Architecture Analysis: tafreeman/prompts Monorepo

A deep-dive technical analysis of a multi-agent workflow orchestration system featuring two parallel execution engines, tier-based LLM routing with circuit breakers, a declarative YAML workflow definition language, and an integrated evaluation framework. This document examines the system's design decisions, architectural strengths, and areas requiring attention.

3

Python Packages

Independent monorepo packages

2

Execution Engines

LangChain + Native

5

Model Tiers

Tier 0–5 LLM routing

10

YAML Workflows

Declarative definitions

Package Structure & Monorepo Layout

The repository is organized as a monorepo containing three independent Python packages, each with its own build system, dependency tree, and runtime requirements. This separation of concerns enforces clear boundaries between the core runtime, the evaluation subsystem, and shared utilities — a pattern that aids independent versioning and release management, though it introduces cross-package integration complexity.

agentic-workflows-v2

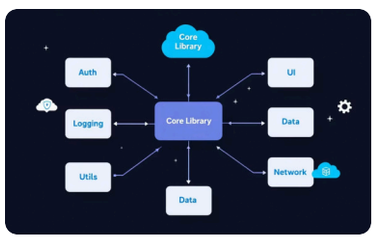
Python 3.11+ · hatchling

- Workflow execution runtime
- Agent orchestration layer
- FastAPI server
- React UI integration

agentic-v2-eval

Python 3.10+ · setuptools

- Scorers & evaluators
- Batch and async runners
- Multi-format reporters
- YAML rubric definitions



