From Chatbot to Orchestrator: Five Phases of Agent System Evolution

Daniel Nakhla

Independent Researcher

This paper analyzes a personal autonomous infrastructure project built and operated outside the author's employer. No employer systems, employer data, or employer resources were used.

Abstract **-** Most agent papers describe architectures as if they arrived fully formed. In practice, useful systems evolve through failure: memory breaks, queues jam, scheduled maintenance starts to dominate real work, and multi-agent freedom creates duplicate execution. This paper documents that process with an evolution log from a personal autonomous infrastructure project operated from February 18 to March 27, 2026. During that 37.21-day span the system executed 5,079 jobs across 43 projects and 679 tasks, then moved through five phases: chatbot/Q&A, task executor, scheduled automation, multi-model coordinator, and what I call the Phase 5 Agent, an autonomous orchestrator with persistent workers, memory, and accountability. Because the underlying log records projects, tasks, jobs, workers, memory events, and incident remediation, each phase change can be tied to a dated failure and a specific architectural response. The result is a bottom-up model of agent evolution that complements existing top-down surveys and framework papers.

Index Terms **-** agents, orchestration, multi-agent systems, autonomy, memory, operations, software architecture

I. INTRODUCTION

There is a mismatch between how agent systems are published and how they are built. Papers and framework launches usually present a finished diagram: planner, memory, tools, evaluator, maybe a few agents talking to each other. Real systems do not look like that on the way up. They begin as a helpful assistant, then become a queue, then a scheduler, then an operations burden, and only later become something that can reasonably be called autonomous. In this paper, I use *evolution log* to mean a dated record tying architectural changes to operating evidence, failures, and corrective action.

This paper offers that record from a single long-running system. Between February 18 and March 27, 2026, I built and operated PennyBot, a personal autonomous infrastructure project for research, writing, operations, and project management. Over that 37.21-day span the system executed 5,079 jobs, with 4,756 done, 221 failed, 96 cancelled, 5 pending, and 1 running, while the database held 43 projects and 679 tasks at the cutoff [7]. Rather than jumping from prompt engineering to orchestration, the system moved through five distinct phases, each with a new control loop and a new failure class.

The contribution is straightforward. I define a five-phase model grounded in logged operating history, attach phase changes to explicit timestamps, analyze the incidents that forced those changes, and compare the resulting model to current agent frameworks. The labels are mine, but the dates and counts are not. Phase boundaries were inferred from logged milestones in projects, tasks, workers, and activity rows. Where the paper draws an inference rather than quoting a field directly, I state that explicitly.

II. RELATED WORK

Existing agent literature provides good components and weak histories. Weng's survey of LLM-powered autonomous agents breaks the problem into planning, memory, and tool use, which is still the cleanest conceptual decomposition for single-agent design [1]. AutoGen turns multi-agent conversation into a reusable software abstraction and shows how multiple agents, tools, and humans can cooperate through programmable dialogues [2]. MetaGPT pushes that idea toward structured role specialization by using an assembly-line paradigm for multi-agent software work [3].

More recent frameworks make the control problem more explicit. CrewAI distinguishes between crews, which emphasize autonomous collaboration, and flows, which emphasize event-driven control and state management [4]. LangGraph focuses on durable execution, human-in-the-loop intervention, comprehensive memory, and production deployment for long-running stateful agents [5]. These are meaningful advances because they recognize that useful agent systems need lifecycle support, not just chat between roles.

Failure analysis has also improved. The MAST study analyzes more than 1,600 traces across seven multi-agent frameworks and derives a 14-mode taxonomy spanning system design, inter-agent misalignment, and task verification [6]. That paper is valuable because it moves the conversation from raw

benchmark scores to structured failure classes. It also reinforces a practical point: coordination is often where agent systems fail, not basic language generation.

The cited surveys, framework papers, and failure taxonomies still do not provide a longitudinal account of one system changing shape under load. They explain how to build, what components exist, and what can fail after deployment. They do not publish the dated sequence by which a helper becomes an orchestrator. That narrower gap is the one this paper addresses.

