POLO: Live Natural Language Programming with LLM Objects

Anonymous ACL submission

Abstract

The dominant paradigm for building software with LLMs is agent-assisted code generation. In this work we ask not whether LLMs can produce software, but whether they can *be* it. We propose to replace traditional software objects with LLM-powered counterparts. The computation model behind object-oriented programming is based on the metaphor of entities with internal private state that communicate by sending messages to each other. With LLMs, this metaphor can be realized directly, having the objects communicate in natural language. As in Smalltalk, messages sent to objects can also communicate behavior changes. Thus, object behavior can also be defined via natural language, and objects can respond in natural language to queries about their behavior. The act of programming now becomes communicating with objects in natural language to query, define, and modify their behavior. The result is a flexible living system that can be modified while it is running and being programmed without any consideration of syntax. The system we propose is also asynchronous and concurrent, borrowing ideas from the actor model. We provide a concrete realization of this approach, including the technical details of asynchronous execution and release software implementing this paradigm. We empirically demonstrate the efficacy of this proposal in the realm of *workflows*, i.e., automation processes defined by individuals for diverse real-time purposes. We deliver a benchmark of 498 automation scenarios and show that our POLO paradigm empirically outperforms the OpenClaw baselines on both workflow completion and live modifications.[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 and Atkinson, 2024). But rather than having LLMs *produce* software, we ask: can LLMs *be* the software?

We propose *Programmable Online LLM-Objects* (POLO), a paradigm in which programs are collections of *LLM-objects*. Like OOP objects, LLM-objects are persistent, encapsulate domain logic, keep private state, and communicate by message passing — yet each live Object is realized as an LLM agent rather than a coded and compiled class. The collection of LLM-objects *is* the software.[2]

This reframes the role of code in LLM-driven software. In contrast to the current paradigm of communicating with agents in natural language in order to create and modify an external software system, in POLO the program is the NL definition; code, when used, is a tool invoked by an LLM-object, not the program itself. The LLM does not *produce* the program; it *is* the runtime executing it.

An *NL program* is a collection of LLM-objects, defined entirely in natural language (name, role, tools, etc). LLM-objects borrow their computational model from object-oriented programming and the actor model (Goldberg and Robson, 1983; Hewitt et al., 1973): objects exchange messages, and in response to a message, an object can create new objects, send messages, and update its state. These primitives are sufficient to express an arbitrary program (Hewitt, 2010). Once running, the system listens for external events — user inputs, arriving emails, new orders — and forwards them as messages to LLM-objects, whose action initiates further message exchanges.

An appealing property of this paradigm is that programs are *live*: they can be inspected and modified while running. Because behavior is specified in NL and interpreted by an LLM, an LLM-object can answer questions about its own behavior ("re-

[1] We will release our code and data upon acceptance.

[2] The name POLO echoes Plain Old Java Objects (POJOs) (Fowler, 2005), which brought flexible programming components to enterprise software.

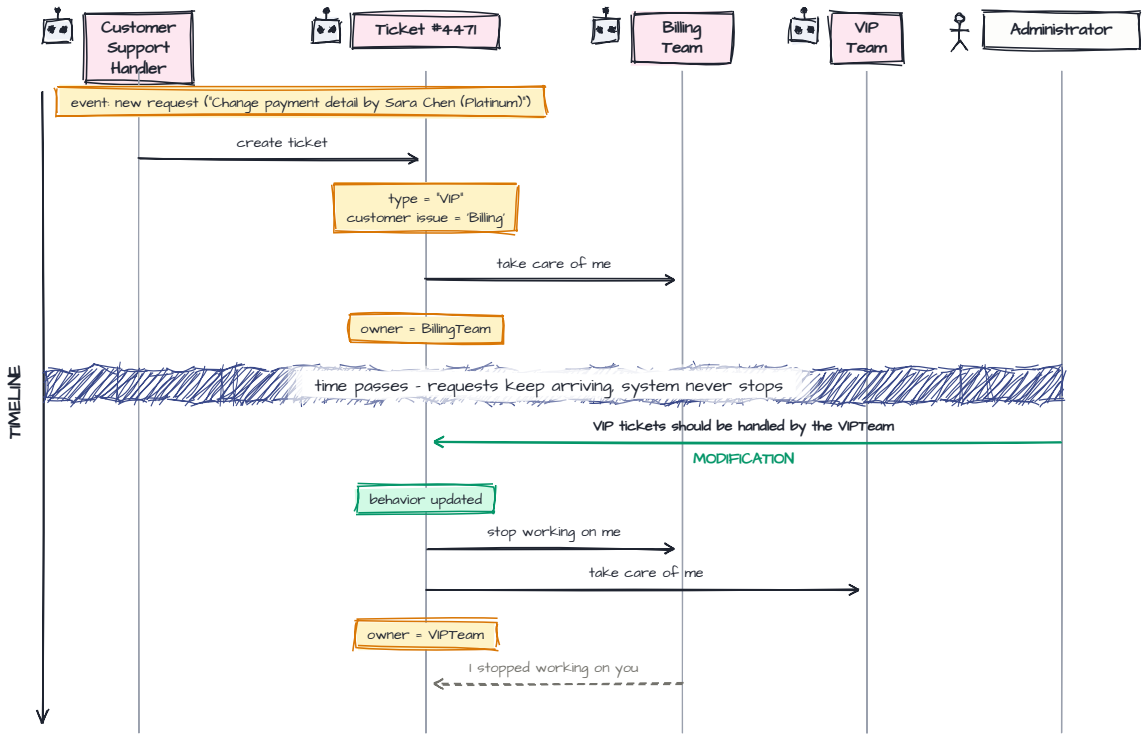

Figure 1: Live program of Programmable Online LLM-Objects. *Customer Support Handler*, *Ticket #4471*, *Billing Team*, and *VIP Team* are all *LLM-objects* — persistent, stateful entities defined entirely in NL. Objects communicate asynchronously with one another through natural-language messages rather than fixed APIs. An event triggers the system, a new ticket is created, and *Ticket #4471* routes itself to *Billing Team*. While the program runs, the *Administrator* sends a natural-language message that alters *Ticket #4471*'s behavior at runtime (labeled MODIFICATION) — editing the live program rather than configuring its execution. The change applies retroactively: *Ticket #4471* re-routes to the newly-relevant *VIP Team* without restart, migration scripts, or state loss.

flection") and accept modifications to that behavior — all through the same message interface, with no restart and no state loss. Unlike traditional software, where the system must be stopped, modified, and restarted to apply changes (Section 3).

LLM-objects are superficially similar to assistant-agents like OpenClaw (OpenClaw, 2026), which are also collections of live objects defined in NL, responding to NL messages, and able to send messages to each other. However, they were designed for short, local operations involving few tools and few inter-agent interactions, not for serving as building blocks of a software with control flow and business logic.

In order to empirically assess the efficacy of POLO, we focus on *workflows* — automation processes defined by individuals for diverse purposes. Consider for example the following workflow, that aims to automate the process of handling support tickets. The administrator can define a few LLM-objects in NL — a Customer Support Handler, a Support Team, a Billing Team, and a VIP Team — instantiate them, and lets the system run (Figure 1). Requests then arrive continuously; each becomes a ticket that asks to be routed. While the system runs, the administrator can query it (*"Billing Team, how many open tickets do you handle?"*) and modify it by sending behavior modification requests (*"VIP tickets should be handled by the VIP Team"*). The change applies immediately and retroactively while requests keep arriving. The system evolves and adapts to new instructions while maintaining its state; it never stops, and no ad-hoc migration is needed. Another example is brought in Section 4.

We release Workflow-bench, a benchmark of 498 automation workflows grounded in real-world scenarios from Zapier, an automation platform. We use it to measure how well POLO executes workflows and incorporates runtime modifications, comparing against OpenClaw single- and multi-agent configurations. Our results show that POLO outperforms both, widens its lead as workflow complexity grows, and runs faster than a single agent despite distributing work across multiple objects.

2 Architecture and Key Components

Our proposed paradigm is built around three main pillars. First, LLM-based elements that *form* the program (rather than generate the program). Second, these elements are defined and communicate solely in natural language. Third, a live program ammenable to updates runs continuously without losing state or resetting. This section describes the core components that realize these pillars.

2.1 LLM-Objects

An LLM-object is the core component of POLO, an agentic LLM harness defined entirely in NL. Each definition specifies the object's name, its role, the peers it communicates with, tools it can use, and the behavior that governs how it processes messages—all expressed in NL.

In contrast with past OOP objects, tht are defined by code that encapsulate logic and state, and interact with one another via message passing, LLM-objects modernize this approach with an LLM agent harness that encapsulates domain logic via NL definitions, owns a private state, and communicates with peer objects via messages in NL. LLM-objects can use code as a tool, but they are *not*, in and of themselves, a code artifact.

Unlike OOP objects, LLM-objects are *live*. Their definitions are not compiled in a static code; they are NL that an LLM interprets on every message. An administrator can modify an object's role, behavior, or peers at runtime, and the object continues from its current state. There is no restart, no state migration, and no separate control plane.

Each object maintains a private *state*—the domain data relevant to its logic, accessible only to the object itself. A Support Ticket, for instance, holds the customer info, a description of the issue, the urgency, and the current status. The object updates its state across messages (e.g., changing the urgency or status), and the state never resets. State is selective: the object keeps only what its logic requires. The state is not a full conversation history.

LLM-objects live for the duration of the program or until explicitly destroyed. New LLM-objects can be created at any point during execution, either by the system administrator or by other LLM-objects — giving the program the flexibility to grow and adapt as needed.

Responding to messages In POLO, the communication medium is NL, not code. *NL messages* are the exclusive medium of communication between LLM-objects. Messages sent to LLM-objects are similar to methods in object-oriented programming, with one major difference. In traditional OOP, the set of messages an object can accept is strictly defined: the methods it exposes and their parameters. The collection of methods determines the behavior of the object. In contrast, an LLM-object's role and behavior determine which messages it can meaningfully respond to, and how. The object interprets each message in light of its NL description and acts on it—sending a message to a peer, invoking a tool, updating its state—or indicates it is unable or not allowed to do so.[3]

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.

2.2 Communication Protocol

Natural language is unstructured and ambiguous; an LLM-object interprets each incoming message in light of its own definition, its state, and the sender. Messages fall into two categories by purpose: *operational* messages carry the program's work, while *administrative* messages modify the program itself. We define a formal NL message-delivery protocol. Each message is a tuple $\langle from, to, intent, payload \rangle$, where the payload is an NL utterance or a tool-call response. The *intent* field captures the conversational shape: *Ask*—a request for information or action that expects a reply; *Tell*—a notification with no reply expected; and *Response*—a reply to a prior Ask.

Communication follows the Actor model (Hewitt et al., 1973): an object that sends an Ask does not block waiting for the Response, and the runtime correlates each Respond with its originating Ask when it arrives. Tool invocations are modeled the same way—an Ask to the tool, returned as a Response. Synchronous tool APIs are wrapped to fit this model.

[3] In the current implementation, the LLM performs computations and state updates directly, but this is not integral to the computation model. In other implementations, the LLM could do so by writing and executing code in a programming language.

2.3 Execution Model

When an LLM-object receives an operational message, it begins processing it. But reliable execution against an object's definition is not something LLM inference provides by default—a single pass often misses parts of the work or halts early. The object processes a message in three phases: *plan* produces a series of steps to fulfill the message, *execute* carries them out, and *verify* checks the execution against the plan. If the message was a Response, the object skips plan and simply continues the execution. Each step is a discrete action—a tool invocation, a message to a peer, or an internal computation. Steps that issue tool calls or peer messages yield immediately; the object resumes when the Response arrives. The LLM-object ends processing by emitting a *finish* block with status *ok* or *failed*: state updates take effect only on a successful finish, and failed processing leaves the object's state unchanged.

2.4 A Natural Language Program

An *NL program* is a collection of persistent LLM-objects, their state, and their interactions. The program has no static code artifact: its logic is the LLM-objects' NL definitions, which evolve as objects are modified during execution. There is no central orchestrator—the program's structure emerges from each object's role and peer declarations, which together dictate how processing flows across the program.

2.5 Runtime

The runtime is an interactive system — the medium — to define LLM-objects, and interact with them through NL administrative messages. Its design draws on Smalltalk (Goldberg and Robson, 1983): a uniform computational medium, a live-persistent system image, and message passing as the sole mechanism of computation—here, with NL as the message medium.

3 Implementation

In the previous section, we described the principles and design of the paradigm. In this section, we describe our implementation of these principles and design.

3.1 The Plan–Execute–Verify Loop

For an LLM-object to *be* the program, its execution must reliably follow its definition—a property LLM inference does not provide by default. We implement a *plan–execute–verify* loop, with each phase a separate LLM call, to close this gap. A *plan* is a DAG $\pi = (V, E, r)$: nodes V are steps, edges E encode data-flow dependencies, and $r : V \to \{0, 1\}$ flags steps that require re-planning before execution proceeds past them. Each step carries a status $q_i \in \{\mathsf{pending}, \mathsf{done}\}$.

Planner. The planner $\mathcal{P}$ maps an incoming message m to an initial plan $\mathcal{P}(m) = \pi_0$ with every step pending. It runs once at the start of processing and is re-invoked in two cases: when the object's definition is modified during execution, and when the executor reaches a step with $r(s_i) = 1$.

