Autonomous Agents as a Self-Conversing Chat Pipeline

An architecture paper on ATAPIC's agent automation system

Enahoro Omoarukhe

Founder & Lead AI Engineer, ATAPIC

TECHNICAL WHITE PAPER • v1.0 • JUNE 2026

ABSTRACT

Most "AI agent" features ship as a separate subsystem: a bespoke pipeline with its own prompts, its own action code, and its own scheduler, bolted onto a product that already has a perfectly good conversational assistant. That duplication is where agents rot. The agent drifts behind the chat, does less, and behaves differently.

ATAPIC takes the opposite stance: an autonomous agent is the existing chat assistant talking to itself on a timer. The same model call, the same action protocol, and the same services power both a human conversation and an unattended agent run. A thin orchestration layer turns "the AI replied with an action" into "run the action, feed the result back, and loop until the goal is done." On top of that loop we add the four things a year-long autonomous agent needs that a single chat turn does not: structured planning with self-correction, cross-run memory, one authoritative scheduler, and a durable run record with crash recovery, all behind a human approval gate for anything that leaves the building. This paper explains the design, the mechanics, and the reasoning, with diagrams.

1. Motivation: why not a separate agent engine?

The first version of our agent system was config-driven and parallel: a 40-field configuration fed a hand-written pipeline that searched, enriched, and emailed. Meanwhile the chat assistant had grown a rich agentic capability of its own. It could search, research, grade, draft, generate images and video, post, and run multi-step plans. The two shared almost nothing.

That split is expensive:

- **Capability drift.** The chat could run dozens of actions; the agent re-implemented about ten. New chat features never reached agents.
- **Behavioural drift.** Two implementations of "save these leads" diverge over time.
- **Maintenance tax.** Every change had to be made twice, or it regressed.
- **Dead weight.** Thousands of lines of legacy pipeline that nothing called.

The fix is not a better second engine. It is to delete the second engine and let agents ride the first one.

2. Why this matters

Traditional agent frameworks duplicate four things: orchestration, tools, prompts, and execution logic. Every duplicate is a place to drift, a surface to maintain, and a reason the agent lags the product.

This architecture removes that duplication by treating autonomous execution as a continuation of the same conversation. The payoff is concrete for four audiences:

- **For engineers:** one codebase to reason about. A capability added to the chat reaches agents for free, with identical behaviour. Less code, fewer bugs, no fork to keep in sync.
- **For enterprise buyers:** the agent does exactly what the assistant does, with the same guardrails and the same human approval gate. There is no shadow system with weaker controls.
- **For investors:** capability compounds. Every improvement to the core assistant is also an improvement to the autonomous product, so effort is not split across two stacks.
- **For prospective hires:** the system is small enough to hold in your head and opinionated enough to be interesting. The hard parts are the orchestration decisions, not babysitting a duplicate pipeline.

In one line: **autonomous execution should be a property of the assistant, not a second product.**

---

3. By the numbers

Verifiable metrics from the codebase (architecture deltas), then the operating parameters that bound a run. Operational telemetry (mean runtime, measured crash-recovery rate) is being collected in the dev environment and will be added in a future revision; we do not quote numbers we have not measured.

Metric	Before	After
Parallel agent pipelines	2 (chat agentic + config-driven agent)	**1** (shared loop)
Schedulers dispatching agents	2 (hourly poll + per-agent cron)	**1** (hourly poll)
Actions available to an agent	~10 (private re-implementation)	**56** (11 native handlers + 45 shared MCP tools)
Legacy agent-pipeline code	present	**~1,150 lines removed**
Implementations of each action	up to 2 (chat + agent)	**1** (shared service)

Operating parameter	Value
Scheduler cadence	hourly poll, config-aware (hours, days, daily cap)
Free-run iteration cap	10 model turns
Plan iteration cap	up to 25 turns, scaled to plan size
Per-step self-correction	1 retry per step
Cross-run memory window	last 5 runs
Run states tracked	4 (Running, Success, Error, Interrupted)
Crash reconcile threshold	30 minutes

The headline: agents went from a shrinking subset of the assistant's abilities to its full action surface, while the system shed a whole pipeline and a whole scheduler.