TABLE I. OBSERVED PHASE BOUNDARIES AND PER-PHASE SCALE

Phase	Date Range (UTC)	Jobs Created	Tasks Created	Projects Created	Workers Created	Transition Trigger
1. Chatbot/Q&A	2026-02-18 to 2026-02-26	538	210	18	0	Architecture v2 project created on 2026-02-26
2. Task Executor	2026-02-26 to 2026-03-06	522	123	5	0	Multi-lane triage queue specified on 2026-03-06
3. Scheduled Automation	2026-03-06 to 2026-03-13	507	35	2	0	Recurring maintenance titles begin on 2026-03-13
4. Multi-Model Coordinator	2026-03-13 to 2026-03-19	1,345	117	5	0	Persistent worker rows appear on 2026-03-19
5. Autonomous Orchestrator (Phase 5 Agent)	2026-03-19 to 2026-03-27	2,167	187	7	1,178	Open phase through latest recorded job

III. SYSTEM DESCRIPTION

PennyBot was built as personal infrastructure, not as a benchmark harness. Its job was to move real work: research, writing, inbox management, project tracking, publishing, and maintenance. The control plane centered on a SQLite database, `pm.db`, with tables for projects, tasks, jobs, messages, activity, facts, and references [7]. The orchestrator wrote planned work into that database, workers executed bounded jobs, and the orchestrator reconciled outputs back into project state. Once multiple workers existed, that ownership boundary became essential.

By March 27 the system was well past toy scale. Weekly job volume grew from 243 jobs in ISO week 07 to 1,767 jobs in week 11 before settling at 1,254 jobs in the partial week 12 snapshot [7]. Model usage was heterogeneous: 2,913 Opus jobs, 887 Sonnet jobs, 263 Haiku jobs, 171 GPT-5 Nano jobs through OpenCode, 141 Moonshot Kimi K2.5 jobs, 101 Gemini 2.5 Pro jobs, and smaller experimental routes [7]. This was not a single-model chatbot wearing more wrappers. Routing across model families became part of the architecture.

The system also had layered memory. The first day exposed a LanceDB schema bug in long-term memory. Daily memory consolidation began on March 1, nightly memory checks on March 2, and explicit core-memory compaction on March 19 [7]. The result was a hybrid design: vector-backed archival memory, compact core memory for identity and priorities, and procedural rules stored as operating constraints. That stack matters because later phases assume context survives beyond the active session.

One more number matters: failure. Across all jobs, 93.64% were done, 4.35% failed, 1.89% were cancelled, and the remaining 0.12% were still pending or running at the cutoff [7]. Those aggregate numbers hide a sharper pattern. Failure rates were highest in phases 2 and 3, then dropped once review gates, maintenance loops, and persistent worker management were added. In other words, the system got more complex and more reliable at the same time because the added complexity mostly improved control rather than raw task generation.

IV. THE FIVE PHASES

I use "phase" in a practical sense: a phase is the dominant control pattern of the system during a given window. Phase boundaries are therefore tied to concrete architectural milestones, not to aesthetic differences in prompts. Each phase below includes its main characteristics, visible capabilities, failure modes, and the trigger that forced the next move.

A. Phase 1: Chatbot/Q&A

Phase 1 ran from the first logged jobs on February 18, 2026 through the creation of the Architecture v2 project on February 26. In that window the system created 538 jobs, 210 tasks, and 18 projects, with no persistent workers [7]. The first fifty jobs are direct asks: draft an outline, research a role, write an email, search for hotels, write an article. The pattern is breadth over continuity. The system can answer, draft, brainstorm, and search, but every request assumes a human is still stitching the work together.

The main capability in Phase 1 is one-shot usefulness. The weakness is that there is no durable execution layer. Progress exists only insofar as the active thread carries it. When long-term memory broke on February 18, the bug was fixed the same day, but the larger lesson was that memory had already become a system dependency, not a nice-to-have [7].

The visible failure mode in this phase is hidden coupling to the operator. Phase 1 looked stable on paper because its inferred failed-job rate was only 1.86%. That number is misleading. The system avoided many failures simply by not attempting long-running autonomous work yet. The real limit was that every unresolved task still depended on human recall, human routing, and a single active session.

The transition trigger was explicit and dated. On February 26, project #28, *PennyBot Architecture v2*, was created, followed by tasks to map main-thread blocking points and design an async triage -> queue -> engine -> poll flow [7]. That is the moment the system stopped being just a helper and started becoming a work dispatcher.

B. Phase 2: Task Executor

Phase 2 ran from February 26 to March 6 and added an asynchronous execution model. The system created 522 jobs, 123 tasks, and 5 projects in this span [7]. The important change was durable queued execution. Work could now be represented as jobs in a queue, executed outside the main conversation, and polled later. That sounds basic, but it marks the difference between "answer this for me" and "own this task until it is done or blocked."