tools

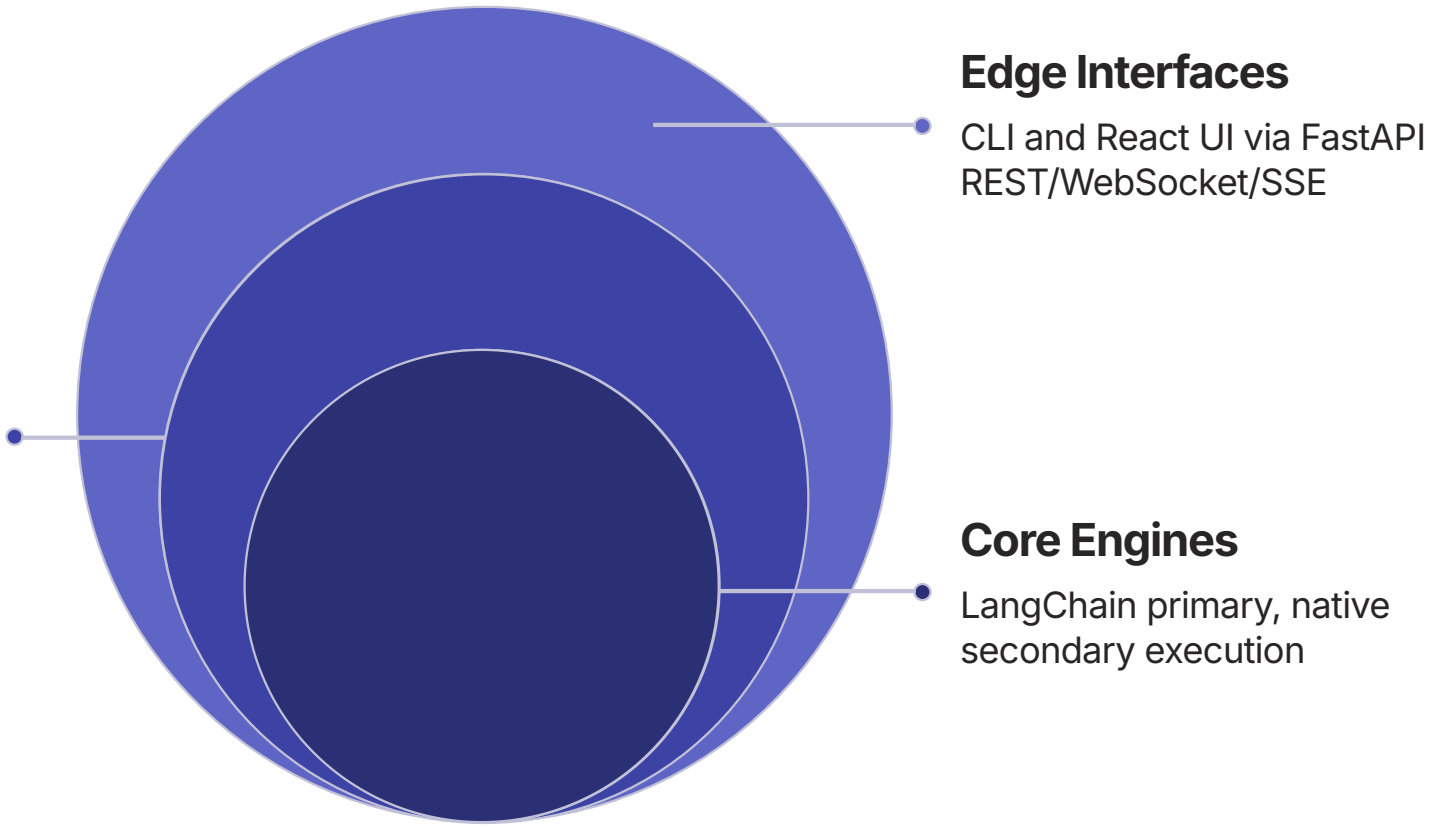
Python 3.10+ · setuptools

- LLMClient multi-backend dispatch
- 8 benchmark definitions
- Config & error classification
- Security gate for remote providers

The `tools` package serves as the shared foundation, with `LLMClient` providing a static `generate_text()` method that dispatches to 9+ backends via model name prefix routing. A notable security design is the opt-in gate for remote providers — they are disabled by default and require `PROMPTEVAL_ALLOW_REMOTE=1` to activate. The split between Python 3.10 and 3.11 minimum versions across packages is a mild inconsistency that could cause environment compatibility issues in CI pipelines.

System Architecture — Top-Down Flow

The system is structured as a classical layered architecture with clean separation across five distinct tiers. Requests originate at the interface layer, flow through the server into one of two execution engines, which then dispatch through the LLM routing subsystem to backend providers. This top-down discipline makes the control flow predictable and auditable, though the dual-engine reality introduces complexity at the execution tier.



The interface layer offers two entry points: a Typer-based CLI and a React 19 UI with React Flow DAG visualization. Both ultimately communicate through the FastAPI server layer, which exposes REST endpoints, WebSocket connections for live execution events, and SSE streams for run output. The `ConnectionManager` provides a WebSocket + SSE multiplex with a 500-event replay buffer, ensuring late-connecting clients can reconstruct execution state without data loss. This is an especially thoughtful design for long-running workflow executions.

Two Execution Engines

The most architecturally significant design decision in this codebase is the maintenance of two fully parallel execution engines. Each implements its own workflow compilation, expression evaluation, step execution, and concurrency model. Understanding the precise differences between them is critical for evaluating the refactoring options in the technical debt section.

PRIMARY

LangChain Engine

Entry: `langchain/runner.py WorkflowRunner`

- Used by CLI and API
- Compiles YAML → LangGraph StateGraph
- ReAct agent per node
- Parallel execution via LangGraph internals
- Reducer annotations for safe state merging
- Checkpointing via MemorySaver
- Graph cached by workflow name

SECONDARY

Native Engine

Entry: `engine/executor.py WorkflowExecutor`

- Library-only — not wired to CLI/API
- Kahn's algorithm DAG executor
- Dynamic scheduling via `asyncio.wait(FIRST_COMPLETED)`
- Mutable ExecutionContext (no reducer safety)
- No checkpointing
- SubprocessRuntime + DockerRuntime isolation
- Concurrency limiting, cascade skip, deadlock detection

The Native Engine's support for `SubprocessRuntime` and `DockerRuntime` is a uniquely valuable capability not present in the LangChain engine. If the Native Engine is deprecated, this isolation feature must be ported or explicitly abandoned. The LangChain Engine's use of `MemorySaver` for checkpointing, combined with immutable state merging via reducer annotations, gives it a meaningful resilience advantage for production workloads. The mutable `ExecutionContext` in the Native Engine is also directly implicated in the immutability violations identified in the technical debt analysis.

LLM Routing — Production-Grade Intelligence

The LLM routing layer is the most sophisticated subsystem in the codebase and represents genuine production-grade engineering. It implements a two-layer architecture with adaptive intelligence, circuit breaking, and multi-provider fallback chains — capabilities typically found in dedicated infrastructure libraries rather than application-layer code.

