# The Convergent Architecture: Harmonizing Agents, Skills, and MCP for Cost-Efficient Enterprise AI

## 1. The Context Imperative: From Static Documents to Dynamic Environments

The contemporary software development landscape is undergoing a radical transformation driven by the integration of Large Language Models (LLMs) into the core engineering workflow. We are rapidly transitioning from an era of human-centric documentation—designed for cognitive parsing by biological developers—to agent-centric context engineering, where the primary consumer of architectural knowledge is a stochastic reasoning engine. In this new paradigm, a critical friction point has emerged: while the models themselves are highly mutable, frequently swapped out as vendors like Anthropic, OpenAI, and Google release more capable or efficient versions, the "ground truth" of a software repository must remain immutable.
The user’s query highlights a sophisticated understanding of this friction, identifying a need to move beyond older, static scaffoldings toward a dynamic, vendor-agnostic infrastructure. The initial "Architecture of LLMs" PDF represents the foundational "Constitutional" approach—defining the static rules of engagement. However, the proliferation of "Skills" and dynamic loading, as detailed in the "Skill architecture" document and the "Confucius Code Agent" white paper, introduces a necessary architectural evolution. As token costs rise and model context windows, while larger, become exponentially more expensive to fill with "noise," the engineering challenge shifts from merely *providing* context to rigorously *curating* it through intelligent orchestration.

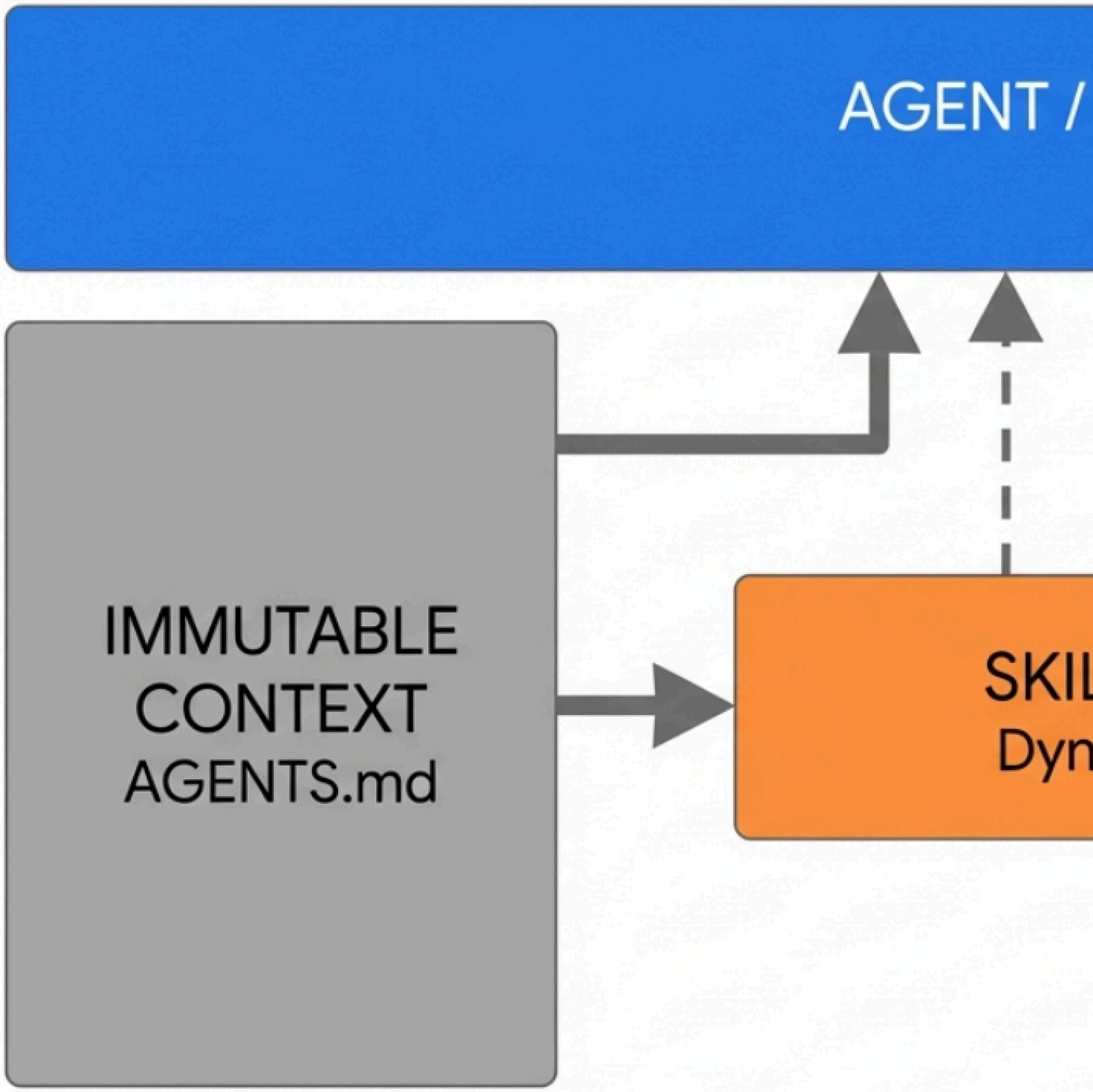
### 1.1 The Tabula Rasa Problem and Context Rot

