The 3 Agentic Patterns That Actually Matter

(and 6 That Don't)

9 patterns are circulating. Most are noise. Here's what works when you're one person, not an enterprise with 50 engineers.

February 2026

v1.0

Florian Ziesche · Ainary Ventures

"Simplicity is the ultimate sophistication."

— Leonardo da Vinci

CONTENTS

1. Executive Summary

Nine agentic workflow patterns are circulating in the ecosystem right now. Most are academic exercises dressed up as production architecture. Three of them actually work for practitioners who ship alone.

- **Router-Specialist is the foundation.** Route tasks to domain-specific agents from a single orchestrator. This is our King → Agent model, and it's the only multi-agent pattern that scales down to one person. We run 7 specialist agents daily.[1][3][5] J
- **Planner-Critic-Executor is the missing quality gate.** Without a review layer between planning and execution, sub-agents produce ~20% error rates. Adding a critic step cut that in half in our experiments.[1][3][6] E
- **Reflection Loop is the cheapest quality multiplier.** Self-critique before finalization catches hallucinations, format errors, and logic gaps. It's not intelligence — it's risk reduction.[1][3][6] E
- **ReAct isn't a pattern — it's a primitive.** Every modern LLM already does reason-then-act. Calling it a "pattern" is like calling "typing" a software methodology. J
- **Tree of Thoughts, LATS, ReWOO, Debate/Consensus, and Plan-and-Execute** are theoretically sound, practically useless for solo operators. They require infrastructure, budget, or team sizes that individual practitioners don't have. J
- **The meta-pattern:** Start with one pattern. Add friction only where the system fails. Complexity is a cost, not a feature. J

Keywords: *Agentic Patterns, Multi-Agent Systems, Router-Specialist, Reflection Loop, Quality Gates, Solo Operator AI, Workflow Architecture*

2. The Landscape: 9 Patterns Everyone's Talking About

In early 2026, the agentic AI discourse converged on a set of recurring workflow patterns. Beam.ai published a comprehensive taxonomy of nine[1]. Dextralabs covered similar ground for enterprise use cases[2]. Rana's Medium piece organized them into four canonical categories: Reflection, Tool Use, Planning, and Multi-Agent[3]. Microsoft Azure and Google Cloud published their own architectural guides[4][5].

The convergence is real. The same patterns appear across independent sources. But convergence is not the same as usefulness. Just because everyone is talking about nine patterns doesn't mean you need nine patterns.

Exhibit 1: The 9 Agentic Workflow Patterns (2026 Landscape)

PATTERN	CORE IDEA	OUR VERDICT
ReAct	Reason + Act in alternating steps	PRIMITIVE
Plan-and-Execute	Strategic plan first, then tactical execution	SKIP
Planner-Critic-Executor	Plan → Review → Execute with quality gate	USE THIS
Reflection Loop	Self-critique and refine before finalizing	USE THIS
Tree of Thoughts	Explore multiple reasoning branches in parallel	SKIP
LATS	Language Agent Tree Search with scoring	SKIP
ReWOO	Reasoning Without Observation — plan around tools explicitly	SKIP
Router-Specialist	Route tasks to domain expert agents	USE THIS
Debate/Consensus	Multiple agents argue to best answer	SKIP

Sources: Beam.ai[1], Dextralabs[2], Rana[3], Microsoft[4], Google Cloud[5]

The question is not "which patterns exist?" but "which patterns work when you're one person running a personal AI system, not a team of 50 shipping enterprise software?" That's a filter most pattern taxonomies never apply.

Our filter has three criteria:

1. **Does it work at scale = 1?** One user, one AI system, limited budget.
2. **Does it pay for itself in daily use?** Not in a research paper. In production. Today.
3. **Can you implement it with markdown files and prompt engineering?** No custom scoring infrastructure. No multi-GPU clusters. No team of ML engineers.

Three patterns pass all three tests. Six don't. Let's start with the three that matter.