Layer 1: Provider Dispatch

`langchain/models.py` — Routes by model name prefix (`gemini:`, `openai:`, `claude:`, `gh:`, `ollama:`, `local:`). Candidate resolution follows a strict precedence chain: step override → env var → probed default → fallback chain → GitHub backup.

Layer 2: SmartModelRouter

`models/smart_router.py` — Implements a full circuit breaker (CLOSED → OPEN → HALF_OPEN) with health-weighted scoring: success rate 60%, latency 20%, recency 20%. EMA-smoothed latency tracking with atomic JSON persistence for durability across restarts.

Adaptive Cooldowns

Exponential backoff formula: `base × 1.5^consecutive_failures`, capped at 600 seconds. This prevents thundering-herd behavior against degraded providers while ensuring recovery is attempted within a bounded window.

The five-tier model hierarchy provides a principled cost-quality tradeoff framework. Tier 0 (deterministic, no LLM) is a particularly clever design: it allows parse and routing steps to bypass LLM invocation entirely for known-format inputs, reducing latency and cost on the critical path. The progression from Tier 1 (gemini-2.0-flash-lite) through Tiers 3–5 (gemini-2.5-flash) creates a natural fallback ladder that the router can traverse based on health signals.

Key Concern: Both `langchain/models.py` and `models/router.py` independently define fallback chain data. These definitions can diverge silently over time. A single source of truth (e.g., a shared YAML or constants module in `tools/`) is strongly recommended.

Workflow DSL & YAML Definition Language

The declarative YAML workflow system is one of the strongest design decisions in the codebase. By externalizing workflow logic into configuration rather than code, it enables rapid iteration on agent compositions without touching the execution runtime. The expression language is particularly well-designed — it supports dynamic data flow between steps while remaining readable to non-engineers reviewing workflow definitions.

Step Declaration Fields

name

Unique step identifier

agent

tier{N}_{role} convention

depends_on

DAG edge declarations

when

Conditional execution

loop_until

Bounded iteration

model_override

Per-step LLM selection

Expression Language

Three-scope reference system for data flow:

- `${inputs.code_file}` — workflow-level inputs
- `${steps.parse_code.outputs.ast}` — prior step outputs
- `${context.some_key}` — shared execution context

Conditional expressions are also supported in `when` fields, enabling branch logic purely in YAML without runtime modifications.

10 Workflow Definitions

- code_review, deep_research
- bug_resolution, fullstack_generation
- tdd_codegen_e2e, plan_implementation

The agent naming convention (`tier{N}_{role}`) elegantly encodes both the cost tier and the functional role in a single string. This allows the runner to resolve the correct persona prompt file from `prompts/{role}.md` and simultaneously know which model tier to target. The absence of schema versioning, however, is a notable gap — any breaking change to the YAML schema will silently corrupt existing workflow definitions with no detection mechanism.

Server Layer & React UI

The server and UI layers form a cohesive real-time application experience. The FastAPI backend exposes a well-structured API surface with both REST and streaming primitives, while the React frontend delivers a sophisticated DAG visualization with live execution monitoring. The dual-channel update architecture — WebSocket for in-flight runs, polling for completed run discovery — is a pragmatic engineering choice that balances real-time fidelity with operational simplicity.

FastAPI Routes

- `POST /api/run` — Execute workflow as background task
- `GET /api/workflows/{name}/dag` — DAG visualization data
- `GET /api/runs/{run_id}/stream` — SSE output stream
- `WS /ws/execution/{run_id}` — Live execution events

ConnectionManager

- WebSocket + SSE multiplex
- 500-event replay buffer for late clients
- Sub-second update latency
- 5s polling for completed run discovery