When an autonomous coding agent initializes a session, it approaches the repository *tabula rasa*. Unlike a human engineer who accumulates tacit knowledge over months of onboarding and social osmosis, the agent is functionally stateless. It possesses no inherent memory of previous architectural decisions, no awareness of the team's unwritten coding conventions, and no understanding of the "tribal knowledge" that often governs complex distributed systems. Without explicit context injection, the model is forced to infer intent from the raw code alone—a process that is computationally expensive, prone to error, and susceptible to "context poisoning," where irrelevant patterns in the codebase mislead the agent's generation logic. This limitation has profound implications for production environments. As noted in recent analyses of agentic workflows, simply dumping the entire repository into the prompt is an anti-pattern known as "context stuffing." This not only introduces latency and increases cost linearly but also degrades performance due to the "lost in the middle" phenomenon, where models struggle to retrieve information buried in the center of a massive context block. Furthermore, research on Recursive Language Models (RLMs) indicates that frontier models suffer from "context rot," where performance on complex reasoning tasks degrades steeply as the input length increases, reinforcing the need for active context management rather than passive loading.

## 1.2 The Stratification of Context: A New Taxonomy

To achieve the "agnostic structure" requested, we must first rigorously define the layers of context. "Context" is not a monolith; it is a stratified architecture where each layer has a specific frequency of change (mutability) and a specific consumer.

## The Universal Context Stack:
## and Capability



The "proper blend" is achieved not by merging these layers, but by creating standard interfaces between them. The Agent (Orchestrator) consults the Constitution (Context) to select the right Procedure (Skill) which executes via a standard Capability (MCP).

#### 1.2.1 The Constitutional Layer (Immutable)

This layer consists of static, invariant rules that define the project's identity. Files like AGENTS.md or CLAUDE.md serve as the repository's constitution. They declare the high-level architecture, the non-negotiable coding standards, and the operational toolchain. This layer answers the question: *"What are the rules of engagement for this codebase?"* It is the "highest leverage point" of the agent harness, often the only file automatically included in every conversation session to align the model's reasoning with business goals.

#### 1.2.2 The Historical Layer (Immutable Log)

Composed of Architectural Decision Records (ADRs) and git history, this layer provides the "why" behind the code. Since agents cannot infer the reasoning behind a past design choice (e.g., "Why did we choose PostgreSQL over MongoDB?"), ADRs serve as an immutable memory log, preventing the agent from relitigating settled decisions or suggesting refactors that violate constraints established years prior.

#### 1.2.3 The Procedural Layer (Dynamic Skills)

This is the layer of "Skills"—specific, task-oriented procedures (e.g., "How to run a migration"). Unlike the constitution, these are loaded *on demand*. They represent the "know-how" of the system. We treat Skills as "Procedures" (stateless SOPs), distinct from "Agents" (identity). Granularity is key here; skills should be "Jira Task" sized, covering specific procedures like "Refactor Component" or "Create Database Migration" rather than broad objectives.

#### 1.2.4 The State Layer (Highly Mutable)

This includes the current file diffs, linter errors, test results, and terminal output. This is the "short-term memory" of the agent session. Through mechanisms like "Hooks" and dynamic skill loading, the repository can inject fresh context in response to the agent's actions, creating a feedback loop where the context evolves within the session itself.