4. Core thesis: the agent is the chat, talking to itself

A single chat turn looks like this:

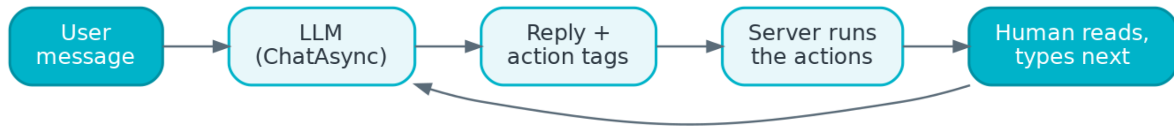

Figure 1. A single chat turn, where the human drives the loop.

An agent run is the same diagram with the loop in place of the human:

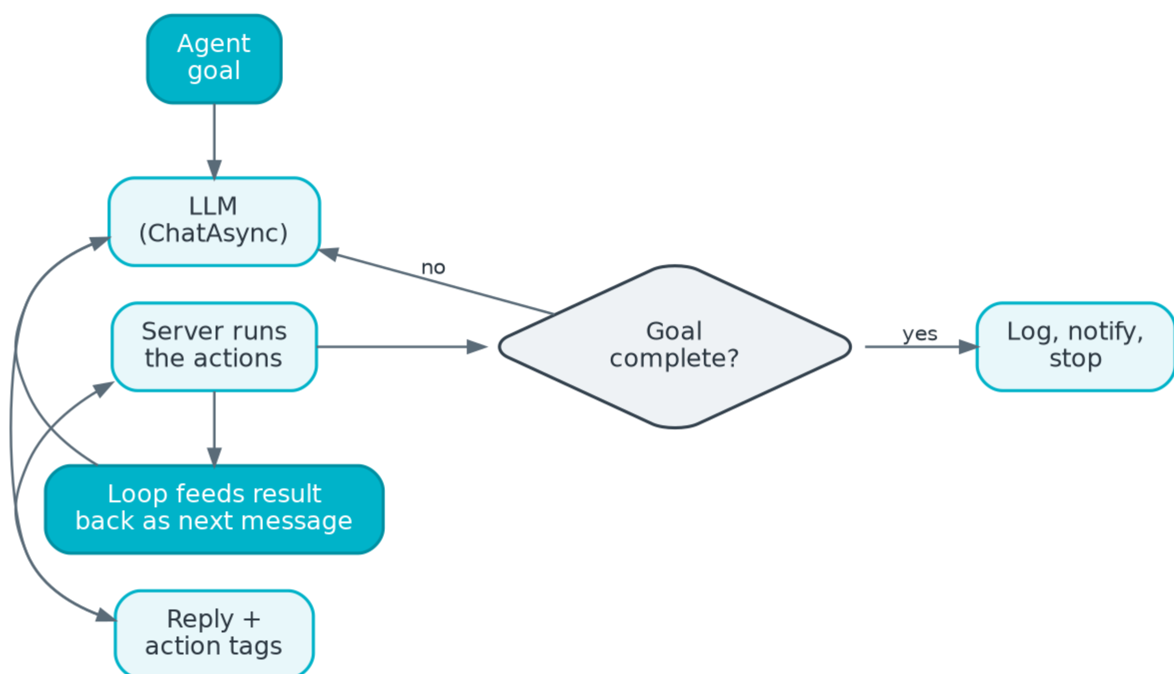

Figure 2. An agent run, where the orchestration loop replaces the human.

From the model's perspective, autonomous execution is indistinguishable from human conversation. That is the whole trick: the conversation is the execution substrate.

The action protocol is a set of inline tags the model emits, for example `<atapic_search>{…}</atapic_search>` or `<atapic_generate_image>{…}</atapic_generate_image>`. The server detects each tag, runs the matching action, and writes a short result back into the thread.