Capabilities increased immediately. The log fills with continuation jobs, scheduled email checks, architecture design jobs, and longer implementation arcs [7]. Work can now survive across turns. The agent is no longer only producing text. It is managing bounded work units.

The downside is that a single queue is still a blunt instrument. Heterogeneous work competes for the same path, and urgent items have no principled way to preempt background items. The inferred failed-job rate rose to 8.43% in this phase because the system was attempting more ambitious tasks without strong scheduling discipline, mature memory routines, or a quality gate.

The transition into Phase 3 was driven by queue pressure. On March 6, task #395 introduced a multi-lane triage queue with priority preemption [7]. That change reflects a new truth: once work is queued, the next bottleneck is prioritization rather than generation.

C. Phase 3: Scheduled Automation

Phase 3 ran from March 6 to March 13. It logged 507 jobs, 35 tasks, and 2 projects [7]. By this point the system had moved beyond async execution into recurring internal work. Daily memory consolidation had already started on March 1, and nightly memory checks on March 2; during this phase those routines became part of the normal operating rhythm [7]. The system was now doing work because time had passed, not only because a user asked.

This phase brought two capabilities: multi-lane scheduling for urgent versus background work, and scheduled memory routines that gave the system limited continuity across days. Job volume rose sharply in week 10 to 883 jobs, with more diversified model routing and more shell-driven execution [7].

Phase 3 also produced the clearest sign that automation was outrunning judgment. On March 12, the Leni's World incident exposed that tasks were being marked done without a mandatory second-model review [7]. This was a control failure, not a prompt failure. The system had learned how to move work faster but had not yet learned how to stop itself from closing work too early. The inferred failed-job rate peaked here at 10.26%.

The architectural response came almost immediately. On March 12 the engine prompt was updated to require a second-pass review before tasks could be marked done [7]. On March 13, recurring maintenance titles began. That date marks the start of a new phase because the system had accepted that self-monitoring was now part of the workload.

D. Phase 4: Multi-Model Coordinator

Phase 4 ran from March 13 to March 19 and expanded faster than any previous phase. The system created 1,345 jobs, 117 tasks, and 5 projects in less than a week [7]. Recurring status snapshots, inbox checks, pulse checks, and project-advancement jobs all became routine here. I still label it the *multi-model coordinator* phase because model heterogeneity became deliberate at the same time the system started maintaining itself.

The evidence is visible in the weekly routing mix. In week 11 the system used Opus, Sonnet, GPT-5 Nano through OpenCode, Moonshot Kimi K2.5, Gemini 2.5 Pro, Haiku, Codex, and smaller routes, all in meaningful volume [7]. The orchestrator was no longer asking one model to be everything. Coordinating across models had become part of the design.

Phase 4 introduced a hard cost that later became the subject of a second paper: maintenance. Exact recurring maintenance titles accounted for 776 of the 1,345 jobs in this phase, or 57.7% [7]. The ratio reflects what happens when a system

graduates from task execution to ongoing coordination. Health checks, snapshots, inbox triage, and project review become structural requirements.

The key failure in this phase was a portal rebuild overlap on March 18. Overlapping plan fan-out created competing rebuild paths and zombie retries for the same target [7]. Again, this was not a raw model failure. It was a coordination failure caused by insufficient rules around plan ownership and retry behavior. The fix was a hard plan-step deduplication rule added to the orchestration skill on March 19 [7]. That rule is pure coordination infrastructure, which is why this incident belongs here.

E. Phase 5: Autonomous Orchestrator

Phase 5 began on March 19, 2026, when persistent worker rows first appeared, and it remains open in the current snapshot. In the first 8.6 days of this phase the system created 2,167 jobs, 187 tasks, 7 projects, and 1,178 worker rows [7]. The swarm included 619 Claude workers, 325 OpenCode workers, 223 Codex workers, and 12 Gemini workers [7]. This is the point at which the system stopped acting like a queue with helpers and started behaving like an operating organism.

I use the term *Phase 5 Agent* for a system that owns routing, persistent execution, memory, review gates, maintenance schedules, and an auditable explanation of why work exists. Many systems can launch agents. Far fewer can explain which project a job belongs to, who owns the decision, what review gate applies, and which worker should resume tomorrow.

The system can now keep several active projects moving at once, supervise persistent workers across backends, surface health state to a dashboard, and formalize engine reliability work as its own project [7]. The inferred failed-job rate fell to about 3.3%, less than one-third of the Phase 3 peak, because the control loops got stronger.