# 2. The Capability Layer: The Model Context Protocol (MCP)

To build a vendor-agnostic infrastructure, we must decouple the *definition* of a tool from the *agent* that uses it. This is the precise role of the Model Context Protocol (MCP). In previous architectures, tool definitions (e.g., git commit, database query) were hardcoded into the agent's system prompt or proprietary SDK. This created vendor lock-in; a tool written for LangChain wouldn't work natively in Cursor or Claude.

## 2.1 MCP as the Universal Driver

MCP creates a standard "Client-Host-Server" architecture for AI tools. In your infrastructure, the **MCP Server** acts as the definitive source of *capability*. It is the "hands" of the agent.

- **Role:** The MCP server exposes atomic, deterministic functions. It does not contain "policy" (how to use the tool) or "identity" (who is using it). It simply exposes read_file, execute_query, or search_docs.
- **Agnosticism:** By defining your internal tools (e.g., a custom deployment script or a database sanitizer) as MCP servers, you ensure they can be consumed by *any* agent that speaks the protocol—whether that is Claude Desktop, a custom internal agent, or a future IDE.
- **Optimization:** MCP servers allow for "resource" abstraction. Instead of feeding a 10MB log file into the context, an MCP server can expose a read_resource tool that allows the agent to read only the relevant lines. This directly addresses your concern about rising request prices.

The decoupling of the capability provider (MCP Server) from the consumer (the Agent) is central to the "proper blend." This allows the infrastructure to swap out the "brain" (the LLM) without needing to rewrite the "hands" (the tools).

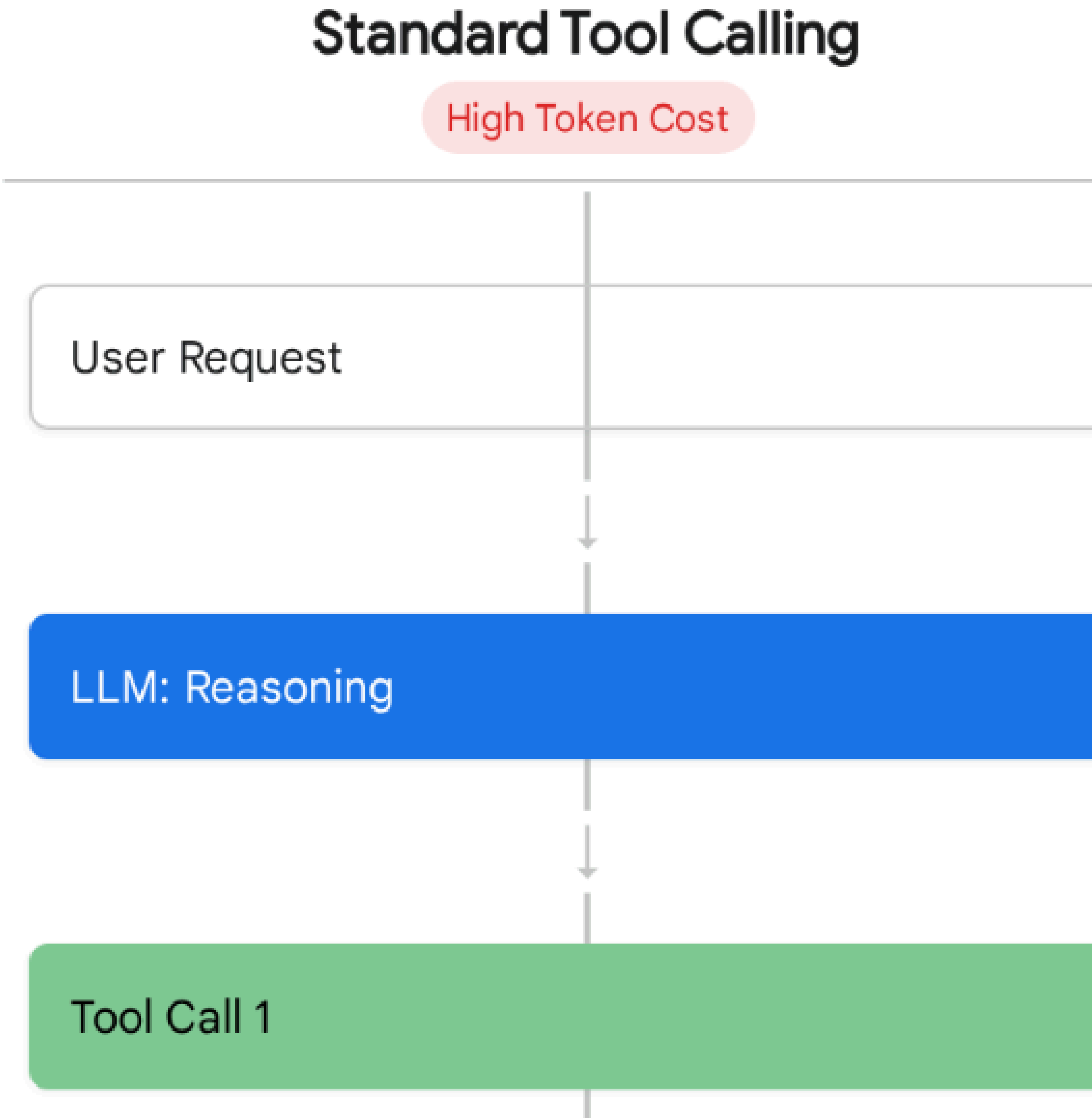
## 2.2 Token Efficiency via Code Execution