5. System overview

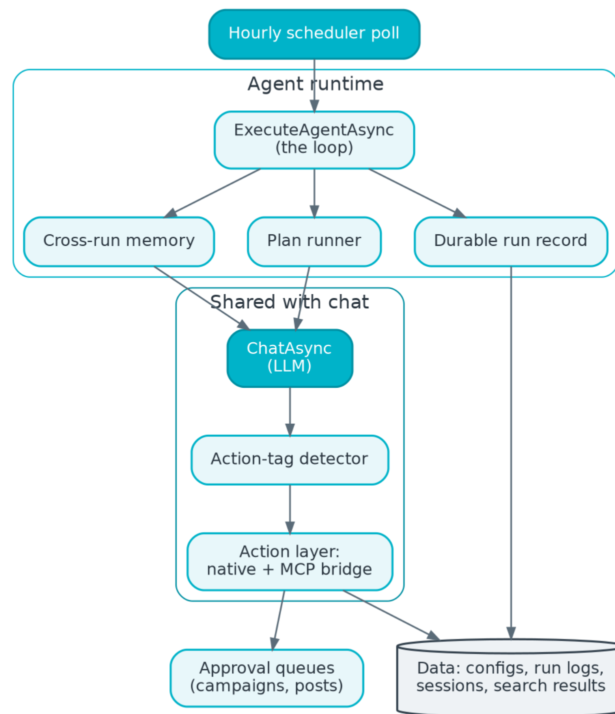

Figure 3. System overview. The agent runtime is thin; the brain and hands are shared with chat; anything outbound passes a human gate.

The agent runtime (left) is thin. The brain and the hands (middle) are shared with the chat. Anything outbound passes through a human gate (right).

6. Components and responsibilities

Component	Responsibility
Scheduler poll (`RunScheduledAgents`, hourly)	The single dispatcher. Selects enabled agents that are in their window, under their cap, and due; runs each. Also triggers crash reconciliation.
`ExecuteAgentAsync`	The agentic loop: build prompt, call the model, detect tags, run actions, feed results back, repeat. Owns memory, planning, token accounting, the run record, and notifications.
`ChatAsync` (shared)	The model call, identical to the chat surface. Builds context from the session and carried results and returns the reply.
Action-tag detector (shared)	Parses `<atapic_*>` tags into a typed list and flags "ghost work" (the model claiming an action without emitting a tag).
Action layer	Native handlers (search, save, enroll, outreach, analyze, post) over the shared services, plus a bridge to the MCP tool registry for everything else (images, video, newsletters, CRM).
Cross-run memory	A compact recap of the agent's recent runs, prepended so it adapts instead of repeating itself.
Plan runner	For a complex goal, the agent emits a plan; the runner walks it step by step with a per-step retry.
Durable run record	An `AgentExecutionLogs` row opened at start and closed at the end; a reconciler flips crashed runs to `Interrupted`.
Approval queues	Drafted campaigns and posts wait here for a human.
Run viewer / Agent Chats	Where humans read exactly what the agent did.

7. The action-tag protocol

The model signals intent to act by emitting tags. The detector validates them against a known set, so a typo or an injected tag is ignored, and returns them in order. Two properties make this hold up:

1. **It is text.** No function-calling schema to keep in sync across two surfaces. One detector serves chat and agents.
2. **It degrades safely.** If the model describes an action but forgets the tag, the ghost-work guard catches the action language, re-prompts once for the tag, and only then treats the turn as complete. This kills the classic agent failure mode: claimed success, did nothing.

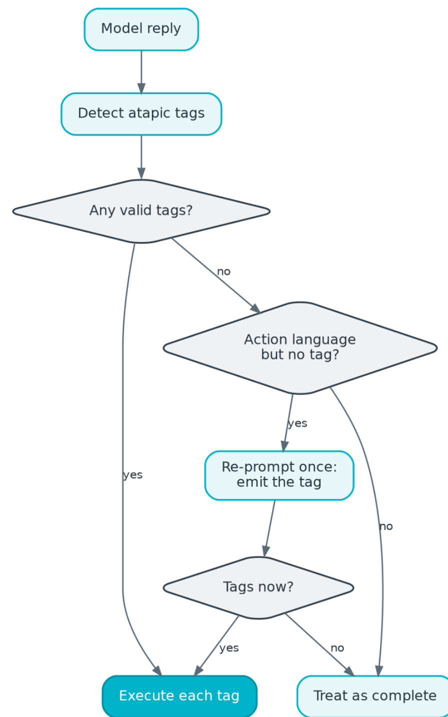

Figure 4. Ghost-work correction. The agent re-prompts itself when it describes an action without emitting a tag.

Agents inherit this guard from the chat unchanged.

---

8. The run lifecycle

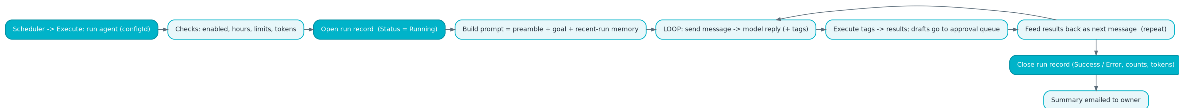

Figure 5. The run lifecycle. The durable run record is opened before the loop and closed after.

