SprintAgent: A Production-Ready Framework for Hierarchical Autonomous Agentic Systems

Ali Shamsaddinlou

March 2026

Abstract

From an engineering perspective, building autonomous agentic systems with large language models (LLMs) demands more deliberate design thinking than traditional software. The trade-offs are sharper and the consequences of getting them wrong are harder to debug: complexity vs. explainability, isolation vs. throughput, context growth vs. performance, and configuration flexibility vs. simplicity. Yet most prototypes lack the engineering infrastructure needed to navigate these trade-offs in production. This paper introduces SprintAgent, an open-source framework that provides a production-ready scaffold for hierarchical autonomous agentic systems, developed with these engineering trade-offs explicitly in mind so that the resulting systems remain manageable, auditable, and explainable. The framework features a planner/designer/critic orchestration loop per agent, Hydra-based Inversion of Control configuration, SQLite-backed session management with turn trimming, checkpoint-based auto-reset on performance degradation, isolated parallel execution, and unified logging for observability. The framework is designed to be highly modular: practitioners can plug their own logic into each module and extend on top with minimal friction. To demonstrate how this looks in practice, an agentic project-management system is built on top of SprintAgent as a reference application; the project-management system is not the contribution itself but rather a demonstration of the scaffold in action.

1 Introduction

Large language models have enabled a new class of autonomous software agents that can plan, reason, and execute multi-step tasks with minimal human intervention [8, 10]. Frameworks such as LangChain [1], AutoGen [9], MetaGPT [5], and CrewAI [2] have lowered the barrier to building agentic applications. However, moving from a notebook prototype to a production-grade agentic system reveals a gap: most existing frameworks focus on the *agent logic* (prompt chaining, tool calling, memory) while leaving the surrounding *engineering infrastructure*—configuration management, fault tolerance, session persistence, observability, and execution isolation—to the developer.

From an engineering perspective, building agentic systems demands more deliberate design thinking than traditional software. The trade-offs are sharper and the consequences of getting them wrong are harder to debug:

- **Complexity vs. explainability** – powerful multi-agent orchestration is useless if no one can understand what the system did and why.
- **Isolation vs. throughput** – strong fault boundaries keep runs safe but add overhead.
- **Context growth vs. performance** – agents accumulate history that must be managed, not just appended.

- **Configuration flexibility vs. simplicity** – real-world systems have dozens of knobs; the config system must scale without becoming its own problem.

What does a production-ready autonomous agentic system look like when it needs to handle complex agents without becoming a mess? This question motivated the design of SprintAgent, an open-source framework developed with these engineering trade-offs explicitly in mind. SprintAgent is a highly modular scaffold where practitioners can plug their own logic into each module and extend on top. For demonstration purposes, an agentic project-management system is built on top of the framework to show what it looks like in practice to build such a system.

The key contributions of this paper are:

1. A **hierarchical agentic architecture** in which each stage (project, module, task, visualization) is handled by an autonomous agent that internally runs a planner/designer/critic refinement loop.
2. An **Inversion of Control configuration system** built on Hydra and OmegaConf with custom resolvers, enabling new agents and experiments to be wired entirely through YAML without code changes.
3. **Production infrastructure** including SQLite-backed session management with turn trimming and summarization, checkpoint-based auto-reset on quality degradation, process-isolated parallel execution, and unified file-and-terminal logging for observability and explainability.
4. An **open-source implementation** accompanied by a demonstration application—an agentic project-management system—that illustrates how to build a complete hierarchical agentic application on the framework.

2 Background and Motivation

2.1 LLM-Based Autonomous Agents

The ReAct paradigm [12] showed that interleaving reasoning and action in LLMs produces more capable agents. Reflexion [7] added verbal self-reflection, enabling agents to learn from mistakes within a single episode. These ideas have been operationalized in multi-agent frameworks: AutoGen introduces conversable agents that collaborate through messages [9]; MetaGPT assigns software-engineering roles to agents in a simulated company [5]; CrewAI orchestrates role-playing agents with configurable workflows [2]; and the OpenAI Agents SDK provides primitives for tool-calling agents with handoff support [6].