A critical insight from recent engineering blogs (e.g., Anthropic's "Code Execution with MCP") is that direct tool calling can be token-expensive. Defining hundreds of JSON schemas for every possible tool consumes vast amounts of the system prompt.

The "proper blend" involves using MCP to expose a **Compute Environment** (like a REPL) rather than just discrete functions. Instead of the agent making ten separate round-trips to read_file, parse_text, and write_file, the agent writes a single Python script that utilizes the MCP environment to perform all three steps in one go. This shifts the burden from "LLM orchestration" (expensive tokens) to "native code execution" (cheap compute).

**Recommendation:** Your infrastructure should expose your core business logic via MCP servers, but your agents should be equipped with a "Code Execution" skill that allows them to interact with these servers programmatically, rather than solely through chat-based tool calling. This aligns with the "Recursive Language Model" findings where agents interact with a REPL to manage their own context.

## Token Economics: Discret
Execution



### 2.3 Semantic Search and MCP

One of the most powerful applications of MCP in an agnostic infrastructure is the integration of semantic search capabilities directly into the agent's toolset. MCP servers can be configured to perform semantic searches over codebases, documentation, or even external knowledge bases. This allows the agent to retrieve relevant context on demand rather than requiring the entire context to be pre-loaded. For instance, an agent could use a "Semantic Context" MCP server to find relevant code snippets based on a natural language query, which it then processes using its procedural skills. This capability is instrumental in realizing the "Universal Context" where agents can navigate vast repositories without being overwhelmed by data.

# 3. The Procedural Layer: The Renaissance of Skills

While MCP provides the *capability*, it does not provide the *wisdom*. This is where "Skills" enter the architecture. Your uploaded document ("Skill architecture.pdf") correctly identifies Skills as "Stateless SOPs" or "Procedures." This distinction is vital for maintaining a clean separation of concerns.

## 3.1 Agents vs. Skills: The Identity Separation

In monolithic architectures, "Agent" and "Skill" were often conflated. An agent was prompted to "be a Data Engineer who knows how to migrate SQL." This is brittle. If the migration process changes, you have to retrain or reprompt the agent.
In the agnostic structure, **Agents** are generic reasoning engines (e.g., "Senior Engineer Persona") that *possess* **Skills** (e.g., "Postgres Migration Skill").

- **Agents (Identity):** Defined in the system prompt. Responsible for planning, error recovery, and ethical judgment.
- **Skills (Procedure):** Defined in markdown files (e.g., docs/skills/migrate-db.md). Responsible for the step-by-step execution logic.

This separation allows you to swap out the agent (e.g., upgrading from GPT-4 to Claude 3.5 Sonnet) without losing the procedural knowledge encoded in your skills. It also allows the same skill to be used by different agents with different personas.

## 3.2 The Skill Schema and Publisher Pattern

Your proposed "Publisher Pattern" is a sophisticated solution to the vendor fragmentation problem. By maintaining a "Rich Source of Truth" in docs/skills/ and compiling it down for specific consumers (Copilot, Cursor), you achieve the "write once, run everywhere" goal.