The run record opens before the loop and closes after it, so a run is durable while it executes, not only once it finishes.

---

9. Planning: free-run vs. structured plan

Simple goals do not need a plan, and over-structuring them wastes model calls. So the agent chooses:

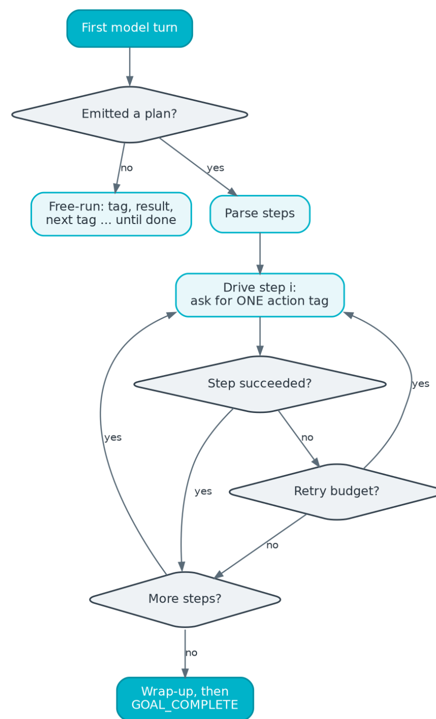

Figure 6. Planning. Simple goals free-run; complex goals run a plan with a per-step retry.

For a multi-stage goal ("find founders, grade them, draft outreach, post a recap"), the agent emits a plan and the runner walks it, retrying a failed step once with a "try a different way" nudge. Crucially, the steps execute through the same action body as free-run. There is no second engine, just a controller around the existing one.

---

10. The tool layer: one set of hands

The agent's power comes from not owning most of its tools.

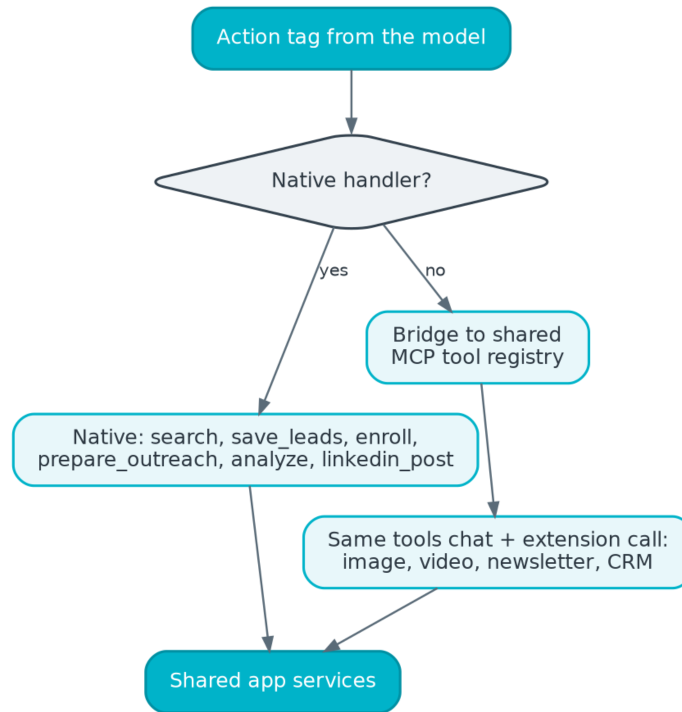

Figure 7. The tool layer. Native handlers and the shared MCP tool bridge resolve to the same services.

Native handlers are thin adapters over the shared services. Any tag without a native handler is dispatched to the MCP tool registry, the exact tools the chat and the Chrome extension use. The payoff: the day an image or video tool is added for chat, agents get it for free, with identical behaviour.

---

11. Cross-run memory and adaptation

A chat turn is stateless beyond its session. A year-long agent must not re-run the same search every hour. Before each run, the loop reads a compact recap of the agent's own recent runs (outcomes, actions, failures) and prepends it to the first message, with an instruction to cover new ground and to broaden or pivot when recent runs kept surfacing leads it already has.

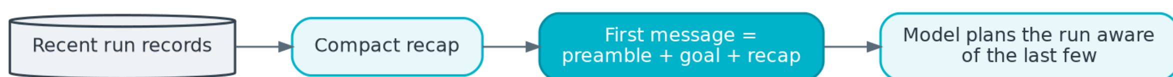

Figure 8. Cross-run memory. Each run is seeded with a recap of the last few.