2.2 The Production Gap

While these frameworks excel at composing agent logic, production deployments require additional concerns:

- **Configuration complexity.** Real-world agentic systems have dozens of knobs (model selection, temperature, scoring thresholds, memory policies, pipeline stages). Hard-coded or flat-file configs do not scale.
- **Fault tolerance.** LLM outputs are non-deterministic; iterative refinement can degrade quality. Systems need mechanisms to detect and recover from regressions.
- **Session persistence and memory management.** Long-running agents accumulate context that exceeds token limits. Persistent sessions with turn trimming and summarization are essential.

- **Observability.** Debugging multi-agent systems requires unified logging that captures what each agent did, when, and why.
- **Execution isolation.** Parallel runs must not interfere with each other; failures in one run should not cascade.

SprintAgent addresses each of these concerns as a first-class framework feature, so developers can focus on domain logic rather than infrastructure.

3 System Overview

SprintAgent transforms a high-level project brief into an execution-ready roadmap through a four-stage hierarchical pipeline: **project** → **module** → **task** → **visualization**. Each stage is handled by a specialized agent that refines its output through an internal planner/designer/critic loop before passing results to the next level.

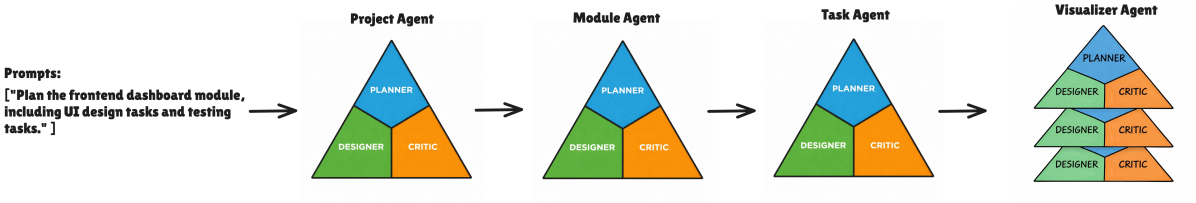

Figure 1: Hierarchical pipeline. Each stage consumes the output of the previous stage and produces a refined plan. The visualization stage generates Mermaid diagrams rendered to PNG/PDF.

Table 1 summarizes the inputs, outputs, and responsible agent for each stage.

Table 1: Pipeline stages and their agents.

Stage	Input	Output	Agent
Project	User brief	`project_plan.md`	StatefulProjectAgent
Module	Project plan	`module_plan.md`	StatefulModuleAgent
Task	Module plan	`task_plan.md`	StatefulTaskAgent
Visualization	All three plans	Mermaid + PNG/PDF diagrams	StatefulVisualizationAgent

The pipeline stages are configurable: users can set `start_stage` and `stop_stage` in YAML to run any contiguous subset of the pipeline.

4 Architecture

4.1 Planner/Designer/Critic Loop

Every stage agent uses the same internal architecture: three sub-agents coordinated by a planner (Figure 2).