### 3.2.1 Validation of the Schema

Research into Semantic Kernel and LangChain templates confirms that your proposed structure (Context / Procedure / Verification) aligns with industry best practices, but it lacks one critical component found in robust systems: **Few-Shot Examples**.

- **Recommendation:** Add a examples section to your SKILL.md schema. Research consistently shows that providing 1-2 examples of "Input -> Reasoning -> Output" significantly improves adherence to the procedure, especially for complex reasoning tasks.

#### 3.2.2 The "Shadow Frontmatter" Pattern

Injecting metadata into the markdown body (as HTML comments or a hidden section) is a proven technique for preserving context in restrictive environments. Models like Claude 3.5 Sonnet and GPT-4o are highly capable of parsing structured data within the text body. This allows you to pass trigger_phrases and required_mcps to the model even if the host IDE (like GitHub Copilot) strips the YAML header.

### 3.3 Dynamic Loading and Context Curation

To address the cost concern, Skills must be loaded dynamically. You cannot stuff 50 skills into the system prompt.

- **Semantic Routing:** The agent's "Constitutional" prompt should include a lightweight index of available skills (Name + Description only).
- **Just-in-Time Injection:** When the agent determines a skill is needed (e.g., user asks "refactor this API"), it requests the full content of skills/refactor-api.md.
- **Unloading:** Once the task is complete, the skill content should be explicitly dropped from the context window to free up space for the next task. This "Context Garbage Collection" is a key feature of advanced frameworks like the Confucius Code Agent, which uses a "Meta-agent" to refine agent configurations and manage memory.

# 4. The Orchestration Layer: Recursive Language Models (RLMs) and Hierarchical Memory

The most forward-looking part of your architecture involves how the agent *thinks* over long horizons. The PDF "Recursive-language-models.pdf" introduces a paradigm shift that directly addresses your fears about context windows and cost.

## 4.1 The Illusion of Infinite Context via RLMs

Standard agents try to fit the entire history into the prompt. RLMs take a different approach: they treat the context as an **external environment** (like a database or a file system) that they can query. The core principle is "Inference-Time Scaling," where the model programmatically examines and decomposes inputs into manageable snippets, rather than digesting the whole block at once.

- **The REPL Pattern:** Instead of "reading" a 10,000-line file, the RLM writes a script to grep the file for relevant lines. It interacts with the text programmatically. The RLM operates within a Read-Eval-Print Loop (REPL) environment where the input prompt is loaded as a variable (e.g., prompt).
- **Recursive Sub-Calls:** If a task is too big, the RLM spawns a sub-agent (a recursive call) to handle a specific chunk, passing only the necessary context to that sub-agent. The sub-agent returns a summarized result. This creates a depth hierarchy where a "root" model delegates sub-tasks to recursive instances.

**Implication for your Architecture:** Your "Custom Agent" should implement this RLM pattern. It should not hold the state of the entire codebase in memory. Instead, it should use MCP tools to "probe" the codebase and "recurse" on specific problems. This keeps the active context window small (low cost) while maintaining access to the massive repository (high utility).

## 4.2 Confucius SDK: A Blueprint for Enterprise Agents

The "Confucius Code Agent" (CCA) white paper provides a blueprint for structuring enterprise-level agents that aligns perfectly with your goals of robustness and cost-efficiency. It introduces distinct "Experience" axes that should be part of your architecture.

### 4.2.1 The Three Axes of Agent Design

- **Agent Experience (AX):** This focuses on the agent's internal cognitive workspace. It dictates how information is distilled and presented to the LLM to ensure stable reasoning. Crucially, AX avoids passing raw human-oriented logs directly to the agent, which degrades performance and wastes tokens.
- **User Experience (UX):** This concerns the transparency and controllability of the agent for human developers. It emphasizes creating interpretable artifacts, such as persistent Markdown notes, that humans can review.
- **Developer Experience (DX):** This focuses on the observability and modularity of the system for those building it. It includes tools for rapid adaptation and automated refinement.

### 4.2.2 Hierarchical Working Memory

To prevent "Context Rot" in long sessions, you should implement a **Hierarchical Working Memory** system as described in the Confucius SDK.

- **Plan (Long-Term):** A plan.md file that stores the high-level objective. This is rarely updated but always present.
- **Task (Medium-Term):** A list of current steps. Updated after every major action.
- **Scratchpad (Short-Term):** The immediate reasoning trace. This is wiped/summarized frequently.

