AiBuddy.software

LLM as Judge

How LLMs Judge: 4 Essential Patterns for Smarter Agent Workflows

The LLM-as-a-Judge pattern originated in eval pipelines to assess output quality, but has since expanded into agent architectures. It now plays a central role in autonomous workflows critiquing outputs, enforcing standards, and triggering rework steps to ensure runtime quality control.

Travis Frisinger

Jul 22, 2025 • 5 min read

AI-driven decision-making is at the edge of automation and intelligence.

As AI agents fragment into discrete, specialized components, we need a way to keep them honest—and that's where the "judge" comes in. A judge is simply an LLM (or other evaluator) wired into your agent pipeline to assess, score, or correct outputs before they move on. Think of it as a built-in QA step that can flag hallucinations, enforce style guidelines, or steer an agent back on track.

Despite its power, this judge role isn't yet a first-class citizen in most architecture docs. You'll find bits and pieces in research papers or scattered examples in SDKs, but there's no unified playbook. That gap leaves teams reinventing the wheel every time they need even a basic critique loop.

In this post, we'll close that gap by breaking the judge concept down into four battle-tested patterns. You'll see when each pattern shines, what trade-offs you're making (speed vs. oversight vs. autonomy), and how to slot a judge into your workflows cleanly—regardless of whether you're using Autogen, CrewAI, or rolling your own orchestration.

Read on for the four core judge patterns that will take your agent pipelines from "it sometimes works" to "it works reliably, every time."

1. Judge-Then-Rework (PDCA Loop)

This pattern treats your agent pipeline like a mini QA process. You let the agent draft an output, then hand it off to a dedicated judge LLM for critique before looping back for revisions—so each pass incrementally tightens quality and preserves a full audit trail.

When to Use: When you've got high-context, multi-step work (long docs, complex code, research reports) that benefits from iterative refinement and a clear audit trail.

Flow: Agent drafts → Judge critiques → Agent revises (loop)

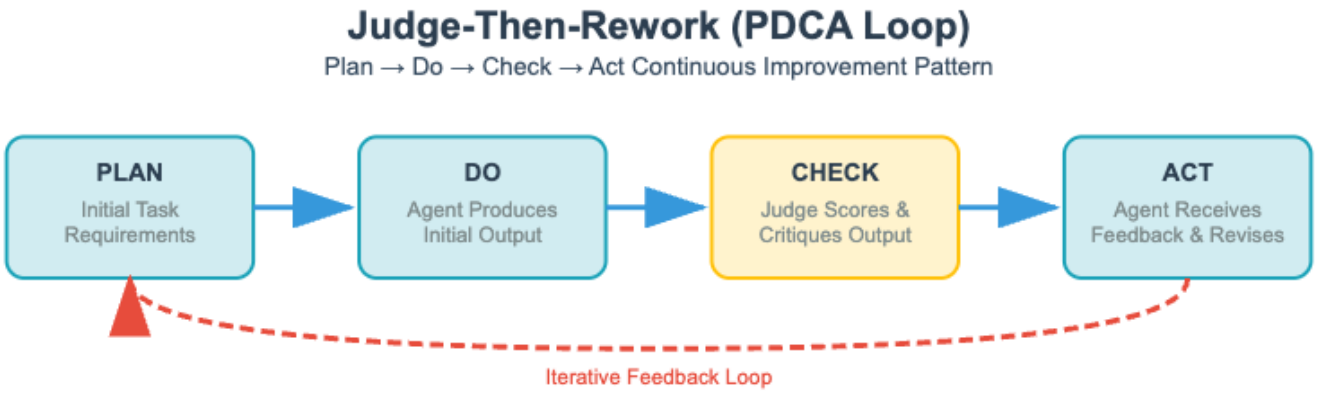

Figure 1: Judge-Then-Rework Pattern

Use Cases: Research, code generation, specification writing.

Why it Works: Maintains agent accountability and traceability. The agent retains ownership of the work, which can be important for multi-step or iterative reasoning.

Context Fit: Ideal for **high-context**, **iterative tasks** where state and reasoning accumulate over time.

Guidelines: Use a reasoning model for your judge.

Right about now you might be thinking: "What about ReAct loop (Reasoning and Acting) as it sounds similar to the PDCA loop above?"

A key difference here is the ReAct loops are about tool use and planning for better interactions between the user and the AI, where the PDCA is an architectural pattern for ensuring quality output in a workflow.