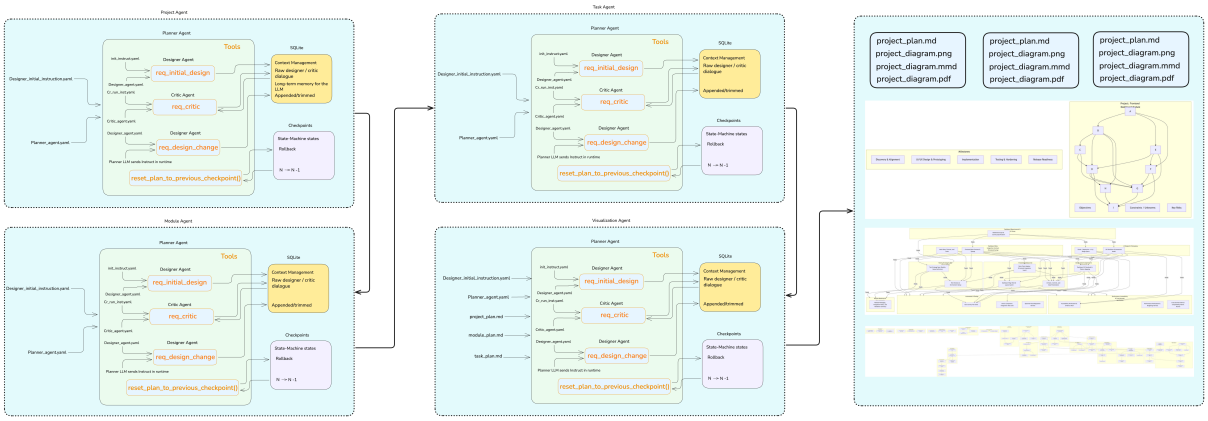

Figure 2: Internal agent architecture. The planner orchestrates the designer and critic through tool calls. Checkpointing enables rollback when quality degrades.

Designer. Creates and refines the artifact (plan text or Mermaid diagram). It receives the context—user brief or parent plan—produces an initial design, and iteratively improves it based on critic feedback. Its full conversation history is persisted in a SQLite session so the critic and subsequent iterations have complete context.

Critic. Evaluates the current artifact against domain-specific criteria (structure, clarity, completeness) and returns structured feedback with per-category scores. For project, module, and task stages the critic reads the designer's session history. For the visualization stage the critic is a *vision-language model* (VLM) that receives the rendered diagram image alongside the plan text.

Planner. Orchestrates the loop via internal tools:

- `request_initial_design` – ask the designer for the first draft.
- `request_critique` – ask the critic to evaluate the current artifact.
- `request_design_change` – ask the designer to address specific issues raised by the critic.
- `reset_plan_to_previous_checkpoint` – roll back to the last good state if scores degrade.

The planner alternates between critique and design-change rounds until quality thresholds are met or the maximum number of rounds is reached. `max_critique_rounds` and `early_finish_min_score` are configurable per agent.

4.2 Checkpoint-Based Auto-Reset

Each critique round updates a checkpoint consisting of the current plan state and associated scores. The framework maintains an *N-1/N* checkpoint pair: the current checkpoint (N) and the previous one (N-1). If the total score after a critique is lower than the previous checkpoint, the planner can call `reset_plan_to_previous_checkpoint`, which:

1. Compares the total scores of checkpoints N and N-1.
2. If N $<$ N-1, restores the plan state from checkpoint N-1.
3. Appends a rollback message to the designer session so subsequent iterations continue from the better state.

This mechanism provides **intra-run fault tolerance**: the system recovers from quality regressions without manual intervention.

4.3 Structured Scoring

Critic output is a structured dataclass (`CritiqueWithScores`) with domain-specific categories. Each category has a name, a grade (0–10), and a textual comment. Domain-specific subclasses define the relevant categories:

- `ProjectCritiqueWithScores` – project-level criteria.
- `ModuleCritiqueWithScores` – module-level criteria.
- `TaskCritiqueWithScores` – task-level criteria.
- `VisualizationCritiqueWithScores` – diagram-level criteria.

A shared `compute_total_score` function sums category grades, and `format_score_deltas_for_planner` formats score changes between iterations so the planner can make informed decisions about when to continue, stop, or roll back.