Executor. The executor advances the plan one cycle at a time. Given the current plan, the object's state, the incoming message, and any verifier feedback, it produces

$$\mathcal{E}(m,\ \pi_k) = (a_{k+1},\ T_{k+1},\ \sigma_{k+1},\ \pi_{k+1}),$$

the outgoing messages a_{k+1}, the tool-call log T_{k+1}, the resulting state σ_{k+1}, and the updated plan π_{k+1} with completed steps marked done. The executor does not block within a cycle: peer messages and tool calls are issued asynchronously, with responses arriving in later cycles.

Verifier. The verifier $\mathcal{V}$ compares the execution against the plan, given the resulting state, and returns a verdict with feedback:

$$\mathcal{V}(a_{k+1},\ \sigma_{k+1},\ \pi_{k+1}) = (v_{k+1},\ \phi_{k+1}).$$

The verdict v is pass when every step the executor should have completed is marked done and the resulting state is consistent with the plan; otherwise it is fail, and the feedback ϕ names the gaps. On a fail, the loop returns to the executor with ϕ as additional input.

The loop. Processing begins with $\pi_0 = \mathcal{P}(m)$ and then alternates $\mathcal{E}$ and $\mathcal{V}$; on a fail verdict, the feedback ϕ is fed back to the executor. The loop halts on the first pass verdict, or after K cycles, where one cycle is an $\mathcal{E}$–$\mathcal{V}$ pair. This loop replaces ReAct (Yao et al., 2023) at the top level—async peer messages and tool calls break ReAct's synchronous observe-act cycle—though ReAct may still operate within a single $\mathcal{E}$ call.

3.2 State Management

LLM-objects in our paradigm are persistent, long-lived entities. This state is crucial for them to make informed decisions, adapt to changing circumstances, and maintain continuity in their behavior. State is implemented as a JSON object, updated by the executor in delta form rather than full replacements—a design that avoids context bloat as state grows. The planner and verifier have read access to the state, while the executor has read-write access.

3.3 Live Modifications

An administrative message is handled by the *Modifier* module—a single LLM call that revises the targeted part of the object's definition (role, behavior, or peer declarations). The change takes effect immediately. Plans in progress are flagged for re-planning by the planner on the executor's next cycle.

4 Programming with LLM-Objects

In this section, we show how a POLO program of LLM-objects is written, run, and changed while running. The full step-by-step walkthrough appears in Appendix A.

4.1 Example: Operations Monitoring

Consider an example: a small business wants to automate the monitoring of its operations. They use a service that posts alerts to an internal dashboard whenever something looks off—a payment fails, a page errors out, an order can't be placed. Alerts arrive continuously, 24/7, and each alert carries a short description of what happened.

Requirements. When a new alert arrives, the system must listen to incoming alerts as they appear on the dashboard, then classify each one along two dimensions: a *severity* (low, medium, high, or critical) and an *affected area* (website, online store, or back office). Once classified, the alert is sent by email to the person on duty for that area—**Sarah** for the online store, **Kevin** for the website, and **Debby** for the back office— including the alert description and the assigned severity. The system then waits for acknowledgment from the recipient: 5 minutes for high or critical alerts, 1 hour for low or medium ones. If the window passes without acknowledgment, the alert is escalated by phone call to the recipient's manager, reporting the alert and the missed acknowledgment. Once acknowledged, the alert is closed, and no further action is taken.

Create the LLM-object program. An automation for this task consists of four LLM-objects. `Alert Watcher` listens to the dashboard's alert feed and turns each new alert into a message in the program. `Alert Classifier` receives the alert from `Alert Watcher`, reads its description, and decides its severity and affected area; it then sends the classified alert onward to the person on duty responsible for that area (Figure 4.1). There is one duty object per area—`Website Duty`, `Store Duty`, and `Backoffice Duty`—each carrying its area's on-duty person, that the person's manager, and the alerts it is currently tracking. A duty officer notifies the on-duty person by email, waits for acknowledgment within the severity-dependent window, and escalates to the manager by phone if the window passes; once acknowledged, the alert is closed. Finally, `Notifier` handles the outbound channels—sending emails and placing phone calls on behalf of the duty officers—keeping the channel mechanics separate from the business logic.

Alert Classifier definition

```
## Name
You are Alert Classifier.

## Role
Your role is to classify each incoming operations alert
      along two dimensions: a severity (low, medium, high,
       or critical) and an affected area (website, online
      store, or back office), then forward it to the duty
      officer responsible for that area.

## Behavior
On a new alert from Alert Watcher, read the alert
      description and decide its severity and affected
      area. Send the alert, with the classification
      attached, according to its area: website alerts to
      Website Duty, online store alerts to Store Duty, and
       back office alerts to Backoffice Duty.

## Peers
- Website Duty: dispatch alerts classified as website.
- Store Duty: dispatch alerts classified as online store.
- Backoffice Duty: dispatch alerts classified as back
      office.
```

Updating an object defition. Following our example, Sarah, the online store's duty person, goes on vacation. Her responsibilities need to be temporarily transferred to Kevin, the website duty person. The administrator updates the `Store Duty` object by sending it a new message that describes the change in plain NL, such as: "*Sarah is sick today, route online store alerts to Kevin!*". `Store Duty` updates immediately by changing its peer declaration to forward online store alerts to `Website`

Duty instead of itself. Alerts assigned to her are automatically transferred. The change is applied without any downtime or disruption to the system.

5 Benchmark

Workflows align naturally with POLO as they are defined in NL, span multiple domains, owned by non-developers, and are susceptible to changes at runtime. This makes them a compelling testbed.

5.1 Workflow-bench

We release WORKFLOW-BENCH, a workflow benchmark that is paradigm-agnostic: Any system that executes and modifies NL workflows at runtime can be evaluated.

What is a workflow? Workflows are how automation systems are specified in practice—tools like Zapier, n8n, and Make let non-developers describe how a system should respond to events. A workflow describes a process: when a new support ticket is created, classify it by content, route it to the appropriate team member (Alice for billing, Sarah for technical issues), and notify the team lead. As requirements evolve, workflows are modified while running: new steps may be added, existing steps changed or removed, and the conditions under which actions are taken updated (*"Skip Sarah, she is on vacation until Friday"*). Both the workflow and its modifications are specified in natural language.

Input. Each example is a tuple $\langle workflow, events, tools \rangle$: the *workflow* description in NL, *events* with explicit ordering, and the *tools* the system may call to interact with the environment. Events are of two kinds: external events that trigger workflow execution (e.g., an incoming email, a support ticket) and modification events that change the workflow (e.g., *"enterprise customers get priority routing"*). See Appendix E for a detailed example.

Expected behavior. Each example is paired with a ground-truth expected behavior: the state changes and tool calls a system executing the workflow should produce at each event. Systems are evaluated against this ground truth (Section 5.3).

5.2 Curation

We sourced 83 workflows from public Zapier templates, varying in size, from one-step automations to multi-step processes with conditional logic and chained events. Each was extended into 6 examples with modifications, yielding 498 examples (83×6).

Construction pipeline. Each example is built through a five-stage pipeline: grounding the workflow with concrete names for entities such as teams, systems, and people; generating modifications; generating events and validating the result. Four stages are LLM-driven, each followed by a manual sampling pass in which the authors inspect a fraction of outputs and either patch them or return the example for regeneration. See Appendix F for the full pipeline.

Modifications. Each workflow was extended with six modifications, one from each category: *Temporal* (activate/expire on time conditions), *Contextual* (adjust behavior based on entity attributes), *Exception* (override for specific cases), *Correction* (fix workflow logic), *Expansion* (add new behavior), and *Removal* (retire an existing behavior).

AI assistance and review. The pipeline used GPT-5.4. After each stage, the authors iteratively reviewed samples with an LLM-judge validator and manual annotation, regenerating or patching examples. See Appendix F.7 for quality measurements.

5.3 Evaluation

Each example introduces a modification event that changes the workflow while the system is running. We look at three kinds of triggering events: a baseline event that test the system under the original workflow (to establish a baseline), an event after the modification that tests its effect, and events unrelated to the modification that test unmodified behavior is preserved.

Timing. The system first receives the workflow and the tools. Events are then injected into the order the example specifies, independent of the system's state: each event is followed by a fixed wait before the next arrives.

Criteria. After each event, the observed state changes and tool calls are compared against the expected behavior for a binary pass/fail verdict. We use an LLM judge, but other methodologies are possible.

6 Experiments

6.1 Setup

We evaluate POLO on Workflow-bench, measuring whether LLM-objects can execute workflow logic,

incorporate modifications correctly, and maintain consistent state over time. OpenClaw (OpenClaw, 2026) is the only live multi-agent platform we are aware of—supporting persistent, modifiable agents at runtime, and serves as our primary baseline.

The Workflow-bench examples are paradigm-agnostic and do not include an object-level decomposition. We derive an *agent graph* for each example through a focused LLM call (Appendix G.1), specifying the entities, their descriptions, and their connectivity. POLO spawns one LLM-object per entity, with the provided description as its object prompt. Multi-agent OpenClaw spawns one agent per node. This shared decomposition isolates architectural differences from translation-quality variance. *Single-agent* OpenClaw receives the original workflow instructions as a single system prompt, without decomposition. External events and modifications are delivered to each system through its native input mechanism.

All configurations use `GPT-5.4-mini` at temperature 0.0; the LLM judge uses `GPT-5.4`. See Appendix G for the full configuration.

6.2 Evaluation Metrics

Pass rate. The fraction of events for which the LLM judge returns a passing verdict. We report pass rate separately for pre-modification events (**Base**), post-modification events (**On-Mod**), and post-modification regression events (**Off-Mod**).

Entropy. LLM inference is non-deterministic. We run each example three times and, per event, estimate the empirical pass probability p; the binary entropy $\overline{H} = -p \log_2 p - (1-p) \log_2 (1-p)$ measures behavioral consistency. $\overline{H} = 0$ means always pass or always fail; $\overline{H}$ peaks at $p = 0.5$.

Cost. We report the mean time for processing an event and the total input/output tokens.

Workflow complexity. To analyze how the system scales, we measure the number of distinct domains every workflow example touches (e.g., email, ticketing, team routing).

7 Results

POLO wins on execution and modification. POLO wins on all three event categories leading single-agent OpenClaw by 7–25 points and multi-agent OpenClaw by 19–35 points (Table 1). Prior work reports that single-agent systems often outperform multi-agent ones at modest task complexity (Tran and Kiela, 2026; Xu et al., 2026; Zhao et al., 2026); our results corroborate that ordering between the two OpenClaw variants, yet POLO—a multi-object system—surpasses both.

POLO's lead widens with workflow complexity. In the 3–6 domain range, POLO's completion rate stays above 49% while single-agent OpenClaw drops from 60% to 21% and multi-agent OpenClaw drops from 21% to 7% (Figure 2, left). At extreme complexity (7+ domains), OpenClaw variants collapse to near zero, while POLO still completes 17% of workflows.

Non-blocking execution keeps POLO fast. POLO issues more LLM calls per example than single-agent OpenClaw, yet runs 2× faster wall-clock time (169s vs. 349s) (Table 1). Multi-agent OpenClaw, configured in async mode, runs slower than single-agent OpenClaw (360s vs. 349s). The non-blocking design of LLM-objects (Section 3.1)—peer messages and tool calls yield immediately rather than serializing—it lets calls overlap. Compared to multi-agent OpenClaw, POLO uses roughly 2× the input tokens, reflecting the cost of plan-execute-verify reasoning per object.

Consistency reflects correctness, not the other way around. Multi-agent OpenClaw has the lowest entropy ($\overline{H} = 0.164$), but this reflects consistent failure rather than reliable behavior: it pairs with the lowest pass rates across the board. POLO and single-agent OpenClaw show similar entropy (∼0.30–0.35), indicating comparable run-to-run variation; POLO's higher pass rates mean its variation occurs at a higher correctness baseline.

8 Related and Future Work

8.1 Natural Language to Code

A large body of work uses NL as input to a code-generation pipeline, from semantic parsers (Quirk et al., 2015; Yin and Neubig, 2018) to neural code assistants (Chen et al., 2021; Li et al., 2022; Lai et al., 2023; GitHub, 2026): the artifact that runs is code, and modifying it means regenerating or editing. In POLO, NL *is* the program—an LLM interprets it on every message, and modifications are NL instructions applied to a running system.

8.2 Multi-Agent Frameworks

Multi-agent frameworks (Wu et al., 2023; Duan and Wang, 2024; Wang and Duan, 2024) and commercial agent systems (Anthropic, 2025b; Cur-

System	Base	On-Mod	Off-Mod	$\overline{\mathrm{H}}$	Time (s)	Tokens
Ours	**67.6 ± 3.1**	**57.0 ± 2.9**	**61.5 ± 3.6**	0.352	**169.39**	770,314/38,031
OpenClaw (single-agent)	43.7 ± 3.4	49.0 ± 3.1	49.2 ± 3.7	0.298	348.76	**74,375/1,723**
OpenClaw (multi-agent)	32.7 ± 4.7	42.6 ± 4.6	42.0 ± 5.6	**0.164**	360.11	336,042/4,910

Table 1: Results on Workflow-bench. POLO outperforms both OpenClaw baselines on pre- (**Base**), post- (**On-Mod**), and regression (**Off-Mod**) modification events.