Why PDCA vs. ReAct Matters

ASPECT	PDCA (JUDGE-THEN-REWORK)	REACT ("THOUGHT→ACTION" L
Goal	Refine a draft over multiple passes	Choose and sequence exte
Structure	Separate "draft → judge → revamp" cycles	Alternating "Thought: … →
Best for	High-context tasks (long docs, multi-step code)	Tool-enabled tasks (data lo
State	Judge feedback persists across iterations	State lives in the prompt his
Overhead	More LLM calls but clearer audit trail	Leaner when you just need

> 💡 **Tip:** You can even nest a tiny ReAct loop inside the **Do** step of PDCA if your agent needs to hit an API mid-generation, then flow the result back to the judge.

2. Judge-as-Fixer (One-Shot Refinement)

Sometimes you just need a single tidy-up pass. Here, the judge LLM acts as an editor/fixer in one go—no loops, no state—making it perfect for quick grammar fixes, style tweaks, or transactional content polish.

When to Use: Quick polish or low-risk edits where you don't need iteration.

Flow: Agent outputs → Judge refines → Final deliverable

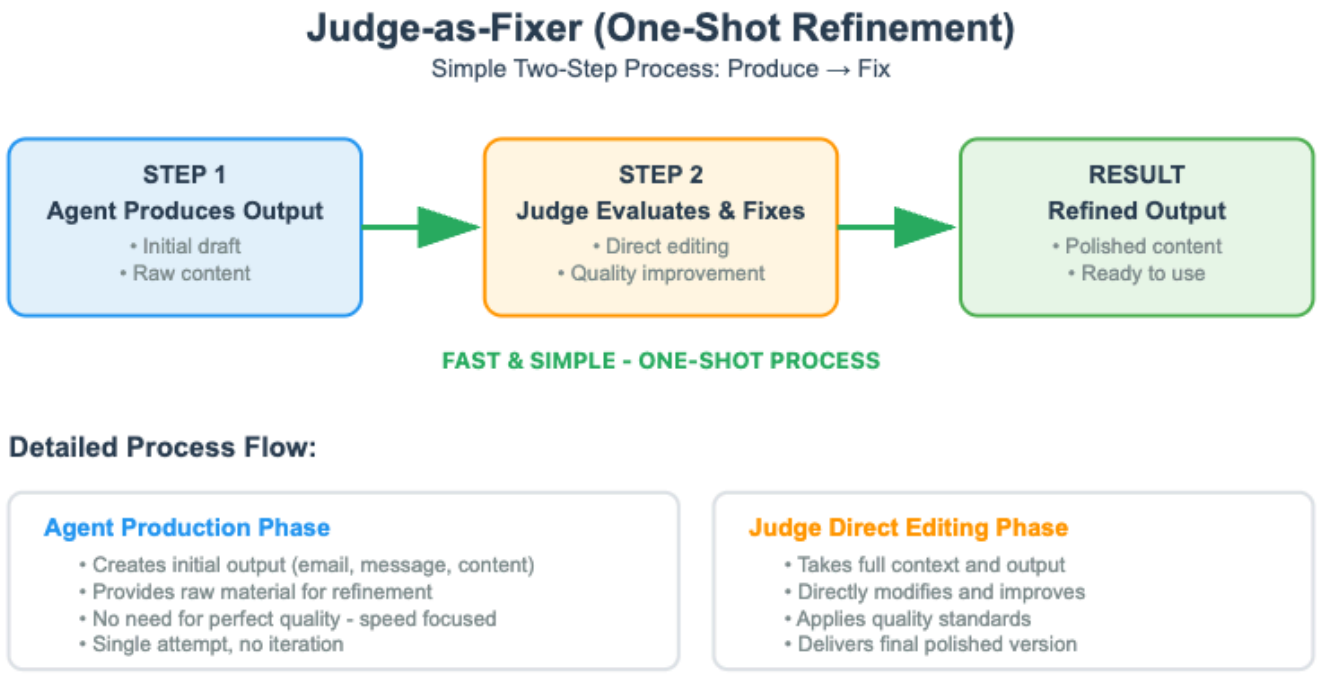

Figure 2: Judge-as-Fixer Pattern

Use Cases: Lead generation messages, cold outreach, content polishing.

Why it Works: Ideal for low-risk, transactional tasks where speed and simplicity are more important than agent autonomy.

Context Fit: Best for **low-context**, **single-pass tasks** where no state or memory is required.

Guidelines: Use a reasoning model for your judge.

