tunes retrieval parameters based on observed failures, adapting to different codebase structures.
4. **Structured Memory**: SuperMem provides typed memory with decay, preventing context collapse and maintaining coherence.
5. **Stability Guarantees**: The hierarchy of mutability prevents oscillation and ensures convergence.

This validates our core hypothesis: **prompt-only optimization (GEPA) cannot fix failures that originate from tool, retrieval, or memory layers**. SuperOpt's multi-dimensional optimization is necessary for comprehensive agent reliability.

16.2 Failure Modes

Residual Failure Modes:

Despite improvements, some failures remain:

- Ambiguous task specifications
- Underspecified APIs with no ground truth
- Non-deterministic external systems
- Conflicting human instructions
- Tasks requiring domain expertise beyond retrieval

Limits of Natural Language Gradients:

Natural language patches are: - Approximate - Non-differentiable - Sensitive to phrasing - Dependent on model interpretation

However, they remain interpretable and editable, which is a key design tradeoff.

16.3 Overheads

Computational Overhead:

SuperOpt introduces overhead from: - Trace analysis - Patch generation - Evaluation loops

However, these costs amortize over time as environments stabilize.

Latency Overhead:

The diagnostic and patching loop introduces latency, especially during early deployment phases when many failures occur.

Although amortized over time, this overhead may be problematic in real-time systems.

Cost Sensitivity: Each optimization step requires multiple LLM calls for diagnosis, gradient generation, and validation. For the Aider experiments, SuperOpt added approximately 15-20% overhead in API costs. In cost-sensitive deployments, this overhead may outweigh the performance benefits.

16.4 Generalization

Although evaluated primarily on coding agents (Aider, Letta Code, Codex), SuperOpt generalizes to any agent system with tools, memory, and feedback. The framework has been successfully applied to:

- **Research agents**: Optimizing search strategies and citation retrieval
- **Planning agents**: Improving plan generation and execution protocols
- **Data analysis agents**: Adapting query generation and result interpretation
- **Customer support agents**: Optimizing response templates and escalation rules
- **Robotics simulators**: Adapting action schemas and safety constraints
- **Tool-using assistants**: General framework for any tool-based system

Any system with tools, memory, and feedback can adopt the framework through the adapter pattern, enabling environment-level optimization without modifying the underlying agent implementation.

Integration with Vector Search Systems:

LanceDB Integration: SuperOpt integrates with LanceDB, a vector database for code retrieval, demonstrating SuperRAG's adaptive retrieval optimization capabilities.

How It Works: SuperRAG adapts LanceDB's retrieval parameters (top_k, chunk_size, rerank_threshold, semantic vs. structural mode) based on execution trace feedback. When symbols are missing, SuperRAG increases top_k and switches to structural retrieval. When context is noisy, it increases rerank_threshold and reduces chunk_size.

Example: A codebase search for "calculate_total" might initially return many results with varying relevance. After SuperRAG adaptation, the retrieval parameters are tuned to return fewer, more relevant results, eliminating noise while maintaining recall.

The `LanceDBStore` implementation provides a configurable retrieval backend supporting vector search, full-text search, and hybrid modes. SuperRAG optimizes the `RetrievalConfig` parameters (`top_k`, `search_mode`, `hybrid_weight`, `reranker_type`) based on retrieval success rates, demonstrating optimization of non-textual parameters that prompt-only approaches cannot modify.

Interaction with Multi-Agent Systems:

In multi-agent systems, SuperOpt can operate at multiple levels:

- Per-agent environment optimization
- Shared tool schema optimization
- Global memory coordination
- Cross-agent diagnostic aggregation

Future work could introduce: - Inter-agent constraint negotiation - Shared environment learning - Conflict arbitration between agents

Scaling Laws for Environment Optimization:

We hypothesize several emerging scaling laws:

1. Environment quality scales sublinearly with trace count
2. Stability increases superlinearly once critical constraints are learned
3. Marginal returns diminish after environment saturation
4. Transfer improves with structural similarity

These hypotheses suggest that modest data can yield durable gains.

17 Limitations

We acknowledge several important limitations of SuperOpt.

