SPAK: A Spec-Driven Kernel for Autonomous Engineering Systems

Abstract

The transition from chatbots to **Autonomous Engineering Systems (AES)** requires agents that can not only generate code but also reason, plan, and verify their actions against strict engineering constraints. Current agent frameworks often lack the architectural rigor needed for such high-stakes tasks, relying on fragile prompt engineering. We propose **Systematic Intelligence Engineering (SIE)**, a methodology that decouples an agent's "System Model" (domain physics) from its "Runtime Execution." To operationalize this, we present **SPAK (Spec-driven Programmatic Agent Kernel)**, a lightweight runtime that injects formal specifications into the Large Language Model's (LLM) latent planning process. SPAK enables a **Cognitive Loop**—*Thought, Plan, Act, Observe, Refine*—within a "Soft Sandbox" environment, allowing agents to iteratively solve problems while adhering to safety invariants. We validate this approach through a **Level 4 Proof-of-Concept**, demonstrating how a "Knowledge Chef" agent can autonomously navigate research tasks by dynamically replanning when initial strategies fail.

1. Introduction

We are witnessing a paradigm shift in AI development: moving from static "Question-Answering" models to dynamic **Autonomous Engineering Systems (AES)** capable of designing, coding, and debugging software. However, building these systems with current tools reveals a critical "Grounding Gap":

1. **Fragility:** Agents struggle to maintain long-term coherence or adhere to strict API constraints.
2. **Lack of Introspection:** When an agent fails (e.g., generates invalid code), it often cannot "realize" the mistake and self-correct without external intervention.

3. **Security Risks:** Direct execution of LLM-generated code poses significant risks.

We argue that reliability in AES is an architectural problem. It requires a kernel that treats **Agent Behavior** as a compilable specification, not just a prompt.

Key Contributions:

- **The SIE Framework:** A layered approach to building AES, separating "Meta-Supervision" from "Runtime Execution."
- **Spec-Guided Latent Planning:** A mechanism where formal constraints (`system_model`) are injected into the LLM's context to guide its reasoning *before* it takes action.
- **The SPAK Kernel:** A Python-based runtime that enforces a "Cognitive Loop" (Plan-Act-Verify) and provides a secure "Soft Sandbox" for tool execution.

2. The Systematic Intelligence Engineering (SIE) Framework

SIE structures the development of autonomous agents into a hierarchy of control. This paper focuses on **Level 4**, where the agent possesses the capability to reason, plan, and self-correct within a defined domain.

2.1 Hierarchical Architecture

Our architecture distinguishes between the *internal* reasoning loop of the agent and the *external* supervision provided by developers (or Meta-LLMs).

```
Layer 4: Meta-Level Supervision (External)
   |      • Monitoring logs / Rewriting Specs / Heuristic Patchin
   ▼
Layer 3: SIE Methodology (The Constitution)
   |      • AgentSpec Grammar / Design Patterns / Domain Constrai
   ▼
Layer 2: Autonomous Engineering System (The Engine)
   |      • Latent Planning / Self-Correction / Domain Agents
   ▼
Layer 1: SPAK Kernel (The Runtime)
```

- "Soft Sandbox" Execution / Algebraic Effects / Traceab

3. The SPAK Architecture

SPAK acts as the operating system for agents. It manages the interface between the **Latent Space** (the LLM's reasoning) and the **Symbolic Space** (Python execution).

3.1 AgentSpec: Defining the "System Model"

Instead of vague textual instructions, SPAK agents are defined using **AgentSpec**, a Domain Specific Language (DSL). A key innovation is the `system_model`, which encodes the "physics" of the domain—axioms, heuristics, and prediction rules.

Listing 1: AgentSpec with System Model

```
system KnowledgeChef {
    // The "Brain": Tells the agent HOW to think about the domair
    system_model ResearchDynamics {
        axiom: "A claim without a primary source is a hallucinati
        heuristic: "Data Scarcity" -> strategy: "Broaden Keywords
        prediction: "Reading binary files" -> result: "Decoding E
    }

    // The "Body": Capabilities (Effects)
    effect Librarian {
        operation search(query: String) -> List<Snippet>;
    }
}
```

3.2 The Cognitive Loop: Spec-Guided Planning

Unlike traditional "Chain-of-Thought" which is linear, SPAK implements a recursive **Plan-Act-Observe-Refine** loop:

1. **Constraint Injection:** The Kernel parses the `system_model` and injects it into the LLM's System Prompt. This sets the "boundaries" for the agent's imagination.

