POLO: LIVE NATURAL LANGUAGE PROGRAMMING WITH LLM OBJECTS

Anonymous authors
Paper under double-blind review

ABSTRACT

The dominant paradigm for building software with LLMs is agent-assisted code generation. We ask not whether LLMs can produce software, but whether they can *be* it. We propose POLO: Persistent LLM-Objects with Self-Authored State (POLO): software as a set of persistent *LLM-objects* that own domain responsibilities, communicate through natural language messages, and are created and modified in natural language while running. The paradigm's core proposition is a new memory discipline: an LLM-object holds structured, private, and self-authored state with the semantics of program variables. The object derives its state schema from its natural language definition and can revise it when that definition changes. Each message handling rewrites the values its rules need—a number, a flag, or a list of entries—and discards everything else. No transcript is kept. Handling depends only on the object's definition, its state, and the incoming message, however long the system has been running. We build FLOWSTATEBENCH, a benchmark of *stateful*, long-horizon workflows spanning weeks to months of simulated time, where correct handling of each event depends on accumulated data across the entire history of the workflow, and on rule changes arriving mid-stream. On FlowStateBench, POLO outperforms agent frameworks representing four memory strategies (LangGraph, Letta, CrewAI, and OpenClaw), and replacing its state with a transcript in the same runtime lowers state-dependent accuracy by 22.9 points. On Memora, an external long-term memory benchmark, a self-authored state schema is not distinguishable from one written by a developer, and both score above transcript-based baselines. The code, benchmark, and generator are released.[1]

1 INTRODUCTION

Large language models (LLMs) have become the dominant tool for powering code generation from natural language (NL) specifications, with remarkable success (Chen et al., 2021; Li et al., 2022; Ziegler et al., 2022; Yang et al., 2024; Kessel & Atkinson, 2024). But rather than having LLMs *produce* software, we ask: can LLMs *be* the software?

We propose *POLO: Persistent LLM-Objects with Self-Authored State* (POLO), a paradigm in which programs are collections of *LLM-objects*. Like OOP objects, they are persistent, encapsulate domain logic, keep private state, and communicate by message passing. Each object is realized as an LLM rather than a class written in code. Objects borrow their computational model from object-oriented programming and the actor model (Goldberg & Robson, 1983; Hewitt et al., 1973): in response to a message, an object can update its state, send messages, and create new objects. The NL definition is the code. Code, when used, is a tool an object invokes. Programs are also live, or *online*: an object can be modified through the same message interface, while running, without restart or state loss.

What sets POLO apart is its memory discipline: each object holds private, structured, persistent, and *self-authored* state. The object derives its state schema from its NL definition, without a developer specifying its fields, and can revise it when the definition changes. Traditional programs use variables to hold the values they need to run. In POLO, state serves the same purpose. Each message handling rewrites the values its rules need and discards the processing trace. The updated state is all that survives an event. Where a rule counts over a window, the state keeps the dated entries that window

[1] https://anonymous.4open.science/r/programmable-online-llm-objects-FC4E/

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

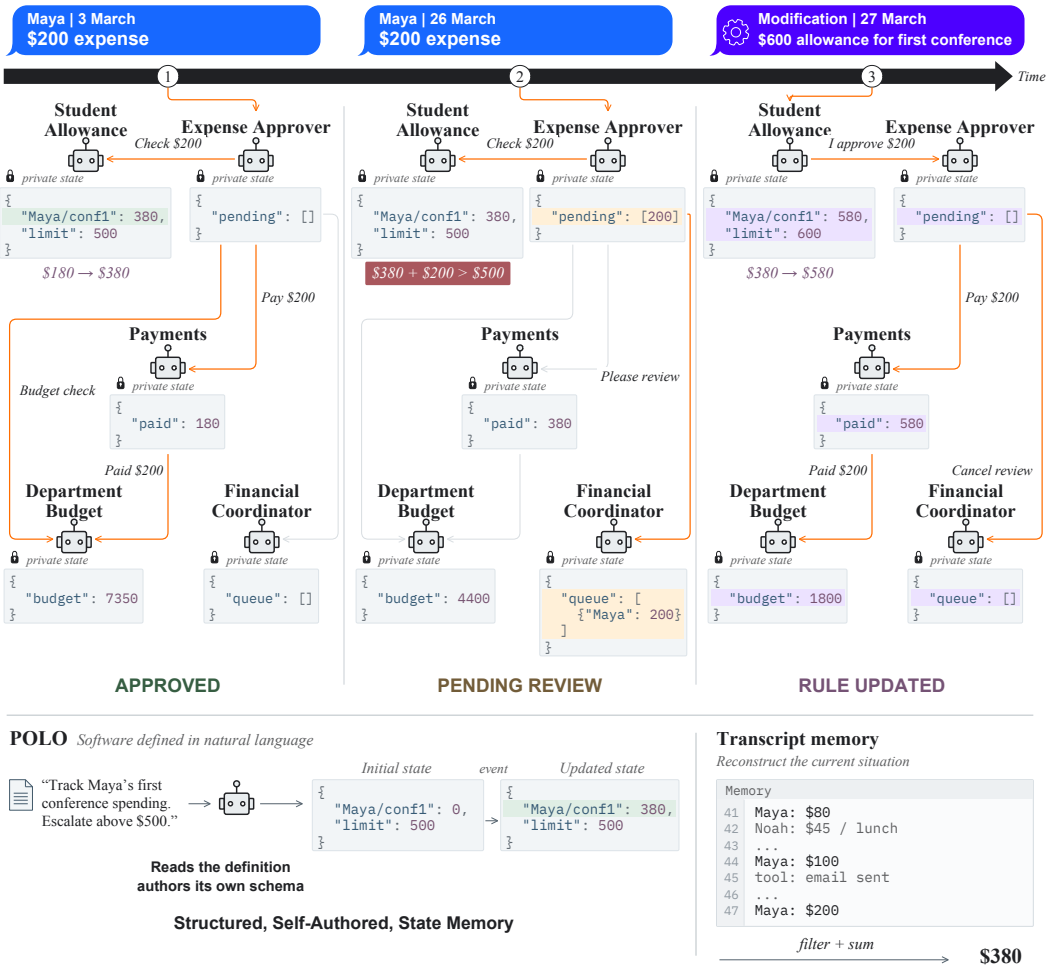

Figure 1: **Self-authored, structured, persistent state in a live program of LLM-objects.** Each object derives its state schema from its NL instructions and retains updated values between events, not messages history. An NL message changes the allowance rule while the program runs, and the same objects use their retained state to approve and pay a previously held expense. Orange arrows: active message. Shaded fields: state updates. Bottom: self-authored state versus transcript memory.

needs and nothing else. Transcript- and notes-based memory instead keep the full record of what happened. To answer how much Maya has spent, such an agent must find the relevant expenses in that record and sum them again.

The objects are the software: their behavior is defined in simple NL without writing code or specifying state fields. We target workflows, automations, policies, and other processes defined and modified in NL while running.

Figure 1 illustrates a small example. A department reimburses students within a $500 per-conference allowance and a $10,000 annual budget. Five objects share the work: `Expense Approver` coordinates decisions, `Student Allowance` tracks spending per conference, `Department Budget` holds the remaining budget, `Financial Coordinator` queues exceptions, and `Payments` pays approved expenses. Maya submits a $200 expense. Her total is $180, so the expense is approved and `Student Allowance` updates her total to $380. When she submits another $200 for the same conference, approving it would exceed $500. The expense is held for review and her total remains $380. With transcript-based memory, the agent must scan the record, filter Maya's expenses for this conference, count each once, sum, and compare at each decision. The `Student Allowance` object holds Maya's number and reads it. The program can also change while running. An administrator sends an NL message: *"Raise the allowance for students' first conference to $600."* This changes the object's

instructions without restarting the program or losing its state. Under the new rule, the same objects approve the held expense, cancel its review, complete its payment, and update Maya's total to $580.

We build FlowStateBench to measure the difference. Its samples are stateful, long-horizon workflows derived from real automation scenarios on Zapier. Events arrive as independent invocations with no history attached. Correct handling requires running totals measured against caps, counts that reset each week or month, and rules changed midway. On the evaluated subset, POLO scores 61.2% overall and 54.2% on state-dependent decisions. The highest baseline scores are 36.2% and 15.9%, respectively (Section 6.1). Replacing folded state with a transcript in our runtime lowers state-dependent accuracy by 22.9 percentage points, with little change on other events. Two limits remain. Accuracy on events after an instruction change is 6.8 points lower than before it when averaged within each workflow (13.3 points pooled), and a single object holding the whole workflow scores at least as well as the multi-object program at lower cost (Section 6.2).

We also evaluate on Memora (Uddin et al., 2026), an external long-term memory benchmark, to test whether an LLM can define a useful state structure without a developer specifying its fields. We compare self-authored and hand-authored state under the same bookkeeping instructions. Neither is reliably better than the other, and both have higher mean scores than our single-agent baselines across the three evaluated durations.

Our contributions are threefold: (1) POLO: Persistent LLM-Objects with Self-Authored State, a programming paradigm in which persistent LLM-objects are the software, with state carrying the semantics of program variables, (2) a runtime with object-owned state, asynchronous messages, and objects defined and modified in NL, released as software, and (3) FlowStateBench, a workflow benchmark whose decisive events require state accumulated across earlier events.

2 Related Work

2.1 Agent Memory

Agent memory takes three broad forms: *records* of past interactions, including editable blocks (Packer et al., 2023), distilled lessons (Shinn et al., 2023), retrieved experiences (Park et al., 2023; Zhong et al., 2024), and memory files (OpenClaw, 2026); *trajectory compression*, through successive chunks or summaries (Yu et al., 2026b; Li et al., 2026a; Ye et al., 2025); and *state tracking*, which maintains goals, inventory, location, or beliefs during execution (Rozanov & Rei, 2025; Yoneda et al., 2024; Jiang et al., 2026; Singh et al., 2026; Yang et al., 2026), sometimes alongside files and recent actions (Yu et al., 2026a). Concurrent work, SKILL.state (Badhe et al., 2026), conditions each step on the specification, current state, and latest observation, discarding intermediate reasoning after a validated update. Its state schema is authored once per domain and remains fixed. POLO combines private state per object with schemas derived from NL definitions and revised when those definitions change during execution.

2.2 Programming Models and Agent Platforms

Semantic parsers (Quirk et al., 2015; Yin & Neubig, 2018) and code assistants (Chen et al., 2021; Li et al., 2022; GitHub, 2026) turn NL into code; code-as-action agents generate and execute code during a task (Wang et al., 2024; 2023), and agent programming frameworks embed LLM calls in code (Beurer-Kellner et al., 2023; Khattab et al., 2024; Zheng et al., 2024). In POLO, the NL definitions are the program. Its objects draw on the actor model's private state and asynchronous messages (Hewitt et al., 1973), Smalltalk's live, modifiable program image (Goldberg & Robson, 1983), and agent-oriented programming (Shoham, 1993), where agents hold state and exchange typed speech acts (Finin et al., 1994): here the interpreter is an LLM and the definitions are NL.

Multi-agent frameworks (Wu et al., 2023; CrewAI Inc., 2024; LangChain Inc., 2024) and commercial agent systems (Anthropic, 2025b; Cursor, 2026; OpenAI, 2026) coordinate and delegate work among LLM-backed agents. Live platforms such as OpenClaw (OpenClaw, 2026) and Letta (Packer et al., 2023) also support persistent agents and runtime changes. These capabilities alone do not distinguish POLO; its contribution is the programmatic model and memory discipline described above. Modularity assigns responsibilities and state to separate components (Parnas, 1972), but does

not guarantee higher accuracy: coordination can introduce failures (Cemri et al., 2025), and single agents can match or outperform teams under equal budgets (Tran & Kiela, 2026).