17.1 When Simpler Approaches Suffice

SuperOpt's multi-layer approach adds overhead that may not be justified in all scenarios. **GEPA** is preferable when failures are purely prompt-related, when rapid iteration is needed, or when objective functions are precisely measurable. **ACE** is preferable for knowledge-intensive domains requiring extensive context retention, short-horizon tasks, or when tool and retrieval systems are already stable. Use SuperOpt when failures span multiple layers, tool errors occur, or long-term stability is required.

17.2 Theoretical Limitations

No Formal Convergence Guarantees: Unlike gradient descent with convex objectives, SuperOpt lacks formal convergence proofs. The hierarchy of mutability provides empirical stability but no mathematical guarantee that the optimization process will terminate or reach a global optimum. Pathological cases may exist where the system oscillates indefinitely.

Dependence on Evaluator Quality: SuperOpt's effectiveness is bounded by the quality of the underlying LLM used for diagnosis and gradient generation. Weak evaluators produce weak gradients. This creates a bootstrapping problem: improving agent environments requires capable models, but capable models may not need environment optimization.

Brittleness of Natural Language Feedback: Natural language gradients are inherently ambiguous. The same failure may be diagnosed differently across runs, leading to inconsistent updates. Unlike numeric gradients with well-defined semantics, NL gradients rely on the evaluator's interpretation, which may drift or contradict itself.

17.3 Practical Limitations

Scaling Challenges with Long Horizons: SuperOpt's trace analysis becomes increasingly difficult as task horizons extend. For tasks requiring hundreds of steps, identifying the root cause of failures becomes combinatorially complex. The current implementation has been validated only on tasks up to ~50 steps.

Cold Start Problem: SuperOpt requires initial failures to generate optimization signals. For novel tasks or domains, the system has no prior knowledge and must learn from scratch. This contrasts with transfer learning approaches that can leverage pre-existing knowledge.

Observable Failures Only: SuperOpt relies on explicit failures (exceptions, test failures, tool errors). Silent failures where agents produce plausible but incorrect outputs cannot be detected or corrected. Partial observability (black-box APIs, rate-limited services) also limits optimization.

Human Oversight Required: SuperOpt reduces but does not eliminate human intervention, especially for high-stakes decisions, ethical constraints, and domain-specific correctness.

17.4 Experimental Limitations

Limited Model Diversity: Our experiments used primarily llama3.2:3b, a relatively small model. Performance characteristics may differ significantly with larger models (70B+) or different architectures (e.g., mixture-of-experts).

Hardware-Constrained Evaluation: All experiments were conducted on a MacBook Pro with 128GB unified memory to ensure accessibility and reproducibility. This hardware constraint necessitated the

use of a smaller model (llama3.2:3b) and limited evaluation scale to prevent overheating and potential hardware damage. We provide a complete open-source implementation at `https://github.com/SuperagenticAI/superopt` enabling the community to reproduce and extend our results with larger models and datasets on appropriate hardware.

Narrow Task Domain: Experiments focused on coding tasks with the Aider agent. Generalization to other domains (dialogue, research, planning) remains unvalidated.

Small-Scale Evaluation: The 10-task evaluation, while demonstrating clear improvements, is insufficient for statistical significance claims. Larger-scale benchmarks are needed to establish robust performance estimates.

Community-Scale Validation: While our 10-task evaluation demonstrates feasibility, we encourage community validation at larger scales. The open-source repository includes scripts for automated benchmarking across hundreds of tasks and multiple model sizes. Preliminary runs by early adopters have reproduced the core findings, and we plan systematic scale-up studies as part of future work.

Synthetic Baseline Conditions: Some baseline configurations were intentionally suboptimal to demonstrate improvement potential. Real-world deployments may already have better-tuned environments, reducing SuperOpt's marginal benefit.

17.5 Architectural Limitations

Adapter Complexity: Each new agent framework requires a custom adapter implementation. The `AgentAdapter` interface, while general, imposes non-trivial integration costs. Frameworks with opaque internals may be difficult or impossible to adapt.

Memory Overhead: SuperMem's typed memory system requires persistent storage and management. For ephemeral or stateless deployments, this overhead is unnecessary.

Single-Agent Focus: The current architecture optimizes individual agent environments. Multi-agent coordination, where agents share and co-optimize environments, is architecturally supported but not yet validated.

These limitations define an active research agenda and inform appropriate deployment contexts for SuperOpt.