4.4 Inversion of Control and Configuration

SprintAgent follows the Inversion of Control principle [4]: the framework owns the execution flow and calls user-supplied components at well-defined extension points. Configuration is managed through Hydra [3] and OmegaConf [11], providing:

- **Hierarchical YAML composition.** Each agent type has a `core_*.yaml` file with shared settings and a `workflow_*.yaml` overlay for workflow-specific overrides. Hydra's defaults list composes them automatically.
- **Cross-config interpolation.** Global settings (e.g., `openai.service_tier`) are defined once and interpolated into agent configs via `${openai.service_tier}`.
- **Custom resolvers.** OmegaConf resolvers for path resolution (`resolve_path`, `resolve_list`), boolean negation (`not`), equality checks (`equal`), and conditionals (`ifelse`) enable expressive config logic without code.
- **Agent wiring.** Each experiment declares a `compatible_*_agents` dictionary mapping config names to Python classes. At runtime the framework reads the `_name` field from the resolved config and instantiates the corresponding class, achieving dependency injection through configuration.

Adding a new agent type requires only a YAML config file and a registration entry in the experiment's compatible agents dictionary—no changes to the core orchestration code.

4.5 Session Management and Turn Trimming

Designer and critic conversations are persisted in SQLite databases (e.g., `prompt_000_project_designer.db`). This provides:

- **Crash recovery.** Sessions survive process restarts.
- **Context for critics.** The critic reads the designer's full session history to evaluate the current state in context.

- **Auditability.** Every interaction is recorded and can be inspected post-hoc.

For long-running agents, unbounded context growth exceeds token limits. SprintAgent addresses this with `TurnTrimmingSession`, a decorator that wraps the base `SQLiteSession`:

1. **Turn parsing.** Conversation items are segmented into turns by user-message boundaries.
2. **Retention window.** The last N turns (configurable via `keep_last_n_turns`) are kept intact.
3. **Image stripping.** Older turns have images replaced with text placeholders to reduce token usage.
4. **Optional summarization.** When `enable_summarization` is active, trimmed turns are summarized by an LLM. Summaries are cached in a dedicated SQLite table (keyed by turn content hash) to avoid redundant API calls.

An additional `IntraTurnImageFilter` strips images from older tool outputs *within* a single turn, keeping only the most recent observations. This is particularly useful for the visualization critic, which may call observation tools multiple times per iteration.

4.6 Parallel Execution and Process Isolation

SprintAgent supports both sequential and parallel execution of prompts. Parallel execution uses `ProcessPoolExecutor`, where each prompt runs in a separate process:

- **Memory isolation.** Each process has its own address space; no shared mutable state.
- **Fault containment.** A crash or exception in one worker does not affect others. The orchestrator collects results and reports per-prompt success or failure.
- **Log separation.** Each worker creates its own `FileLoggingContext` writing to `prompt_XXX/plan.log`, with `suppress_stdout=True` to prevent log interleaving.

The number of workers is controlled by the `num_workers` configuration parameter; setting it to 1 enables sequential execution for debugging.

4.7 Unified Logging

SprintAgent uses a `FileLoggingContext` context manager that attaches a file handler to Python's root logger, capturing output from all modules without per-module configuration:

- **Experiment-level log.** The main process writes to `experiment.log` with both file and console output.
- **Per-prompt logs.** Parallel workers write to isolated `plan.log` files with console output suppressed.
- **Consistent format.** All log entries follow `timestamp - logger_name - level - message`.

Combined with the SQLite session databases, this creates a full audit trail: for every prompt one can inspect the experiment log, the per-prompt log, and the designer/critic conversation history.

4.8 Prompt Management

Prompts are stored as YAML files with Jinja2 templates under a structured directory (`src/prompts/data/<agent`
A `PromptRegistry` provides type-safe access through domain-specific enums (e.g., `ProjectAgentPrompts.PLANN`
and enforces strict template-variable validation: the set of provided keyword arguments must exactly match the declared `template_variables`, preventing silent errors from missing or extra parameters.

Prompts are loaded and cached by a `PromptManager` with LRU caching, and rendered at runtime with the appropriate context (e.g., the user brief, a parent plan, or critic feedback).