2.3 Long-Horizon and Memory Benchmarks

Long-running agents must retain state and recover across extended workflows (Anthropic, 2025a; Kwa et al., 2026; Tran et al., 2025; Jaye et al., 2026; Du, 2026). OdysseyBench (Wang et al., 2025) and AgencyBench (Li et al., 2026b) evaluate long-horizon task completion. LoCoMo (Maharana et al., 2024) and LongMemEval (Wu et al., 2025) test recall from long histories. MemoryArena (He et al., 2026) evaluates actions under previously stated constraints. Memora (Uddin et al., 2026) pose a task or question after long conversational history. FlowStateBench streams events one at a time, and tests the system's ability to maintain and update state across them. Its decisions depends on earlier events, which require continuous state tracking and aggregation across earlier events, under interleaved counting windows, decoys, and mid-stream rule changes (Section 4).

3 Paradigm

3.1 LLM-Objects

An LLM-object is the core component of POLO, a special class of agentic LLM with a harness. LLM-objects are persistent and stateful, and live for the duration of the program (or until explicitly destroyed). An object's definition specifies the object's name, its role, the peers it communicates with through NL messages, tools it can use, and the behavior that governs how it processes messages—all expressed in NL. The object's definition can be modified at runtime, and the object continues from its current state.

Self-authored state. The object derives its state schema from its NL definition and general state-design principles. These guide it to identify durable entities and their keys, distinguish updates from appending, summing, or counting, and retain fields needed to handle later messages (e.g., tracking student spending requires a total keyed by student ID). The schema remains fixed unless the instructions change, when the object can revise it to match.

For a fixed model and inference configuration θ, the object's initial state is derived from its initial NL definition:

$$\sigma_0 \sim \mathsf{Init}_\theta(D_0). \tag{1}$$

An LLM-object then reads only three things on each invocation: its definition, state, and the incoming message. We write this as

$$(o_t, \sigma_{t+1}) \sim \mathsf{Agent}_\theta(D_t, \sigma_t, m_t). \tag{2}$$

Here, D_t is the object's current definition, σ_t its state, and m_t the incoming message, including its sender and intent. The output o_t consists of outgoing messages, tool requests, and any reply. σ_{t+1} is the updated state. Modifying the object's instructions changes D_t. For a fixed model and inference configuration, the same definition, state, and message yield the same output distribution, although individual outputs may vary.

Structured, persistent state. Each object holds structured state that persists between messages. Each handling updates its values and discards its processing trace. The updated state is all that survives. The object stores the current values directly: a running total, or, for a rule that counts over a window, the dated entries that window needs. It does not keep the messages that produced them. State is owned by the object, and other objects communicate with it through messages rather than reading its state. The evaluated runtime keeps state in object-owned partitions but does not enforce this privacy rule, and it provides counting tools over recorded entries (Section 5). Tool results and peer replies also arrive as messages, and their retained effects are written into state.

The design goal is *history independence*: handling depends only on the definition, the state, and the incoming message, so different histories that lead to the same state are handled the same way. In practice, the runtime adds one more input, the compiled plan cached per message kind (below), and repeated handling of the same event does not always agree (Pass^3 in Section 6). The substantive claim is about the form of the state: it is a small structured record rather than the transcript that produced it.

Algorithm 1 Handling an operational message in an LLM-object

Require: definition D, persistent state σ_t, message m
1: $P \leftarrow \textsc{Compile}(D, kind(m))$
2: **repeat**
3: $\quad (\Delta\sigma, r) \leftarrow \textsc{Execute}(P)$
4: $\quad P \leftarrow \textsc{Verify}(P)$
5: **until** $\textsc{Complete}(P)$
6: $\sigma_{t+1} \leftarrow \textsc{UpdateState}(\sigma_t, \Delta\sigma)$
7: **return** r

An agent carrying its transcript also carries the history that produced its state, so its input can differ even when the current values are the same.

Message handling. The NL instructions in the definition are compiled by an LLM call into a *plan*: tool calls and messages to peer LLM-objects connected by data dependencies, branches, and loops. The harness caches the plan by message kind. A modification to the definition invalidates the cache. The executor receives the object's definition, state, and incoming message, fills in symbolic values, and dispatches steps whose inputs are available. Verification checks what the plan required against what the executor actually did and reopens only missing work, leaving completed work intact. Once every step is done, failed, or skipped and no repair remains, the harness writes the updated values to the object's persistent state and discards the processing trace. When a reply is expected, the harness collects it from the executor and returns it to the sender. Algorithm 1 summarizes the process. Appendix B gives the full procedure (Algorithm 2).

Emitters and external events. LLM-objects only receive NL messages, but external events—an incoming email, a new order, a webhook firing—don't arrive in NL. An *emitter* bridges this gap: a small program, outside the object model, that watches an external system and translates each relevant occurrence into an NL message. LLM-objects subscribe to emitters and interpret each event against their role and state, just like any other message.

3.2 Communication Protocol

Messages fall into two categories by purpose: *operational* messages carry the program's work, while *administrative* messages modify the program itself. Each message is a tuple $\langle$*from*, *to*, *intent*, *payload*$\rangle$, where the payload is an NL utterance or a tool-call response, and the intent is *Ask* (expects a reply), *Tell* (no reply), or *Response* (a reply to a prior Ask). Following the actor model (Hewitt et al., 1973), a sender does not block on an Ask, and the runtime correlates each Response with its Ask. Tool invocations are an Ask to the tool returned as a Response, and synchronous tool APIs are wrapped to fit. The evaluated runtime does not authenticate administrative messages or separate them from the operational channel.

4 Benchmark

Workflows align naturally with POLO: they are defined in NL, span multiple domains, are owned by non-developers, and are changed at runtime. They also test memory: what a workflow should do with an incoming event often depends on counts accumulated over time and rules changed along the way.

4.1 FlowStateBench

We release FlowStateBench, a benchmark of *stateful*, long-horizon workflows spanning weeks to months of simulated time. A workflow describes a process, its steps, and the conditions under which each step is taken. Each step may call a tool to act on the environment, such as sending an email or creating a ticket. We add thresholds that require counting or arithmetic, for example, *"For the 3rd alert in the last 24 hours, set priority to high"*. A mid-stream modification changes a rule in NL, such as *"lower the team limit to 2"*.

Each sample contains a workflow description in NL, the tools available to the system, and 60–72 events with absolute timestamps. Events include external events that trigger the workflow and a single modification event. Each event is paired with its expected behavior: the tool calls the system should produce, given the full history. Appendix D gives a full sample.

4.2 What makes a sample test memory

A long event stream is not, by itself, a memory test if each event can be answered on its own or the answers follow a guessable pattern. Each workflow therefore tracks multiple interleaved counts over different subjects, different clocks, or both. Some later events depend on evidence from many events earlier, while decoys superficially match a rule but must not be counted. The system must maintain separate tallies, age out rolling windows, reset calendar counts, and apply the mid-stream modification only from that point on and within its named scope.

We check long-range dependence for a selected set of events by removing earlier evidence and verifying that the correct answer changes. During construction, deterministic probes also replay each design with limited memory, incorrect counting, or incorrect modification handling. Appendix D.2 reports what those probes distinguish across the benchmark.

4.3 Curation

The benchmark contains 100 samples, each built from a hand-authored workflow grounded in a public Zapier template. The template supplies the workflow's domain, systems, and vocabulary. A sample has 65.6 events on average (6,600 events in total, 6,000 graded). The authors write a specification of the workflow, its thresholds, and its modification. From this specification, we deterministically generate the events and expected tool calls. An LLM (`GPT-5.4`) renders each event in NL under validators that preserve every fact the answers depend on. The authors review every event for clarity and correctness. Appendix E gives the construction procedure.

4.4 Evaluation

The system receives the workflow and tools, then processes events in the specified order with a fixed time allowed per event. It is reset between samples. After each event, an LLM judge compares the tool calls produced against the expected calls for a binary pass/fail verdict. The judge sees only the actions taken for that event, not the system's replies or internal state. Events fall into three stages: *base* events test the original rules, *post-modification* events test the changed rule, and *regression* events test behavior the change should leave intact. In the reported agent results, *decisive* events are those labelled as threshold triggers or post-change flips. A stricter subset has counterfactual proof that removing earlier evidence changes its answer (Appendix D.2). We report overall pass rates and pass rates on decisive events.

5 Experiments

We run three experiments: the full evaluation on FlowStateBench, an ablation of POLO's design choices, and an external evaluation on Memora (Uddin et al., 2026). The main comparison uses `GPT-5-nano` with low reasoning effort for execution and the stronger `GPT-5.4` for compilation. Baselines use `GPT-5-nano` only, so the comparison does not separate the paradigm from the stronger compiler model. The evaluated runtime also exposes a read-only counting and arithmetic-aiding tools (Appendix F.1). We also report cost in dollars, including compilation, to account for the different model prices. All systems act on the same deterministic mock API environment. Full configurations, baseline descriptions, judging protocol, and metric definitions are in Appendix F.

FlowStateBench. To evaluate the complete paradigm, we compare POLO against four agent frameworks—OpenClaw (OpenClaw, 2026), LangGraph (LangChain Inc., 2024), CrewAI (CrewAI Inc., 2024), and Letta (Packer et al., 2023)—each in a single-agent and a multi-agent configuration. These frameworks provide different memory mechanisms and support instruction changes while retaining memory. We manually author an agent graph for each workflow and use the same graph for POLO and all multi-agent baselines. Single-agent baselines receive the full workflow instructions.

The four frameworks keep the past as records—a transcript replayed, notes searched, interactions retrieved, memory blocks the agent may edit—whereas in POLO folding is a mandatory stage and the folded state is all that survives an event. Modifications are delivered to every system as ordinary messages through its native input. A Gemini judge scores each system's tool calls against the expected calls for each event. We report pass rate over all graded events and over the *decisive* events—those whose correct handling depends on accumulated state—plus Pass^3 (Yao et al., 2024), the fraction of events passed in all three runs, and cost per event.

Ablations. We ablate the three choices that define POLO, changing one at a time. *Memory*: folded state versus the conversational transcript. *Execution*: disabling the compiler or verifier, with bare ReAct (Yao et al., 2023) and a simple single-pass execution (one model call per event) as additional references. *Topology*: multi-objects versus a single-object holding the workflow.

Memora. To test the memory discipline—state representation and schema authorship—outside our own benchmark, we evaluate on Memora's reasoning task, whose structure matches our regime: quantities accumulated over months of sessions against standing goals, with updates and invalidations. We compare self-authored and hand-authored schemas within the same multi-object architecture, using identical bookkeeping requirements, routing, tools, and answering procedure. Both retain raw dated entries for later querying. They differ in who specifies the state schema. We also evaluate a single-agent ReAct + search baseline, a plain single-agent transcript, and a no-memory baseline. We implemented these basic baselines. The comparison with ReAct + search changes both memory and architecture and therefore does not isolate the memory discipline. We report FAMA (Forgetting-Aware Memory Accuracy, Memora's metric, which credits answers that use current information and penalizes answers that rely on invalidated information) and token cost across the evaluated durations, using repeated runs and paired comparisons by persona. Memora's published memory systems are not included. Cost profiling details are in Appendix G.4.

6 Results

6.1 FlowStateBench: Performance and Efficiency

POLO has the highest reported accuracy. POLO scores 1.7× better than the best baseline overall and 3.4× better on state-dependent events (Table 1). Different baselines represent different memory mechanisms, but none reach POLO's accuracy at the budget they were run with. After an instruction change, POLO scores 1.7× the best baseline but 13.3 points below its pre-change accuracy (6.8 averaged within workflows; 95% CI: 1.0–12.7). Accuracy also falls at later positions (Table 9), which overlap with post-change events, so the two effects are not separated. Its Pass^3 is about 2× the best baseline, largely following from its higher accuracy.

This accuracy comes with a price. Higher cost originates from the strong compiler model (gpt-5.4), accounting for almost half of the cost. Output tokens, mostly reasoning, are more than half the cost. Since plans are compiled once per message kind and reused, this cost grows with the number of message kinds, not events. Cheaper executors with comparable accuracy, prompt optimization, and better caching are left for future work.

POLO input tokens do not increase with session length. Figure 2 shows that POLO's input tokens per model call stay steady across event positions, while the baselines' grow. An LLM-object makes several calls per event, but each call's input stays stable. The state itself stays small: it grows as entries are added (e.g., list items) and shrinks when they are no longer needed. Appendix G.1 details the call input and shows an example state after events 10 and 60.