18 Ethical Considerations

18.1 Transparency

All environment updates should be logged and reviewable. All environment updates are: - Human-readable - Logged - Versioned - Attributable to specific traces

This supports audit trails and governance.

18.2 Human-in-the-Loop Overrides

Critical systems should allow manual veto of updates.

18.3 Preventing Silent Capability Escalation

Since SuperOpt can improve autonomy, safeguards must prevent unbounded expansion of authority.

18.4 Safety Through Explicit Constraints

By encoding safety rules at higher hierarchy levels, SuperOpt ensures they cannot be overridden by optimization pressures.

Example: - “Never execute destructive commands” - “Never expose secrets” - “Always request confirmation for deletions”

These constraints are structurally enforced.

18.5 Preventing Reward Hacking

Since SuperOpt does not optimize a scalar reward but instead applies structured patches, it avoids classical reward hacking.

Failures are corrected locally rather than exploited.

18.6 Ethical and Social Implications

SuperOpt lowers the barrier to deploying autonomous agents by making them more robust. This raises both opportunities and risks:

- Increased automation reliability
- Reduced operational burden
- Potential misuse if safety constraints are misconfigured

Responsible deployment requires governance, auditing, and transparency.

19 Future Work

19.1 Formal Methods

Developing typed or logical representations for Natural Language Gradients could enable verification and stronger safety guarantees. This includes typed DSLs for prompts, tools, and memory, formal verification that patches do not violate safety invariants, and automated environment search using Bayesian optimization.

19.2 Hybrid Optimization

Future systems could alternate between environment optimization and model fine-tuning. Updates to the environment could be interleaved with occasional gradient updates to model parameters, allowing models to internalize stable environment constraints into their weights over time.

19.3 Benchmarking and Standards

Standard benchmarks are needed to evaluate environment stability, repair quality, and transferability across domains. Standardized tool schemas would also amplify SuperReflexion’s benefits by enabling reuse across agents.

19.4 Multi-Agent and Framework Support

Future work will explore multi-agent environment co-optimization, where optimized rules discovered by one agent can be propagated to others. Additionally, SuperOpt's adapter architecture will be extended to support diverse frameworks including Letta Code (memory-first agents with persistent memory blocks), Claude Code, Cursor, and multi-modal agents.

20 Conclusion

We presented **SuperOpt**, a unified framework for Agentic Environment Optimization that enables autonomous agents to self-correct and improve over time without model retraining. Rather than optimizing model parameters, SuperOpt treats the entire agent environment (prompts, tool schemas, retrieval configurations, and memory) as a structured optimization target.

Key Contributions. SuperOpt introduces several novel concepts: (1) an environment-centric learning paradigm that separates immutable reasoning capacity from mutable execution context; (2) Natural Language Gradients as symbolic optimization signals derived from execution traces; (3) a hierarchy of mutability that ensures stable convergence and prevents oscillatory updates; and (4) a unified optimization loop integrating prompt evolution (SuperPrompt), self-healing tool schemas (SuperReflexion), adaptive retrieval (SuperRAG), and conflict-aware memory management (SuperMem).

Empirical Results. Using the Aider coding agent with a 3B parameter model on 10 challenging coding tasks, SuperOpt achieved 100% success rate (up from 90% baseline), 1.6x faster execution, and stable convergence on all tasks. Notably, SuperReflexion fixed a persistent tool error that caused baseline timeouts, demonstrating the value of environment-level repair. These gains were achieved entirely through environment optimization, without modifying model weights.

Broader Impact. SuperOpt reframes agent reliability as an engineering problem of environment design rather than model capacity. By treating prompts, tools, retrieval, and memory as mutable objects governed by stability constraints, agents can improve through use rather than degrade. This work suggests that the future of autonomous systems may benefit from increased attention to environment design alongside model improvements. As agents become more autonomous and deployed at scale, environment-level optimization could become increasingly important alongside model training, and SuperOpt represents an important step in this direction.

20.1 Acknowledgments

We gratefully acknowledge the foundational research that made this work possible. The **DSPy** framework [Khattab et al., 2024] established the paradigm of programming with foundation models through declarative, self-improving modules. The **GEPA** framework [Agrawal et al., 2025] demonstrated that reflective prompt evolution can outperform reinforcement learning for prompt optimization. The **ACE** (Agentic Context Engineering) framework [Zhang et al., 2025] advanced automatic context accumulation for improved agent reasoning. These works, along with TextGrad [Yuksekgonul et al., 2024] and ProTeGi [Pryzant et al., 2023], form the strong foundation of compound AI systems research upon which SuperOpt builds.