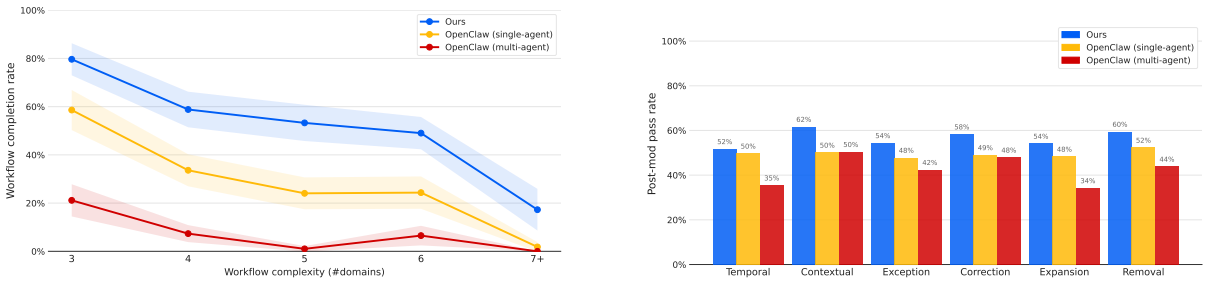

Figure 2: (Left) Completion rate is more robust as complexity grows for POLO than both OpenClaw variants. (Right) POLOhandles all types of modification consistently better than both OpenClaw variants. The performance gap is largest on the most complex modification type, which requires reasoning across multiple domains.

sor, 2026; OpenAI, 2026) coordinate LLM-backed agents to complete tasks. In POLO, agents run continuously, call each other like functions rather than passing artifacts, and accept runtime modifications without stopping.

8.3 Live AI Agents Systems

Live AI multi-agents platforms such as OpenClaw (OpenClaw, 2026) are the closest existing systems to Programmable Online LLM-Objects: agents are persistent, interactive, and modifiable at runtime. But the platform has no model of the program as a whole, no execution planning and no evaluation loop that is required when operating in heavy coordination. The consequence is in how an event entering the system is handled.

8.4 Long-running Systems

AI agents are increasingly deployed on long-running workflows spanning hours or days (Wang et al., 2025; Li et al., 2026; Anthropic, 2025a; Yu et al., 2026; Kwa et al., 2026). Over an extended horizon, agents must coordinate (Tran et al., 2025; Jaye et al., 2026), persist state (Du, 2026), and accept runtime changes without restart or state loss (Gao et al., 2025b,a). POLO contributes an orthogonal direction: a programmatic approach to coordination, runtime modification, and a benchmark to measure it.

8.5 Workflow Benchmarks

Recent agent benchmarks target realistic long-running workflows: OdysseyBench evaluates long-horizon office workflows (Wang et al., 2025); AgencyBench scales to million-token real-world scenarios with rubric-based deliverables (Li et al., 2026). None isolate runtime modification of a stateful program. WORKFLOW-BENCH adds this axis: the workflow definition is altered mid-run by NL modification requests, and the system is judged on whether the change takes effect while prior state and behavior are preserved.

8.6 Future Work

LLM-object is a new abstraction, and there are many avenues for future work. Some overarching future work borrows classical program-execution concerns for LLM-objects: lifecycle management (creation, destruction, and a semantically-driven garbage collector), object extension and composition, and error handling.

9 Conclusion

We introduced POLO, a runtime for programming multi-agent systems built around LLM-objects: persistent, stateful agents defined in NL and communicating asynchronously through NL messages. We release a benchmark of 498 workflow and modification cases derived from Zapier templates. POLO matches OpenClaw's single-agent baseline on simple workflows and maintains its pass rate as complexity grows, while both OpenClaw configurations degrade. Treating agents as live programming components opens a design space, this paper has only begun to map.

Limitations

First, our experiments were constrained by budget. Multi-agent configurations emit large volumes of tokens, and running the full benchmark across three systems configurations with multiple seeds imposed a substantial cost that limited the scope of further ablations.

Second, our findings may not generalize beyond the LLM family used in this work. We evaluate with `GPT-5.4-mini` for the systems under test and `GPT-5.4` as the judge, stronger or more varied model families could reveal different patterns of LLM-object interaction and shift the quantitative gaps we observe.

Third, at the time of this work, only a single live multi-agent platform —OpenClaw—was available for comparison. A broader baseline set, particularly other systems that support runtime modification, would strengthen the empirical picture.

Finally, WORKFLOW-BENCH is constructed with substantial AI assistance and partial human verification. The final dataset reaches a CLEAN verdict on 77% of examples; the remainder carry minor stylistic or causal-naming issues, documented in Appendix F. WORKFLOW-BENCH opens a new evaluation axis—runtime modification of stateful programs—and we expect the broader community to refine and extend it.

Ethical Considerations

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.

References

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

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

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

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

Zhihua Duan and Jialin Wang. 2024. Exploration of llm multi-agent application implementation based on langgraph+crewai. *arXiv preprint arXiv:2411.18241*.

Martin Fowler. 2005. Pojo. https://www.martinfowler.com/bliki/POJO.html.

Dawei Gao, Zitao Li, Yuexiang Xie, Weirui Kuang, Liuyi Yao, Bingchen Qian, Zhijian Ma, Yue Cui, Haohao Luo, Shen Li, Lu Yi, Yi Yu, Shiqi He, Zhiling Luo, Wenmeng Zhou, Zhicheng Zhang, Xu He, Ziqian Chen, Wei-Guo Liao, and 4 others. 2025a. Agentscope 1.0: A developer-centric framework for building agentic applications. *ArXiv*, abs/2508.16279.

Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Yiran Wu, and 1 others. 2025b. A survey of self-evolving agents: What, when, how, and where to evolve on the path to artificial super intelligence. *arXiv preprint arXiv:2507.21046*.

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

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

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

Carl E. Hewitt. 2010. Actor model of computation: Scalable robust information systems. *arXiv: Programming Languages*.

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. 2026. Corpgen: Simulating corporate environments with autonomous digital employees in multi-horizon task environments. *ArXiv*, abs/2602.14229.

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

Thomas Kwa and 1 others. 2026. Measuring ai ability to complete long tasks. *arXiv preprint*. METR.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR.

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

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, and 1 others. 2022. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*.

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

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

Chris Quirk, R. Mooney, and Michel Galley. 2015. Language to code: Learning semantic parsers for if-this-then-that recipes. In *Annual Meeting of the Association for Computational Linguistics*.

Dat Tran and Douwe Kiela. 2026. Single-agent llms outperform multi-agent systems on multi-hop reasoning under equal thinking token budgets. In *unknown*.

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

Jialin Wang and Zhihua Duan. 2024. Agent ai with langgraph: A modular framework for enhancing machine translation using large language models. *arXiv preprint arXiv:2412.03801*.

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

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. 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*.

Jiawei Xu, A. Koesdwiady, Sisong Bei, Yan Han, Baixiang Huang, Dakuo Wang, Yutong Chen, Z. Wang, Peihao Wang, Pan Li, and Ying Ding. 2026. Rethinking the value of multi-agent workflow: A strong single agent baseline. *ArXiv*, abs/2601.12307.

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

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

Pengcheng Yin and Graham Neubig. 2018. Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *Conference on Empirical Methods in Natural Language Processing*.

Chenglin Yu, Yucheng Wang, Songmiao Wang, Hongxia Yang, and Ming Li. 2026. Infiagent: An infinite-horizon framework for general-purpose autonomous agents. *ArXiv*, abs/2601.03204.

Yuxuan Zhao, Sijia Chen, and Ningxin Su. 2026. When does multi-agent collaboration help? an entropy perspective. In *unknown*.

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

A Appendix: Support Ticket Routing Example

We continue the example of the small business operations monitoring system introduced in the main text (Section 4). Here, we provide the full definitions for the LLM-objects in this system: Alert Watcher, Alert Classifier, Store Duty, Website Duty, Backoffice Duty, and Notifier. Each definition follows the same format as the Ticket Router definition in the main text, specifying the object's name, role, behavior, peers, and tools.

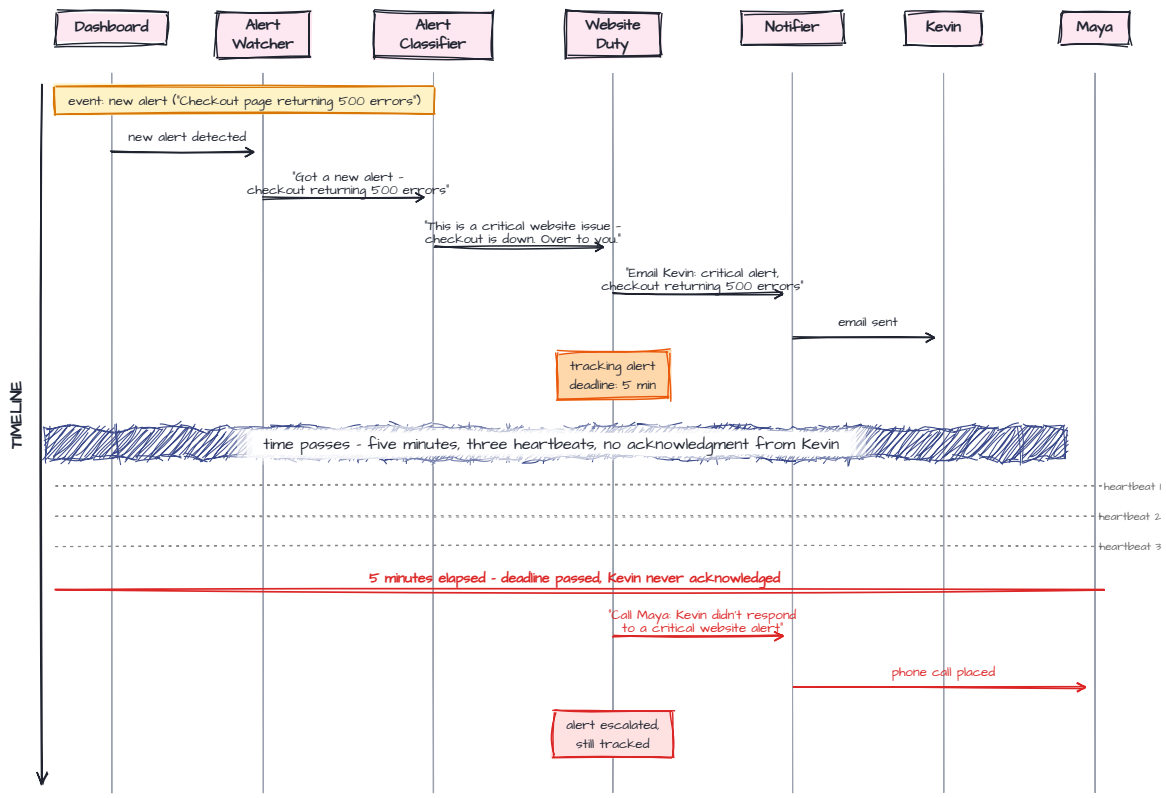

Figure 3: A critical alert flows through the program as a sequence of natural-language messages. `Alert Watcher` detects the alert, `Alert Classifier` reads it and routes by area, `Website Duty` tracks it against a five-minute deadline, and on heartbeat expiration escalates through `Notifier`. No central controller orchestrates the flow; each object decides what to send next.

A.1 LLM-objects Definitions

Alert Watcher definition

```
## Name
You are Alert Watcher.

## Role
Your role is to listen to the operations dashboard's alert
      feed and bring each new alert into the program for
      classification. You do not classify alerts yourself;
      you only detect them and pass them on.

## State
The set of alert ids you have already forwarded, so the
      same alert is never sent twice.

## Behavior
On every heartbeat, call fetch_new_alerts to retrieve
      alerts posted to the dashboard. For each returned
      alert whose id is not in your state, send a message
      to Alert Classifier containing the alert id and the
      alert description, then add the id to your state.

## Peers
- Alert Classifier: send each newly detected alert for
      classification.

## Tools
- fetch_new_alerts(): returns a list of alerts currently
      on the dashboard, each with an id and a description.
```

Website Duty definition

```
## Name
You are Website Duty.

## Role
Your role is to handle classified alerts for the website
      area. You notify the on-duty person by email, wait
      for acknowledgment within a severity-dependent
      window, escalate to the manager by phone if the
      window passes, and close the alert once acknowledged.


The on-duty person for the website is Kevin, email
      kevin@acme.com. His manager is Maya, phone
      +1-555-0142.

## State
For each alert you are currently tracking: the alert id,
      its description, its severity, the deadline by which
       acknowledgment is expected, and whether it has
      already been escalated.

## Behavior
On a classified alert from Alert Classifier, send a
      message to Notifier requesting an email to
      kevin@acme.com with subject "Alert: <severity>" and
      a body containing the alert description and severity.
       Add the alert to your state with a deadline of 5
      minutes from now if its severity is high or critical,
       or 1 hour from now if its severity is low or medium.
       Mark it as not yet escalated.

On every heartbeat, for each tracked alert whose deadline
      has passed and which has not yet been escalated:
      send a message to Notifier requesting a phone call
      to +1-555-0142 with a message stating the alert id,
```

```
      its description, its severity, and that Kevin did
      not acknowledge it within the window. Mark the alert
       as escalated. Keep tracking the alert until it is
      acknowledged.

On an acknowledgment event naming an alert id you are
      tracking, remove that alert from your state.

## Peers
- Alert Classifier: receive classified alerts for the
      website area.
- Notifier: request email notifications and phone
      escalations.
```