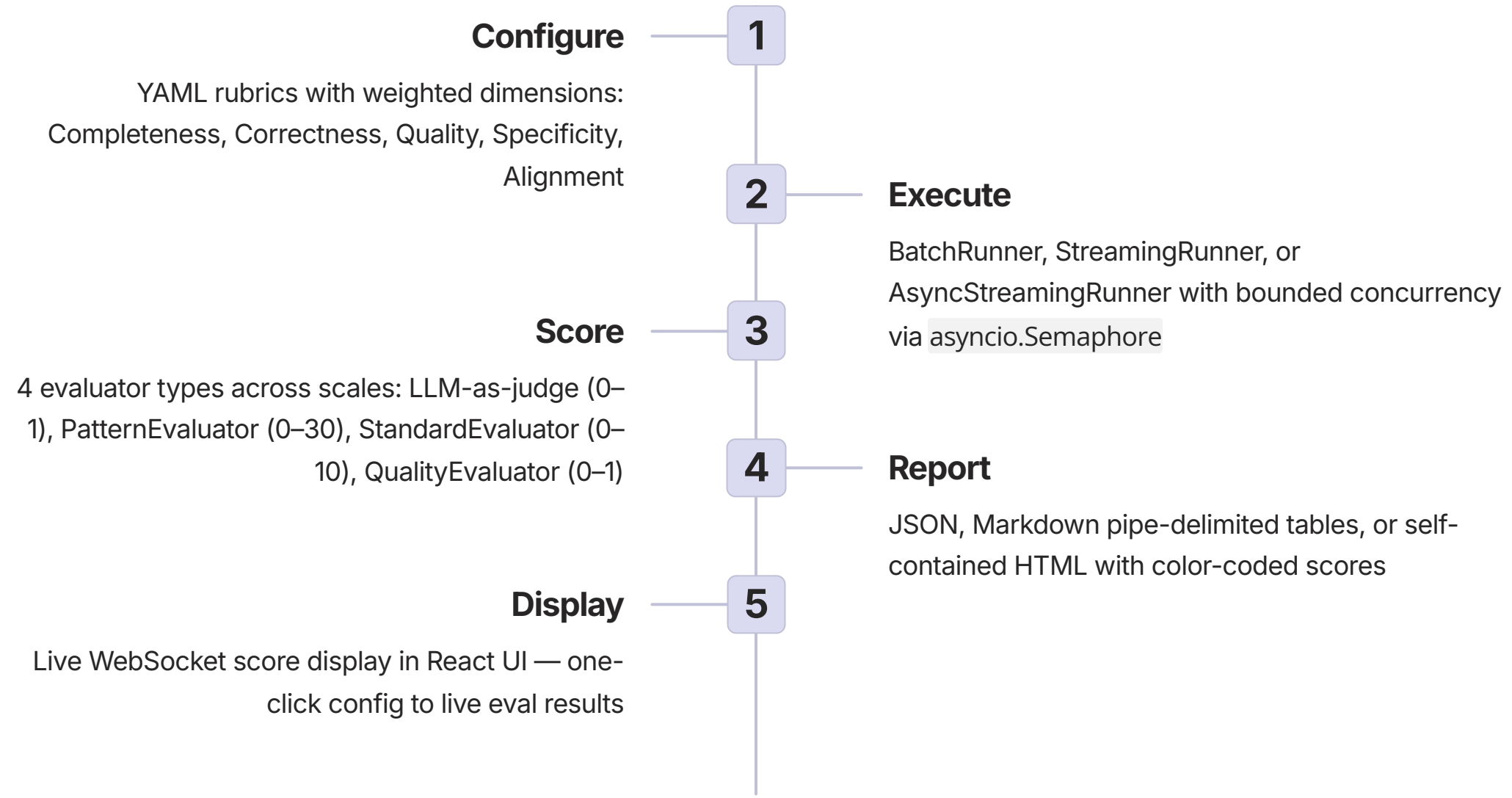
React DAG Visualization

- React 19 + Vite 6 + TypeScript 5.7
- React Flow 12 for DAG rendering
- Optimistic running status display
- Auto-pan to active step (user-interaction aware)
- Edge coloring by traversal state
- 250ms live elapsed timer per step

The optimistic running status display in the DAG visualization is an especially thoughtful UX decision — marking steps as "running" before the backend confirms prevents the UI from feeling laggy during high-latency LLM invocations. The user-interaction detection for auto-pan is also critical: without it, auto-panning would fight user navigation and create a disorienting experience. TanStack Query 5 manages the REST layer caching, while the 6-route structure covers the full workflow lifecycle from discovery to post-run evaluation.