3. Pattern 1: Router-Specialist 85%

(Confidence: High — we run this daily)

Route tasks from a single orchestrator to domain-specific specialists. This is the only multi-agent pattern that scales down to one person — and it's the one we've built our entire system around.

What It Is

A central "router" agent receives all incoming requests, classifies them by domain, and delegates to specialized agents optimized for that domain. Beam.ai describes it as "routing work from a single entry point to the right domain expert" [1]. Rana calls the multi-agent supervisor pattern "the gold standard of 2026"[3]. Microsoft's architecture guide lists it as a core orchestration pattern[4].

The pattern is conceptually simple. The value is in the separation of concerns: the router doesn't need to know how to write, research, or analyze — it just needs to know *who* does.

Our Implementation: King → Agent

We run this as a King → Agent model with 7 active specialists[6]:

Exhibit 2: Our Router-Specialist Architecture

AGENT	ROLE	TRIGGER
🎯 HUNTER	VC Job Search	Applications, interviews, networking
✍️ WRITER	Content & Blog	Posts, articles, social media
🔬 RESEARCHER	Deep Dives	Research, fund analysis, market maps
🧱 OPERATOR	Systems	Automation, process optimization
💼 DEALMAKER	Freelance & Sales	Proposals, outreach, pricing
📊 ANALYST	Data & Metrics	Revenue, performance, goals
🧠 STRATEGIST	Thinking Partner	Big decisions, trade-offs, strategy

Source: Our AGENTS.md configuration[6]

Each agent has isolated context, specialized prompts, and domain-specific tools. The King (main agent) handles routing, maintains session state, and coordinates handoffs. This maps directly to the 5-Specialist Architecture documented in the OpenClaw community: "isolated sessions for complex tasks prevent context pollution"[7].

Why It Works for Solo Operators

Router-Specialist scales *down* as elegantly as it scales up. With one person:

- **Context isolation prevents confusion.** Your job search context doesn't bleed into your content writing.
- **Specialization is free.** Each agent's system prompt can be optimized for its domain without bloating a single monolithic prompt.
- **Routing is trivial.** Unlike enterprise systems that need labeled training data for routing[1], a solo operator's request domains are small enough that the LLM routes correctly with zero-shot classification.
- **It compounds.** Every specialist gets better over time as its domain-specific memory grows.

SO WHAT?

If you build one pattern, build this one. Start with 2-3 specialists (the domains where you spend most time). Add more only when you notice distinct task types bleeding into each other. The router doesn't need to be fancy — it needs to be consistent.

4. Pattern 2: Planner-Critic-Executor 78%

(Confidence: High — we measured the error rate)

Add a review layer between planning and execution. Without it, sub-agents produce approximately 20% error rates. With it, you cut errors in half. This is the quality gate most agent systems are missing.

What It Is

Beam.ai describes it as a workflow that "adds another review layer between planning and execution to ensure high-quality results"[1]. Rana frames planning as "cognitive load management" and emphasizes that "no long-running agent should operate without an explicit plan object"[3]. The Planner proposes a plan. The Critic reviews it for gaps, errors, and risks. The Executor carries out the approved plan.

This is not the same as Plan-and-Execute (pattern 2 in the Beam.ai taxonomy). Plan-and-Execute has no review step — it plans and runs. Planner-Critic-Executor adds the critic as a structural quality gate. The difference is the difference between "write a report" and "write a report, have it reviewed, then publish."

The Problem It Solves: Sub-Agent Error Rates

We observed this directly. Before implementing critic-style quality gates, our sub-agents had a roughly 20% error rate on complex tasks[6][8]. The failure modes were consistent:

- **Missing requirements.** Sub-agents would complete 80% of a task and silently skip the rest.
- **Context contamination.** Cached context from previous sessions led to stale data in output.
- **Format drift.** Output formatting would deviate from specifications over multiple iterations.

- **Confidence without accuracy.** Sub-agents would present incorrect information with high confidence.