Store Duty definition

```
## Name
You are Store Duty.

## Role
Your role is to handle classified alerts for the online
      store area. You notify the on-duty person by email,
      wait for acknowledgment within a severity-dependent
      window, escalate to the manager by phone if the
      window passes, and close the alert once acknowledged.


The on-duty person for the online store is Sarah, email
      sarah@acme.com. Her manager is Maya, phone
      +1-555-0142.

## State
For each alert you are currently tracking: the alert id,
      its description, its severity, the deadline by which
       acknowledgment is expected, and whether it has
      already been escalated.

## Behavior
On a classified alert from Alert Classifier, send a
      message to Notifier requesting an email to
      sarah@acme.com with subject "Alert: <severity>" and
      a body containing the alert description and severity.
       Add the alert to your state with a deadline of 5
      minutes from now if its severity is high or critical,
       or 1 hour from now if its severity is low or medium.
       Mark it as not yet escalated.

On every heartbeat, for each tracked alert whose deadline
      has passed and which has not yet been escalated:
      send a message to Notifier requesting a phone call
      to +1-555-0142 with a message stating the alert id,
      its description, its severity, and that Sarah did
      not acknowledge it within the window. Mark the alert
       as escalated. Keep tracking the alert until it is
      acknowledged.

On an acknowledgment event naming an alert id you are
      tracking, remove that alert from your state.

## Peers
- Alert Classifier: receive classified alerts for the
      online store area.
- Notifier: request email notifications and phone
      escalations.
```

Backoffice Duty definition

```
## Name
You are Backoffice Duty.

## Role
Your role is to handle classified alerts for the back
      office area. You notify the on-duty person by email,
       wait for acknowledgment within a severity-dependent
       window, escalate to the manager by phone if the
      window passes, and close the alert once acknowledged.


The on-duty person for the back office is Debby, email
      debby@acme.com. Her manager is Raj, phone
      +1-555-0188.

## State
For each alert you are currently tracking: the alert id,
      its description, its severity, the deadline by which
       acknowledgment is expected, and whether it has
      already been escalated.

## Behavior
On a classified alert from Alert Classifier, send a
      message to Notifier requesting an email to
      debby@acme.com with subject "Alert: <severity>" and
      a body containing the alert description and severity.
       Add the alert to your state with a deadline of 5
      minutes from now if its severity is high or critical,
       or 1 hour from now if its severity is low or medium.
       Mark it as not yet escalated.

On every heartbeat, for each tracked alert whose deadline
      has passed and which has not yet been escalated:
      send a message to Notifier requesting a phone call
      to +1-555-0188 with a message stating the alert id,
      its description, its severity, and that Debby did
      not acknowledge it within the window. Mark the alert
       as escalated. Keep tracking the alert until it is
      acknowledged.

On an acknowledgment event naming an alert id you are
      tracking, remove that alert from your state.

## Peers
- Alert Classifier: receive classified alerts for the back
       office area.
- Notifier: request email notifications and phone
      escalations.
```

Notifier definition

```
## Name
You are Notifier.

## Role
Your role is to deliver outbound notifications on behalf
      of the duty officers. You send emails and place
      phone calls exactly as requested; you do not decide
      who to notify or what to say.

## Behavior
On an email request from a duty officer containing a
      recipient address, a subject, and a body, call
      send_email with those arguments.

On a phone call request from a duty officer containing a
      phone number and a message, call place_phone_call
      with those arguments.

## Peers
- Website Duty, Store Duty, Backoffice Duty: receive
      notification requests from each.

## Tools
- send_email(to, subject, body): send an email to the
      recipient with the given subject and body.
- place_phone_call(number, message): place a phone call to
       the number and read out the message.
```

A.2 Example Walkthrough

Walkthrough. Consider a single alert flowing through the program end-to-end. At 9:14 a.m., the monitoring service posts a new entry to the dashboard: *"Checkout page returning 500 errors."* On its next heartbeat, `Alert Watcher` fetches the new entry, sees that it has not been forwarded before,

records its id, and sends `Alert Classifier` the message *"Got a new alert — checkout page returning 500 errors."* `Alert Classifier` reads the description, reasons that a broken checkout is a critical website issue, and forwards it to `Website Duty`: *"This is a critical website issue — checkout is down. Over to you."* `Website Duty` knows from its definition that Kevin is on duty for the website and that critical alerts must be acknowledged within five minutes; it asks `Notifier` to *"Email Kevin: critical alert, checkout returning 500 errors,"* and adds the alert to its state with a deadline of 9:19 a.m. `Notifier` sends the email and holds no further state.

For the next five minutes the runtime broadcasts heartbeats to every object. On each one, `Website Duty` scans its tracked alerts, finds the checkout alert still within its window, and takes no action. At 9:19 the deadline passes without acknowledgment from Kevin. On the next heartbeat, `Website Duty` detects the expired deadline and asks `Notifier` to *"Call Maya: Kevin didn't respond to a critical website alert — checkout is down,"* then marks the alert as escalated in its state. The alert remains tracked, because the requirement specifies that alerts are closed only on acknowledgment. End-to-end, every inter-object handoff was a natural-language message on the bus, interpreted by the receiving object's LLM brain according to its own role and state—no schemas, no field names, no central controller deciding what happens next.

B Appendix: Building an LLM-Objects Program

In POLO, a program is the constituent of its underlying LLM-objects. The program is assembled incrementally: the programmer defines objects one at a time, each in a plain Markdown file, and spawns them into a shared runtime. This mirrors the mental model users already hold about the world — independent entities, each with a clear role, communicating by passing natural-language messages. There is no schema to compile and no wiring step; objects discover each other through the peer declarations written in their definitions.

In this section, we walk through the process of building a simple LLM-objects program from scratch. We cover the mechanics of creating a program through a CLI specification, defining and spawning objects, registering listeners for external events, and modifying the system at runtime. The example is a three-object quote-approval workflow that integrates with HubSpot and Slack; the full definitions are included in the figures.

B.1 Creating an LLM-Objects Program

The runtime is a daemon process installed once on the system. The programmer interacts with it through a CLI that supports creating programs, adding objects, sending messages, and inspecting state. The runtime hosts multiple concurrent programs, each in its own namespace.

A program is created in three steps: declare the program, register its object definitions, and start it. For a simple program with two objects:

```
polo program new myWorkflow
polo myWorkflow new object1.md
polo myWorkflow new object2.md
polo myWorkflow start
```

The first command creates a new program namespace. The next two register the object definitions from local Markdown files; each file is parsed by the runtime and the resulting object is added to the program's registry. The final command starts the program: the runtime instantiates the registered objects, connects them to the message bus, and the system begins running.

Once running, the programmer can interact with the program through further CLI commands—sending messages to objects, querying their state, or adding new objects at runtime. The CLI's `program` flag selects the target program for each command; if omitted, the CLI defaults to the most recently active program. The next sections walk through the process of building the LLM-objects within the app.

B.2 Defining and Spawning Objects

Each object is a Markdown file with three sections: **Role** (what the object is), **Behavior** (how it responds to messages), and **Peers** (the objects it is allowed to contact). The programmer writes these in natural language; the runtime turns each file into a live agent.

```
$ polo <program> new --id <object name> --description <
     description> --peers <peer1,peer2,...>
Spawned: <object name>
```

Creating an object is a two-step process: write the markdown file definition, then spawn it into the runtime. One can create a new object directly from the CLI with `polo new`, which generates a template Markdown file. The programmer fills in the sections and saves the file; the runtime detects

the change, parses the new definition, and spawns the object automatically.

```
$ polo <program> new --id <object name> --description <
      description> --peers <peer1,peer2,...>
Spawned: <object name>
```

B.3 A Three-Object Quote Approval Workflow

We illustrate with a three-object quote-approval program. When a sales rep submits a quote in HubSpot, the system identifies the required approver based on the deal size, posts an approval request to Slack, and — once the manager responds — updates the quote status in HubSpot and notifies the rep. Three objects cover the full lifecycle: `hubspot-quotes` is the entry point that receives new submissions; `quote-approvals` holds the approval policy and orchestrates the response; and `slack-notifier` is a write-side sink that records every outbound Slack message. Figure 4 shows the resulting message flow.

Object 1 — HubSpot Quotes. The first object is the workflow's entry point. It listens for new quote submissions from HubSpot and forwards each one to the approval policy.

hubspot-quotes.md

```
# HubSpot Quotes

## Role
Receives new quote submissions from sales reps and
      forwards them for approval processing.

## Behavior
When a new quote submission arrives, record the submitting
       rep, deal amount, and concessions in state, then
      forward the full submission to quote-approvals.

## Peers
- quote-approvals: Forward new quote submissions for
      approval processing.

## Event Sources
- hubspot: new-quote-submitted
```

Figure 5: Definition of the `hubspot-quotes` object. The **Event Sources** declaration registers a listener for new HubSpot submissions; the runtime delivers each event as a natural-language message. The single peer declaration wires the object into the approval cascade.

```
$ polo myWorkflow new objects/hubspot-quotes.md --provider
      openai --llm gpt-5.4-mini
Spawned: hubspot-quotes
```

Object 2 — Slack Notifier. The second object represents the Slack write-side: it records every notification routed through it.

slack-notifier.md

```
# Slack Notifier

## Role
Records all notification messages sent through it. Acts as
       a write-side service representing Slack.

## Behavior
When a message is received, record it in state with the
      sender and content. Keep a running log of all
      messages.
```

Figure 6: Definition of the `slack-notifier` object. No peers are declared: this object only accepts inbound write requests and maintains its own log.

```
$ polo my automation new objects/slack-notifier.md
Spawned: slack-notifier
```

Object 3 — Quote Approvals. The third object holds the approval policy and orchestrates the workflow. Its **Peers** section names the two objects already running; the runtime uses these declarations to route outgoing messages.

quote-approvals.md

```
# Quote Approvals

## Role
Processes quote approval requests and notifies the
      required approver via Slack.

## Behavior
When a quote submission arrives, check the deal amount
      against the approval rules and notify the required
      approver via slack-notifier.

Approval rules:
- Quotes under $1,000: auto-approved
- Quotes $1,000-$10,000: notify the submitter's direct
      manager
- Quotes over $10,000: notify the VP of Sales

## Peers
- slack-notifier: Send approval notifications and requests
       to the required approver.
```

Figure 7: Definition of the `quote-approvals` object. All approval logic is expressed as natural language in **Behavior**; the single **Peers** entry is the only wiring the programmer writes.

```
$ polo myWorkflow new objects/quote-approvals.md
Spawned: quote-approvals
```

B.4 Listeners and External Events

Peer declarations wire objects to each other; **Event Sources** declarations wire objects to the outside world. When the runtime spawns an object that contains an `## Event Sources` section, it reads each line as a descriptor of the form `<provider>`:

<event-type> and registers a *listener* for that stream.

A listener is a thin adapter that sits outside the object model. It subscribes to an external system — a webhook endpoint, a polling loop, a message queue — and, whenever a relevant event occurs, translates it into a natural-language message and delivers it to the registered object. The object receives the event the same way it receives a peer message: as plain text, routed through the message bus. No glue code is required from the programmer.

```
$ polo myWorkflow provider openai new objects/hubspot-quotes.
      md
Spawned: hubspot-quotes
Listener bound: hubspot/new-quote-submitted -> hubspot-quotes
```

The binding is resolved at spawn time from a runtime registry that maps provider names to listener factories. If no registered factory matches a descriptor, the runtime falls back to an injectable listener that can be triggered manually — useful for testing without a live external connection (see §B.5). Listeners are released automatically when an object is stopped.

This design keeps external-integration concerns out of the object definitions entirely. The programmer declares *what* event a given object cares about; the runtime decides *how* to deliver it.

B.5 Triggering the Workflow

With all three objects live, an incoming HubSpot event triggers the cascade automatically via the registered listener. The programmer can also inject an event directly to test the program:

```
$ polo myWorkflow provider openai send hubspot-quotes \
    "Alice submitted a $5,000 quote for Acme Corp."
```

The runtime delivers the message to `hubspot-quotes`, which records the submission and forwards it to `quote-approvals`. That object determines that $5,000 falls in the manager-approval tier and sends a notification to `slack-notifier`: *"Approval required from Bob (Alice's manager) for her $5,000 Acme Corp quote."*. The programmer never specified this cascade; it emerges from the natural-language definitions alone.

B.6 Modifying an Object at Runtime

Definitions can be updated while the program is running. The programmer sends a plain instruction to any object; the runtime injects it as a behavioral update without restarting the system:

```
$ polo --provider openai send quote-approvals \
    "For purchase orders above $10,000, also CC carol@company.
      com
    on every approval notification."
```

Subsequent messages to `quote-approvals` reflect the new rule immediately; prior state is preserved. This live-edit capability is the defining feature of POLO: the programmer interacts with a running program the same way they would instruct a human colleague.

C Appendix: POLO Agent Prompts

All three prompts use `{placeholder}` notation for runtime-injected fields.

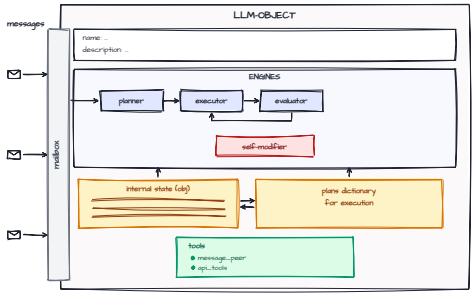

Figure 8: Architecture of an LLM-object.