The remaining failure mode is epistemic rather than mechanical. The Media Framing study was pushed toward distribution before literature review and deeper research had been completed, leading to a build-before-research reversal and a new research-first rule written into core memory [7]. In Phase 5, the risk is acting in the wrong order at scale rather than failing to act at all. That distinction matters for Phase 6.

TABLE II. TRANSITION FAILURES AND CORRECTIVE ACTION

Transition Context	Logged Incident	Root Cause	MAST Lens	Corrective Action
Phase 1 -> 2	Memory schema failure made long-term recall an explicit system problem on 2026-02-18	State was implicit and brittle; the helper was already depending on external memory without treating it as core infrastructure	System design issue	LanceDB schema repaired the same day; Architecture v2 work started on 2026-02-26
Phase 2 -> 3	Async execution scaled task count but exposed single-lane queue pressure	No priority preemption for heterogeneous work; the system could execute jobs but not yet schedule them well	System design issue	Multi-lane triage queue introduced on 2026-03-06
Phase 3 -> 4	Leni's World tasks were marked done without second-pass review on 2026-03-12	Throughput was rewarded more than verification	Task verification failure	Mandatory second-pass review added to engine prompts on 2026-03-12
Phase 4 -> 5	Portal rebuild overlap and zombie plan fan-out on 2026-03-18	Overlapping execution paths and no sibling-step cancellation rule	System design and coordination failure	Plan-step deduplication hard rule added on 2026-03-19
Within Phase 5	Media Framing build-before-research reversal on 2026-03-15 to 2026-03-16	The system could publish faster than it could establish epistemic grounding	Task verification and governance failure	Research-before-building rule written into core memory

V. FAILURE ANALYSIS

Several patterns emerge when the incidents are read as a sequence rather than as isolated bugs. Failures move upward in abstraction as the system matures. Early failures are storage and session failures. Mid-stage failures are workflow failures such as poor prioritization and premature completion. Late failures are governance failures: duplicated execution, bad ordering of work, and maintenance burden. Capability at one layer simply exposes weakness at a higher layer.

Every successful transition added a control loop rather than a larger model. Phase 2 added execution, Phase 3 added scheduling, Phase 4 added maintenance, and Phase 5 added persistent workers plus accountability. Those loops explain why the system got more reliable after Phase 3 even while handling more work and more backends. Inferred failed-job rates rose from 1.86% in Phase 1 to 8.43% in Phase 2 and 10.26% in Phase 3, then fell to 3.27% in Phase 4 and about 3.28% in Phase 5 once coordination controls were in place.

The system also did not suffer a major direct inter-agent message-loop incident of the type highlighted in MAST [6]. I interpret that absence cautiously, but it supports the hub-and-spoke design choice. Workers wrote artifacts and returned them to an orchestrator. That constraint reduced a class of inter-agent misalignment at the cost of higher orchestration load. In this system, that trade was worth it. Maintenance is the other side of that choice: exact recurring maintenance jobs accounted for 57.7% of Phase 4 volume and 57.54% of Phase 5 volume [7].

TABLE III. WHERE EXISTING FRAMEWORKS FIT IN THE FIVE-PHASE MODEL

Framework	Primary Abstraction	Native Phase Fit	Ceiling Without Extra Systems	What Is Still Missing
AutoGen [2]	Programmable multi-agent conversation	Phase 2 to early Phase 3	Phase 4	Persistent accountability, maintenance accounting, project-state ownership
CrewAI [4]	Crews for autonomy, flows for event-driven control	Phase 2 to Phase 3	Phase 4	Long-running governance, operator memory policy, auditable work provenance
LangGraph [5]	Durable graph execution with memory and human checkpoints	Phase 3 to Phase 4	Near Phase 5	Worker organization, task ownership boundary, maintenance economics
MetaGPT [3]	Assembly-line role specialization	Phase 3 to Phase 4	Phase 4	General project management, external operations, self-maintenance loops
MAST [6]	Failure taxonomy, not a runtime	Best used across Phase 4 and Phase 5	Not applicable	Longitudinal evolution sequence and operational economics

VI. COMPARISON WITH EXISTING FRAMEWORKS

AutoGen, CrewAI, LangGraph, and MetaGPT all fit inside the five-phase model, but they fit at different layers. AutoGen is closest to the Phase 2 idea that work can be decomposed into cooperating conversational agents [2]. That is a real step beyond the chatbot phase, but conversation alone does not create accountability, scheduling discipline, or maintenance awareness. CrewAI sits slightly farther along because its split between crews and flows mirrors a practical split between autonomy and control [4]. Crews map to the work pattern of Phase 2 and Phase 3. Flows begin to formalize Phase 4 behavior.