The OpenClaw community documented identical patterns: "Sub-agents load cached context. Flag stale issues that were already fixed in main session."[7]

Our Implementation: The Self-Audit Gate

We added a mandatory self-audit step to every sub-agent task[6]:

OUR QUALITY GATE (FROM AGENTS.MD)

BEFORE COMPLETING: Audit your own output:
1. Re-read the original task requirements
2. Check every requirement against your output — what's missing?
3. If files were edited: verify no unintended changes (diff check)
4. If deploying: test the build locally first
5. Rate your confidence: <80% → flag what's uncertain

This is a simplified Planner-Critic-Executor: the sub-agent plays both planner and executor, while the audit step forces it to also play critic. It's cheaper than a dedicated critic agent but captures most of the value. For the OpenClaw community, a separate Editor agent with a scoring rubric and 40% rejection rate achieved even stronger quality control[7]. E

Cost-Benefit

Beam.ai warns that "the critical layer increases latency but ensures precision"[1] and recommends routing only high-value outputs through the critic. We agree. For quick tasks (Slack replies, calendar checks), skip the critic. For high-stakes output (research reports, code deployments, client-facing content), the critic step is non-negotiable.

SO WHAT?

If your agents produce output without a review step, you have a ~20% error rate and you probably don't know it. Add the self-audit gate to every sub-agent task. It costs one extra LLM call. It catches the errors that would otherwise silently degrade your system's reliability.

5. Pattern 3: Reflection Loop 82%

(Confidence: High — multiple independent sources confirm value)

Self-critique before finalization. It's not about making the AI smarter — it's about making it less wrong. Reflection is risk reduction, not intelligence enhancement.

What It Is

The agent generates output, then critiques its own output, then refines based on the critique. Beam.ai: "allows your AI agent to critique and refine its outputs before finalizing them"[1]. Rana states it plainly: "Reflection is not for intelligence. It is for risk reduction."[3]

The mechanism is simple: generate → evaluate → regenerate. The trick is knowing when to apply it and when to skip it.

When It's Worth the Cost

Rana's framework is useful here[3]:

Exhibit 3: When to Use Reflection

USE REFLECTION	SKIP REFLECTION
Code generation	Real-time latency paths
Legal / compliance text	Deterministic pipelines
RAG answers (fact-checking)	Simple lookups / routing
Financial logic	Chat responses where speed matters
Research reports	Low-stakes internal notes

Adapted from Rana[3] and Beam.ai[1]

Our Implementation: Structural, Not Aspirational

We implemented reflection at two levels:

Level 1: Sub-agent self-audit (described above in Planner-Critic-Executor). This is the minimum viable reflection — a checklist at the end of every complex task.

Level 2: Evening distillation. A nightly process reviews all daily logs and extracts only what would change future behavior in 30 days. This is reflection applied to memory, not just output — and it's what prevents our memory files from becoming junk drawers[6][7].

Our evolution experiment (SYNTHESIS-v2) validated this further. Across 10 experimental groups, the principle that "failures contain more information than successes" scored 8/10 in convergence[8]. The Integrity Engine — built on Red/Blue team review, belief graveyards, and anti-sycophancy counters — is essentially reflection applied at the system level[8].

The Compound Effect

Reflection is cheap individually — one extra LLM call per task. But the compound effect is significant. Rana's framework shows it: without reflection, you get hallucinations, silent errors, and non-deterministic output. With reflection, you get self-correction, explicit critique, and converging output quality[3].

Our SYNTHESIS-v2 experiment quantified this: the single most robust improvement signal was "corrections per session ↓" — and reflection is the primary mechanism that drives that metric down[8].

> SO WHAT?
>
> Add reflection to high-stakes output. Skip it for routine tasks. The implementation is trivial: append "Before finalizing, critique your output for errors, gaps, and hallucinations. Fix what you find." to any complex prompt. The cost is one extra generation. The payoff is measurably fewer errors over time.

6. The Other 6: Fair Assessment, Honest Verdict

These six patterns are not bad. They're not wrong. They're designed for contexts that don't apply to most practitioners. Here's a fair description of each and why we skip them.