This closes the loop on adaptation across runs, not just within one. It reuses the durable run records (Section 13) as its source, so it needs no extra storage.

---

12. Scheduling and concurrency

There is exactly one dispatcher: an hourly poll. It is config-aware (honours each agent's hours, days, and daily cap) and idempotent (guards on "last run about an hour ago"). An earlier design also registered a per-agent cron, which double-fired; that path was removed and now actively cleans up stale per-agent jobs.

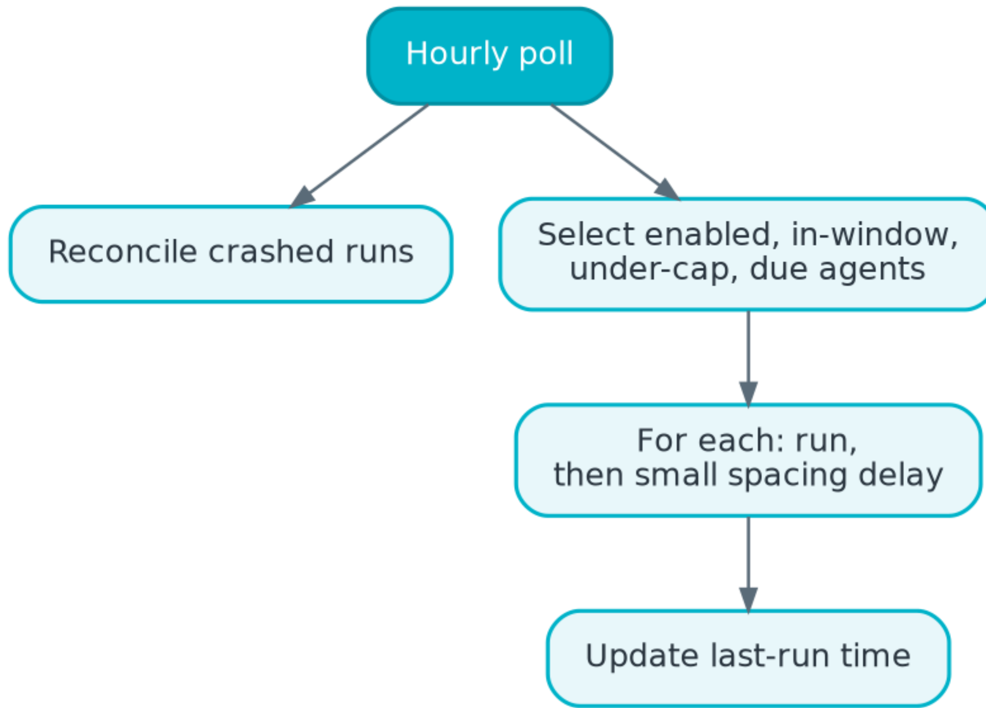

Figure 9. Scheduling. One config-aware hourly poll dispatches the whole fleet.

Why a poll, not a per-agent timer? A single poll is the one place that knows the whole fleet's state, which makes fairness, spacing, caps, and crash reconciliation trivial to reason about. Per-agent timers scatter that logic and race each other.

---

13. Durable run state and crash recovery

A run used to be recorded only on completion. If the worker restarted mid-run, the run vanished, invisible to the user and to the agent's own memory.

The fix is a lifecycle record. The run row opens at start with `Status = Running` and closes to a terminal status at the end. A crashed run leaves an orphaned `Running` row that the hourly reconciler flips to `Interrupted`.