This stratification ensures that the agent never "forgets" the main goal (Plan) even if it gets stuck in a deep debugging rabbit hole (Scratchpad). It couples hierarchical memory with **adaptive context compression**, allowing the agent to retain essential states while supporting long-horizon reasoning without exceeding context limits.

### 4.2.3 The Meta-Agent Loop

The Confucius architecture also introduces a "Meta-agent" that automates the **build-test-improve loop**. This meta-agent synthesizes, evaluates, and refines agent configurations, allowing for rapid adaptation to new tasks and tool stacks. In your "agnostic structure," this could be implemented as a CI/CD process that tests your agents against a set of "benchmark" tasks whenever you update your AGENTS.md or Skills, ensuring that changes don't degrade performance.

# 5. Synthesizing the "Proper Blend": The Agnostic

# Directory Structure

To answer your core question—*"what's the proper blend to achieve an agnostic structure?"*—we propose the following repository structure. This structure separates concerns, respects the distinct roles of Agents, Skills, and MCP, and is "compile-ready" for various vendors.

## 5.1 The Directory Schema

We propose a unified directory structure that acts as the "Rich Source of Truth," which is then compiled or symlinked for specific tools.

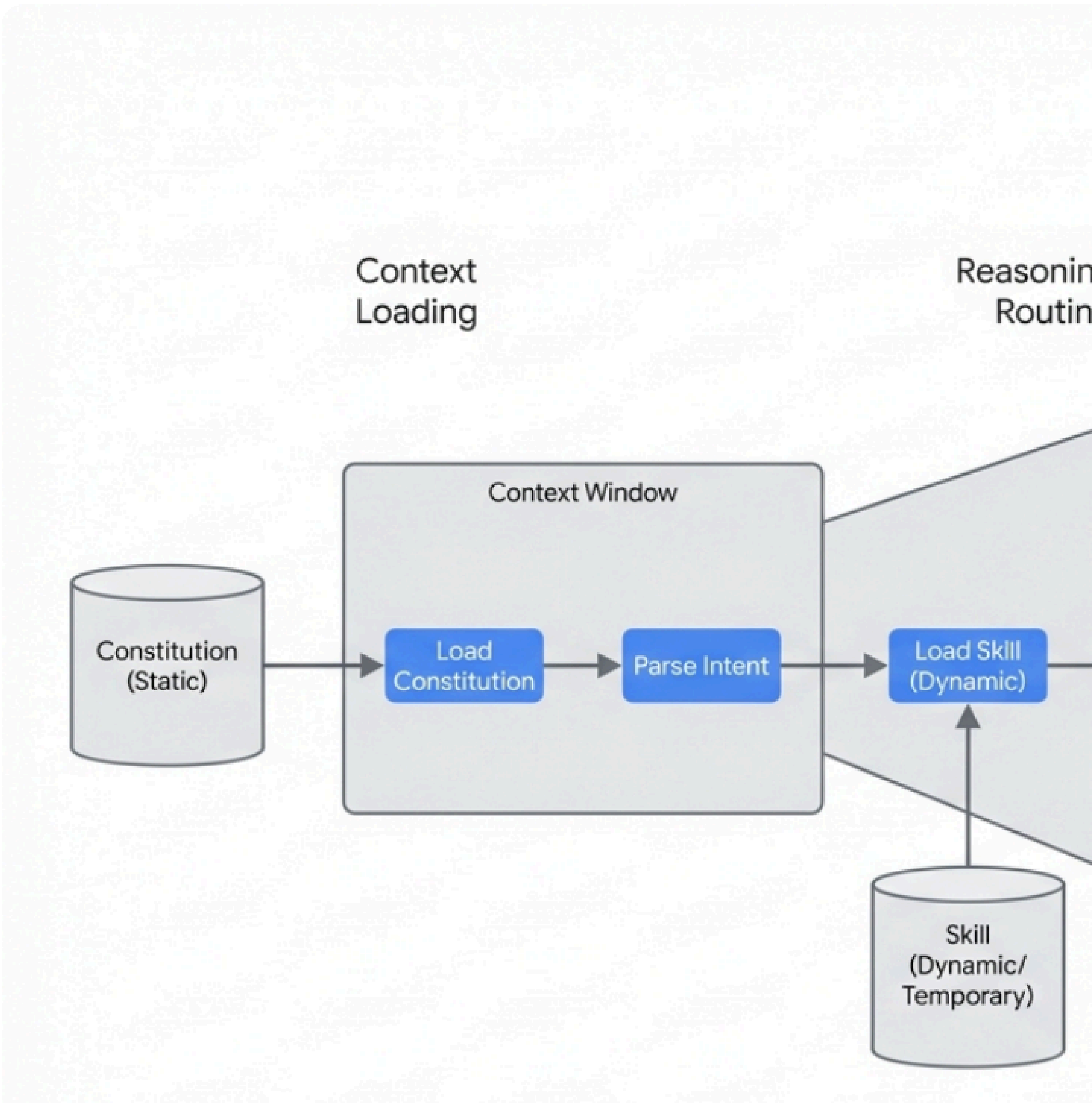
| Directory / File | Description | Consumer |
|---|---|---|
| **docs/constitution/** | **The Constitutional Layer (Immutable)** | All Agents |
| docs/constitution/AGENTS.md | The Master Constitution. Defines high-level architecture, coding standards, and operational toolchain. | All |
| docs/constitution/ADRs/ | Architectural Decision Records. The "Why" behind the code. | All |
| **docs/skills/** | **The Procedural Layer (Dynamic Source)** | Dynamic Loaders |
| docs/skills/db-migration.md | A specific skill. Contains Rich YAML frontmatter (trigger phrases, MCP dependencies) + Procedure + Verification. | Custom Agent |
| docs/skills/refactor.md | Another skill. | Custom Agent |
| **mcp/** | **The Capability Layer (Execution)** | MCP Clients |
| mcp/database-tool/ | Source code for a custom MCP server (e.g., exposing database sanitization tools). | All |
| **.github/skills/** | **Compiled Artifacts (Generated)** | GitHub Copilot |
| *.md | Stripped versions of skills generated by publish-skills.py for Copilot compatibility. | Copilot |
| **.cursor/rules/** | **Compiled Artifacts (Generated)** | Cursor |
| *.mdc | Stripped and formatted versions of skills generated for Cursor. | Cursor |
| **scripts/** | **Infrastructure Tools** | CI/CD |
| scripts/publish-skills.py | The "Publisher" script that transforms source skills into vendor-specific artifacts (Shadow Frontmatter pattern). | Build System |