6.2 Design Ablations

Each variant ran three times on a workflow subset; 95% CIs are in Appendix G.3. Table 2 shows that **structured state impacts accuracy mainly on state-dependent decisions**: a transcript lowers state-dependent accuracy by 22.9 points. **The compiler model choice has the largest effect on accuracy**. A strong model compiler delivers better performance. Weaker models reduce accuracy,

Table 1: Accuracy (%), input/output tokens per event, and cost (USD). Decisive: state-dependent events. Base: original tests. Post-mod: tests of changed instructions. Regr.: separate regression tests. Pass^3: events passing all three runs. Appendix G.1 gives coverage and confidence intervals.

System	Agents	All	Decisive	Base	Post-mod	Regr.	Pass^3	Input/output tokens/event	$/event	$/success
POLO	Multi	**61.2**	**54.2**	**62.5**	**49.2**	**64.7**	**39.1**	238.4k/19.7k	0.024	0.038
CrewAI	Single	25.8	11.9	29.1	12.6	15.2	12.4	178.2k/12.0k	0.014	0.053
CrewAI	Multi	16.7	1.4	18.8	7.9	9.4	11.5	20.7k/5.4k	0.003	0.019
LangGraph	Single	36.2	15.9	36.9	28.5	39.4	19.3	127.1k/0.9k	**0.001**	**0.004**
LangGraph	Multi	30.8	12.4	30.3	27.6	38.6	14.1	116.4k/1.4k	**0.001**	0.005
Letta	Single	24.7	3.4	25.6	17.6	25.9	10.7	97.7k/0.7k	0.005	0.021
Letta	Multi	15.1	1.7	17.3	6.4	7.0	9.8	259.4k/5.3k	0.015	0.100
OpenClaw	Single	24.9	11.4	26.0	19.2	21.8	10.1	105.9k/0.1k	**0.001**	**0.004**
OpenClaw	Multi	9.7	0.0	11.5	1.9	2.6	8.7	34.3k/0.3k	**0.001**	0.006

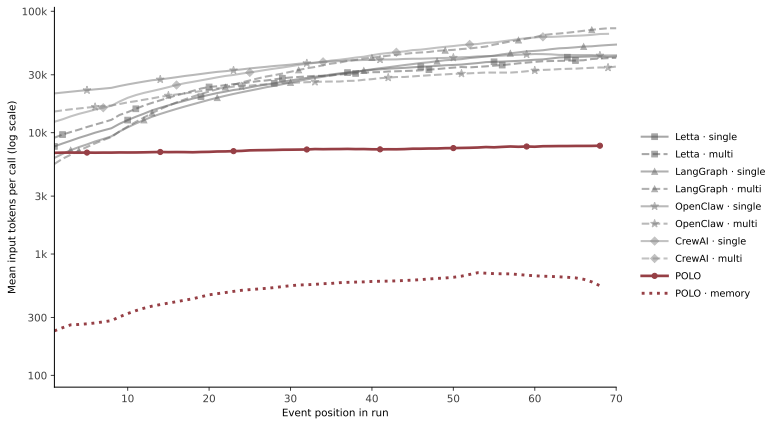

Figure 2: Mean input tokens per model call by event position (log scale). POLO remains stable with session length.

Table 2: The compiler matters most. State matters for decisive events. Accuracy changes are percentage points. Token and cost changes are per event.

Change	Δ Overall	Δ Decisive	Δ Input/output tokens/event	Δ $/event
Full POLO				
Transcript instead of structured state	−11.4	−22.9	+219k/−9k	+0.002
No compiler	−27.2	−36.2	−22k/+1k	−0.037
Verifier disabled	−22.1	−35.2	−31k/−2k	−0.006
Single object	+4.2	+11.4	−105k/−12k	−0.029
Compiler on `GPT-5.4-mini`	−18.5	−26.7	−130k/−7.5k	−0.007
Compiler on `GPT-5-nano`	−45.2	−50.0	−96k/−4.3k	−0.010
Bare ReAct	−18.7	−39.5	−138k/−19k	−0.050
Single-pass execution	−23.3	−41.0	−218k/−20k	−0.052

Table 3: Memora FAMA, mean ± 95% CI from three run-level scores. Weekly and monthly authored-state runs build separate memories. Quarterly runs repeat answering over one memory, so their intervals cover answering variation only.

Configuration	Weekly	Monthly	Quarterly
Hand-authored	55.1 ± 7.3	22.1 ± 11.3	10.0 ± 4.5
Self-authored	47.4 ± 8.1	28.0 ± 5.9	8.2 ± 5.9
ReAct + search	20.9 ± 0.9	13.3 ± 8.0	5.7 ± 2.9
Plain transcript	15.5 ± 5.9	9.5 ± 9.2	4.0 ± 6.4
No memory	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0

and the effect is larger on state-dependent events. The gpt-5-nano executor can skip a step or perform one incorrectly. **Verification mitigates the weak execution model**.

Neither the single-object nor the multi-object program is clearly better than the other. The single-object program is cheaper, but the multi-object program is more modular and easier to modify. Appendix Table 12 also gives the change in cost per success and error taxonomy.

6.3 MEMORA: SCHEMA AUTHORSHIP

We test whether an LLM-object can self-author its state—i.e., define a useful state structure without a developer specifying its fields, comparing both state authoring approaches under the same bookkeeping instructions. Baselines are our own ReAct + search, plain transcript, and no-memory agents.

Table 3 shows that both self-authored and hand-authored state LLM-objects paradigms have higher mean scores than ReAct + search and plain transcript. All methods with transcript-based memory score lower on longer histories. *Hand-authored and self-authored state are not distinguishable*: The weekly and quarterly advantages of hand-authored state are not statistically significant, and self-authored state has the higher mean monthly. Appendix G.4 gives preliminary query-cost profiling.

7 CONCLUSION

We introduced POLO, a paradigm in which software is a set of persistent LLM-objects defined and modified in NL, each holding structured, self-authored state that is rewritten at every event. We released FlowStateBench, 100 workflows whose decisive events depend on accumulated counts and mid-stream rule changes. POLO scores 61.2% overall and 54.2% on state-dependent events, against 36.2% and 15.9% for the best of eight framework baselines, at about 5× the cost per correct state-dependent decision ($0.09 vs. $0.02). Replacing its state with a transcript in the same runtime lowers state-dependent accuracy by 22.9 points, and on Memora a self-authored schema is not distinguishable from a developer-written one. Treating LLM-objects as live programming components opens a design space this paper has only begun to map.

LIMITATIONS

All systems execute with `GPT-5-nano`, chosen for price and latency: an ongoing workflow handles many small events that need speed and a modest budget more than long reasoning, and a `GPT-5.4` executor would cost about 23x more.

The paradigm depends on compilation: a plan is compiled once per message kind and reused, but producing it requires a strong model. POLO compiles with `GPT-5.4` while routing and executing with the smaller model the baselines use throughout. Removing compilation lowers state-dependent accuracy by 36.2 points (Table 2). Compiling with `GPT-5-nano`, the baselines' model, lowers accuracy by 45.2 points (Table 2), so part of POLO's advantage comes from the stronger compiler. All systems get counting tools, but only POLO gets a tool that reads another object's state; we do not measure its effect on the gap (Appendix F).

AI USE STATEMENT

We used generative AI tools to refine the conceptual framework and experimental methodology, develop and debug research code, interpret experimental results, and draft and revise text and figures. LLMs also assisted benchmark construction and served as evaluation judges, as described in Section 4 and Appendix F. The authors are responsible for the paper's claims, results, and artifacts, including AI-assisted material.

ETHICS STATEMENT

POLO lowers the barrier to programming and creating multi-agent systems through NL. This carries dual-use risk: the same interface that lets non-programmers compose workflows also makes it easier to deploy systems that act on the world without close monitoring. Runtime modification through administrative messages complicates accountability, since the behavior of a deployed system may diverge by any modification entering the system — through the administrative message or an event message interface — carrying potentially significant consequences. The token cost reported in our experiments also implies a meaningful compute and energy footprint at scale.

A separate concern is the security surface of such systems. A compromised object could propagate malicious instructions through the peer graph, and administrative messages provide an interface that, if unauthenticated, would allow object behavior to be rewritten without authorization. We leave the defense of the LLM-object networks against such attacks to the broader community.

REPRODUCIBILITY STATEMENT

The code, benchmark, generator, and run configurations are available at `https://anonymous.4open.science/r/programmable-online-llm-objects-FC4E/`. Appendix C provides the agent prompts; Appendices D and E describe benchmark examples and construction; and Appendix F details the baseline configurations, shared environment, and evaluation protocol. Table 3 specifies the repetitions and uncertainty estimates for Memora.

REFERENCES

Anthropic. Effective harnesses for long-running agents. `https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents`, 2025a.

Anthropic. Subagents in the claude code sdk. `https://code.claude.com/docs/en/agent-sdk/subagents`, 2025b. Accessed: 2026-05-13.

Sanket Badhe, Priyanka Tiwari, and Jonghyun Chung. Skill. state: Scalable long-horizon agent skills. *arXiv preprint arXiv:2608.26263*, 2026.

Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting is programming: A query language for large language models. *Proceedings of the ACM on Programming Languages*, 7 (PLDI), 2023. doi: 10.1145/3591300. URL `https://doi.org/10.1145/3591300`.

Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent LLM systems fail? *arXiv preprint arXiv:2503.13657*, 2025. URL `https://arxiv.org/abs/2503.13657`.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

CrewAI Inc. CrewAI: Framework for orchestrating role-playing, autonomous AI agents. `https://github.com/crewAIInc/crewAI`, 2024.

Cursor. Cursor: The best way to code with ai. `https://cursor.com/docs/subagents`, 2026. Accessed: 2026-05-13.

Pengfei Du. Memory for autonomous llm agents: Mechanisms, evaluation, and emerging frontiers. *arXiv preprint arXiv:2603.07670*, 2026.

Tim Finin, Richard Fritzson, Don McKay, and Robin McEntire. KQML as an agent communication language. In *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM)*, pp. 456–463, 1994.

Martin Fowler. Pojo. https://www.martinfowler.com/bliki/POJO.html, 2003.

GitHub. Github copilot. https://github.com/features/copilot, 2026. Accessed: 2026-05-13.

Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

Zexue He, Yu Wang, Churan Zhi, Yuanzhe Hu, Tzu-Ping Chen, Lang Yin, Ze Chen, Tong Arthur Wu, Siru Ouyang, Zihan Wang, et al. Memoryarena: Benchmarking agent memory in interdependent multi-session agentic tasks. *arXiv preprint arXiv:2602.16313*, 2026.

Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, pp. 235–245, 1973.

Abubakarr Jaye, Nigel Boachie Kumankumah, Chidera Biringa, Anjel Patel, Sulaiman Vesal, Dayquan Julienne, Charlotte Siska, Manuel Ra'ul Mel'endez Luj'an, Anthony Twum-Barimah, Mauricio Velazco, and Tianwei Chen. Corpgen: Simulating corporate environments with autonomous digital employees in multi-horizon task environments. *ArXiv*, abs/2602.14229, 2026. URL https://api.semanticscholar.org/CorpusID:285615667.

Haitao Jiang, Lin Ge, Hengrui Cai, and Rui Song. PABU: Progress-aware belief update for efficient LLM agents. *arXiv preprint arXiv:2602.09138*, 2026. URL https://arxiv.org/abs/2602.09138.

Marcus Kessel and Colin Atkinson. Morescient gai for software engineering. *ACM Transactions on Software Engineering and Methodology*, 2024.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into self-improving pipelines. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=sY5N0zY5Od.

Thomas Kwa, Ben West, Joel Becker, Amy Deng, Katharyn Garcia, Max Hasin, Sami Jawhar, Megan Kinniment, Nate Rush, Sydney Von Arx, et al. Measuring ai ability to complete long software tasks. *Advances in Neural Information Processing Systems*, 38:92213–92266, 2026.

LangChain Inc. LangGraph: Build resilient language agents as graphs. https://github.com/langchain-ai/langgraph, 2024.

Jiatong Li, Samuel Yeh, and Sharon Li. Multi-head recurrent memory agents. *arXiv preprint arXiv:2607.01523*, 2026a. URL https://arxiv.org/abs/2607.01523.