ReAct — Reason + Act PRIMITIVE

What it is: The agent reasons about what to do, takes an action, observes the result, reasons again, acts again. Beam.ai calls it the foundation for "fast-moving tasks that need continuous thinking and acting"[1].

Fair assessment: ReAct was genuinely important when it was published (Yao et al., 2022). It demonstrated that LLMs could interleave reasoning and action effectively. It was a breakthrough.

Why we skip it as a "pattern": Every modern LLM already does this. Claude, GPT-4, Gemini — they all reason-then-act natively. Calling ReAct a "pattern" in 2026 is like calling "using a keyboard" an input methodology. It's the baseline, not a design choice. You don't "implement" ReAct. You use an LLM. J

Plan-and-Execute SKIP

What it is: Create a complete strategic plan upfront, then execute each step sequentially. Beam.ai recommends it for "report generation, research summaries, or data enrichment"[1].

Fair assessment: Useful when the problem space is well-defined and predictable. Enterprise data pipelines benefit from this structure. Dextralabs positions planning as essential for enterprise workflows[2].

Why we skip it: Plans are rigid. When your first sub-task reveals the problem is different than expected — which happens constantly in research, content, and exploratory work — the plan becomes a constraint, not a guide. Beam.ai acknowledges this: "Plans can be rigid and fail when conditions change"[1]. For a

solo operator doing varied creative and analytical work, adaptive execution beats rigid planning every time. J

Tree of Thoughts SKIP

What it is: Explore multiple reasoning branches in parallel before converging on the best answer. Like a chess engine that evaluates many moves ahead[1].

Fair assessment: Genuinely powerful for complex logical reasoning and creative problem-solving. The research results are real. For mathematical proofs, strategic planning, and complex code architecture, branching exploration can find solutions that linear reasoning misses.

Why we skip it: Cost. Beam.ai warns that "branching can multiply costs quickly" [1]. For a solo operator paying token costs out of pocket, running 5-10 parallel reasoning branches on every decision is economically irrational. The 80/20 applies: a single well-prompted reasoning chain gets you 80% of the quality at 10% of the cost. J

LATS — Language Agent Tree Search SKIP

What it is: Structured search over possible actions, guided by real-time tool feedback and scoring functions[1].

Fair assessment: Elegant architecture. The idea of using tool feedback as a scoring signal to guide search is theoretically sound and could produce superior results in well-instrumented environments.

Why we skip it: "Success depends on having strong scoring signals from those tools"[1]. That's the problem. Building reliable scoring infrastructure for your personal AI tasks — writing quality scores, research relevance metrics, decision quality ratings — is a full engineering project. LATS solves a problem (guided search) that requires infrastructure (scoring) that solo operators don't have and shouldn't build. J

ReWOO — Reasoning Without Observation SKIP

What it is: The agent explicitly plans its entire chain of tool calls upfront, referencing expected outputs before executing any of them[1]. Rana describes it

as best for "knowledge-oriented research tasks"[3].

Fair assessment: The transparency gain is real. ReWOO plans are auditable — you can see exactly which tools will be called and why before anything executes. For compliance-sensitive environments, this traceability has genuine value.

Why we skip it: Beam.ai is honest about it: "Slightly more setup effort, but you gain transparency and traceability"[1]. For a solo operator, that transparency isn't worth the setup overhead. You *are* the compliance department. You can audit your own agent's actions in the logs after the fact. Pre-planning every tool call adds friction without adding safety in a single-user context. J

Debate/Consensus Multi-Agent SKIP

What it is: Multiple agents argue from different perspectives, then converge on the best answer. Beam.ai recommends it for "high-stakes decisions that benefit from multiple perspectives"[1].

Fair assessment: The strongest use case for Debate is de-risking critical decisions — policy checks, risk assessments, major financial decisions. If you're an enterprise making a $10M decision, having three AI agents argue about it is cheap insurance.