## 5.2 The Unified Interaction Flow

The interaction flow demonstrates how these components work together in a live session.

1. **Initialization:** The Agent loads AGENTS.md. This defines its Persona ("Senior Engineer") and loads the *Index* of available Skills (metadata only). This establishes the "Constitutional Layer."
2. **Intent Recognition:** The user asks a question. The Agent scans the Skill Index to see if a specific procedure applies.
3. **Dynamic Injection:**
    - *If a Skill matches:* The Agent requests the full content of docs/skills/my-skill.md. The system injects this into the active context.
    - *If no Skill matches:* The Agent relies on general reasoning and core MCP tools.
4. **Execution (MCP Layer):**
    - The Skill prescribes a procedure (e.g., "Check database schema").
    - The Agent invokes the database-schema MCP tool to execute this. Crucially, if using the RLM pattern, the agent might write a script to utilize the MCP tool rather than calling it directly.
    - The MCP Server performs the action deterministically.
5. **Verification:** The Agent checks the result against the "Verification" section of the Skill.
6. **Cleanup:** The Agent summarizes the outcome into its "Historical Memory" (part of the Hierarchical Memory system) and creates a "Hindsight Note" if it encountered a failure (as suggested by the Confucius paper ). The raw skill text is flushed from the context to save tokens.

# The Interaction Lifecycle: Ho
Converge



## 6. Strategic Recommendations for Cost and Future-Proofing

Your intuition that "prices are going to go up per request" is a valid strategic constraint. While base model prices (per token) are dropping, the *complexity* of agentic requests (number of turns, depth of reasoning) is exploding, leading to a higher total cost of ownership.

### 6.1 The "Strip-Markdown" and "Prompt Compiler" Strategy

To minimize waste, your "Publisher" script should do more than just move YAML. It should implement a **"Prompt Minification"** step.