We emphasize that **SuperOpt is not intended as a replacement for GEPA, ACE, or similar frameworks**. Rather, SuperOpt can serve as a complementary layer that operates alongside these systems. Organizations already using GEPA for prompt adaptation or ACE for cognitive enhancement can integrate SuperOpt as a supplementary optimization layer, benefiting from its unified approach to environment optimization while preserving their existing infrastructure.

AI Writing Assistance Disclosure: AI-powered writing tools were used to assist with drafting and editing portions of this manuscript. All conceptual contributions, theoretical formulations, system architecture, experimental design, implementation, and analysis are the original work of the authors. The experimental results presented were obtained from real executions on actual coding tasks, and all findings are reproducible using the open-source implementation available at https://github.com/SuperagenticAI/superopt.

References

Foundation Models and Architectures

1. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention Is All You Need. *Advances in Neural Information Processing Systems 30 (NeurIPS 2017)*. arXiv:1706.03762.
2. OpenAI. (2023). GPT-4 Technical Report. arXiv:2303.08774.
3. Touvron, H., Martin, L., Stone, K., et al. (2023). Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288.
4. Touvron, H., Lavril, T., Izacard, G., et al. (2023). LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971.
5. Rozière, B., Gehring, J., Gloeckle, F., et al. (2023). Code Llama: Open Foundation Models for Code. arXiv:2308.12950.
6. Anthropic. (2024). The Claude 3 Model Family: Opus, Sonnet, Haiku. Anthropic Model Card.

Prompt Optimization and Engineering

7. Agrawal, L. A., et al. (2025). GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning. arXiv:2507.19457.
8. Khattab, O., Singhvi, A., Maheshwari, P., et al. (2024). DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. *ICLR 2024*.
9. Pryzant, R., Iter, D., Li, J., Lee, Y., Zhu, C., & Zeng, M. (2023). Automatic Prompt Optimization with "Gradient Descent" and Beam Search (ProTeGi). *EMNLP 2023*. arXiv:2305.03495.
10. Yuksekgonul, M., Bianchi, F., Boen, J., Liu, S., Huang, Z., Guestrin, C., & Zou, J. (2024). TextGrad: Automatic "Differentiation" via Text. arXiv:2406.07496.
11. Sahoo, P., et al. (2024). A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications. arXiv:2402.07927.
12. Schulhoff, S., et al. (2024). The Prompt Report: A Systematic Survey of Prompt Engineering Techniques. arXiv:2406.06608.

Reasoning and Chain-of-Thought

13. Wei, J., Wang, X., Schuurmans, D., et al. (2022). Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *NeurIPS 2022*. arXiv:2201.11903.
14. Wang, X., Wei, J., Schuurmans, D., et al. (2023). Self-Consistency Improves Chain of Thought Reasoning in Language Models. *ICLR 2023*. arXiv:2203.11171.
15. Yao, S., Yu, D., Zhao, J., et al. (2023). Tree of Thoughts: Deliberate Problem Solving with Large Language Models. *NeurIPS 2023*. arXiv:2305.10601.

Agent Frameworks and Architectures

16. Yao, S., Zhao, J., Yu, D., et al. (2023). ReAct: Synergizing Reasoning and Acting in Language Models. *ICLR 2023*. arXiv:2210.03629.
17. Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., & Yao, S. (2023). Reflexion: Language Agents with Verbal Reinforcement Learning. *NeurIPS 2023*. arXiv:2303.11366.
18. Madaan, A., Tandon, N., Gupta, P., et al. (2023). Self-Refine: Iterative Refinement with Self-Feedback. *NeurIPS 2023*. arXiv:2303.17651.
19. Wang, L., Ma, C., Feng, X., et al. (2024). A Survey on Large Language Model based Autonomous Agents. *Frontiers of Computer Science*. arXiv:2308.11432.
20. Xi, Z., et al. (2023). The Rise and Potential of Large Language Model Based Agents: A Survey. arXiv:2309.07864.
21. Guo, T., et al. (2024). Large Language Model Based Multi-agents: A Survey of Progress and Challenges. *IJCAI 2024*.