LangGraph is the nearest existing substrate to the Phase 4 to Phase 5 boundary. Durable execution, human-in-the-loop intervention, and long-term memory are exactly the primitives that mattered once PennyBot stopped being a task executor and became a coordinator [5]. I still do not classify LangGraph itself as a Phase 5 Agent because a graph runtime is infrastructure, not an operating identity. A Phase 5 Agent needs a stable notion of who owns work, why work exists, what review standard applies, and how maintenance jobs are justified.

MetaGPT fits cleanly into Phase 3 to Phase 4 because its assembly-line paradigm and role assignment are strongest when the domain resembles structured production [3]. MAST sits differently because it is not a runtime but an analytic lens [6]. The two models are complementary: MAST says how multi-agent systems fail, while the phase model says when those categories of failure first become likely and why.

VII. DISCUSSION

A. Predicting Phase 6

Phase 6 is not "more agents." It is a governed organization. If Phase 5 is an autonomous orchestrator with persistent workers and accountability, Phase 6 is an orchestrator that can manage its own policy surface: typed peer channels, explicit budget allocation, experiment governance, automatic retirement of low-yield maintenance work, and stronger separation between epistemic work and execution work. The defining shift is from owning tasks to owning institutional rules.

That prediction follows directly from the incidents above. By Phase 5, the limiting problems were not lack of task execution. They were duplicated execution, premature publication, and maintenance economics. A Phase 6 system therefore needs formal mechanisms for approving routes, constraining peer collaboration, evaluating maintenance yield, and deciding when not to act.

B. Practical Implications

For practitioners, the pattern points to a small set of habits. Keep the evolution log from day one so incidents, fixes, and new rules stay linked over time. Add quality gates before widening agent-to-agent freedom, because the Leni's World and portal-rebuild failures were solved by tighter control rather than wider delegation. Report maintenance as part of system cost as well, because once later phases spend more than half their jobs on coherence, that workload belongs in the architecture rather than in a footnote.

There is also an evaluation implication. Many teams still assess agent systems by prompt quality or benchmark completion. That misses the harder question: can the system

own work over time without confusing itself, closing work too early, or multiplying unseen maintenance? A more useful signal may be whether the team can produce an honest evolution log.

VIII. CONCLUSION

This paper documented one agent system's path from helper to orchestrator across a 37.21-day span, 5,079 jobs, 43 projects, and 679 tasks. The core result is that agent evolution follows a control sequence. The system first learns to answer, then to execute, then to schedule, then to coordinate across models, and finally to orchestrate persistent work with memory and accountability. Each stage solves a real problem and introduces a new failure class.

The main practical lesson is simple. Production autonomy starts when you can explain, from the log, why a job exists, who owns it, what review gate it must pass, which worker should do it, and what rule changed after the last failure. That is what an evolution log makes visible. Without it, final architectures are just cleaned-up stories.

REFERENCES

1. L. Weng, "LLM-Powered Autonomous Agents," *Lil'Log*, Jun. 23, 2023. [Online]. Available: https://lilianweng.github.io/posts/2023-06-23-agent/
2. Q. Wu *et al.*, "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation," arXiv:2308.08155, Oct. 2023. doi: 10.48550/arXiv.2308.08155.
3. S. Hong *et al.*, "MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework," arXiv:2308.00352, Nov. 2024 version. doi: 10.48550/arXiv.2308.00352.
4. crewAIInc, "CrewAI," GitHub repository, accessed Mar. 27, 2026. [Online]. Available: https://github.com/crewAIInc/crewAI
5. langchain-ai, "LangGraph," GitHub repository, accessed Mar. 27, 2026. [Online]. Available: https://github.com/langchain-ai/langgraph
6. M. Cemri *et al.*, "Why Do Multi-Agent LLM Systems Fail?," arXiv:2503.13657, Oct. 2025 version. doi: 10.48550/arXiv.2503.13657.
7. D. Nakhla, "PennyBot operating log (`pm.db` export), February 18-March 27, 2026," personal dataset and event log, Mar. 27, 2026.