5 Use Case: Agentic Project Management

To demonstrate how the framework looks in practice, an autonomous project-management pipeline is built on top of SprintAgent as a reference application. This use case is not the primary contribution; it serves to illustrate how practitioners can wire their own agents, tools, and domain logic into the scaffold. Given a natural-language brief such as *"Plan the frontend dashboard module, including UI design tasks and testing tasks,"* the system produces:

1. **Project plan** – High-level components, boundaries, and dependencies (`project_plan.md`).
2. **Module plan** – Module structure per component, refined from the project plan (`module_plan.md`).
3. **Task plan** – Concrete, assignable tasks per module (`task_plan.md`).
4. **Visual diagrams** – Mermaid-based hierarchy diagrams rendered to PNG and PDF for each plan level.

The pipeline is fully autonomous: the user provides a brief, and agents iteratively refine plans until quality criteria are met. Each stage consumes the output of the previous one, ensuring alignment from high-level strategy down to individual tasks.

The visualization stage is noteworthy because its critic is a vision-language model. After the designer produces a Mermaid diagram and the system renders it to an image, the VLM critic compares the rendered diagram against the source plan, checking for structural accuracy, completeness, and readability. This closes the loop between textual plans and visual artifacts.

Outputs are organized under `outputs/<timestamp>/<run>/prompt_<id>/`, with all plans, diagrams, logs, and session databases co-located for easy inspection.

6 Implementation Details

SprintAgent is implemented in Python and built on the OpenAI Agents SDK [6] for agent execution. Key technology choices include:

Table 2: Technology stack.

Component	**Technology**
Agent execution	OpenAI Agents SDK
Configuration	Hydra + OmegaConf
Session persistence	SQLite
Diagram rendering	Mermaid CLI (Node.js)
Parallel execution	`ProcessPoolExecutor`
Prompt templating	Jinja2 + YAML
Package management	`uv`

The codebase is organized as follows:

- `src/agent_utils/` – Shared agent infrastructure: base stateful agent, checkpointing, scoring, turn trimming, and image filtering.
- `src/{project,module,task,visualization}_agents/` – Domain-specific agent implementations.
- `src/experiments/` – Experiment orchestration and pipeline execution.
- `src/prompts/` – Prompt registry, manager, and YAML templates.
- `src/utils/` – Logging, parallel execution, OpenAI helpers, and Mermaid rendering.
- `configurations/` – Hydra YAML configs for agents, experiments, and pipelines.

7 Design Patterns

Table 3 summarizes the software design patterns employed in SprintAgent and their roles.

Table 3: Design patterns in SprintAgent.

Pattern	Location	Purpose
Template Method	`BaseStatefulAgent`	Domain-specific hooks for prompts and behavior
Decorator	`TurnTrimmingSession`	Wraps sessions with trimming/summarization
Strategy	Prompt registry, scoring	Pluggable prompts and scoring schemas
Factory Method	`_create_*_agent()`	Agent creation from configuration
Inversion of Control	Hydra + compatible agents dict	Config-driven agent wiring
Filter	`IntraTurnImageFilter`	Pre-call input filtering for token control
Abstract Base	`CritiqueWithScores`	Shared scoring interface across domains
N-1/N Checkpoint	Checkpoint state	Rollback on quality degradation

8 Discussion

As outlined in Section 1, building agentic systems demands deliberate engineering trade-offs. This section revisits each trade-off and discusses how SprintAgent addresses it.

Complexity vs. explainability. A recurring tension in agentic systems is between the power of complex multi-agent orchestration and the need for humans to understand what the system did and why. SprintAgent addresses this by making every agent interaction auditable (SQLite sessions), every decision logged (unified logging), and every quality judgment structured (scored critiques). The checkpoint mechanism further supports explainability: when a rollback occurs, the system records what degraded and what was restored.