Planner Prompt

```
You are a planner. Your single job is to produce a multi-
      step execution plan for an LLM-object so it can
      handle an incoming event reliably across
turns. You DO NOT execute any action; another component
      will execute the plan. Independent steps will be
      dispatched IN PARALLEL in the same turn whenever
      their dependencies are satisfied -- design your plan
       as a DAG, not a sequence.

# The object you are planning for

## Identity
- object_id: {object_id}

## Role
{role}

## Behavior (what this object does when it receives an
      event)
{behavior}

## Declared peers (the ONLY object_ids you may target for
      tell/ask steps)
{peers}

## Current state (operational state at this moment)
{current_state}

# The incoming event you are planning the response to

Sender: {sender}
Type: {message_type}
Content:
{message_content}
{prior_plan_context}
{replan_mode_note}

# Your task

Produce a numbered, multi-step plan describing the actions
       this object must take to handle this event. Each
      step is a SINGLE atomic action.
Steps form a directed acyclic graph (DAG) via their `
      depends_on` lists -- every step whose `depends_on`
```

```
      is empty (or whose dependencies have all completed)
      is eligible to dispatch in the same turn as every
      other eligible step.

**Four step kinds:**
- `tell` / `ask` -- dispatch to a declared peer (target =
      peer_id)
- `tool`         -- invoke a registered tool (target =
      tool name)
- `reason`       -- record a state requirement or non-
      dispatch decision
                     (target = "self")

Note: a fifth kind `wait` exists for multi-stage workflows
       that span asynchronous external events. Do NOT
      author wait steps in this initial plan -- the
      executor amends the plan with wait steps from inside
       its iteration loop. Plan the steps for the FIRST
      event you have been given.

## Output format -- strict JSON, no other text

{{
  "goal": "<one short sentence summarizing what the plan
      accomplishes>",
  "steps": [
    {{
      "id": "s1",
      "step_number": 1,
      "kind": "tell" | "ask" | "tool" | "reason",
      "target": "<peer-id | tool-name | 'self'>",
      "description": "<one short sentence: what to
      dispatch, call, or record>",
      "depends_on": [],
      "reasoning": "<detailed reasoning: why this step
      exists, what data
                     it carries, what role it plays in
      completing the workflow>"
    }},
    {{
      "id": "final",
      "step_number": <N>,
      "kind": "final",
      "target": "final",
      "description": "All required actions complete; no
      further actions.",
      "depends_on": [],
      "reasoning": "<why the workflow is complete after
      prior steps>"
    }}
  ]
}}

**Step IDs:**
Every executable step must have a stable string id in the
      form `s1`, `s2`, `s3`, ... -- sequential, 1-indexed,
       no gaps. The `final` marker uses id "final". Ids
      are immutable; later steps reference earlier results
       by id (e.g. "post the URL from s2.result to channel
      ").

**The order of steps in the JSON list does NOT determine
      dispatch order.**
The runtime dispatches every step whose `depends_on` is
      empty (or whose deps have all completed) in the SAME
       turn, regardless of list position. Author `
      depends_on` deliberately -- every dep you add
      removes a parallelism opportunity.

## Planning principles (follow ALL)

1.  One action per step. A step dispatches to exactly one
      peer OR describes one state requirement. Never
      combine actions.

2.  `depends_on` encodes data flow, not preference. For
      every candidate dep, apply this test: Does step sN's
       payload reference a specific field, id, URL, status,
       or value produced ONLY by step sM's reply or tool
      result?
    YES -> sN.depends_on must include "sM".
    NO  -> sM MUST NOT appear in sN.depends_on, even if sM
       "logically comes first." Extra deps serialize work
      and add a full LLM turn of latency per false dep.

3.  Independent dispatches must be parallel. Every step
      that does not pass the test in principle 2 gets
      depends_on: []. Common scenarios:
    - Multi-peer fan-out: notify A, B, C with the event
      payload -> three tell steps with depends_on: [] each.

    - Multi-peer ask fan-out: ask A, B, C for status ->
      three ask steps with depends_on: [] each (provided A
      's question doesn't embed B's answer).
    - Write + notify with shared input: if the
      notification uses the ORIGINAL event payload (not a
      tool-returned id), the tool step and the tell step
      both have depends_on: [].
    - reason does not gate dispatches. A reason step
      records state; do not add it as a dep of tell/ask/
      tool steps.

4.  Use only declared peers for tell/ask. The target of
      every tell/ask step must appear in the Declared
      peers list. Never invent peer IDs.

5.  Every step has detailed reasoning. The reasoning field
       explains WHY this step exists, WHAT data it carries,
       and WHAT it accomplishes.

6.  The last step is always "final".

7.  Tools and peers MUST be exercised. If this object has
      tools AND its role contains action verbs (write,
      send, post, store, notify, persist, publish, append,
       update, trigger, log, deliver), the plan MUST
      include at least one tool step. A reason-only plan
      for such an object is a bug.

8.  Do not execute. description is a short summary, not
      the full payload.

9.  Peers MUST be dispatched to when behavior requires it.
       reason/tool steps supplement but never replace
      dispatch steps.

10. Conditional steps must declare the condition in
      description. Use `if`, `when`, `only when`, or `
      provided that`. The executor evaluates the condition
       before dispatching; without an explicit condition
      clause the step is treated as unconditional.

11. Sinks (zero declared peers): use tool steps (with all
      required args) or reason steps with concrete field-
      level requirements.

## Anti-patterns (do NOT do these)

- "Sequential vibes" deps: s3.depends_on: ["s2"] because
      s2 "happens before" s3 in the narrative, even though
       s3 reads only the original event. -> Use [].
- Same-source serialization: two tell steps carry the same
       event payload but are chained. -> Both get [].
- Reason-as-prerequisite: putting a reason step as a dep
      of every dispatch step. State recording is not a
      value dispatches consume.
- Tool-first ordering: always chaining tell after tool
      even when the tell does not use the tool's return
      value.
- Bundling independent asks into a chain.

## Example: fan-out orchestrator (two independent peer
      dispatches)

{{
  "goal": "Forward the captured event to two downstream
      peers in parallel",
  "steps": [
    {{"id":"s1","step_number":1,"kind":"tell","target":"<
      peer-A>",
      "description":"Forward event payload to <peer-A>",
      "depends_on":[],
      "reasoning":"Behavior says <peer-A> persists the
      record. Independent of <peer-B>."}},
    {{"id":"s2","step_number":2,"kind":"tell","target":"<
      peer-B>",
      "description":"Notify <peer-B> of the event",
      "depends_on":[],
      "reasoning":"Behavior says <peer-B> sends
      notifications. Independent of <peer-A>."}},
    {{"id":"final","step_number":3,"kind":"final","target
      ":"final",
```

```
      "description":"All required dispatches complete; no
      further actions.",
      "depends_on":[],"reasoning":"s1-s2 cover every
      required peer and ran in parallel."}}
  ]
}}

## Example: chained step where depends_on IS required

{{
  "goal": "Write record to storage and notify channel with
       the row id",
  "steps": [
    {{"id":"s1","step_number":1,"kind":"tool","target":"
      write_row",
      "description":"Write order_id, customer_name, items,
       status=completed",
      "depends_on":[],"reasoning":"Behavior requires
      persistence; tool returns row_id needed by s2."}},
    {{"id":"s2","step_number":2,"kind":"tell","target":"<
      peer-channel>",
      "description":"Post completion notice including
      row_id from s1.result",
      "depends_on":["s1"],
      "reasoning":"Channel post must reference the
      persistent row_id, which only s1 can provide."}},
    {{"id":"final","step_number":3,"kind":"final","target
      ":"final",
      "description":"All required actions complete; no
      further actions.",
      "depends_on":[],"reasoning":"s1 + s2 cover the full
      storage + notification requirement."}}
  ]
}}

## Example: ask fan-out (multiple independent queries,
      then combine)

{{
  "goal": "Gather status from three services, then notify
      on-call",
  "steps": [
    {{"id":"s1","step_number":1,"kind":"ask","target":"<
      service-A>",
      "description":"Query service-A current status",
      "depends_on":[],"reasoning":"Independent of B and C
      -- fans out in turn 1."}},
    {{"id":"s2","step_number":2,"kind":"ask","target":"<
      service-B>",
      "description":"Query service-B current status",
      "depends_on":[],"reasoning":"Independent of A and C
      -- fans out in turn 1."}},
    {{"id":"s3","step_number":3,"kind":"ask","target":"<
      service-C>",
      "description":"Query service-C current status",
      "depends_on":[],"reasoning":"Independent of A and B
      -- fans out in turn 1."}},
    {{"id":"s4","step_number":4,"kind":"tell","target":"<
      oncall>",
      "description":"Notify on-call with combined status (
      s1+s2+s3.result)",
      "depends_on":["s1","s2","s3"],
      "reasoning":"Combined payload embeds all three
      replies."}},
    {{"id":"final","step_number":5,"kind":"final","target
      ":"final",
      "description":"All required actions complete; no
      further actions.",
      "depends_on":[],"reasoning":"Three parallel asks +
      one downstream notify cover the behavior."}}
  ]
}}

Begin. Output ONLY the JSON object.
```

Figure 9: Planner system prompt. The planner decomposes an incoming event into a DAG-structured plan; steps with no unmet dependencies are dispatched in parallel within the same executor turn. {{...}} denotes literal JSON braces in the Python format-string template.

Executor Prompt

```
You are '{object_id}'.

## My Definition
Role: {role}
Behavior: {behavior}
Peers (the ONLY object_ids you may message): {peers}
Skills: {skills}
Event Sources: {event_sources}

## Tools
{tools}

## Operating Discipline
You are an executor. A planner produced your active plan
      decomposing the incoming event into ordered steps.
      An evaluator grades each step's sub-items after your
       response. Your job: execute the plan's current step
       accurately on each turn.

- tell/ask steps: emit the outgoing message with EVERY
      field the step names, addressed to the step's target
       peer.
- tool steps: emit a tool_call with EVERY argument named,
      taking values from the incoming message and prior
      step results.
- reason steps: emit a state_update setting EVERY field
      named, using completion terms (stored/written/sent/
      created), never readiness terms (pending/queued/
      ready_to_write).
- When graded FAIL on a sub-item, fix exactly that sub-
      item -- don't redo PASS work.

## Hard rules
- State and plan updates apply ONLY on action="finish".
- Never ask for missing information; infer, apply a
      sensible default, or record your assumption.
- Act on this message only; do not simulate future
      messages.
- Most-recent input wins over state or prior instructions.
- Action target != message sender.

## State
Own your state schema. Keep only operational data. Use
      queryable top-level
keys for entity collections (e.g., {"T-042": {status,
      priority, assignee}}).
  Adding:    set <id> = {<initial attrs>}
  Modifying: re-set <id> = {<full updated attrs>} (same
      key, always)
  Retracting (cancelled/closed/removed): delete <id>
Retraction is YOUR responsibility regardless of message
      source.

## Peer Communication & Completeness
- Fan-out is all-or-nothing: include EVERY required peer
      in outgoing_messages in a single finish.
- Send the complete payload verbatim; downstream peers
      cannot recover dropped fields.
- Intent != Action: "pending"/"queued"/"will send later"
      are FORBIDDEN as substitutes for real outgoing
      messages.

## Multi-stage workflows -- emit a `wait` step when a
      later event is expected
If your behavior has a two-clause pattern (first trigger +
       awaited approval/callback/confirmation) AND this
      turn handles the FIRST clause, emit a wait step via
      plan_update.add_steps:

  {
    "id": "s_wait",  "kind": "wait",
    "description": "Wait for <second trigger> for <entity
      id>",
    "depends_on": ["<last first-stage step id>"],
    "wait_predicate": "<NL description + concrete
      identifier>",
    "wait_source": "<expected sender/peer id>"
  }

## Response format -- strict JSON, no other text
{
  "thought": "<explicit reasoning about what to do this
```

```
    turn>",
  "action": "tool_call" | "finish",

  // if action = "tool_call":
  "tool_calls": [{"id", "tool", "arguments", "
      plan_step_index?"}],

  // if action = "finish":
  "finish": {
    "reply": "<reply to sender>",
    "outgoing_messages": [{"recipient", "content", "
      expects_reply?"}]
  },
  "state_update":  {"op": "set|delete|append", "key", "
      value"}, // optional
  "plan_update": {
            // optional
    "step_updates": [{"id": "s1", "status": "done"}],
    "add_steps": [...]
  },
  "updated_definition": ...  // only on Admin messages
      changing definition
}

## Active Plan
{active_plan}

## Current State
{current_state}
```

Figure 10: Executor system prompt. The executor carries out one plan step per turn and handles evaluator-driven self-correction cycles.