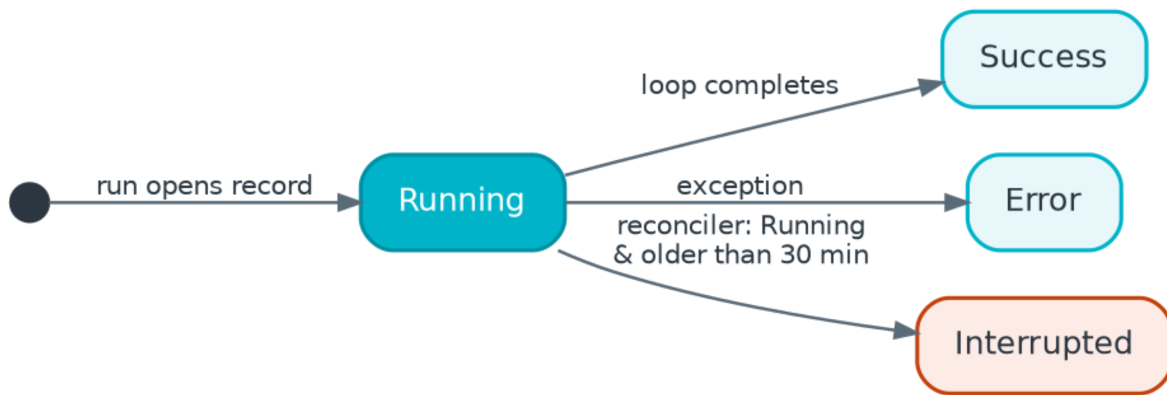

Figure 10. Run states. A crash leaves a Running row the reconciler flips to Interrupted.

Because this reuses the existing run-log table (new status values only, no schema change), the History UI immediately shows live `Running` runs and crashed `Interrupted` ones, and the change is purely additive and best-effort: logging never fails a run.

On "resume": full mid-loop resume (rehydrate the conversation and continue from step *k*) was considered and deliberately not built. A resumed run is nearly equivalent to a fresh run, because the next poll re-runs the agent and the lead-freshness filter prevents re-discovering the same people. So the record's value is observability and crash detection, not literal continuation, a conscious trade-off.

---

14. Safety: a human gate in front of anything outbound

An agent that runs unattended for months must not send email or publish posts on its own. Actions split into two classes:

- **Internal and reversible** (search, grade, save leads, create tasks): executed automatically.
- **Outbound and irreversible** (send outreach, publish a post): the agent drafts and queues for human approval. It never sends.

Figure 11. Safety. Anything outbound is drafted and queued for human approval before it sends.

This single rule is what makes "leave it on for a year" safe.

---

15. Data model

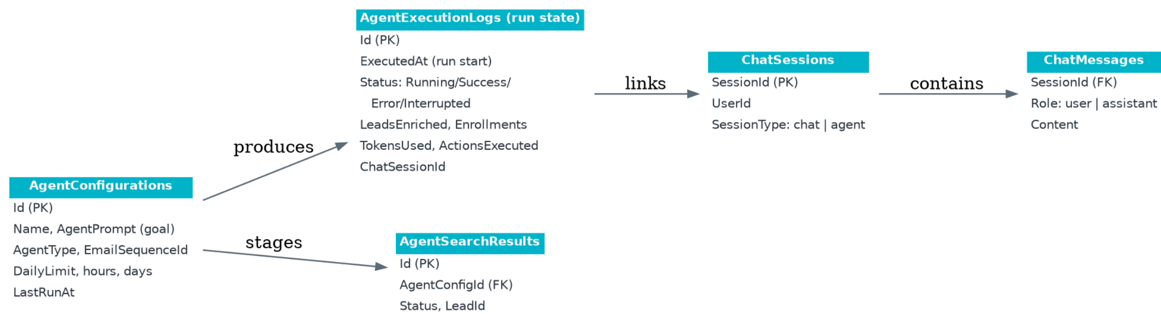

Figure 12. Data model. Agents own a config, a stream of run records, and a normal chat session.

The agent owns very little state: a config (the goal plus guardrails), a stream of run records, and a normal chat session for its conversation. There is no separate "agent message" store. Agent runs live in the same session and message tables as human chats, which is why the chat UI can render them.

---

16. Observability: where humans see the agent

- **Run viewer** (Agent Center, History, a run): the plan as a checklist, each step, action-result cards, generated-image thumbnails, and action badges. Live runs show as `Running`; crashed ones as `Interrupted`.
- **Agent Chats** (chat sidebar): the agent's session appears with an AGENT badge and renders like any conversation, because it is one.
- **Approval queues**: drafted posts and campaigns awaiting a click.
- **Email**: a per-run summary, respecting notification preferences.

The design goal: a human can always answer "what did this agent do, and why?" by reading its thread.

---

17. Design principles distilled

1. **One brain, one set of hands.** Reuse the chat's model call and action layer; never fork them. The agent is orchestration, not a second engine.
2. **Text protocol over rigid schemas.** Inline action tags keep chat and agents on the same rails and degrade gracefully.
3. **Goal-driven, not config-driven.** The form is Name, Goal, Schedule, and an optional sequence. Intent lives in the goal; the agent picks the actions.
4. **Structure only when it pays.** Plans for complex goals, free-run for simple ones, per-step retry for self-correction.
5. **One scheduler.** A single config-aware poll, not scattered timers.