3. Multi-Agent Vote (Arbitration)

Leverage the wisdom of the crowd—well, multiple LLMs. Fan out your prompt to several peer agents, then use a judge LLM to compare, rank, or even merge their outputs. This ensemble approach boosts creativity and guards against single-agent blind spots.

When to Use: Ideation, summarization, or any scenario benefiting from diverse hypotheses.

Flow: Agents propose in parallel → Judge ranks/selects/merges → Chosen output

Judging Modes:

- **Select Best**: Compare all output, select the highest quality.
- **Synthesize**: Extract best elements, merge relate ideas, create hybrid.
- **Rank & Explain**: Order by quality, provide reasoning, show trade offs.

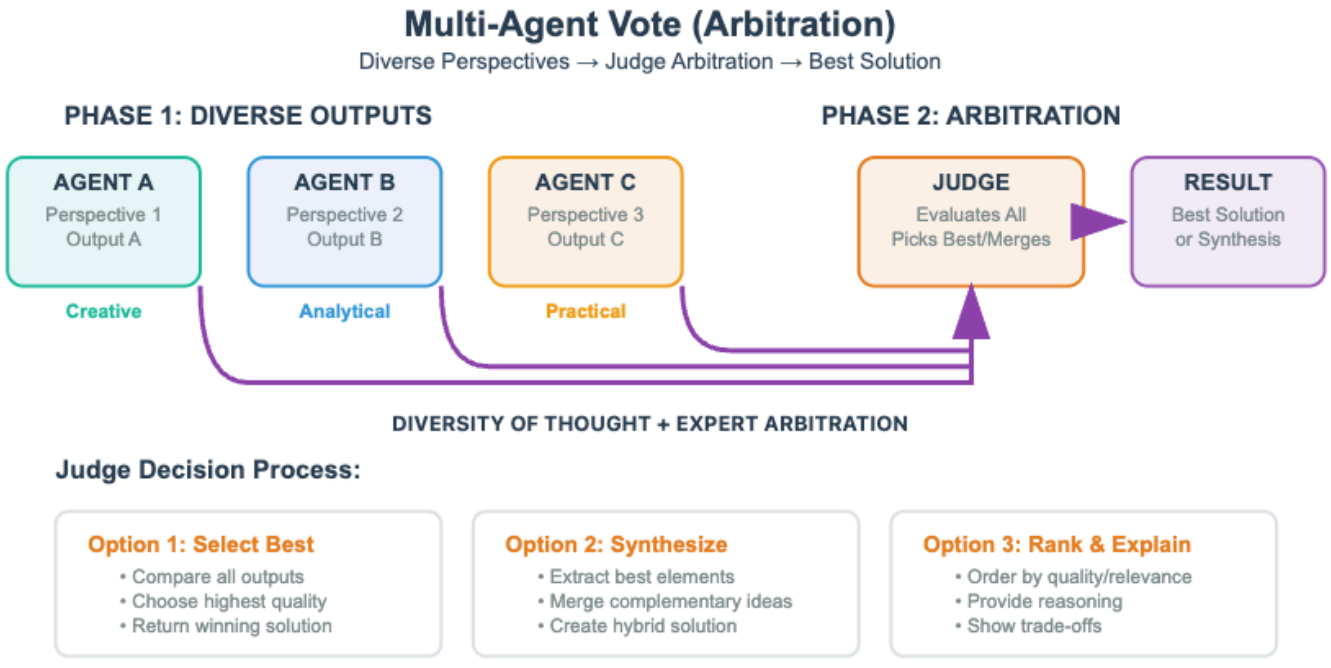

Figure 3: Multi-Agent Vote Pattern

Use Cases: Ideation, summarization, ensemble reasoning.

Why it Works: Encourages diversity of thought and leverages the LLM's ability to adjudicate conflicting viewpoints.

Context Fit: Useful when **different agent perspectives** contribute to solving a **non-trivial, high-context task**.

Guidelines: Use a reasoning model for your judge. Use temperature variance when creating the phase 1 agents to ensure diversity in perspective.

4. Self-Critique Loop (Internal Reflection)

When you want the agent to police itself, this pattern is your go-to. The agent generates, then immediately reflects on its own work and revises—all inside a single chained prompt—giving you lightweight alignment checks without a separate evaluator.

When to Use: Quick alignment checks, fine-tuning loops, or low-overhead quality guards.