Evaluator Prompt

```
You are an evaluator. Your single job is to review what an
       LLM-object ("the executor") just did, and produce a
       STRUCTURED criterion-level grade with specific
      actionable diagnostics for any failures.
You do NOT execute, plan, or dispatch.

## The object being evaluated
Identity: {object_id}  |  Role: {role}
Behavior: {behavior}
Declared peers: {peers}

## The message the executor was processing
{incoming_message}

## The plan
{plan_section}

## What the executor just did (this turn's outputs)
State after this turn:  {current_state}
Outgoing messages:      {outgoing_messages}
Reply text:             {reply}
Tool calls executed:    {tool_calls_this_turn}

(tool_calls_this_turn is authoritative: if a tool name isn
      't here,
  it was NOT called -- state or reply text cannot override
       this.)

## Task
For EACH plan step, enumerate the required sub-items and
      grade each sub-item individually. ONE criterion per
      sub-item, NOT one per step.

Sub-item enumeration by step kind:
  tell/ask -- one criterion per required PEER + one per
      required FIELD
  tool     -- first: did the tool execute? (check the log)
       then (only if called): one criterion per required
      ARGUMENT
  reason   -- one criterion per required STATE FIELD /
      completion marker
  fan-out  -- one criterion PER DESTINATION
```

```
## Output format -- strict JSON, no other text
{
  "verdict": "PASS" | "FAIL",
  "criteria": [
    {
      "step_index": <0-based int>,
      "step_id":   "<e.g. 's1'>",
      "sub_item":  "<short NL label for what is being
      checked>",
      "status":    "PASS" | "FAIL" | "SKIP",
      "diagnostic":"<one sentence naming the specific
      missing item if FAIL>"
    },
    ...
  ],
  "feedback": "<summary of remaining work, or empty string
       if PASS>"
}

## Grading rules
- verdict PASS: every criterion is PASS or SKIP.
- verdict FAIL: any criterion is FAIL.
- SKIP: sub-item is conditionally not needed per the
      workflow.
- Tool execution criterion FAIL diagnostic MUST read:
  "Plan step <s_id> requires calling tool `<name>` but the
       executed-tool list shows it was NOT called. Call it
       now." Do NOT emit argument criteria if the tool was
       not called.

## Diagnostic quality bar
Bad:  "step didn't run" / "content is incomplete"
Good: "outgoing to slack-channel missing `summary` field;
      add it from the incoming message content"
Good: "tool call to write_row missing `customer_email`
      argument; the incoming message specifies email=
      jane@example.com"

## Things to IGNORE
- Cosmetic status label differences describing the same
      lifecycle stage.
- Steps already marked done with consistent state evidence.

- Self-dispatch errors; the synthetic `final` step.

## Things you MUST FAIL
- A planned tell/ask with NO outgoing AND no completion
      evidence.
- An outgoing that omits a required field.
- A tool step whose target does NOT appear in the executed
      -tool log.
- A reason step whose required state field is absent post-
      turn.
- Replies that defer ("will send later") without producing
       the needed artifact (URL, ID, link).
```

Figure 11: Evaluator system prompt. The evaluator grades each plan sub-item independently and returns structured PASS/FAIL criteria that drive the executor's self-correction loop.

D Appendix: Workflow Specification Example

A POLO program is driven by three kinds of input: *workflow steps* describe the intended behavior in natural language; *modifications* inject behavioral changes at a specified point in time; and *operational events* carry concrete payloads that trigger and probe the system. We illustrate all three using the HubSpot quote-approval workflow.

D.1 Workflow Steps

Workflow steps are written in plain natural language, one step per intended behavior. Each step describes a trigger class and the expected system response; concrete values (deal amounts, names, dates) are supplied later at evaluation time via event payloads.

Workflow: HubSpot Automated Quote Approval

```
{
  "workflow_id": "hubspot-quote-approval",
  "description": "Automates the quote approval lifecycle
      for a B2B sales team across HubSpot, email, and
      Slack.",
  "raw_steps": [
    "A sales rep submits a new quote for approval in
      HubSpot Quotes.",
    "The system identifies the required approvers based on
       the concessions requested and the rep's reporting
      chain.",
    "An approval-request email is sent to each designated
      approver.",
    "The approver reviews the quote and takes action --
      approve, reject, or request changes -- via a Slack
      thread.",
    "If approved, the quote is marked approved in HubSpot
      and the rep is free to send it.",
    "If concessions are not approved, the quote is marked
      rejected and the rep may resubmit.",
    "If no decision arrives within 24 hours, stakeholders
      are tagged in the existing Slack thread as a
      reminder."
  ]
}
```

Figure 12: Workflow specification for the HubSpot quote-approval example. Steps are trigger-response pairs written in natural language; the system infers from them which objects to create and how they should behave.

D.2 Modifications

A modification injects a behavioral change at a specified time. Each modification carries a `when` timestamp (relative to the evaluation timeline), a `mod_type` classifying the nature of the change, an `ambiguity` level indicating how much interpretation is required, and a free-text `intent`. `mod_type` and `ambiguity` are assigned by the data-generation pipeline; the LLM supplies only the `intent`.

Modification M001 — expansion / precise

```
{
  "id":        "M001",
  "when":      "W03-1T09:00",
  "mod_type":  "expansion",
  "ambiguity": "precise",
  "intent":    "When a quote is approved, in addition to
      existing steps, post a message to the #deal-wins
      Slack channel tagging the sales rep by name with a
      congratulatory note that includes the deal name and
      ARR amount."
}
```

Figure 13: M001 (`expansion` / `precise`) adds a new output path on approval. The instruction is fully specified; the system applies it with no interpretation required.

Modification M002 — temporal / vague

```
{
  "id":        "M002",
  "when":      "W04-2T08:00",
  "mod_type":  "temporal",
  "ambiguity": "vague",
  "intent":    "Send the 24-hour reminder a bit earlier so
      approvers have more time to act before the deadline
      ."
}
```

Figure 14: M002 (`temporal` / `vague`) adjusts reminder timing. "A bit earlier" is underspecified; the system must interpret a reasonable interval from context.

Modification M003 — contextual / precise

```
{
  "id":        "M003",
  "when":      "W05-1T09:00",
  "mod_type":  "contextual",
  "ambiguity": "precise",
  "intent":    "For enterprise deals (ARR above $100,000),
      also notify the CFO by email in addition to the
      standard approvers, regardless of discount level."
}
```

Figure 15: M003 (`contextual` / `precise`) adds a conditional approval path. The condition (ARR $>$ \$100k) and the action (email CFO) are fully specified.

D.3 Operational Events

Each event carries a concrete `input` payload and an `expect` block. The two fields are intentionally decoupled: `input` is delivered to the system; `expect` is withheld during execution and consulted only by the judge afterward to assess whether the system's observable output matches the intended behavior. `expect.action` describes the required terminal-state output; `expect.reason` provides a one-sentence causal justification for human auditing. Events are typed by `role`: `pre_mod` probes baseline behavior before any modification is active; `post_mod` probes the system after a specific modification has been applied; `irrelevant` tests functionality orthogonal to the current modification.

Event E001 — pre_mod

```
{
  "id":   "E001",
  "role": "pre_mod",
```

```
  "when": "W02-5T10:00",
  "input": "Priya Nair submitted a new quote for Vertex
      Systems.
            Deal: $95,000 ARR.
            Concessions: 16% discount, net-45 payment
      terms.",
  "expect": {
    "action": "1) Look up Priya Nair's reporting chain.
               2) Send approval-request email to required
      approvers.
               3) Post approval request to #deal-approvals
       .",
    "reason": "Quote triggers standard approval flow: 16%
      discount and net-45 terms both require approver sign-
      off."
  }
}
```

Figure 16: E001 (`pre_mod`) fires before M001 is applied and verifies the baseline three-step approval flow.

Event E003 — post_mod

```
{
  "id":           "E003",
  "role":         "post_mod",
  "when":         "W03-1T14:30",
  "triggered_by": "E002",
  "input": "Approver action received in #deal-approvals.
            Quote ID: Q-3112.
            Action: approve.
            Approver: Rachel Kim, VP of Sales. All
      required approvers have responded.",
  "expect": {
    "action": "1) Q-3112 status updated to approved in
      HubSpot.
               2) Confirmation email sent to sales rep.
               3) [M001] Message posted to #deal-wins
      tagging the rep with deal name and ARR amount.",
    "reason": "Standard approval flow executes (steps 1-2).
       M001 is active since W03-1T09:00, so the system
      must also post to #deal-wins (step 3)."
  }
}
```

Figure 17: E003 (`post_mod`) fires after M001 is active. Item 3 of `expect.action` is the new output introduced by M001 and is the primary assertion under test. `triggered_by: E002` indicates that this event is causally downstream of a prior quote submission.

E Appendix: Benchmark

Workflow-bench measures how well a system tracks the evolving behaviour of a real-world automation as it is amended over time. Each example fixes a baseline workflow, a sequence of in-flight *modifications* delivered in plain NL, and a stream of *operational events* that probe the system before and after each modification takes effect. The data is sourced from public automation templates (Zapier catalogues) so that workflows reflect the structure and vocabulary of real business processes rather than synthetic prompts; the modifications are written as the kinds of mid-flight instructions an operator might send to a human colleague.

Design choices. Three conventions distinguish Workflow-bench from prompt-driven evaluation: (i) *expectations are decoupled from execution* — each event's `expect` block is consulted only by the judge after the fact, so the system processes raw natural-language inputs without seeing the oracle (§E.5); (ii) *modifications are typed* along two orthogonal axes — `mod_type` (six categories from *temporal* to *removal*) and `ambiguity` (four levels from *precise* to *implicit*) — which lets us report per-axis pass rates and separate fidelity issues from interpretation issues (Table 3); (iii) *events are role-tagged* as `pre_mod`, `post_mod`, or `irrelevant`, so baseline fidelity and modification uptake can be scored independently (Table 4).

E.1 Construction Pipeline

Each example is produced by a six-stage curation pipeline that transforms an ungrounded automation template into a fully specified example (full details in §F). At a high level the stages are:

1. **Raw template** — a short, abstract step list scraped from a public Zapier catalogue.
2. **Grounding** — an LLM call replaces every placeholder with a concrete value (specific channels, thresholds, deadlines) and strips platform-specific vocabulary.
3. **Step authoring** — an LLM call writes the canonical workflow steps together with their expected terminal outputs.
4. **Modification generation** — an LLM call produces a natural-language `intent` for a modification whose `mod_type` and `ambiguity` are pre-assigned by the script.
5. **Event generation** — an LLM call produces the operational events that probe the modified workflow; the script assigns each event a `role`.
6. **Automated validation** — a battery of structural and stylistic validators gates the example; only examples passing all blocking checks enter the benchmark.

Every LLM-generation stage is followed by a manual sampling pass: a human reviewer inspects a fraction of the outputs and either patches the artifact in place or returns the example for regeneration.

E.2 A Worked Example: HubSpot Automated Quote Approval

We make the format concrete by walking through one Workflow-bench example end-to-end. The chosen workflow automates a B2B sales team's quote-approval lifecycle across email, Slack, and HubSpot — a pattern that exercises every component of the schema: external triggers (a new submission, an approver action, a deadline check), conditional routing (by deal size and concession profile), write-side notifications, and a deadline-driven reminder path that the modification will extend.

E.3 Workflow

The workflow captures how a B2B sales team routes quotes for approval. When a rep submits a quote, the system identifies the required approvers based on the concessions requested (e.g., discounts above a threshold, non-standard payment terms) and the rep's reporting chain, sends approval-request emails, and posts a structured request to a dedicated Slack channel (`#deal-approvals`). Approvers act in Slack (approve / reject / request changes); the system propagates the decision to HubSpot and notifies the rep by email. Quotes unanswered after 24 hours trigger a reminder tagging the relevant stakeholders in the original thread. The workflow is characterised by three distinct *trigger types*, each encoded as a step that specifies the trigger class and its conditional logic. Steps describe logic, not named instances; concrete values are supplied only at evaluation time via the test event payloads. Ground-truth terminal-output expectations are recorded separately and used exclusively by the judge (see §E.5).

S1. A sales rep submits a new quote for approval in HubSpot Quotes.

S2. The system identifies the required approvers based on the specific concessions requested and the rep's reporting chain.

S3. An approval-request email is sent to the designated approvers.

S4. The approver reviews the quote details and takes action—approve, reject, or request changes—in Slack.

S5. If the quote is approved, it is marked as such in HubSpot Quotes and the rep is free to send it.

S6. If concessions are not approved, the quote is marked rejected and the rep may resubmit.

S7. If a quote is not approved within 24 hours, the relevant stakeholders are tagged in the existing Slack thread as a reminder.

E.4 Modifications, Operational Events, and Tools

Modification. A single modification is injected into this example at `W03-1T09:00` (week 3, day 1, 09:00) — before events E002 and E003 are evaluated. It is typed `expansion` (adds a new output path) with `precise` ambiguity (no interpretation required).

Modification M001

```
{
  "id":        "M001",
  "when":      "W03-1T09:00",
  "mod_type":  "expansion",
  "ambiguity": "precise",
  "intent":    "When a quote is approved, in addition to
      the existing confirmation steps, also post a message
       to the #deal-wins Slack channel tagging the sales
      rep by name with a congratulatory note that includes
       the deal name and ARR amount."
}
```

Figure 18: Modification M001 (`expansion` / `precise`). Applied at `W03-1T09:00`; extends the workflow with a new `#deal-wins` notification on approval.

Operational events. Four events probe the system at different points relative to M001 (Table 2). Listing 19 shows the full record for E003, the key `post_mod` event.

Event E003 — post_mod

```
{
  "id":           "E003",
  "role":         "post_mod",
  "when":         "W03-1T14:30",
  "triggered_by": "E002",
  "input": "Approver action received in the #deal-
      approvals Slack channel.
            Quote ID: Q-3112.
            Action: approve.
            Approver: Rachel Kim, VP of Sales.
            All required approvers have responded.",
  "expect": {
    "action": "1) Quote Q-3112 status updated to approved
      in HubSpot.
                2) Email sent to sales rep confirming
      approval.
                3) [M001] Message posted to #deal-wins
      tagging the sales rep by name with deal name and ARR
       amount.",
    "reason": "The system executes the standard approval
      flow (HubSpot update + rep notification). Because
      M001 is active since W03-1T09:00, it must also post
      to #deal-wins  this third output is the post-
      modification assertion."
  }
}
```

Event	Input (excerpt)	Expected action (excerpt)
E001 `pre_mod` W02-5T10:00	Quote Q-3105 by Priya Nair; Vertex Systems, $95k ARR, 16% discount, net-45 terms.	(1) Look up Priya Nair's reporting chain. (2) Send approval-request email to required approvers. (3) Post approval request to `#deal-approvals`.
E002 `irrelevant` W03-1T10:00	Quote Q-3112 by Daniel Cho; Meridian Health, $210k ARR, 19% discount, net-60, two free onboarding sessions.	(1) Email sent to required approvers for Q-3112. (2) Approval request posted to `#deal-approvals`.
E003 `post_mod` W03-1T14:30 *triggered by E002*	Q-3112 approved by Rachel Kim, VP of Sales. All required approvers have responded.	(1) Q-3112 updated to *approved* in HubSpot. (2) Confirmation email sent to sales rep. (3) **[M001]** Message posted to `#deal-wins` tagging the rep with deal name and ARR amount.
E004 `irrelevant` W03-2T10:00	24-hour deadline check. Q-3105 still pending. Required approver: Tom Bassett, Sales Director.	Follow-up reminder posted in the existing `#deal-approvals` thread tagging Tom Bassett.