- **Whitespace Removal:** Markdown often contains excessive whitespace for human readability. Agents don't need this.
- **Comment Stripping:** Remove HTML comments that are for developers, leaving only the instructions for the agent.
- **Pseudo-Code Summaries:** As suggested in the "Architecture of LLMs" PDF, for massive files, inject a "Generative Pseudo-Code" summary instead of the raw code. This reduces token usage by ~40% while preserving logic.

### 6.2 Leveraging "Reasoning Models" (System 2 Thinking)

The trend towards "Reasoning Models" (like OpenAI's o1 or DeepSeek R1) changes the equation. These models "think" before they speak, consuming more compute but potentially requiring *less* verbose context because they are better at inferring intent.

- **Recommendation:** For your "Custom Agent," implement a "Router" pattern. Use a cheaper, faster model (e.g., GPT-4o-mini) for the initial routing and skill selection. Only invoke the expensive "Reasoning Model" for the execution of complex Skills. This hybrid approach optimizes the "Intelligence/Cost" ratio.

### 6.3 The "Gardener" Pattern

Finally, immutability requires maintenance. As your repository grows, AGENTS.md and Skills will rot. Implement a "Gardener Agent" that runs periodically (e.g., via GitHub Actions).

- **Task:** It reads recent PRs and ADRs.
- **Action:** It suggests updates to AGENTS.md or new Skills based on emerging patterns.
- **Value:** This keeps your context "fresh" without requiring constant human intervention, ensuring the "Immutable Context" evolves in lockstep with the code.

## 7. Conclusion

The "proper blend" for your enterprise infrastructure is not a static template but a dynamic lifecycle. It requires elevating **Context** to a constitutional level, treating **Skills** as compiled software artifacts, and utilizing **MCP** as the universal hardware interface.

By adopting the **Runtime Skill Injection** model and the **Publisher Pattern**, you insulate your organization from vendor shifts. By implementing **Recursive Language Model** principles and **Hierarchical Memory** from the Confucius SDK, you solve the "context rot" and cost problems.

This architecture positions your team not just to use AI agents, but to govern them—transforming them from chaotic experimental tools into reliable, cost-effective engineering partners.
The future of AI engineering is not about having the smartest model; it is about having the most intelligible context. Your proposed infrastructure, refined with these recommendations, is perfectly poised to capture that value.

## Citations

- ****: Skill architecture.pdf - For Runtime Skill Injection, Publisher Pattern, and Skill Schema.
- ****: The architecture of LLMs.pdf - For Immutable Context, AGENTS.md, and Context Stratification.
- ****: Agent scaffolding white paper.pdf - For Confucius SDK, Hierarchical Memory, and AX/UX/DX.
- ****: Recursive-language-models.pdf - For RLM, REPL environment, and Inference-time scaling.
- : Anthropic & Claude Blogs - For MCP definitions and Code Execution patterns.
- : Cursor Documentation - For.cursorrules best practices.
- : Community & Skywork Blogs - For Semantic Search MCP servers.
- : MCP Specification & Logto Blog - For MCP Architecture details.

### Works cited

1. Going Beyond the Context Window: Recursive Language Models in ..., https://towardsdatascience.com/going-beyond-the-context-window-recursive-language-models-in-action/ 2. Architecture - Model Context Protocol, https://modelcontextprotocol.io/specification/2025-06-18/architecture 3. MCP server vs Tools vs Agent skill - Logto blog, https://blog.logto.io/mcp-tools-agentskill 4. Model Context Protocol (MCP) vs. APIs: Architecture & Use Cases, https://www.codecademy.com/article/mcp-vs-api-architecture-and-use-cases 5. How Skills compares to prompts, Projects, MCP, and subagents, https://claude.com/blog/skills-explained 6. Code execution with MCP: building more efficient AI agents - Anthropic, https://www.anthropic.com/engineering/code-execution-with-mcp 7. My First MCP Server: Semantic Code Search - DEV Community, https://dev.to/paradoxy/my-first-mcp-server-semantic-code-search-3520 8. Unlocking AI: A Deep Dive into Semantic Context MCP Servers, https://skywork.ai/skypage/en/unlocking-ai-semantic-context/1979069004350345216 9. Rules | Cursor Docs, https://cursor.com/docs/context/rules 10. Best practices when using Cursor the AI editor. - GitHub, https://github.com/digitalchild/cursor-best-practices