Keyu Li, Junhao Shi, Yang Xiao, Mohan Jiang, Jie Sun, Yunze Wu, Dayuan Fu, Shijie Xia, Xiaojie Cai, Tianze Xu, et al. Agencybench: Benchmarking the frontiers of autonomous agents in 1m-token real-world contexts. *arXiv preprint arXiv:2601.11044*, 2026b.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. Evaluating very long-term conversational memory of LLM agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13851–13870. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.747. URL https://aclanthology.org/2024.acl-long.747/.

OpenAI. Subagents: Use subagents and custom agents in codex. https://developers.openai.com/codex/subagents, 2026. Accessed: 2026-05-06.

OpenClaw. Openclaw: Your own personal ai assistant. https://github.com/openclaw/openclaw, 2026. Accessed: 2026-03-16.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560*, 2023. URL https://arxiv.org/abs/2310.08560.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pp. 1–22, 2023.

David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972. doi: 10.1145/361598.361623.

Chris Quirk, R. Mooney, and Michel Galley. Language to code: Learning semantic parsers for if-this-then-that recipes. In *Annual Meeting of the Association for Computational Linguistics*, 2015. URL https://www.aclweb.org/anthology/P15-1085.pdf.

Nikolai Rozanov and Marek Rei. StateAct: Enhancing LLM base agents via self-prompting and state-tracking. In *Proceedings of the 1st Workshop for Research on Agent Language Models (REALM 2025)*, pp. 367–385. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.realm-1.27. URL https://aclanthology.org/2025.realm-1.27/.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://papers.nips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

Joykirat Singh, Zaid Khan, Archiki Prasad, Justin Chih-Yao Chen, Akshay Nambi, Hyunji Lee, Elias Stengel-Eskin, and Mohit Bansal. Agent-BRACE: Decoupling beliefs from actions in long-horizon tasks via verbalized state uncertainty. *arXiv preprint arXiv:2605.11436*, 2026. URL https://arxiv.org/abs/2605.11436.

Dat Tran and Douwe Kiela. Single-agent llms outperform multi-agent systems on multi-hop reasoning under equal thinking token budgets. *arXiv preprint arXiv:2604.02460*, 2026. URL https://arxiv.org/abs/2604.02460.

Khanh-Tung Tran, Dung Dao, Minh-Duong Nguyen, Quoc-Viet Pham, Barry O'Sullivan, and Hoang D Nguyen. Multi-agent collaboration mechanisms: A survey of llms. *arXiv preprint arXiv:2501.06322*, 2025.

Md Nayem Uddin, Kumar Shubham, Eduardo Blanco, Chitta Baral, and Gengyu Wang. From recall to forgetting: Benchmarking long-term memory for personalized agents. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 26814–26841, 2026.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Weixuan Wang, Dongge Han, Daniel Madrigal Diaz, Jin Xu, Victor Rühle, and Saravan Rajmohan. Odysseybench: Evaluating llm agents on long-horizon complex office application workflows. *arXiv preprint arXiv:2508.09124*, 2025.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. *arXiv preprint arXiv:2402.01030*, 2024.

Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. LongMemEval: Benchmarking chat assistants on long-term interactive memory. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2410.10813.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024.

Zixu Yang, Hang Zheng, Nan Jiang, Zhiyang Tang, Situo Zhang, Xiaobao Wu, Lu Chen, and Kai Yu. No action without a NOD: A heterogeneous multi-agent architecture for reliable service agents. *arXiv preprint arXiv:2605.12240*, 2026. URL https://arxiv.org/abs/2605.12240.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2023.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *ArXiv*, abs/2406.12045, 2024. URL https://arxiv.org/abs/2406.12045.

Rui Ye, Zhongwang Zhang, Kuan Li, Huifeng Yin, Zhengwei Tao, Yida Zhao, Liangcai Su, Liwen Zhang, Zile Qiao, Xinyu Wang, Pengjun Xie, Fei Huang, Siheng Chen, Jingren Zhou, and Yong Jiang. AgentFold: Long-horizon web agents with proactive context management. *arXiv preprint arXiv:2510.24699*, 2025. URL https://arxiv.org/abs/2510.24699.

Pengcheng Yin and Graham Neubig. Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *Conference on Empirical Methods in Natural Language Processing*, 2018. URL https://www.aclweb.org/anthology/D18-2002.pdf.

Takuma Yoneda, Jiading Fang, Peng Li, Huanyu Zhang, Tianchong Jiang, Shengjie Lin, Ben Picker, David Yunis, Hongyuan Mei, and Matthew R. Walter. Statler: State-maintaining language models for embodied reasoning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2024. doi: 10.1109/ICRA57147.2024.10610634. URL https://statler-lm.github.io/.

Chenglin Yu, Yuchen Wang, Songmiao Wang, Hongxia Yang, and Li Ming. Infiagent: An infinite-horizon framework for general-purpose autonomous agents. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 35884–35894, 2026a.

Hongli Yu, Tinghong Chen, Jiangtao Feng, Jiangjie Chen, Weinan Dai, Qiying Yu, Ya-Qin Zhang, Wei-Ying Ma, Jingjing Liu, Mingxuan Wang, and Hao Zhou. MemAgent: Reshaping long-context LLM with multi-conv RL-based memory agent. In *The Fourteenth International Conference on Learning Representations*, 2026b. URL https://arxiv.org/abs/2507.02259.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL https://papers.nips.cc/paper_files/paper/2024/hash/724be4472168f31ba1c9ac630f15dec8-Abstract-Conference.html.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. MemoryBank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19724–19731, 2024. doi: 10.1609/aaai.v38i17.29946. URL https://ojs.aaai.org/index.php/AAAI/article/view/29946.

Albert Ziegler, Eirini Kalliamvakou, Shawn Simister, Ganesh Sittampalam, Alice Li, Andrew Rice, Devon Rifkin, and Edward Aftandilian. Productivity assessment of neural code completion. *arXiv preprint arXiv:2205.06537*, 2022.

A PROGRAMMING WITH LLM-OBJECTS

Why "LLM-object"? The name echoes the plain old Java object (POJO). Fowler and colleagues named ordinary Java objects when frameworks encouraged special object types (Fowler, 2003). An LLM-object is the unit of an NL program. It owns state and behavior and receives messages. Unlike a POJO, it follows a natural-language definition through an LLM and a runtime. The analogy concerns program structure, not execution accuracy.

This example shows how a POLO program is created through conversation, run, and changed while running. The example uses nine objects whose responsibilities connect through messages (Figure 3).

A.1 EXAMPLE: EXPENSE APPROVALS

A university wants to automate expense approvals. Students email a receipt and a short note for each expense. Expenses from different students and conferences arrive in mixed order.

Requirements. Each expense under $250 is approved automatically if it fits within both the student's allowance for that conference and the lab's $3,000 monthly budget. The university determines each student's allowance according to their academic achievements and its current funding policy. The lab budget starts fresh each month. The student's conference spending does not. Larger expenses, expenses beyond either limit, and incomplete allowance checks go to a manager through `Travel and Reimbursements`. The manager can decline, defer, or request approval, but every approval must still fit within both current limits.

Approved expenses count against both totals in the month of approval, then go to the university finance ERP for payment. Payment confirmation does not count the expense again. Students receive separate approval and payment updates. If the manager has not responded to an expense after 48 hours, they receive one reminder.

A.2 CREATING OBJECTS THROUGH CONVERSATION

We illustrate creation through an example command-line interface. The administrator first creates a program:

```
polo program new expenses
Created program expenses.
```

Subsequent commands add objects to this program. Each `polo object new` command prompts for instructions in free text, shown after `>`. The administrator describes the object's responsibility and names the objects it works with. No separate definition file is required. A script can supply the same instructions inline.

This example assumes the runtime is already connected to the expense inbox, university records, manager, finance ERP, and outgoing email. The instructions describe how the objects use these connections.

Expense Intake

```
polo object new "Expense Intake"
Enter instructions:
```

> Watch the expenses inbox and send complete expenses to Expense Approver. Make sure each has a student, conference, amount, receipt, and reference. If something is missing, ask Student Notifications to request it. Keep the expense until the student fills in the details.

```
Created Expense Intake in expenses.
```

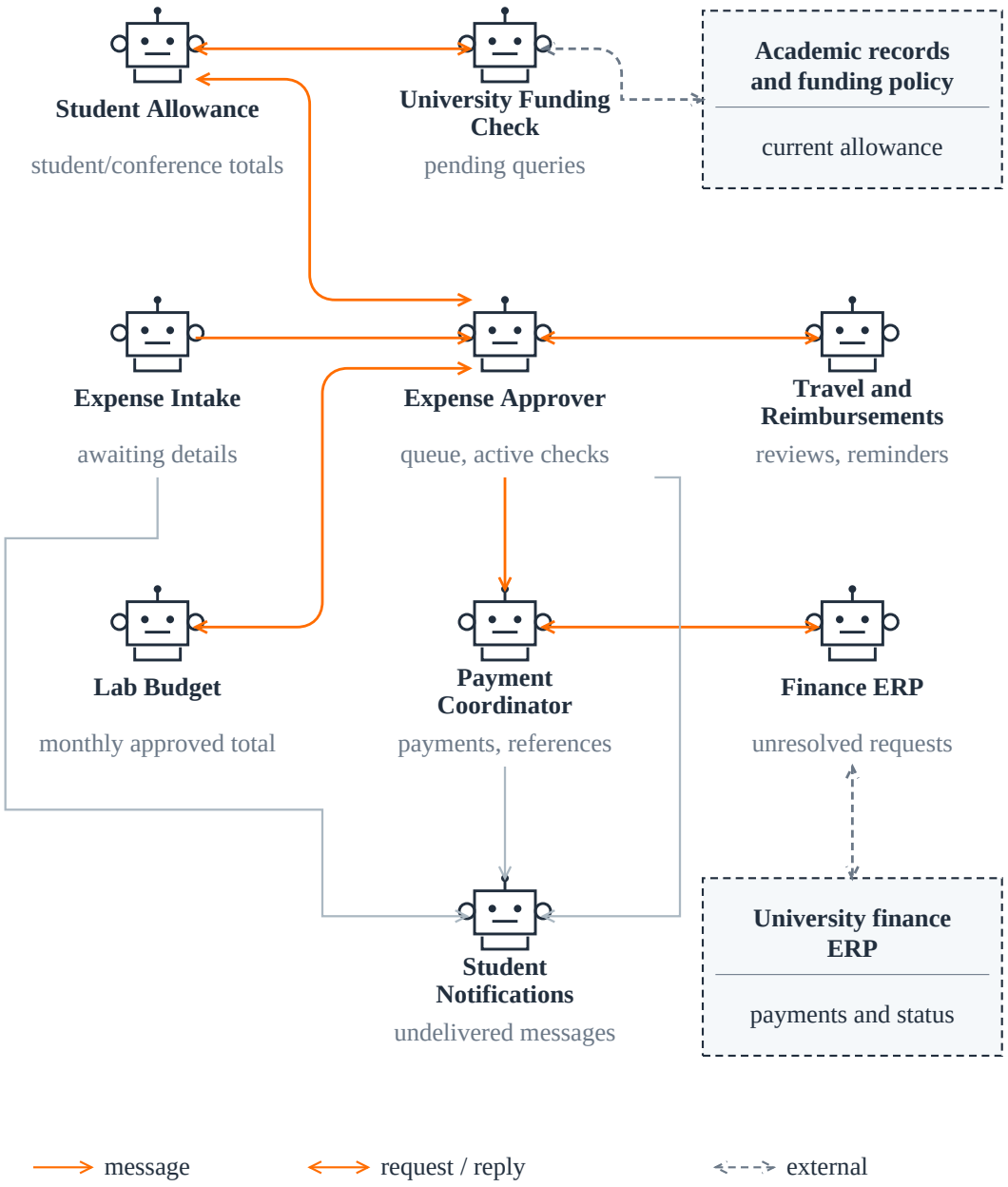

Figure 3: The nine-object expense program. Robot icons mark LLM-objects, with each object's name and private state below it. Orange arrows show messages between objects. Two-headed arrows include replies. Light arrows show selected notification requests. Dashed arrows and boxes mark university systems. Routine acknowledgments, student emails, manager decisions, and heartbeats are omitted.