2. **Latent Planning:** The LLM generates a **Plan-IR** (Intermediate Representation)—a structured list of intended steps (e.g., `[Search, Analyze, Write]`).
3. **Soft Sandbox Execution:** The Kernel executes the planned tools. Since Docker is resource-intensive for this PoC, we implement a **Restricted Python Scope**:
    - Code is executed via `exec()` with a whitelist of allowed modules (`math`, `re`, `json`).
    - File I/O and Network calls are banned in the code and must be performed via Kernel **Effects**.
4. **Reflection & Re-planning:** The agent observes the result. If the result is poor (e.g., search returned zero results), the `system_model` triggers a heuristic (e.g., "Broaden Keywords"), causing the agent to generate a *new* plan.

4. Implementation: The Level 4 Proof-of-Concept

To validate this architecture without the complexity of full containerization, we built a lightweight Python implementation using **Lark** for parsing and a local LLM (or API) for reasoning.

4.1 The Kernel Runtime

The Kernel is an event loop that mediates all interactions. It does not allow the agent to talk directly to the OS.

- **Input:** `AgentSpec` file and a User Goal.
- **Process:**
    1. Parser loads the Spec and extracts the `system_model`.
    2. Kernel initializes the `Orchestrator`.
    3. `Orchestrator` prompts the LLM to generate a Plan.
    4. Kernel validates the Plan against `invariants` (e.g., "No infinite loops").
    5. Kernel executes the Plan step-by-step.

- **Output:** A verified artifact (e.g., a Report or Code Snippet).

4.2 Handling Failure: The "Self-Correction" Mechanism

In standard scripts, an error (e.g., `FileNotFound`) crashes the program. In SPAK, an error is an **Observation**.

- **Scenario:** Agent tries to read `data.csv`.
- **Kernel:** Returns `Error: File not found`.
- **Agent (Latent Reasoning):** "My plan failed. My `system_model` says I should check the directory listing first."
- **New Plan:** `[ListFiles, ReadFile(actual_name)]`.
- **Result:** Success.

5. Case Study: The Knowledge Chef

We deployed the "Knowledge Chef" agent to synthesize a technical report on "HBM3 Memory Architecture."

5.1 Experiment Setup

- **Goal:** "Write a summary of HBM3 bandwidth based on available papers."
- **Constraint:** The agent must only use data found in the provided mock file system.
- **Simulated Fault:** The initial search query "HBM3 specs" returns no exact matches; the files are named "HighBandwidthMemory_v3.txt".

5.2 Execution Trace

1. **Initial Plan:** `Librarian.search("HBM3 specs")`
2. **Observation:** `[]` (Empty List).
3. **Reflection:** The agent triggers the `Data Scarcity` heuristic defined in its Spec.
4. **Re-planning:** It generates a new plan: `[Librarian.list_files("/data"), Librarian.read("HighBandwidthMemory_v3.txt")]`.

5. **Execution:** The file is found and read.
6. **Final Output:** A report citing the correct file.

This demonstrates **Level 4 Autonomy**: the ability to modify one's own plan at runtime based on environmental feedback, without human intervention.

6. Discussion & Future Work

This work establishes the foundation for **Autonomous Engineering Systems**. By combining the flexibility of LLMs with the rigor of formal specifications (SPAK), we create agents that are both creative and reliable.

Future Work (Level 5): We envision a "Meta-Supervisor" layer where a superior LLM analyzes the execution logs of SPAK. If an agent consistently fails to plan correctly, the Meta-Supervisor will autonomously edit the `AgentSpec` text file—effectively "patching" the agent's brain—closing the loop on fully autonomous software evolution.

7. Conclusion

SPAK provides a pragmatic, engineer-friendly path to building reliable AI