Table 2: Operational events for the HubSpot Quote Approval example. The **[M001]** annotation in E003 marks the new output that must be present for the event to pass; it is the primary assertion under test.

Figure 19: Full record for event E003 (`post_mod`). Item 3 of `expect.action` is the new behaviour introduced by M001 and is the primary assertion under test.

External service interfaces. The example exposes one scripted read tool and three write-side service endpoints.

Read tool (`org_directory_data`). A scripted mock that returns the employee directory on demand. Each record includes the employee's title, department, manager, full reporting chain, and per-employee approval authority thresholds (maximum approvable discount percentage; authority over non-standard payment terms; authority to grant complimentary services). This tool is invoked during S1 to identify the required approver(s) for the concessions in a submitted quote.

Write-side endpoints. Three external services receive outgoing messages from the system:

- **Email notifications.** Sends approval-request emails to the designated approver(s) and a confirmation email to the sales rep once all approvers have responded.
- **Slack `#deal-approvals`.** Posts structured approval-request cards with approve / reject / request-changes prompts; posts follow-up reminder messages when a quote exceeds the 24-hour deadline.
- **HubSpot Quotes.** Updates the quote's approval-status record once all required approvers have responded.

Modification M001 adds a fourth write-side interaction: posting a congratulatory note to Slack `#deal-wins` tagging the sales rep by name with the deal name and ARR amount.

E.5 Evaluation

Each example is evaluated event by event. The `expect` block is *decoupled from execution*: it is never passed to the system and plays no role in how the system processes the event. After the system produces its output, a judge receives the observable terminal-state alongside `expect.action` and returns a binary pass/fail. `expect.reason` provides a one-sentence causal justification for human auditing; it is not scored.

Step-level ground truth. Each workflow step carries an `expect.action` string describing the terminal-state output the system should produce for that trigger class. It is authored at benchmark-construction time and serves as the reference for all events whose `role` is `pre_mod` or `irrelevant`; `post_mod` events extend it with the additional output introduced by the active modification.

S1 — Quote submission. Approval-request email sent to the designated approver(s); structured approval-request card posted to `#deal-approvals` with approve / reject / request-changes prompts.

S2 — Approver action. Quote status updated in HubSpot (approved or rejected); sales rep notified by email of the outcome (free to send, or must resubmit).

S3 — 24-hour deadline check. Follow-up message posted in the existing `#deal-approvals` thread tagging the relevant stakeholders as a reminder.

The Judge. A single LLM judge is shared across all evaluated setups — POLO and the OpenClaw baseline — ensuring a consistent scoring standard. The judge receives the `expect.action` condition together with the system's observable evidence and returns a binary verdict with a one-sentence rationale. For the OpenClaw baseline we hook into the agent's internals via the OpenClaw SDK, surfacing tool invocations and agent state as evidence; for POLO we surface the same evidence from the message bus and object state.

Judge Prompt

```
You are an evaluation judge. Determine whether the
      condition is satisfied by the evidence.

Respond with JSON: {"passed": true/false, "reasoning": "
      one sentence"}

## Evidence sources (any one is sufficient)

- **Tool calls** are the PRIMARY confirmation of write-
      side actions. The "Tool calls:" section lists every
      external API invocation. A successful tool call with
       the right arguments IS the action -- you do not
      need any additional confirmation.
- **Message bus**: a message from object A to object B
      confirms A dispatched that work. A message TO a
      write-service object is sufficient evidence that the
       external action was performed.
- **Object state**: a recorded entry in an object's state
      confirms the action occurred. Object state is
      supplementary evidence -- its ABSENCE is NOT a
      failure when a tool call or bus message already
      shows the action happened.
- **Object registry**: maps each object ID to its role;
      use this to understand what external system an
      object represents.

CRITICAL: If the tool call appears with the right
      arguments, the action happened. Do NOT fail because
      "the object state is empty" or "the state does not
      record ..." -- state mirroring is best-effort
      bookkeeping by the agent. The tool call is ground
      truth.

## Empty / null conditions

If the condition is empty, null, the string "null", "
      action: null", or otherwise specifies no observable
      action -- automatically PASS with reasoning "no
      condition to verify".

## How to judge

- Match on INTENT and CORE FACTS, not exact wording.
      Semantic equivalence passes.
- Trust the system's routing. If the condition names a
      specific person or email and the agent dispatched
      via a generic placeholder that represents that same
      role in this workflow, treat it as PASS -- the
      routing identified the right role.
- Ignore UX elements (buttons, dropdowns, forms).
- Ignore minor content omissions if the core action
      occurred and the recipient, target, and primary
      subject are correct: missing optional fields,
      missing URLs/IDs/timestamps when the core action and
       recipient match, "stored as null" auxiliary fields
      when the required fields are stored.
- **Computed/derived values pass when the inputs are
      recorded.**
- **Superset answers pass.** Only fail if required
      information is missing or wrong.
- **Field-value matching is semantic, not lexical.** PASS
      when values describe the SAME FACT in different
      words. FAIL when values describe DIFFERENT FACTS:
      opposite decisions, distinct entities, or workflow
      stages that imply different downstream actions.

## Suppression / negative conditions

If the condition says an action SHOULD NOT happen and the
      evidence shows it DID happen via tool calls or bus
      messages, that is FAIL.

## Retraction / deletion

**Absence IS retraction.** If the condition says entity X
      was retracted and X does NOT appear in any active
      state dict, that IS sufficient confirmation. An
      entity appearing only in an audit log counts as
      retracted.

## Current-value queries (state-probe)

When the condition is a query about current state, judge
      against the OBJECT STATE section, not the message
      bus. Object state is the authoritative source for
      current values.

## Prior state

If a 'PRIOR STATE' section is present, use it to resolve
      abstract references in the condition before judging.
```

Figure 20: Judge prompt, shared across all evaluation setups. The judge receives `expect.action` and the system's observable evidence; it returns `{"passed": true/false, "reasoning": "..."}`.

Events are partitioned by `role` (Table 4) to separate the baseline signal from the modification signal. Modifications are characterised by `mod_type` and `ambiguity` (Table 3); both are assigned by the data-generation pipeline, not generated by the LLM. Modifications fall into six categories: (i) *Temporal* — activate or expire rules on time conditions (*"Skip Sarah, she is on vacation until Friday"*); (ii) *Contextual* — route or adjust behavior on entity attributes (*"enterprise customers get priority routing"*); (iii) *Exception* — override normal behavior for specific cases (*"Acme Corp tickets always go to the Customer Success VP"*); (iv) *Correction* — fix prior logic, potentially retroactively (*"threshold should be $100, not $50; reclassify last week's tickets"*); (v) *Expansion* — add behavior without changing existing logic (*"also send a weekly summary to the team lead"*); (vi) *Removal* — retire existing behavior (*"stop CC'ing the legal team on routine contracts"*).

F Appendix: Benchmark Curation

Each benchmark example is produced by a five-stage pipeline that transforms an ungrounded au-

Value	Description
mod_type	
temporal	Activate or expire rules on time conditions
contextual	Route or adjust behavior on entity attributes
exception	Override normal behavior for specific cases
correction	Fix prior logic, potentially retroactively
expansion	Add behavior without changing existing logic
removal	Retire existing behavior
ambiguity	
precise	Fully specified; no interpretation required
semantic	Requires interpretation of domain terminology
vague	Underspecified; multiple valid interpretations exist
implicit	Must be inferred from context; not stated directly

Table 3: Modification taxonomy: mod_type and ambiguity values.

role	Description
pre_mod	Fires *before* the modification is applied; tests that baseline behaviour is unaffected.
post_mod	Fires *after* the modification is applied; tests that the system correctly reflects the change.
irrelevant	Tests functionality orthogonal to the modification; expected output is unchanged regardless of which modifications are active.

Table 4: Event role taxonomy and its use in scoring. The benchmark reports pre_mod and post_mod pass rates separately to distinguish baseline fidelity from modification uptake.

tomation template into a fully specified example with modifications and events. Four of the five stages use a focused LLM call; the fifth is automated validation. Every LLM stage is paired with a *manual sampling pass*: a human reviewer inspects a fraction of the outputs, flags drift or low-quality generations, and either patches the artifact in place or returns the example to the previous stage for regeneration. This appendix walks through the pipeline using the HubSpot quote-approval template that runs through the rest of the appendix. The resulting example is paradigm-agnostic: steps, modifications, and events describe behaviour without committing to a particular agent decomposition. Adapting an example to a specific experimental configuration — in particular deriving the Objects graph used by POLO and the OpenClaw multi-agent baseline — is a separate step, described in §G.1.

F.1 Stage 1 — Raw Template

The pipeline ingests automation templates from public Zapier catalogues. Each template is a short list of plain-language steps describing the intended behaviour, with no concrete values, no system boundaries, and no expected outputs.

Raw template (Zapier)

```
id: deal-desk-manage-hubspot-quote-approvals-slack
name: HubSpot Automated Quote Approval
raw_steps:
  - A sales rep submits a new quote for approval in
      HubSpot Quotes
  - The system identifies approvers based on the specific
      concessions asked for and the rep's reporting chain
  - An email approval request gets sent to the designated
      approvers, and an actionable Slack approval message
      is posted to each approver
  - The approver reviews the quote and takes action ---
      approve, reject, or request changes --- in Slack
  - If approved, the quote is marked as such in HubSpot
  - If concessions aren't approved, the quote is marked
      rejected and the rep can resubmit
  - If quotes aren't approved in 24 hours, stakeholders
      are tagged in the thread as a reminder
```

Figure 21: Raw template, as scraped from Zapier. Every step is abstract: “the designated approvers”, “stakeholders”, “24 hours” are not resolved to specific channels, roles, or thresholds.

F.2 Stage 2 — Template Grounding

A first LLM call rewrites the steps with concrete values. It names specific Slack channels, picks concrete approval thresholds and deadlines, and strips Zapier-specific vocabulary. Step count and intent are preserved one-to-one.

Grounded template (excerpt)

```
grounded_steps:
  - A sales rep submits a new quote for approval in
      HubSpot Quotes.
  - The system identifies the required approvers based on
      the concessions requested (discount above 15%,
      payment terms longer than net-30, or complimentary
      services) and the rep's reporting chain.
  - An approval-request email is sent to each designated
      approver, and a structured approval request is
      posted to the #deal-approvals Slack channel.
  ...
```

Figure 22: Grounded template. Concrete channels (`#deal-approvals`), thresholds (15%, net-30), and concession categories replace the original placeholders.

Manual sampling. A reviewer checks each grounded template against its source to confirm one-to-one step coverage, replacement consistency within a workflow (e.g., a single channel name reused across steps), and domain-appropriate threshold values. Edits are made in place; the corrected text propagates to all downstream stages.

F.3 Stage 3 — Step Authoring

A second LLM call writes the canonical workflow steps and their expected terminal outputs. Each step pairs an external trigger class with the observable response the system should produce, expressed in terms of the mock-data identifiers the runtime resolves at evaluation time.

Canonical steps (excerpt)

```
steps:
  - id: S1
    trigger: A sales rep submits a new quote for approval
      in HubSpot Quotes.
    expect: Approval-request email sent to required
      approvers; structured approval card posted to #deal-
      approvals.
  - id: S2
    trigger: An approver acts on the quote in Slack.
    expect: Quote status updated in HubSpot; sales rep
      notified by email of the outcome.
  ...
```

Figure 23: Canonical steps with expected outputs. The `expect` field becomes the ground truth used by the judge (§E.5).

Manual sampling. Steps are checked against the grounded template for one-to-one coverage and against the mock data to confirm every named entity (approver, deal ID, channel) resolves to a runtime identifier. Steps that reference fabricated entities are rewritten.

F.4 Stage 4 — Modification Generation

A third LLM call generates modifications. The script controls `mod_type` (one of six categories) and `ambiguity` (one of four levels); the LLM produces only the natural-language `intent`, the `when` timestamp, and a stable identifier. Each modification targets exactly one step so that downstream events can be scoped cleanly to pre-/post-modification windows.

Modification M001

```
{
  "id":        "M001",
  "when":      "W03-1T09:00",
  "mod_type":  "expansion",
  "ambiguity": "precise",
  "intent":    "When a quote is approved, in addition to
      the existing confirmation steps, also post a message
       to the #deal-wins Slack channel tagging the sales
      rep with the deal name and ARR amount."
}
```

Figure 24: Generated modification. The script chose the type and ambiguity values; the LLM produced the intent text.

Manual sampling. Modifications are checked for type adherence (an `expansion` really adds, a `removal` really subtracts) and ambiguity calibration (a `precise` modification leaves no room for interpretation). Intents that drift from the requested type or ambiguity level are rewritten in place.

F.5 Stage 5 — Event Generation

A fourth LLM call generates the operational events that probe the modified workflow. Each event carries an `input` payload and an `expect` block; the script assigns the event `role` (`pre_mod`, `post_mod`, or `irrelevant`). Causal chains between events are recorded via `triggered_by` (e.g., E003 follows the submission posted in E002).