Expense Approver

```
polo object new "Expense Approver"
Enter instructions:
```

> Check each expense with Student Allowance and Lab Budget. Approve it if it is under $250 and both say it fits. Otherwise send it to Travel and Reimbursements with the reasons. For approvals, have both totals updated, then ask Payment Coordinator to arrange payment and Student Notifications to tell the student.

A follow-up: Handle one expense at a time until both updates are confirmed. When the manager requests approval, check both limits again. They can approve larger expenses, but the limits still apply. Tell Travel and Reimbursements the outcome.

Student Allowance

```
polo object new "Student Allowance"
Enter instructions:
```

> Keep approved spending separately for each student and conference, across months. When Expense Approver asks whether an expense fits, ask University Funding Check for the current allowance and compare it with what the student has spent. Reply with whether it fits and how much remains. Only add spending when Expense Approver confirms approval, then confirm the update.

University Funding Check

```
polo object new "University Funding Check"
Enter instructions:
```

> When Student Allowance asks, check the university systems for the student's current conference allowance, based on their academic achievements and the funding policy. Return the amount and any conditions. If you cannot get an answer, say the check is incomplete and keep the request pending.

Lab Budget

```
polo object new "Lab Budget"
Enter instructions:
```

> The lab has $3,000 each month. Keep its approved spending and start fresh each month. Tell Expense Approver whether an expense fits and how much remains. Only add spending when it confirms approval, then confirm the new total.

Travel and Reimbursements

```
polo object new "Travel and Reimbursements"
Enter instructions:
```

> Bring forwarded expenses to the manager with the receipt and reason for review. Keep them until resolved. Remind the manager once after 48 hours without a reply. Send approval requests back to Expense Approver for fresh checks. Bring failed checks back to the manager. Ask Student Notifications to report declines. Leave deferred expenses on hold.

Payment Coordinator

```
polo object new "Payment Coordinator"
Enter instructions:
```

> Ask Finance ERP to pay approved expenses. Keep track until payment and notification are confirmed. Check pending payments regularly, always using the same expense reference. If a request times out, check what happened before trying again. Ask Student Notifications to report payment or a new delay. Don't count the spending again.

Finance ERP

```
polo object new "Finance ERP"
Enter instructions:
```

> Handle Payment Coordinator's requests in the university finance system. Look up the expense reference before submitting a payment, so an existing request is not submitted again. Return the payment reference and current status. If the result is unclear, keep checking it. Only report payment when the university confirms it.

Student Notifications

```
polo object new "Student Notifications"
Enter instructions:
```

> Send students the requests, decisions, and payment updates the other objects give you. Include the expense reference. Keep messages until delivery is confirmed and tell the requesting object whether delivery succeeded.

Each plain-language definition describes an object's role, what it tracks, how it acts, and which peers it contacts. These are the parts of an object definition in Section 3. Each object derives its state fields from its instructions and initializes its private state. The administrator supplies neither state fields nor update code. The runtime keeps the definition and state between messages. An instruction change updates the behavior while preserving state needed under the new instructions.

Two pairs show how the objects divide work. `Student Allowance` asks `University Funding Check` for an allowance, and `Payment Coordinator` asks `Finance ERP` to arrange or check payment. `Student Allowance` owns the conference spending totals. `Payment Coordinator` owns the pending payments. The other two objects handle the university systems.

Every expense carries the same reference through the program, and the ERP treats repeated submissions with that reference as the same payment request. Once all nine objects are in place, the administrator tells `Expense Intake`: "Start watching for expenses."

The instructions determine what survives. The administrator describes responsibilities, not JSON fields. From "keep approved spending separately for each student and conference," `Student Allowance` needs a total identified by both student and conference. From "start fresh each month," `Lab Budget` needs the current month and its spending total. Waiting for a reply requires a pending request. The objects choose fields for these values when they author their state.

The following snapshots show one possible set of state fields. The authors did not supply these fields to the objects, and the snapshots are not from a measured run. They show the state before Maya's March 3 expense, after earlier expenses have been approved. Each block belongs to one object.

Student Allowance: before Event17

```
{
  "approved": {"Maya": {"Conference A": 180}},
  "pending_checks": {}
}
```

Lab Budget: before Event17

```
{
  "month": "2026-03",
  "limit": 3000,
  "spent": 740
}
```

The two objects have different keys and reset rules. A message to `Lab Budget` uses its monthly values. It does not receive the student's conference totals. To obtain those, `Expense Approver` asks `Student Allowance`, which answers from its own state and a fresh university funding check.

A.3 Running the Program

External events. The runtime delivers new expense emails and student replies to `Expense Intake`. Manager decisions arrive at `Travel and Reimbursements`. It also sends heartbeats that carry the current time, so objects can act when a deadline passes even if no new expense arrives. `Payment Coordinator` uses them to request payment status through `Finance ERP`. Each object handles these messages using its definition and current state, just as it handles a message from a peer.

An expense moves through the objects. An incoming email starts the workflow through `Expense Intake`. For a manual walkthrough, the administrator can also send it a message:

To Expense Intake

March 3, 2026: expense Event17 from Maya, $200 for Conference A. Receipt R17 confirms a $200 registration payment.

Consider the program after earlier approvals have brought Maya's spending for Conference A to $180 and the lab's March spending to $740. The following messages illustrate the flow. Their wording need not be fixed in advance.

1. `Expense Intake` sends `Expense Approver` the complete expense: "Maya requests $200 for Conference A, expense Event17. The receipt is present."
2. `Expense Approver` asks `Student Allowance`: "Does Maya have room for another $200 for Conference A?"
3. `Student Allowance` asks `University Funding Check`: "What is Maya's current allowance for Conference A?"
4. `University Funding Check` consults the university systems and returns their decision: "Maya's current allowance for Conference A is $800, based on her academic achievements."
5. `Student Allowance` compares that allowance with its own $180 total and replies: "$620 remains for Maya at Conference A. Event17 fits."
6. `Lab Budget`, responding to the approver's separate check, reports: "$2,260 remains in the lab's March budget. Event17 fits."
7. `Expense Approver` records the approval with both objects and waits for confirmation: Maya's conference total becomes $380, and the lab's March total becomes $940.
8. `Expense Approver` sends Event17 to `Payment Coordinator` and asks `Student Notifications` to tell Maya it is approved, with payment pending.
9. `Payment Coordinator` asks `Finance ERP` to arrange the reimbursement.
10. `Finance ERP` checks for an existing request under Event17, submits it if absent, and returns the university's payment reference and current status.

The $800 in this example is a university response for Maya, not an allowance assigned to every student by the program. The approver does not consult academic records itself. It asks `Student Allowance`, which combines a reply from another object with the spending it owns.

State at work. On March 26th at 10:00, Maya submits another $200 expense for the same conference, Event18. Her approved conference spending is still $380, while intervening expenses have brought the lab's March total to $2,900. `Student Allowance` obtains a fresh university check. Suppose Maya's allowance remains $800. It reports that $420 remains and the expense fits. `Lab Budget`, however, reports that only $100 remains. `Expense Approver` forwards the expense to `Travel and Reimbursements`, noting that the student allowance is sufficient but the lab budget is not. Neither spending total changes while the expense awaits a decision.

The requests are otherwise identical, but accumulated spending changes the answer. `Lab Budget` holds the current month and its running total, not a ledger it must sum again. `Student Allowance` holds the current total for each student and conference. The university allowance is checked again because academic records and funding policy can change outside the program.

3 March | Event17

$200 approved

Student Allowance

```
{
  "approved": {
    "Maya": {"Conference A": 380}
  },
  "pending_checks": {}
}
```

180 ⟶ **380**

Lab Budget

```
{
  "month": "2026-03",
  "limit": 3000,
  "spent": 940
}
```

740 to 940

26 March | Event18

$200 held for review

Student Allowance

```
{
  "approved": {
    "Maya": {"Conference A": 380}
  },
  "pending_checks": {}
}
```

380 unchanged

Lab Budget

```
{
  "month": "2026-03",
  "limit": 3000,
  "spent": 2900
}
```

unchanged

Figure 4: Illustrative private state after each decision. Event17 updates both spending totals. Event18 is held for review, so neither total changes. Highlighted fields are written while handling the expense. The lab's other spending between the two dates is already in its March total.

The state after each decision. After Event17 is approved, `Student Allowance` replaces Maya's total of 180 with 380. `Lab Budget` replaces 740 with 940. These values survive after the messages used to make the decision are discarded. Neither object needs those messages to recover the new total.

By Event18, other approved expenses have raised the lab's total to 2,900. Event18 is held, so it does not increase either spending total. Instead, the review object adds an unresolved request to its own state:

Travel and Reimbursements: Event18 awaiting review

```
{
  "pending": {
    "Event18": {
      "student": "Maya",
      "conference": "Conference A",
      "amount": 200,
      "reason": "Only 100 remains in the lab budget",
      "remind_at": "2026-03-28T10:00",
      "reminder_sent": false
    }
  }
}
```

This entry describes unfinished work. It is removed when the review is resolved. The totals describe approved spending and remain until their own reset conditions apply. A queue can therefore hold several requests without becoming a transcript of every message the program has received.

From approval to payment. Suppose the ERP accepts Event17 and returns payment reference P904 with status *pending*. `Payment Coordinator` keeps Event17, the $200 amount, P904, and that current status. On the next heartbeat, it asks `Finance ERP` to check P904. It does not submit another $200 payment. When the university confirms that P904 has been paid, `Payment Coordinator` replaces the pending status and asks `Student Notifications` to tell Maya. After delivery is confirmed, the payment leaves its pending collection. The lab and student spending totals stay unchanged throughout: the $200 was counted when the expense was approved.

The ERP owns the payment record and may update it between messages. `Finance ERP` therefore obtains a fresh status from the university, while `Payment Coordinator` holds what the program is still waiting for. If a request times out, the objects keep the expense and payment references needed to check its outcome, rather than treating the timeout as proof that no payment was created.

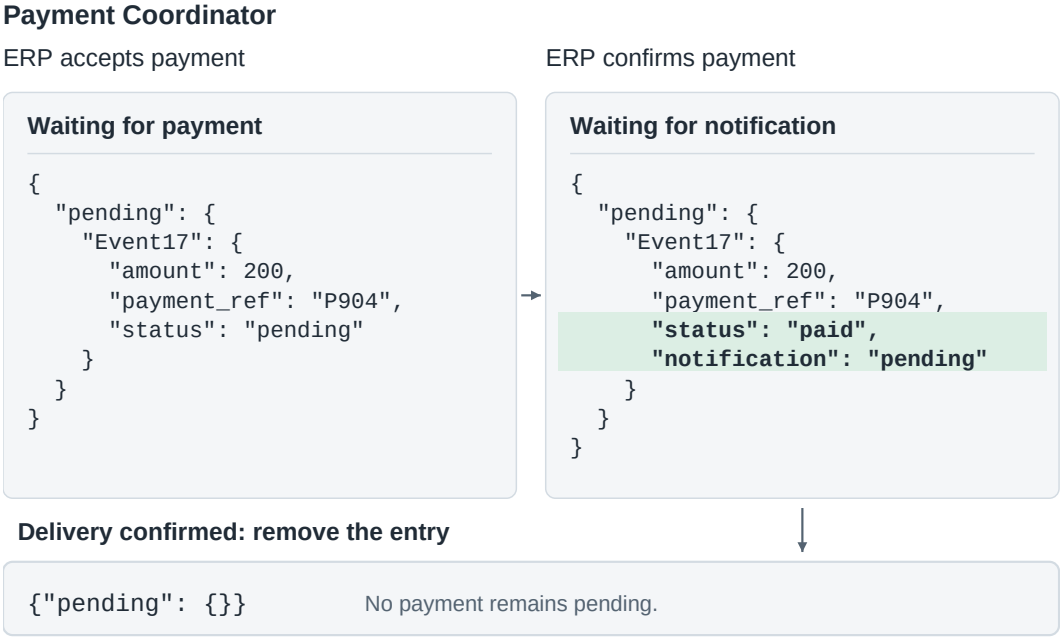

Figure 5: Illustrative Payment Coordinator state for Event17. The expense and payment references survive while payment is pending. Payment confirmation changes the status. Notification confirmation removes the entry. The approved spending totals do not change during payment.