Why we skip it: For a solo operator, you are the debate partner. You bring the domain expertise, the values, and the judgment. Having two AI agents argue with each other when both lack your context is theater, not de-risking. Beam.ai's own advice: "Only trigger consensus when confidence or compliance thresholds aren't met"[1]. For most solo decisions, a single well-prompted agent with a reflection loop achieves the same outcome at a fraction of the cost. J

Exhibit 4: Pattern Suitability by Context

PATTERN	SOLO OPERATOR	SMALL TEAM (3-10)	ENTERPRISE (50+)
Router-Specialist	✅ Essential	✅ Essential	✅ Essential
Planner-Critic-Executor	✅ High value	✅ High value	✅ High value
Reflection Loop	✅ High value	✅ High value	✅ High value
ReAct	— (built-in)	— (built-in)	— (built-in)
Plan-and-Execute	⚠️ Rarely	✅ For pipelines	✅ For pipelines
Tree of Thoughts	❌ Too expensive	⚠️ Selective	✅ For complex reasoning
LATS	❌ No infra	❌ No infra	⚠️ If scoring exists
ReWOO	❌ Overhead	⚠️ For compliance	✅ For audit trails
Debate/Consensus	❌ Overkill	⚠️ For critical decisions	✅ For risk management

Source: Author analysis based on S1-S8. J

7. The Meta-Pattern: Simplicity > Complexity

The most important pattern isn't in any taxonomy: start with one, add friction only where the system fails, and resist the temptation to architect for problems you don't have yet.

Our SYNTHESIS-v2 experiment — 10 groups, 33,000 words of transcripts, synthesized into one protocol[8] — discovered something no single group could see: the 24-hour testability filter eliminates more bloat than any other principle.

> THE 24-HOUR TESTABILITY FILTER (FROM SYNTHESIS-V2)
>
> Before adding ANY new protocol element, ask: "Can I measure whether this works within 24 hours?" If no → it's speculative. Either redesign it to be testable or don't implement it.

Applied to agentic patterns, this means:

1. **Start with Router-Specialist.** Can you measure routing accuracy within 24 hours? Yes — track how often the wrong agent gets a task. Start here.
2. **Add Reflection when errors appear.** Are you seeing hallucinations or format errors in output? Add a reflection step to the agents that produce them. Measure: did error rate drop within 24 hours?
3. **Add Planner-Critic-Executor for high-stakes tasks.** Are sub-agents missing requirements on complex tasks? Add the critic gate. Measure: did completion quality improve within one work session?
4. **Stop.** Three patterns. That's your system. If you find yourself reaching for Tree of Thoughts or Debate/Consensus, you're almost certainly solving the wrong problem. The issue is more likely prompt quality, context management, or task decomposition — not pattern sophistication.

Rana's "Architect's Golden Rules" converge on the same insight[3]: "Never trust a single-shot answer. State is more important than prompts. Tools beat tokens.

Reflection reduces risk." These are principles, not architectures. The patterns exist to serve the principles — not the other way around.

Our evolution experiment put it in systems terms[8]: watch for the "Limits to Growth" archetype — where growing competence creates complexity that eventually limits further growth. The fix is to expand capability *before* hitting limits, not to pre-build for every possible scenario. Simplicity is not laziness. It's discipline.

> **SO WHAT?**
>
> The best agentic architecture is the simplest one that handles your actual failure modes. Start with Router-Specialist. Add Reflection and Critic gates where errors appear. Resist adding more until you have evidence — not theory — that you need it. Complexity is a cost. Every pattern you add is a pattern you maintain.

8. Recommendations

If you're starting from zero

1. **Week 1:** Implement Router-Specialist with 2-3 domain agents. Define each agent's role in a markdown file. Route by keyword or intent.
2. **Week 2:** Add the self-audit gate to all agent outputs: "Before completing, re-read requirements and check every one against your output."
3. **Week 3:** Add reflection to your highest-error agent. Measure the change.
4. **Week 4:** Review. What broke? What worked? Adjust. Do *not* add new patterns.