6. **Durable, best-effort state.** Record the run from the moment it starts; never let bookkeeping fail the work.
7. **Human gate on anything outbound.** Draft and queue; a person sends.
8. **Memory makes it long-lived.** Seed each run with a recap of the last few so it adapts over a year instead of repeating itself.

---

18. Related work, and how this differs

This design borrows liberally from the agent literature and differs in one deliberate way: it refuses to build a second execution stack.

- **ReAct** [1] interleaves reasoning and acting in a single model loop. Our loop is ReAct-style, but the "act" half is the production chat's action layer, not a research harness, and every action is a real, audited product operation.
- **Toolformer** [2] teaches a model to call tools inline. We do not fine-tune for tool use; tools are invoked through a text tag protocol shared with the chat, so the same vocabulary serves humans and agents.
- **AutoGPT** [3] popularised long-running autonomous loops but is known for drifting and looping without progress. We bound runs (iteration caps, per-step retry, ghost-work detection) and gate all outbound actions behind human approval.
- **LangGraph** [4] and similar frameworks model agents as explicit graphs or state machines you author separately. We invert this: the "graph" is an optional plan the model emits at runtime, executed by the same loop that already runs chat.
- **OpenAI function calling** [5] and **Anthropic tool use** [6] provide structured tool invocation at the API layer. We sit above that: a single tag protocol plus a shared tool registry, so a provider's calling convention is an implementation detail, not the agent's architecture.
- **The Model Context Protocol (MCP)** [7] standardises how assistants reach external tools. We use an MCP-style shared registry as the agent's hands, which is exactly why agents inherit the chat's full tool surface.
- **Generative Agents** [8] showed memory and reflection driving believable long-lived behaviour. Our cross-run memory is a pragmatic, production-scoped version: a compact recap of prior runs that steers the next one.

The common thread in prior work is a dedicated agent runtime. Our contribution is the opposite: **autonomous execution as a thin loop over an existing conversational system**, which eliminates the duplication those runtimes reintroduce.

---

19. Future work

- **Mid-loop resume** for very long plans, only if a real need appears; today the poll plus the freshness filter cover most of it.

- **Per-goal budgets** (token and wall-clock caps) separate from the org pool, so heavy plans are priced correctly.
- **Live run streaming** to the run viewer over the same progress channel the chat uses, so a human can watch and intervene mid-run.
- **Published operational metrics** (mean runtime, measured crash-recovery rate, approval-acceptance rate) once enough dev-environment telemetry accrues.

References

[1] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, Y. Cao. "ReAct: Synergizing Reasoning and Acting in Language Models." ICLR, 2023. arXiv:2210.03629.

[2] T. Schick, J. Dwivedi-Yu, R. Dessi, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, T. Scialom. "Toolformer: Language Models Can Teach Themselves to Use Tools." 2023. arXiv:2302.04761.

[3] Significant Gravitas. "AutoGPT." Open-source project, 2023. github.com/Significant-Gravitas/AutoGPT.

[4] LangChain. "LangGraph: Building Stateful, Multi-Actor Applications with LLMs." Documentation, 2024.

[5] OpenAI. "Function calling." API documentation, 2023.

[6] Anthropic. "Tool use with Claude." API documentation, 2024.

[7] Anthropic. "Introducing the Model Context Protocol." 2024.

[8] J. S. Park, J. C. O'Brien, C. J. Cai, M. R. Morris, P. Liang, M. S. Bernstein. "Generative Agents: Interactive Simulacra of Human Behavior." UIST, 2023. arXiv:2304.03442.

Appendix A. Glossary

- **Action tag -** an inline `<atapic_*>` marker the model emits to request an action; the server detects and runs it.
- **Ghost work -** the model claiming an action in prose without emitting the tag; caught and re-prompted.
- **MCP tool registry -** the shared catalogue of tools (image, video, newsletter, CRM) used by chat, the Chrome extension, and agents.
- **Free-run -** the agent emitting one action tag per turn without an upfront plan.
- **Plan mode -** the agent emitting a structured plan that the runner walks step by step.
- **Run record -** the durable run-log row tracking a run's lifecycle from `Running` to a terminal status.
- **Approval queue -** where outbound drafts (campaigns, posts) wait for a human.