Waiting without a transcript. The runtime sends heartbeat messages at fixed intervals, each carrying the current time. Suppose Event18 reaches `Travel and Reimbursements` at 10:00 on March 26th. The object keeps the unresolved expense with a reminder deadline of March 28th at 10:00 and a flag indicating that no reminder has been sent. On each heartbeat before the deadline, it leaves the expense pending. On the first heartbeat at or after the deadline, if the manager has not responded, it sends a reminder and changes the flag, so later heartbeats do not trigger the same reminder again. For a manual walkthrough, a heartbeat can also be supplied as a message:

To Travel and Reimbursements

Heartbeat: March 28, 2026, 10:00.

The object holds the current review status, deadline, and reminder flag, not the sequence of heartbeats that brought it there. Once the expense is resolved, it removes the pending entry. This timed behavior

belongs to `Travel and Reimbursements`. The spending objects continue to handle their own checks and updates.

Persistent and private state. These states persist between events, but they are not combined into one shared memory. When `Lab Budget` handles a message, it receives its definition, its own month and spending total, and the incoming message. Maya's conference spending and the manager's pending queue belong to other objects. `Student Allowance` receives its own conference totals and pending checks, not the lab's monthly spending. Other objects keep the work they own, such as unfinished university queries, manager reviews, payments, and messages awaiting delivery.

The distinction matters: $420 remaining for Maya at one conference and $100 remaining for the lab in March answer different questions. Treating one as the other would change the approval decision. The two balances are computed by different objects, and their replies identify which balance they report. The approver uses both replies, but it does not maintain either spending total. This separation keeps unrelated totals out of each budget object's state. It does not guarantee that an LLM will never confuse the replies. On the first April handling, `Lab Budget` starts a new monthly total, while Maya's conference spending remains $380. Likewise, `Payment Coordinator` may hold a $200 reimbursement that is still pending, while `Lab Budget` already includes that $200 in approved spending. The payment status and approved total are different state values with different owners. Confirming payment must not count the expense a second time.

A.4 Changing the Program While It Runs

Updating an object definition. In December, the department raises the lab budget for that month. The administrator updates `Lab Budget` with an NL message:

To Lab Budget

The lab can spend $4,000 in December 2026. Keep what it has already spent. Go back to $3,000 per month in January.

The budget it checks against is now $4,000 for December, while its running total for the month is preserved. The student allowance calculation is unchanged: `Student Allowance` still obtains the university's current allowance through `University Funding Check`. Subsequent expenses are handled under the changed definition without recreating the objects or resetting their state.

Changing one responsibility. The university may also change how allowances are determined. For example, if its funding policy now includes an achievement-based supplement recorded in a separate system, the administrator can update only `University Funding Check`:

To University Funding Check

Conference funding now includes the achievement supplement. Check it along with the base allowance in the university systems, and return the combined amount and conditions.

The example assumes the connected university tools can access the supplement. `Student Allowance` continues to ask the same question and compare the returned allowance with its own spending total. Its definition need not describe the new lookup, and its totals stay in its own state. The payment pair works the same way. If the university replaces its finance interface, the connection and `Finance ERP` definition can change while `Payment Coordinator` tracks pending payments, provided their references remain valid in the new system.

Changing what an object tracks. Changing the December limit adjusts a value without requiring a new kind of state. The administrator can also add a responsibility that requires additional state:

To Lab Budget

From now on, also count how many expenses the lab approves. Start the count at zero now, reset it at the start of each month, and include it whenever you report the remaining budget.

`Lab Budget` now maintains an approval count alongside its spending total. In the month of the change, the count covers only approvals after the change. It does not reconstruct earlier approvals from discarded history. The definition sets both the additional state and how the object uses it.

The object chooses a field for the new count. For example, if the change arrives during December when approved spending is 1,240, its state can become:

Lab Budget: an additional state field

```
{
  "month": "2026-12",
  "limit": 4000,
  "spent": 1240,
  "approved_count_since_change": 0
}
```

The existing spending total remains. The next approved expense updates both the total and the new count. The administrator requested this behavior in NL without naming a field or writing update code.

B MESSAGE HANDLING

Algorithm 2 Handling an operational message in an LLM-object

Require: definition D, persistent state σ_t, message m
$w \leftarrow \text{WORKINGCOPY}(\sigma_t)$
$P \leftarrow \text{COMPILE}(D, kind(m))$
attach P to w as transient execution state
repeat
  while P has ready steps or outstanding results **do**
    for all ready steps s in P **do**
      if s is a branch or loop **then**
        $P \leftarrow \text{RESOLVECONDITION}(P, s, w)$
      else
        $(a, \Delta\sigma, r) \leftarrow \text{EXECUTESTEP}(D, w, m, s)$
        apply $\Delta\sigma$ and retain reply r in w
        DISPATCH(a)
        UPDATESTATUS(P,s)
      end if
    end for
    if no step is ready but a result is outstanding **then**
      $(w, P) \leftarrow \text{AWAITRESULT}(w, P)$
    end if
  end while
  $P \leftarrow \text{VERIFY}(P)$
until $\text{COMPLETE}(P)$
$\sigma_{t+1} \leftarrow \text{UPDATESTATE}(w)$ ▷ update state
return $\text{REPLY}(w)$

Algorithm 2 gives the full handling procedure summarized in Section 3. The working state holds the object's persistent values, the active plan, step results, and any reply while a message is being handled.

Compile. The compiler turns the object's NL instructions into an executable, step-by-step plan. Steps specify tool calls, messages to other objects, and the values needed from earlier steps. Branches and loops remain open until their conditions can be checked. The plan uses placeholders for message-specific values and results. The router reuses a stored plan for the message kind when one fits.

Otherwise, the compiler creates a plan and stores it for later messages. An instruction change invalidates the object's stored plans.

Execute. The executor receives the definition, current state, incoming message, and ready steps. It fills in the required values and makes the tool calls, sends messages, and records state updates and any reply. Steps that need a tool result or peer reply remain open until it arrives. The runtime uses an LLM to resolve branch and loop conditions from the available values. It then runs newly ready steps. If no step is ready and a result is outstanding, it waits for that result. Failed steps can be retried within the configured limit.

Verify. The verifier checks the steps against the recorded actions. Missing work that can still be retried is reopened. Completed steps remain complete. Processing ends when every step is done, failed, or skipped and no repair remains. The ablation in Appendix G.3 specifies which checks are disabled.

Update state. The object retains the updated domain values and removes the processing trace. A running total keeps its new value. A pending request keeps the fields needed to finish it. Tool results and intermediate messages are not retained unless their contents are needed in the object's state. The object uses this state for the next message.

Return a result. If a reply was requested, the runtime returns the reply collected during execution. The reply is separate from the plan. If none was requested, it returns ∅.

C Runtime Prompt Excerpts

These excerpts come from code revision `d3058c98`, used in the reported POLO run. They show compilation, state updates, and condition evaluation. They omit parts of the prompts and have minor punctuation edits. The source files are under `config/prompts/llm_objects/` in the runtime repository. Names in braces are filled in at runtime.

Compile. The compiler produces a reusable plan for a kind of message. It leaves conditions open until execution. The excerpt is from `planner_ast.yaml`, field `template_contract_kind`.

Compile: reusable plans

```
## BLUEPRINT MODE -- this plan is a reusable PROGRAM for ONE KIND of event

The plan you emit will be COMPILED: stored under a description of the KIND of event it handles, and re-
      executed for every future event of that same kind, until the object's definition is modified.

- NO event literals anywhere. Never copy this event's ids, names, amounts, dates, or free text into a
      description or condition. Refer to data symbolically: `the lead id from the incoming message`, `the rep
       the rotation read returns`, `this event's scenario time`.

- Steps must be the GENERAL procedure FOR THIS KIND: the reads, the decision rule, the actions, the record.
      If the behavior says round-robin under a daily cap, the plan encodes 'find the first rep in rotation
      order whose count for today is under the cap' -- never the name of whichever rep qualifies today.
```

State authoring. The schema compiler receives the object's role, behavior, tools, and an example incoming message. It defines fields for rules that span messages. The excerpt is from `planner_ast.yaml`, field `memory_schema_contract`. These state types are choices made in this runtime. They may not suit every program.

State schema: types and keys

```
Read the behavior's CROSS-EVENT rules (caps, quotas, rolling windows, rotations, availability, budgets) and
      declare the typed registers that make them mechanically evaluable. Output STRICT JSON:
{{"registers": [{{"name": "<snake_case>", "kind": "stream|flag|counter", "key_scheme": "<literal:{{
      placeholder}} template, e.g. assignment:{{rep}}:{{tier}}>", "count_mode": "actions|arrivals", "default":
       <flags: the value when never set, or counters: {{"start": 0, "limit": <N>}}>, "doc": "<the behavior
      clause this serves>"}}]}}
```

```
Rules:
- ONE stream per countable rule. The key_scheme's placeholders are the rule's 'per X' pivots -- every
      dimension the cap distinguishes (rep AND tier if the caps differ by tier). Never a date, never the item
      's own id.
- count_mode=actions when the rule caps ACTIONS TAKEN (a blocked/declined request is NOT recorded). Use
      count_mode=arrivals when the rule fires on the Nth ARRIVAL (recorded even when nothing was done).
```

Execute and update state. The executor changes state through operations on JSON fields. Setting a field replaces its value. Appending adds an entry to a collection. The excerpt is from `executor.yaml`, field `system_prompt`.

Executor: state updates

```
Your state is a **nested JSON object**. You modify it by emitting
`state_update` -- a LIST of action items applied in order. Each action is:

```
{"op": "<set|merge|delete|append>", "path": "<dotted.path>", "value": <any>}
```

- `path` is a **dotted path** into the nested tree, e.g.
  `"tickets.T-042.status"`. Use the empty string `""` to address the
  whole root.
- `op` is one of:
  - **`set`**: write `value` at `path`, replacing whatever was there.
    Parent dicts along the path are auto-created.
  - **`merge`**: deep-merge `value` (must be a dict) into the existing
    dict at `path`. Existing siblings stay. Overlapping keys get
    replaced. Use this to add/update a few fields of a nested entity
    without re-writing the whole record.
  - **`delete`**: remove the leaf at `path`. No-op if missing. Omit
    `value`.
  - **`append`**: push `value` onto the array at `path`. Creates an
    empty array if `path` doesn't exist yet.

Emit `state_update` ONLY when state genuinely changed. Omit it (or
send `[]`) when nothing changed -- do not invent updates.
```

Condition evaluation. An LLM checks whether a plan's branch condition holds for the supplied values. It does not score task completion. The verifier separately checks for missing actions and requests repairs. The excerpt is from `verifier.yaml`, field `condition_runner_stepwise`.

Condition evaluation

```
You decide ONE yes/no question: is the given CONDITION true or false for the data shown?

You are not the object and you do not do its work. Do not plan, route, record or
send anything, and do not decide what the object should do next. Answer only
whether the condition, as written, holds.

## How to decide

1. Read the condition. Say in a few words what fact it asks about and where that
   fact lives: the incoming event, a named step result (s1, s3.result, ...), a
   counting-tool result, or the object's stored state.
2. Find that fact in the data and quote the value you use.
   - A condition that names a step (for example "s2.result", "the shared state
     from s1") is answered from THAT step's result. Other state shown elsewhere is
     background. When it disagrees with the named step, the named step wins.
   - "The incoming event/message" means the Incoming event block only. Stored
     entries, attached source data and your instructions are not the message.
   - What happened for THIS event (answers from other objects, tool results) is
     in the step results: "Declared dependencies" and "Other completed-step
     results". The object state and the accumulated store hold what was saved
     from EARLIER events. When they disagree with this event's step results, this
     event's step results win.
3. Compare, and let `passed` follow from the comparison.
```

Implementation scope. The evaluated executor also receives instructions for each object's state and counting tools. The prompt permits reading another object's state. Appendix F explains how this differs from the private-state model illustrated in Appendix A.