If you're already running agents

1. **Audit your error rate.** If you're not tracking it, start. "Corrections per session ↓" is the single most useful metric[8].
2. **Add the critic gate to complex tasks.** The 40% rejection rate from the OpenClaw community's Editor agent is not a bug — it's a quality bar[7].
3. **Implement evening distillation.** A nightly process that reviews daily logs and extracts only what changes future behavior. This is reflection applied to memory, and it prevents the "junk drawer" failure mode[7].

If you're tempted by the other 6

Ask yourself: "What specific failure am I experiencing that this pattern would fix?" If the answer is "none yet, but it seems smart" — stop. You're pre-optimizing. The 24-hour testability filter exists for exactly this moment.

The one exception: **Plan-and-Execute** has legitimate value for well-defined data pipelines with predictable steps. If you're running the same 8-step research pipeline every week, planning that pipeline explicitly is good engineering. But that's a workflow, not a pattern — and you don't need a framework to do it.

Exhibit 5: Implementation Priority Matrix

PRIORITY	PATTERN	IMPLEMENTATION	TIME TO VALUE
1	Router-Specialist	AGENTS.md with role definitions + routing logic	Same day
2	Reflection Loop	Self-audit prompt appended to complex tasks	Same day
3	Planner-Critic-Executor	Quality gate checklist in sub-agent instructions	1-2 days
—	Everything else	Only if specific failure evidence demands it	Never (hopefully)

9. Methodology & References

Methodology

This report synthesizes 8 primary sources: 2 comprehensive pattern taxonomies (Beam.ai, Rana/Medium), 2 enterprise architecture guides (Microsoft Azure, Google Cloud), 1 enterprise consulting analysis (Dextralabs), and 3 internal sources (our AGENTS.md configuration, OpenClaw community research, and SYNTHESIS-v2 evolution experiment). The thesis — that 3 patterns matter for solo operators while 6 don't — is an opinionated judgment based on 4+ months of daily production use, not a controlled experiment.

Limitations: This analysis is written from the perspective of a solo operator running a personal AI system. The "skip" verdicts apply to that context. Enterprise teams, ML research labs, and compliance-heavy environments have different constraints — and several of the patterns we dismiss may be essential in those contexts. We tried to be fair to each pattern regardless of our verdict.

Conflict of interest: The author runs an AI consulting practice and has a commercial interest in agentic AI adoption. This report reflects genuine production experience, not marketing.

References

[1] Beam.ai, "The 9 Best Agentic Workflow Patterns to Scale AI Agents in 2026," beam.ai/agentic-insights, February 2026.

[2] Dextralabs, "Top AI Agentic Workflow Patterns Enterprises Should Use in 2026," dextralabs.com/blog, January 2026.

[3] D. Rana, "Agentic AI Design Patterns (2026 Edition)," Medium, January 2026.

[4] Microsoft, "AI Agent Orchestration Patterns," Azure Architecture Center, learn.microsoft.com, 2025-2026.

[5] Google Cloud, "Choose a Design Pattern for Your Agentic AI System," Cloud Architecture Center, docs.cloud.google.com, October 2025.

[6] F. Ziesche, AGENTS.md — Workspace & Agents Configuration, Ainary Ventures internal, February 2026.

[7] OpenClaw Community Research Compilation, openclaw-research-2026-02-17.md, sourced from r/ClaudeAI, r/openclaw, r/AI_Agents, r/LocalLLaMA, February 2026.

[8] F. Ziesche, "The Grand Synthesis v2.0 — Full-Transcript Analysis, 10 Groups × ~3,300 Words," SYNTHESIS-v2.md, Ainary Ventures internal, February 2026.

About the Author

Florian Ziesche is the founder of Ainary Ventures, an AI strategy and implementation practice. He builds the systems first, then writes about what works. This report is based on 4+ months of daily production use of multi-agent AI systems — not theory, not demos, not pitch decks.

AI strategy · research · implementation. By someone who built the systems first.

Ainary

AI Strategy · Research · Implementation

Want to implement agentic patterns that actually work?
Let's talk about what your system needs — not what's trending.

ainary.com · florian@ainary.com

AR-035 · February 2026 · v1.0