Isolation vs. throughput. Process isolation provides strong fault containment but incurs overhead from process creation and inter-process serialization (all arguments must be picklable). For most agentic workloads, where LLM API latency dominates, this overhead is negligible. The framework also supports sequential execution for debugging and development.

Context growth vs. performance. Agents accumulate conversation history that, if left unmanaged, exceeds token limits and inflates API costs. SprintAgent balances context richness against performance through turn trimming (keeping only the last N turns intact), image stripping for older observations, and optional LLM-based summarization with hash-keyed caching to avoid redundant calls. The result is bounded context windows without losing the essential reasoning trace.

Configuration flexibility vs. simplicity. Real-world agentic systems have dozens of knobs—model selection, scoring thresholds, memory policies, pipeline stages—and the configuration system must scale without becoming its own problem. SprintAgent addresses this through Hydra's hierarchical YAML composition, cross-config interpolation, and custom OmegaConf resolvers. The Inversion of Control approach means that adding a new agent type, experiment, or scoring schema requires only YAML and a Python class—no changes to the orchestration core. This separation of wiring from logic reduces the cost of experimentation and lowers the barrier for new contributors.

Limitations. The current implementation is tightly coupled to the OpenAI Agents SDK for agent execution. Supporting alternative LLM providers would require an adapter layer. The checkpoint mechanism operates at the plan level and does not yet support fine-grained rollback of individual tool calls. Evaluation is currently qualitative; integrating automated benchmarks for plan quality is left for future work.

9 Related Work

Self-improving agents. ReAct [12] and Reflexion [7] introduced reasoning-and-acting loops and verbal self-reflection. SprintAgent's planner/designer/critic loop can be seen as a structured variant of self-reflection, where the critic provides quantitative scores and the planner makes rollback decisions based on score trajectories.

Configuration and IoC. The Inversion of Control pattern [4] is well-established in enterprise software but underutilized in LLM-agent frameworks. Hydra [3] and OmegaConf [11] are widely used in ML experiment management; SprintAgent adapts them for agent orchestration.

10 Conclusion

This paper presented SprintAgent, a production-ready framework for hierarchical autonomous agentic systems, developed with four core engineering trade-offs in mind: complexity vs. explainability, isolation vs. throughput, context growth vs. performance, and configuration flexibility vs. simplicity. The framework provides planner/designer/critic orchestration per agent, Inversion of Control-based configuration, SQLite session management with turn trimming, checkpoint auto-reset on degradation, process-isolated parallel execution, and unified logging—all driven by YAML configuration. Its modular design allows practitioners to plug their own logic into each module and extend on top. Through a demonstration application—an agentic project-management system built on the framework—the paper showed what it looks like in practice to build a complete hierarchical agentic application on SprintAgent, confirming that the scaffold is both functional and extensible.

Future work includes supporting alternative LLM backends, adding quantitative benchmarks for plan quality, extending the checkpoint mechanism to sub-agent granularity, and exploring distributed execution beyond single-machine process pools.

SprintAgent is open source and available at `https://github.com/alishams21/sprintagent`.

References

[1] Harrison Chase. LangChain: Building applications with LLMs through composability. 2023. https://github.com/langchain-ai/langchain.

[2] CrewAI. CrewAI: Framework for orchestrating role-playing autonomous AI agents. 2024. https://github.com/crewAIInc/crewAI.

[3] Facebook AI Research. Hydra – a framework for elegantly configuring complex applications, 2019. https://hydra.cc.

[4] Martin Fowler. Inversion of control containers and the dependency injection pattern. 2004. https://martinfowler.com/articles/injection.html.

[5] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. MetaGPT: Meta programming for a multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

[6] OpenAI. OpenAI agents SDK, 2025. https://github.com/openai/openai-agents-python.

[7] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366*, 2023.

[8] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 2024.

[9] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

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

[11] Omry Yadan. OmegaConf: A hierarchical configuration system, 2019. https://github.com/omry/omegaconf.

[12] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2023.