D Benchmark Specification and Example

Each workflow has instructions, tools, an event stream with timestamps, and an instruction change partway through. Each event has an expected outcome. The system receives the event but not that outcome. Appendix E explains how we write the workflows and generate the events.

D.1 Workflow Specification

A workflow specifies its steps in NL, its tools, the initial data available through those tools, its agent graph, and its event stream. The steps describe the trigger, required lookups, actions, and memory-dependent rules. These are the instructions supplied to the systems, not a compiled execution plan.

Seed. The initial data contains records such as an owner directory and coverage roster. Tools can return these records, but they do not contain the answers to later events. Each run starts with the same data. The generator also records a random seed so the event stream can be recreated from its specification. The task-tracking example below uses seed `20260809`.

Events and expected fields. Each event contains an identifier, sender, recipient, timestamp, and NL input. Its expected behavior is stored separately as `expect.action`, the required tool calls and values, and `expect.reason`, why they follow from the rules and prior events. The event also records its test stage and whether its decision depends on accumulated state. Table 4 summarizes these fields. Expected behavior and scoring labels are withheld from the system.

Table 4: Fields in a benchmark event. The expected fields describe behavior, including required tool arguments or the absence of an action.

Field	Content
`id`, `when`	Event identifier and simulated time.
`source`, `recipient`	Sender and object receiving the input.
`input`, `call_type`	NL message and delivery type.
`expect.action`	Required calls, targets, and argument values, or no action.
`expect.reason`	Applicable rules and the counts or conditions behind the answer.
`role`, `after_mod_ids`	Test stage and instruction changes already delivered.
`is_gate_trigger`, `is_gate_flip`	Labels used to identify state-dependent decisions.

Agent breakdown. The authors manually divide each workflow into objects with named responsibilities, tools, state requirements, and peers. The graph separates duties such as intake, counting, choosing an action, and writing to an external system. The same graph is used by all multi-agent systems. Single-agent systems receive the complete workflow instructions.

Three stages. *Base* events test the original instructions. An ordinary message then changes a named rule while the workflow continues. *Post-mod* events test the changed rule. *Regression* events test behavior that should remain unchanged. In the stored records, regression events have the role `irrelevant`. This refers to the change's scope, not to inputs that should be ignored. Both later stages use new events, and state is retained across stages.

D.2 Memory Requirements and Construction Probes

The samples are designed to require the types of memory described in Section 4.2. During construction, deterministic probes test specific ways to fail. Their results are stored with the benchmark. These checks are separate from the later system evaluation.

Two interleaved counts. Every workflow tracks two counts at once: over different subjects, different clocks, or both, for example, a per-customer count over a rolling 7-day window and a per-team count over the calendar month. The two interleave in the event stream, so the system must maintain separate tallies, age one out over time, and reset the other at period boundaries.

Long-range dependence. Stored tests identify events whose answers depend on earlier evidence. For each event, we rerun an independent oracle after removing or changing that evidence. The answer must change. The current session or any one input shard must not be enough on its own.

Decoys. The stream includes events that appear to match a rule but must not count, such as rejected requests or sales below a stated minimum. A system that counts all arrivals instead of accepted work reaches the thresholds too early.

Controlled failure modes. The rule simulator replays a design with one capability switched off at a time (`src/data/rule_engine.py`). Some probes forget all prior events or retain only the current session or one input shard. Others track only one of two counts, disable window aging or calendar resets, count rejected requests or ignore eligibility, or ignore, pre-apply, or over-apply the instruction change. A further probe repeats a recent answer pattern. The packer also computes fixed-answer controls for always taking an action, never taking it, and choosing the most common outcome (`src/data/baselines.py`).

Results of the construction probes. For each probe, the stored results give the percentage of design-decisive events on which its outcome differs from the correct simulator. We compare the decision, target, applicable rule, and any action added by an instruction change. Across all 100 benchmark samples, the probes cover 3,493 design-decisive events. Table 5 shows how many samples had at least one difference. Some probes apply only when a design has the relevant feature. The benchmark builder uses these results to place events and check the fixed-answer controls. It does not require every probe to differ on every sample. No sample violates a fixed-answer control in the stored results.

Table 5: Deterministic construction probes across the benchmark. Counts are samples with at least one changed design-decisive outcome, divided by samples where the probe was recorded.

Limited policy	Samples with a difference
No prior event memory	100/100
Current session only	100/100
One input shard only	100/100
Only one count	100/100
Window handling disabled	84/97
Calendar reset disabled	3/3
Count rejected requests	14/14
Ignore eligibility precondition	0/2
Ignore instruction change	100/100
Apply instruction change retroactively	8/100
Apply instruction change beyond its scope	94/94
Repeat a recent answer pattern	100/100

The retroactive-change probe matches the correct outcome on the design-decisive events in most samples. We therefore cannot claim that every sample detects this error. Separate construction checks test agreement between oracles, whether earlier evidence changes the answer, and rejection of listed wrong answers.

D.3 Example: Out-of-Office Task Tracking

This example is taken from the benchmark sample `absent-owner-task-capture-n52-v4`. It contains 66 operational events from January 6 to April 18, 2026, and one instruction change on March 1. The following excerpts show the rules, state requirements, and expected actions.

Workflow. A Slack reaction with the emoji `:ooo-capture:` creates a task for an absent colleague. Other emoji reactions are ignored. The system reads the message, its identifier and link, and the named owner. It looks up the owner's coverage channel, coverage person, and handle. It then creates a task with normal priority and no watchers, subject to two rules:

1. From the owner's third task within a rolling ten-day window, set priority to urgent.
2. From the owner's sixth task across all time, add the coverage person to the watchers. This count never resets.

Steps. The workflow first checks the reaction, then reads the message and named owner, looks up the owner's coverage record, applies both count-based rules, and creates the task with the chosen priority and watchers. For an ignored reaction, it stops before counting or creating a task. The seed supplies the owner directory and coverage records, so expected destinations are fixed across systems.

Both counts include the task currently being handled. A reaction with another emoji changes neither count. Counts for different owners are interleaved in the event stream.

Memory required. The all-time rule needs a count per owner. The rolling rule also needs to know when earlier tasks leave the window. A single all-time total cannot answer both questions. The system must distinguish the owner, the time window, and whether the reaction qualifies before it updates either count.

Object graph. The graph has five objects. Intake filters reactions. Two objects track the ten-day and all-time counts. A policy object chooses priority and watchers. A writing object looks up the coverage record and creates the task. Both counting objects report to the policy object. Every multi-agent system receives the same graph.

Events and expected actions. Table 6 shows selected events. Each row is interpreted using the preceding events in the full stream. The counts include the current qualifying task.

Table 6: Selected events from the task-tracking workflow. Counts and expected actions are ground truth. The rows omit intervening events.

Event	Date	Owner	10-day	All-time	Expected action
S009	Jan. 8	Lucas	3	3	Create an urgent task, no watchers.
S047	Feb. 9	Jonah	2	9	Create a task with the coverage person as watcher.
S048	Feb. 9	Farah	–	–	Ignore the `:eyes:` reaction.
PM004	Mar. 12	Quentin	1	3	Create a normal task, no watchers.
PM005	Mar. 13	Quentin	2	4	Create a task with the coverage person as watcher.
IRR005	Apr. 8	Uma	2	4	Create a normal task, no watchers.

Instruction change. On March 1, the all-time threshold changes from six to four for Quentin. The ten-day threshold and every other owner's all-time threshold remain unchanged. The change is delivered as an ordinary message to the all-time counting object.

PM005 must therefore add a watcher at Quentin's fourth task. IRR005 must not do so at Uma's fourth task. Both have a ten-day count of two, so neither requires urgent priority. The system must apply the change to the named owner without changing the other rules.

Tools. `coverage_roster_data` returns the coverage records. `tasks_create` creates the task using its owner, channel, priority, watchers, title, message identifier, and link. The mock tools return fixed responses and record the arguments for evaluation.

D.4 Evaluation Evidence

The judge compares an event's expected behavior with recorded API calls. A reply that claims success or an internal message requesting an action does not count as a completed external action.

The scoring prompt is `config/prompts/llm_objects/judge_ledger.yaml` at revision `d3058c98`. The following excerpt states its evidence rule.

Evaluation judge: evidence

```
You are shown the API calls the system made -- the actions it performed on the
outside world -- and nothing else. There is no state document, no reply, and no
internal message traffic. This is deliberate: an action a workflow performs
becomes an API call, so the calls are what the system DID. Anything the system
might have said about its own behaviour is not evidence and is not shown.
```

The scoring rule accepts equivalent wording and checks the action, target, and required values. It distinguishes completed calls from attempts. It also accepts a derived value when the call arguments contain the inputs needed to compute it. This does not show that the system computed that value itself. The judge can use earlier API calls when the evaluation runtime supplies them. Appendix F gives the number of scoring votes and the evaluated sets.

E Benchmark Construction

E.1 Workflow Specifications and Event Streams

Workflow specifications. Each workflow is based on a public Zapier template, which supplies the domain, systems, and vocabulary. The authors write the process and steps in NL. They also specify the data in each event, the thresholds and time windows, the instruction change, and the tool calls that count as correct actions.

Event streams and ground truth. We generate each event stream from its specification using fixed rules. The specification sets the timestamps, entities, qualifying events, decoys, and tests of changed and unchanged rules. It also sets the expected tool calls for every event. `GPT-5.4` writes the events in NL. Automated checks preserve the facts needed for the answers, and the authors review every event for clarity and correctness.

Agent graphs. The authors manually define each workflow's objects, their responsibilities, and their connections. The same graph is used by POLO and every multi-agent baseline. Single-agent baselines receive the full workflow instructions. Appendix F.2 describes how the graph is used.

Memory checks and design probes. We check whether selected events depend on earlier evidence. During sample construction, deterministic probes replay the rule with one capability disabled at a time. The benchmark builder also checks that fixed-answer controls stay below set limits. Appendix D.2 reports the results and limits of these checks.

F Experimental Setup

F.1 Systems and Inputs

We compare POLO with LangGraph, Letta, CrewAI, and OpenClaw in single-agent and multi-agent setups. These frameworks use different forms of memory and can change instructions while keeping that memory. Single-agent systems receive the full workflow instructions. Multi-agent systems use the same graph written by the authors, with one agent per object and the same duties and connections. All systems receive the same events and instruction changes and use the same deterministic mock tools.

The main POLO comparison uses runtime revision `d3058c98`.

Memory configurations. LangGraph retains conversation messages through its checkpointer. Letta carries editable memory blocks alongside its conversation. CrewAI retrieves stored memory records. OpenClaw's single-agent run retains its session transcript. Its multi-agent run uses persistent files across events. POLO updates structured state between messages. These are comparisons of complete systems, including their execution and communication mechanisms.

Models. All agent calls use `GPT-5-nano` with low reasoning effort, including routing, execution, and condition evaluation. The exception is POLO compilation, which uses `GPT-5.4`. This choice reflects the latency and cost of processing ongoing workflows. Reported costs include compiler and router calls. Evaluation judges and benchmark generation are separate from the agent calls.

State in the evaluated runtime. The runtime stores each object's state separately. The executor prompt also gives it `read_shared_state`, a read-only tool for another object's state, and tools that count recorded events. These instructions are present in the saved run. The implementation therefore allows reads that the programming model's private-state rule forbids. Tools can also compute counts and window sums. Every baseline receives the same counting tools. The comparison does not test strict private-state enforcement or the models' ability to count without tools. The schema compiler receives the object's instructions, registered tools, and an example incoming message that shows the input format (Appendix C).

F.2 Manually Authored Agent Graphs

An agent graph lists the objects, their NL duties, their tools, and the peers they can message. The authors write one graph for each workflow. Every multi-agent system receives that graph, so graph design does not vary between systems.

Each node supplies its object's definition. An edge identifies a peer that it can contact. Entry objects receive external messages and pass work to other objects according to their instructions. For example, the task-capture workflow in Appendix D.3 has an intake object, two counting objects, a policy object, and a writing object. The two counts have different time windows and are kept by different objects.

Each event record names its recipient. Multi-agent systems deliver the event to that entry object. Single-agent systems deliver it to their only agent. The systems do not choose the graph or recipients during evaluation.