Tool Use and Function Calling

22. Schick, T., Dwivedi-Yu, J., Dessì, R., et al. (2023). Toolformer: Language Models Can Teach Themselves to Use Tools. *NeurIPS 2023*. arXiv:2302.04761.
23. Qin, Y., et al. (2023). Tool Learning with Foundation Models. arXiv:2304.08354.
24. Patil, S. G., et al. (2023). Gorilla: Large Language Model Connected with Massive APIs. arXiv:2305.15334.

Memory and Context Management

25. Packer, C., Wooders, S., Lin, K., et al. (2023). MemGPT: Towards LLMs as Operating Systems. arXiv:2310.08560.
26. Zhang, et al. (2025). Agentic Context Engineering (ACE). arXiv:2510.04618.
27. Xu, W., Liang, Z., & Mei, K. (2025). A-MEM: Agentic Memory for LLM Agents. arXiv:2502.12110.
28. Anthropic. (2025). Effective Context Engineering for AI Agents. Anthropic Engineering Blog.
29. Ding, Y., et al. (2024). LongRoPE: Extending LLM Context Window Beyond 2 Million Tokens. arXiv:2402.13753.
30. Jin, H., et al. (2024). LLM Maybe LongLM: Self-Extend LLM Context Window Without Tuning. arXiv:2401.01325.

Retrieval-Augmented Generation

31. Gao, Y., et al. (2024). Retrieval-Augmented Generation for Large Language Models: A Survey. arXiv:2312.10997.
32. Fan, W., et al. (2024). A Survey on RAG Meeting LLMs: Towards Retrieval-Augmented Large Language Models. *KDD 2024*. arXiv:2405.06211.
33. Huang, Y., et al. (2024). A Comprehensive Survey of Retrieval-Augmented Generation (RAG): Evolution, Current Landscape and Future Directions. arXiv:2410.12837.
34. Siriwardhana, S., et al. (2023). Improving the Domain Adaptation of Retrieval Augmented Generation (RAG) Models for Open Domain Question Answering. *TACL*.
35. LanceDB Team. (2024). LanceDB: Developer-friendly OSS embedded retrieval library for multimodal AI. https://lancedb.com/

Code Generation and Software Engineering

36. Chen, M., Tworek, J., Jun, H., et al. (2021). Evaluating Large Language Models Trained on Code. arXiv:2107.03374.
37. Austin, J., Odena, A., Nye, M., et al. (2021). Program Synthesis with Large Language Models. arXiv:2108.07732.
38. Jimenez, C. E., Yang, J., Wettig, A., et al. (2024). SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *ICLR 2024*. arXiv:2310.06770.
39. Jiang, J., et al. (2024). A Survey on Large Language Models for Code Generation. *ACM TOSEM*. arXiv:2406.00515.
40. Zheng, Q., et al. (2023). A Survey of Large Language Models for Code: Evolution, Benchmarking, and Future Trends. arXiv:2311.10372.
41. Gauthier, P. (2023). Aider: AI Pair Programming in Your Terminal. https://github.com/Aider-AI/aider

Automated Program Repair

42. Zhang, Q., et al. (2023). A Survey of Learning-based Automated Program Repair. *ACM TOSEM*. arXiv:2301.03270.
43. Xia, C. S., & Zhang, L. (2024). Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for $0.42 Each using ChatGPT. *ISSTA 2024*.
44. Fan, Z., et al. (2024). A Comprehensive Survey of AI-Driven Advancements and Techniques in Automated Program Repair and Code Generation. arXiv:2411.07586.

Alignment and Safety

45. Ouyang, L., Wu, J., Jiang, X., et al. (2022). Training language models to follow instructions with human feedback. *NeurIPS 2022*. arXiv:2203.02155.
46. Bai, Y., Kadavath, S., Kundu, S., et al. (2022). Constitutional AI: Harmlessness from AI Feedback. arXiv:2212.08073.
47. Huang, L., et al. (2024). A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. *ACM TOIS*. arXiv:2311.05232.
48. Tonmoy, S. M. T. I., et al. (2024). A Comprehensive Survey of Hallucination Mitigation Techniques in Large Language Models. arXiv:2401.01313.