Flow: Agent outputs → Agent self-critiques → Agent revises (*Logic, Accuracy, Clarity, Completeness)*

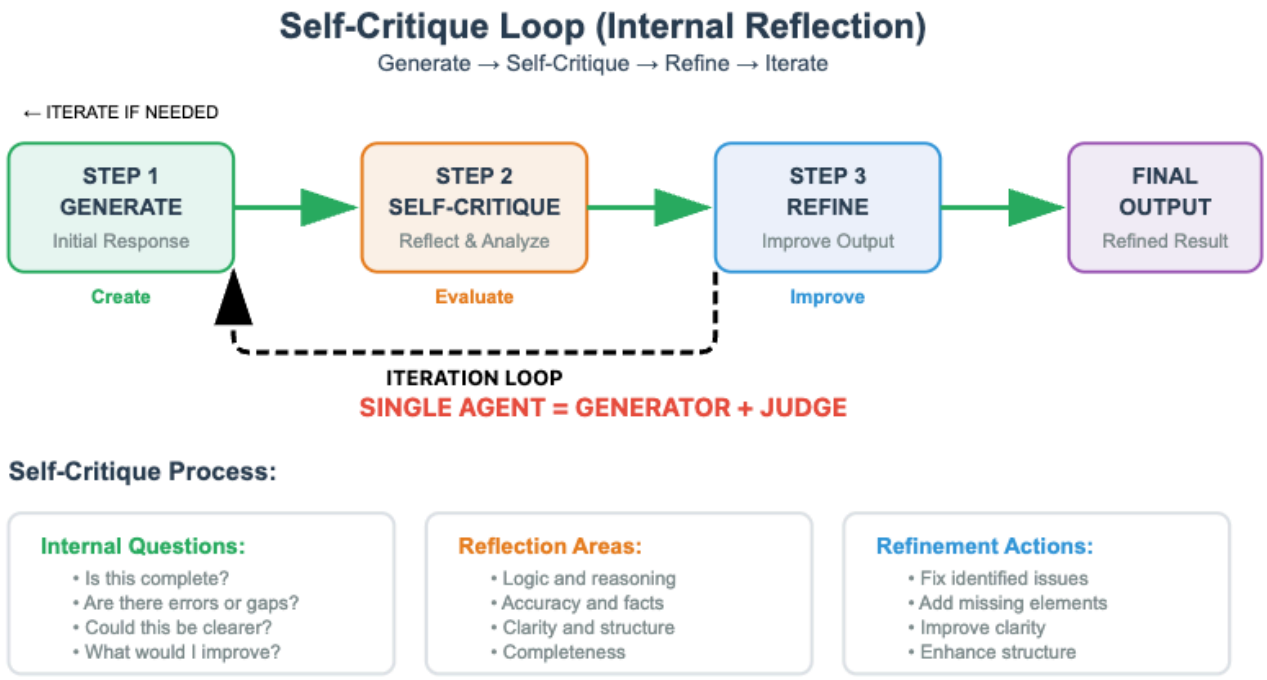

Figure 4: Self-Critique Pattern

Use Cases: Training loops, simple alignment tasks, fine-tuning pipelines.

Why it Helps: Lightweight and self-contained. Good for rapid iteration or embedding judgment directly into an agent's internal loop.

Context Fit: Effective for **low to medium complexity tasks** with **limited iteration depth**.

Guidelines: Critique may echo generation biases; limited by context window.

Closing Thoughts

Each of these judge loop patterns reflects a different balance between speed, autonomy, modularity, and oversight. Whether you're building autonomous researchers, scalable messaging pipelines, or self-healing code agents,

picking the right pattern can significantly influence the quality and traceability of your system's outputs.

An important dimension to consider is **context complexity**. Patterns like **Self-Critique** and **Judge-as-Fixer** work well for **low-context, single-shot tasks** where limited iteration or judgment is sufficient. In contrast, **Judge-Then-Rework** and **Multi-Agent Arbitration** shine when solving **larger, more complex problems** where **context accumulates**, and quality hinges on deliberate refinement and comparison.

Notably, while the idea of judgment in LLM workflows is gaining traction, it's not yet codified. If you're building agentic systems, understanding these patterns gives you a solid vocabulary and design space to work from.

Which judge loop are you using—and why? Let me know!

Better Bets, Not Bigger Burdens: Why "Agile+AI" Isn’t Delivering the Gains

Most teams are “doing agile” but really just running faster on old rails. Layering AI on to...

Sep 24, 2025 3 min read

Better Bets, Not Bigger Burdens: Shaping Delivery with Specs, Prototypes, and Agent Factories

Exploring how specs, prototypes, and agent factories are reshaping delivery in the AI er...

Sep 17, 2025 6 min read