F.3 Environment and Judging

The tools represent external systems such as email, Slack, and task trackers. They return fixed responses from the workflow's data and record calls and arguments. Events are delivered in the specified order. Each workflow run starts with a fresh system instance.

After each graded event, the judge compares the expected behavior with the recorded API calls. Internal state, replies, and messages between objects do not establish that an external action occurred. Appendix D.4 gives the evidence rules. Every scored event, in the main comparison and in the ablations, is judged by one `Gemini 3.1 Pro` vote.

F.4 Metrics

Accuracy. Overall pass rate is the fraction of scored event instances that pass. Decisive events require accumulated state. Base events test the original workflow. Post-mod events test the changed rule, and regression events test behavior that should remain unchanged. These roles are separate from a chronological split into all events before and after the change. Following Yao et al. (2024), Pass^3 is the fraction of events that pass in all three repeated runs.

Uncertainty. We calculate confidence intervals by resampling workflows while keeping each workflow's events and repeated runs together. The intervals use 2,000 draws. We report the scored events alongside the results. An interval for each system does not show the uncertainty in the difference between systems, which requires a paired interval.

Cost. Input tokens per event include cached input tokens. $/event is total model cost divided by scored events. $/success is the same cost divided by passing events. Cost per correct state-dependent decision divides it by passing decisive events. These costs include failed attempts, compilation, routing, execution, and retries. They exclude evaluation judging in both the main comparison and the ablations.

F.5 Ablations

We replace folded state with a transcript, disable the compiler, disable the verifier, and place the workflow in one object. Bare ReAct and single-pass execution are additional reference systems that change several components. The matched subset, model assignments, and remaining verification check are specified in Appendix G.3. We report accuracy changes from full POLO on the same events.

G Additional Results

G.1 FlowStateBench: Detailed Performance and Cost

Evaluation coverage. Table 7 shows the runs and scored events for each system in Table 1. We exclude infrastructure errors and timeouts when a system never made an agent call. The POLO row includes three runs per evaluated workflow. Gemini judged all of them.

Table 7: Evaluation coverage behind Table 1: runs, scored events, and decisive events per system.

System	Agents	Runs	Events	Decisive events
POLO	Multi	120	7,490	3,480
LangGraph	Single	120	7,525	3,492
LangGraph	Multi	120	7,526	3,494
Letta	Single	120	7,533	3,498
Letta	Multi	120	7,498	3,483
CrewAI	Single	120	7,126	3,254
CrewAI	Multi	120	7,140	3,254
OpenClaw	Single	30	1,932	948
OpenClaw	Multi	30	1,932	948

The OpenClaw rows use three runs on ten workflows for each topology, with 1,932 scored events and 948 decisive events per topology. Input/output tokens and agent cost come from raw Azure per-call usage attributed by worker and event completion time. The per-event figures include calls assigned to scored events; they exclude modification calls and calls after the final scored event. One single-agent workflow run finished before a restart, and its Azure calls were not logged. It counts toward accuracy but not toward tokens and cost, which use the remaining 1,872 single-agent events. For OpenClaw single/multi respectively, attributed totals are 198.30M/66.32M input tokens, 191.8k/486.2k output tokens, and \$1.862/\$1.178 agent cost. The full raw Azure logs including those excluded calls contain 201.32M/67.94M input tokens and \$1.904/\$1.228 agent cost. Costs use the measured cached-token subset and the rates in `config/pricing.yaml`; judge cost is excluded.

The Letta multi-agent input/output tokens and cost come from a separate run with a per-call usage log, on the same ten workflows (one run, 644 scored events). In the multi-agent setup, Letta's own token count includes only the agent that receives the event, not the peer agents it calls, so it undercounts: on these events the full input is 6.8× and the output 11.4× the harness count. The per-call log records every call, and each call is assigned to its event by worker and event time. Calls made during instruction changes (11) and calls that fall outside any event (216) are excluded. The attributed totals are 167.05M input tokens, 3.39M output tokens, and \$9.709 agent cost, or 259.4k/5.3k tokens and \$0.015 per event. Accuracy still comes from the main runs. As for Letta single-agent, cost uses the rates in `config/pricing.yaml` without a cache discount; judge cost is excluded. For Letta single-agent, the per-call log matches the harness count on every event.

The POLO accuracy in Table 1 comes from a run with three state-recording votes per write. For token and cost reporting, we divide recorder usage by three to estimate one vote per write, leaving other calls unchanged. Token and cost totals include compilation, routing, execution, and state recording. The judge tokens were excluded from the total tokens count and cost. Model reasoning accounts for 77–81% of POLO's output across its components.

Compilation and routing use. The compiler uses `GPT-5.4`. Routing, execution, and condition evaluation use `GPT-5-nano` with low reasoning effort. A router checks if an incoming message has a

plan and if not, the compiler creates a plan. Instruction changes invalidate stored plans, and repairs can require further compilation.

Composition of an executor call. Only the saved state carries information from earlier events. The rest of the executor's input is either fixed (harness instructions) or changes only with the instructions (tool and response schemas, object definition) or the event kind (plan). The executor's input tokens per call remain stable across event positions (Figure 2).

Table 8: **Executor call input by part.** Share of input tokens over 774 executor calls (mean 10.1K tokens per call).

Part	Changes with	Share (%)
Tool and response schemas	instruction change	36
Harness instructions	never	23
Object definition (behavior, role, peers, tools)	instruction change	10
Plan	event kind	11
State (saved and working)	event	10
Event text and fetched data	event	10

Saved state over a workflow. In the concession-review sample workflow, the saved state holds 221 tokens after event 10 and 297 after event 60. Time-window lists keep only recent entries, and pending items are removed once decided.

The LLM-object not only accumulates information but also clears it as needed. The `directorate` object keeps a list of items in a rolling window. At the 60th event point, this list has multiple empty entries as the window has moved past the last items for most directors. The small executor model also makes mistakes: it saves one key under different spellings (`Amara-Diop`, `Elif_Sahin`), leaves empty items (`REQ-1013`), and removes or keeps items at the wrong time.

Saved state, concession-review: 10th event

```
concession-feed:   pending: {REQ-1004: {submitter: Darius Metz, sum: 5000}, REQ-1006: {submitter: Carys Evans
      , sum: 4000}}
pool counter:      granted_concessions: [{when: Jan 12, id: REQ-1001, amount: 8000}, {Jan 15, REQ-1002,
      8000}]
directorate:       {Amara-Diop: [{Jan 12, REQ-1001, 8000}], Benoit Claire: [{Jan 15, REQ-1002, 8000}]}
```

Saved state, concession-review: 60th event

```
concession-feed:   pending: {REQ-1018, REQ-1021, REQ-1019, REQ-1020: {submitter, sum}, REQ-1013: {}}
pool counter:      granted_concessions: [{Mar 29, REQ-1028, 7000}, {Apr 5, REQ-1031, 8000}]
directorate:       {Amara-Diop: [], Benoit Claire: [], Carys Evans: [], Elif_Sahin: [], ...8 empty..., Jules
      Perrin: [{Mar 29, REQ-1028, 7000}]}
```

Accuracy by event position. In the main evaluation, decisive-event pass rates vary by event position (Table 9). Later events may also be harder, so these rates do not measure the effect of session length alone.

G.2 Error Analysis

Table 10 defines six causes by where the first incorrect rule, value, or action appears. An event can have several causes.

Table 9: Gate-event pass rate by event position in the main evaluation.

Event positions	Pass (%)
1–10	62.0
11–20	58.0
21–30	62.9
31–40	55.4
41–50	45.0
51–60	27.4
61–70	38.6

Table 10: Root-cause taxonomy. These categories describe causes by where the first incorrect rule, value, or action appears.

Cause	Definition	Evidence needed
Compilation error	Compiled rules omit or change required behavior.	Compare the instructions with the accepted plan or memory rules.
Memory-label error	A read and write use different labels for the same value.	The required record exists, but the query reads another key.
Memory-update error	A required update is missing, extra, or assigned to the wrong count.	Compare the event and action with the resulting stored state.
Decision-use error	A correct decision is lost before the final action.	A correct count or branch is followed by a conflicting message or argument.
Call-execution error	A tool call or message to another object is omitted or sends wrong values.	Compare the planned action, outgoing messages, and recorded calls.
Instruction-update error	An instruction change removes rules that should remain.	Compare the old instructions, requested change, and applied replacement.

G.3 Ablation Setup

Each ablation variant was run three times on a workflow subset chosen to cover the benchmark's types of state rules and a range of decisive-event frequencies. Every variant is compared with full POLO on the same events. We also ran the variants without a compiler on the full workflow set because they cost less. On the full set, removing the compiler lowers overall accuracy by 38.2 points, bare ReAct by 33.5, and single-pass execution by 35.1. These changes have the same direction as in Table 2. Gemini judges each event once.

In the ablations, full POLO uses `GPT-5-nano` with low reasoning effort for routing, execution, and condition evaluation. It uses `GPT-5.4` for compilation. Table 2 reports differences from this full configuration on matched events. The 95% confidence intervals resample workflows and keep each workflow's events together. Costs include all agent-side model calls for compilation, routing, execution, condition evaluation, and retries. They exclude the scoring judge.

The single-object variant retains compilation and verification. Bare ReAct and single-pass execution change several components, so they serve as reference systems rather than single-component ablations. Removing the compiler removes its `GPT-5.4` calls and the executable plan. The verifier ablation disables repairs for ready steps left unexecuted, missing outgoing calls, and message steps closed without a message. A separate executor repair for omitted tool calls remains active, so this ablation does not remove every repair.

Table 11 gives the paired 95% confidence intervals for the differences in Table 2. Table 12 gives the matching token and cost changes.

Table 11: Paired 95% confidence intervals, in percentage points, for the accuracy changes in Table 2. Intervals resample workflows (2,000 draws) and keep all events of a workflow together.

Change	Δ Overall	Δ Decisive	Δ Non-dec.
Transcript instead of folded state	-23.4 to -0.7	-42.8 to -1.7	-17.6 to $+6.5$
No compiler	-42.0 to -12.7	-54.2 to -13.2	-38.1 to -7.7
Verifier disabled	-38.1 to -6.5	-55.5 to -10.6	-34.5 to $+0.5$
Single object	-12.2 to $+21.2$	-9.8 to $+30.9$	-16.4 to $+18.5$
Bare ReAct	-35.8 to -2.4	-61.6 to -12.8	-25.2 to $+6.0$
Single-pass execution	-39.2 to -8.0	-61.2 to -15.3	-31.0 to $+0.7$

Table 12: Changes in input/output tokens per event and agent cost from full POLO on the same events. Negative token values mean fewer tokens, and negative cost values mean lower cost. Costs include compilation, routing, execution, and retries but exclude evaluation judging.

Change	Δ Input/output tokens/event	Δ $/event	Δ $/success
Full POLO			
Transcript instead of folded state	+219k/−9k	+0.002	+0.037
No compiler	−22k/+1k	−0.037	−0.031
Verifier disabled	−31k/−2k	−0.006	+0.064
Single object	−105k/−12k	−0.029	−0.061
Compiler on `GPT-5.4-mini`	−130k/−7.5k	−0.007	+0.009
Compiler on `GPT-5-nano`	−96k/−4.3k	−0.010	+0.061
Bare ReAct	−138k/−19k	−0.050	−0.093
Single-pass execution	−218k/−20k	−0.052	−0.103

G.4 MEMORA: QUERY COST

The cost figures below cover one persona for each duration. They are not averages across the benchmark.

Fewer tokens per question in initial profiling. In these examples, self-authored state uses 8.0% of the plain single-agent transcript's input tokens per monthly question and 6.2% per quarterly question (Table 13). Across 155 to 1,991 sessions, its question-answering input stays between 15.4K and 20.7K tokens. Building the memory still makes POLO more expensive overall than the plain transcript.

Table 13: Preliminary input tokens per question, not averages across personas. POLO uses one profiled persona per duration. Plain-transcript tokens are counted from the answer prompt. *The quarterly transcript is truncated to the context window.

	Weekly	Monthly	Quarterly
Sessions	155	615	1,991
POLO (self-authored)	20,715	18,704	15,437
Plain transcript	59,880	232,882	249,825*
POLO / plain transcript (%)	34.6	8.0	6.2