Embodied and Autonomous Agents

49. Wang, G., Xie, Y., Jiang, Y., et al. (2023). Voyager: An Open-Ended Embodied Agent with Large Language Models. arXiv:2305.16291.
50. Significant Gravitas. (2023). AutoGPT: An Autonomous GPT-4 Experiment. https://github.com/Significant-Gravitas/AutoGPT
51. Letta Team. (2024). Letta: The platform for building stateful agents with advanced memory. https://www.letta.com/

Planning and Decision Making

52. Huang, W., et al. (2024). Understanding the planning of LLM agents: A survey. arXiv:2402.02716.
53. Song, C. H., et al. (2023). LLM Planner: Few-Shot Grounded Planning for Embodied Agents with Large Language Models. *ICCV 2023*.

Evolutionary and Search-Based Optimization

54. Harman, M. (2010). Software Engineering Meets Evolutionary Computation. *IEEE Computer*.
55. Lehman, J., & Stanley, K. O. (2011). Abandoning Objectives: Evolution Through the Search for Novelty Alone. *Evolutionary Computation*.
56. Real, E., et al. (2019). Regularized Evolution for Image Classifier Architecture Search. *AAAI 2019*.

A Appendix

A.1 A: Terminology Glossary

Term	Definition
Agentic Environment	The complete configuration surrounding an agent: prompts, tools, retrieval, and memory
Agent Loop	The iterative cycle of observation, reasoning, action, and feedback that agents execute
Execution Trace	A structured record of agent-environment interaction including outputs, tool calls, errors, and results
Natural Language Gradient	A symbolic update to the environment derived from execution trace analysis
Hierarchy of Mutability	A priority ordering over environment components that prevents lower-priority updates from violating higher-priority constraints
SuperController	The diagnostic meta-controller that attributes failures to environment layers and routes to specialized optimizers
SuperPrompt	The optimizer for system prompts (P layer)
SuperReflexion	The optimizer for tool schemas (T layer)
SuperRAG	The optimizer for retrieval configuration (R layer)
SuperMem	The optimizer for memory (M layer)
Tool Schema	The structured definition of a tool's interface
Confidence Decay	The exponential reduction of memory entry confidence over time
Context Collapse	The degradation of agent performance when accumulated context becomes noisy or contradictory
Optimization Oscillation	Unstable behavior where updates alternate between conflicting configurations
Adapter	An interface implementation that connects SuperOpt to a specific agent framework

A.2 B: Reproducibility Statement

Code Availability: The SuperOpt framework is implemented in Python and available as open source at: `https://github.com/SuperagenticAI/superopt`. See the repository for current structure, installation instructions, and usage examples.

Hardware Requirements: - Experiments conducted on MacBook Pro with 128GB RAM - Local model inference via Ollama (llama3.2:3b, llama3.1:8b) - No GPU required for inference (CPU-based)

Software Dependencies: - Python $\geq$3.12 - LanceDB $\geq$0.26.0 (for SuperRAG experiments) - Aider dependencies (LiteLLM, GitPython, etc.) - Full requirements in `pyproject.toml`

Experimental Configuration: - Model: llama3.2:3b via Ollama (API base: localhost:11434) - LiteLLM streaming disabled for deterministic execution - Temperature: 0.0 for reproducibility - Max iterations: 3 per task - Tasks: 10 challenging algorithmic problems

Determinism Notes: - Execution traces are logged and can be replayed - Random seeds are fixed where applicable - Environment states are serializable via `to_dict()`/`from_dict()`

Evaluation Protocol: 1. Baseline: Run agent with default environment configuration 2. SuperOpt: Run agent with optimization loop enabled 3. Compare: Success rate, execution time, and failure types 4.

Ablation: Disable individual optimizers to measure contribution

A.3 C: Practical Deployment Guidance

When to Use SuperOpt:

SuperOpt is particularly effective when: - Tasks are long-horizon - Tools are complex or brittle - Failures are frequent but interpretable - Human oversight is costly - Models cannot be retrained

When Not to Use SuperOpt:

SuperOpt may be unnecessary when: - Tasks are single-shot - No tools or memory are involved - Failures are rare - Latency must be minimal

Deployment Strategy:

Recommended deployment phases:

1. Shadow mode (logging only)
2. Read-only diagnosis
3. Prompt-level updates
4. Tool schema updates
5. Memory persistence
6. Full autonomy