Evaluation Framework

The evaluation framework is a first-class subsystem rather than an afterthought, providing a comprehensive LLM output quality measurement pipeline that is fully integrated into the run lifecycle. The combination of four distinct evaluator types — each targeting a different quality dimension — enables nuanced assessment that goes well beyond simple pass/fail metrics.



The `PatternEvaluator` is a standout component — evaluating agentic reasoning patterns like ReAct and Chain-of-Verification (CoVe) across 6 dimensions on a 0–30 scale goes significantly beyond standard output quality metrics. This enables detection of reasoning quality degradation, not just output format failures. However, the organizational placement of `evaluation.py` and `judge.py` under the `server/` directory rather than a dedicated `evaluation/` package represents a domain boundary violation that will complicate future modularization. The scale heterogeneity across evaluator types (0–1 vs 0–10 vs 0–30) also warrants a normalization layer for aggregate reporting.

Architectural Scorecard

Across eight evaluated dimensions, the system demonstrates particular strength in LLM routing intelligence, workflow DSL expressiveness, and evaluation integration — areas where deliberate design investment is evident. The scores below reflect both implementation quality and architectural soundness, with notes on where gaps exist.

95%

LLM Routing

Circuit breaker, health scoring, adaptive cooldowns, fallback chains — near-production-grade

85%

DAG Execution

Kahn's algorithm, cycle detection, cascade skip, deadlock detection. Penalized for dual-engine burden.

92%

Workflow DSL

Declarative YAML, full expression language, conditional execution. Penalized for missing schema versioning.

80%

Defensive LLM Handling

Robust normalization of freeform LLM outputs. Silent exception swallowing is a deduction.

70%

Observability

Pluggable TraceAdapter (OTel, console, file, composite). No structured metrics aggregation yet.

72%

Contract Design

Pydantic v2 with computed fields, but meaningful immutability violations in core types.

75%

Test Coverage

37 test files, autouse fixtures, async support. Coverage depth unknown without metrics.

88%

Eval Integration

End-to-end from YAML config to live WebSocket score display. Domain placement is a minor deduction.

Known Issues & Technical Debt

The issues below are ranked by severity. Two HIGH-priority items represent systemic architectural risks that will compound over time if unaddressed. The MEDIUM items are operational friction points. The LOW items are housekeeping concerns that should be addressed in the next refactoring sprint.

HIGH

Dual Engine Maintenance Burden

Two parallel implementations of workflow compilation, expression evaluation, and step execution mean every bug fix must be applied twice. The Native Engine is not wired to CLI/API, making it a maintenance liability without active consumer value. **Recommendation:** Deprecate the Native Engine or extract a shared core library. Preserve `SubprocessRuntime`/`DockerRuntime` as a standalone capability if needed.

HIGH

Immutability Violations

`TaskInput.with_context()`, `StepResult.mark_complete()`, and `WorkflowResult.add_step()` all mutate in place despite "Immutability First" being a stated non-negotiable design principle. This creates subtle concurrency bugs in parallel execution scenarios. **Recommendation:** Implement copy-on-write semantics using Pydantic's `model_copy(update=...)`.

MEDIUM

Empty Configuration Module + Global Singletons

`config/__init__.py` exports nothing — configuration is scattered across env vars, dataclass defaults, and hardcoded constants. Additionally, 6+ module-level singletons with `get_*()`/`reset_*()` patterns bypass the existing `ServiceContainer`. **Recommendation:** Centralize configuration in a `Pydantic BaseSettings` class and route all singletons through `ServiceContainer`.

MEDIUM

Silent Exception Swallowing

Multiple bare `except: pass` blocks violate the stated error handling standards. Silent failures are especially dangerous in an LLM orchestration system where routing fallbacks depend on accurate error signals from upstream calls. **Recommendation:** Replace all bare excepts with structured logging at `WARNING` or `DEBUG` level and re-raise or return typed error results.

LOW

Domain Misplacement + Duplicated Data + No Schema Versioning

Evaluation domain modules (`evaluation.py`, `judge.py`) live under `server/` rather than a dedicated `evaluation/` package. Fallback chain data is duplicated between `langchain/models.py` and `models/router.py`, risking silent divergence. No schema versioning mechanism exists for YAML workflow definitions, making breaking changes undetectable. These are low-urgency but should be addressed before the next major version boundary.