Event E003 (excerpt)

```
{
  "id":           "E003",
  "role":         "post_mod",
  "when":         "W03-1T14:30",
  "triggered_by": "E002",
  "input": "Approver action received in #deal-approvals.
            Quote: Q-3112. Action: approve.
            Approver: Rachel Kim, VP of Sales.",
  "expect": {
    "action": "1) Q-3112 status updated to approved in
      HubSpot.
               2) Confirmation email sent to sales rep.
               3) [M001] Message posted to #deal-wins
      tagging the rep with deal name and ARR amount."
  }
}
```

Figure 25: Generated event. `expect.action` is the ground truth used by the judge; the `[M001]` annotation marks the new output introduced by the active modification.

Manual sampling. Events are checked for two things: that `expect.action` cleanly distinguishes the pre- and post-modification behaviour (otherwise the example cannot discriminate), and that `triggered_by` chains form a valid DAG with no orphan parents. Events that fail either check are rewritten or regenerated.

F.6 Stage 6 — Automated Validation

The final stage applies a battery of automated validators to every example. Blocking validators reject structural defects: disconnected peer graphs, unreachable objects, dangling peer declarations, missing mock tools, and duplicate or malformed identifiers. Warning validators flag stylistic risks that are unlikely to block evaluation but can degrade signal: mock-data field mismatches, unnatural identifiers, and trigger configurations that would raise at runtime.

Blocking issues are either auto-patched by deterministic rules, repaired by hand, or removed; the example only graduates to the benchmark once all blocking validators pass. After Stage 6, the HubSpot quote-approval example contains seven canonical steps, one modification, and four events — a paradigm-agnostic example that can be replayed against any system that consumes the benchmark's input format. Adapting the example to a particular experimental configuration is described in §G.1.

F.7 Iterative Refinement Ablation

To quantify the contribution of the validation pipeline we measured benchmark quality across successive refinement passes on a 498-example draft. Each pass combines two ingredients: an automated LLM-judge validator (`validate_sample_events`) that flags candidate issues, and a **manual review-and-annotation step**. The review is deliberately broader than just the flagged set, and was carried out meticulously: at each pass reviewers also doubt the validator, sampling items the judge marked `CLEAN`, re-checking items whose verdict felt borderline, and cross-reading events against the workflow's modifications to surface defects the codebook does not capture. When a flagged item is confirmed as a real defect, the reviewer annotates the correct expected output directly; when an item is a false positive, it is cleared. This step is critical for three reasons: the validator itself carries non-trivial label noise (we measure this below); a fraction of flagged items are false positives that should be cleared rather than regenerated; and certain defect classes (input-level contradictions, mod-text ambiguity) require hand-patching that no rewrite can produce. Without it the loop drifts — the validator's mistakes get amplified by the next regeneration pass, and the items the judge silently misses never get caught.

The validator labels each event sequence with one of four verdicts and tags individual events with one of eight issue codes:

- **Verdicts.** `CLEAN` (no issues), `MILD_ISSUES` (minor stylistic flags), `PARADOX` (mutually contradicting events), `INCOMPLETE` (missing observable output, null-expect misuse, or causal dead-end).
- **Issue codes.** `causal_orphan`, `mod_effect_not_reflected`, `expect_incomplete`, `expect_null_invalid`, `sequential_paradox`, `mod_unsuppression_invalid`, `redundant`, `expect_leak`.

Starting from the baseline draft, each pass followed the same three-step loop: (1) run the validator over every example; (2) conduct the manual review described above — confirm flagged issues, dismiss false positives, sample items the validator marked `CLEAN` to verify the negative label, and, where the defect is local, annotate the correct expected output directly; (3) regenerate only the still-flagged examples after adding a single targeted rule to the generation prompt aimed at the most common code in that pass. Five expect-side regeneration passes (v1→v5) and one Stage 2 input-regeneration pass (v6) were applied. Table 5 shows the verdict distribution across passes; Table 6 shows the per-issue trajectory.

Takeaways. Four findings from the ablation shape the production curation pipeline: (i) *prompt iteration has diminishing returns* — the first pass recovered 95 examples, the fifth recovered 7 (Table 5); (ii) *input-level defects are unreachable from the expectations* — five expect-regeneration passes could not move `sequential_paradox` below 14, while a single Stage 2 input regeneration cut it to 4 (Table 6); (iii) *the judge validator carries non-trivial noise* — a re-validation of

Pass	Added rule	CLEAN	MILD	PARADOX	INCOMPL.
v1	— (baseline)	216 (43%)	212	14	56
v2	expect isolation	311	166	11	10
v3	cross-event consistency	340	133	13	12
v4	entity naming + thresholds	367	110	14	7
v5	defer-until rule	374	104	14	6
v6	Stage 2 input regen	**382 (77%)**	107	**4**	**5**

Table 5: Verdict distribution across six refinement passes on a 498-example draft. Each pass added one targeted rule and regenerated only the flagged examples. The final pass regenerates the *inputs* (Stage 2) rather than the expectations, since input-level paradoxes cannot be repaired by expect-side rewrites.

Issue code	v1	v6	Reduction
`expect_null_invalid`	68	1	−99%
`mod_effect_not_reflected`	168	12	−93%
`expect_incomplete`	129	15	−88%
`mod_unsuppression_invalid`	25	4	−84%
`sequential_paradox`	19	4	−79%
`causal_orphan`	216	133	−38%
`expect_leak`	7	6	−14%
`redundant`	48	51	+6%

Table 6: Per-issue trajectory from v1 to v6. Mod-effect, expectation, and paradox codes all reduce by 79% or more; `causal_orphan` reduces more slowly because it requires consistent identifier naming across events rather than a single targeted rule; `redundant` is unaffected by expect-side iteration.

v6 with the same model and temperature flipped roughly 2% of CLEAN/MILD labels, so final numbers should be read as a ±2% range rather than point estimates; (iv) *manual review and annotation are load-bearing, not optional* — because of the noise in (iii) and the limits in (ii), a fully automated validator→regenerate loop diverges; the human pass at each iteration is what kept regenerations targeted, dismissed false positives, caught defects the judge missed (by re-sampling `CLEAN` items), and resolved the cases that no rewrite could produce. A residual 1.8% of examples (9 of 498) resisted both expect and input regeneration and were hand-patched directly or retained as known-imperfect cases.

G Appendix: Experimental Setup

We compare POLO against two configurations of OpenClaw (OpenClaw, 2026), a commercial conversational-agent framework that supports both standalone agents and multi-agent orchestration. All three configurations consume the same benchmark examples, share the same external mock tools, and are scored by the same judge; they differ in how the workflow is partitioned across agents and, by extension, in the form of specification each configuration consumes.

We first describe the *agent graph* derived from each benchmark example—a workflow-decomposition abstraction shared by the multi-agent configurations—before defining the three configurations themselves.

G.1 The Agent Graph Abstraction

A benchmark example is paradigm-agnostic: it consists of the original workflow instructions (a sequence of business-level steps), a set of modifications, and a timed sequence of external events. This specification does not commit to any particular agent decomposition.

For configurations that distribute the workflow across multiple agents, we derive an *agent graph* from the example via a focused LLM call. The agent graph is a directed graph $G = (V, E)$ in which each node carries a *structured natural-language specification*, not just a label. Formally:

- Each node $v \in V$ is an agent specification with four text fields: a **role** (one-line statement of responsibility), a **behavior** (a paragraph describing how this agent responds to incoming messages, including conditional branches and downstream actions it takes), an **interface** (the message shapes it accepts), and an initial **state** description (may be empty).
- Each edge $(u, v) \in E$ denotes a permitted message channel from u to v, annotated with the relationship type (request, notification, query, etc.). Edges constrain communication but carry no behavior of their own; behavior lives on nodes.
- Distinguished nodes are tagged with one or more *event sources*: external systems whose events enter the workflow through that node.

Together the node text fields plus the edge set make the agent graph a self-contained specification: an agent runtime can instantiate it by reading each node's behavior text as that agent's system prompt

and honoring the edge set as the permitted outgoing message destinations.

The derivation call returns one node per externally-named system referenced by the workflow (entry-point and service nodes) plus business-logic nodes that route between them.

Agent graph (excerpt

```
nodes:
  - id: entry-point-a
    role: Ingest submissions from external-system-a.
    behavior: >
      When a new submission event arrives, capture the
      payload, assign a case identifier, and forward the
      case to the orchestrator. Do not perform policy
      decisions; that is the orchestrator's responsibility.

    event_sources: [external-system-a: new-item-submitted]
    neighbors: [orchestrator]
  - id: lookup-service
    role: Read-only directory of organisational facts.
    behavior: >
      When asked for an entity's attributes or
      relationships, return the matching record from the
      directory. Never write.
    tools: [lookup_data]
  - id: orchestrator
    role: Apply the workflow policy and route to services.
    behavior: >
      For each incoming case, query the lookup-service to
      resolve the relevant entities, evaluate the policy
      conditions in the workflow steps, dispatch to notify-
      service and response-service. On a terminal decision,
       write status back via response-service.
    neighbors: [lookup-service, notify-service,
                response-service, entry-point-a]
```

Figure 26: Agent graph derived from a benchmark example. Each node carries a role line and a behavior paragraph (truncated here for space); together these populate the agent's system prompt at instantiation time. Edges constrain communication; behavior lives on nodes.

Quality control of the derived graph. Each derived agent graph is inspected for two properties before use: (i) *interface connectivity*—every event-source node must be reachable to the orchestrator over the edges, so an arriving event can trigger the cascade; (ii) *role consistency*—any node serving lookup queries is typed as a read service, any node persisting effects is typed as a write service. Misclassifications are corrected by hand. This step is the same regardless of which downstream configuration the graph is fed into.

G.1.1 The agent Event Association

The benchmark example includes a timed sequence of external events. The events are not associated with any particular node in the agent graph; they are paradigm-agnostic. For the graph-based configurations, we associate each event with the node that listens to its source system. This is a straightforward matching of the event's `triggered_by` field to the `event_sources` tags on the nodes. The single-agent configuration receives the same events but without any node association, since it has no internal notion of nodes.

The association is purely for routing: the event is delivered as a message to the associated node at the scheduled timestamp. The event content is identical regardless of configuration, and all configurations receive the same events in the same order; the only difference is which agent receives each event as a message.

Let the event sequence be $E = [e_1, e_2, \ldots]$. For the graph-based configurations, we define a mapping M from event to nodes: for each node $e \in E$, we add entries to M mapping those sources to v. Then each event e_i is delivered as a message to node $M[e_i]$ at time t_i. For the single-agent configuration, all events are delivered to the same agent without any source-based routing.

The mapper M is an LLM call that takes the agent graph and the event sequence as input and returns the mapping as output. This is a one-time call per benchmark example, performed before any of the three configurations are run; the resulting mapping is shared by both graph-based configurations. The single-agent configuration does not use the mapping at all, since it has no internal notion of nodes.

Event-to-node mapper: $M(\mathcal{E}) \to \mu$

```
Inputs:
  G = (V, E)                          the agent graph
  S                                   the workflow steps
  Ev = {e_1, e_2, ..., e_n}           the events of one
      example

Output:
  mu : Ev -> V,
       a mapping that assigns each e_i in Ev to a node v_i
       in V.

Definition:
  Given the agent graph G, the workflow steps S, and the
      event set Ev, M returns the assignment mu such that,
       for every e_i in Ev, mu(e_i) is the unique node v_i
       in V whose role and behavior, under the policy
      expressed by S, make it the correct entry point for
      e_i.
```

Figure 27: Specification of the event-to-node mapper $M(\mathcal{E}) \to \mu$. Given an agent graph $G = (V, E)$, the workflow steps S, and the full event set $\mathcal{E} = \{e_1, \ldots, e_n\}$ of a benchmark example, M produces a single assignment $\mu : \mathcal{E} \to V$ used by the graph-based configurations to dispatch every e_i to its entry-point node $\mu(e_i) = v_i$. From v_i the cascade continues across edges in E per S.

G.2 Configurations

POLO (agent-graph runtime). POLO instantiates one agent per node of the derived graph on top of its own runtime: a message bus that routes between neighbors, a per-agent Planner–Executor loop, and an asynchronous mailbox model. Each agent's system prompt is composed from its node's role and behavior text; the edge set determines which destinations its outgoing messages may target.

OpenClaw multi-agent (agent graph on a hosted orchestrator). The same agent graph is materialized as a set of OpenClaw agents—one agent per node, each with a system prompt composed from its node's role and behavior text—wired together by OpenClaw's hosted agent-to-agent (A2A) protocol. The edge set E becomes the set of permitted A2A dispatches: when an upstream agent decides a neighbor should act, it dispatches a message to that neighbor's session by name, and the neighbor's response is returned as a tool result. This configuration honors the same decomposition as POLO; the difference is the runtime (a hosted orchestrator rather than the POLO bus and Planner–Executor loop).

OpenClaw single-agent (workflow on one agent). The entire workflow is handled by a single OpenClaw agent. The agent receives the *original workflow instructions* (the example's business-level steps, as written before any decomposition) as its system prompt at session start—it is *not* given the agent graph, neither the edge set nor the per-node behavior text. Workflow modifications arrive as additional messages at their scheduled timestamps; events follow in chronological order. The agent maintains all workflow state in a single conversation thread and performs internally any routing that the graph-based configurations would express as edges. This represents a strong, practical baseline: a competent practitioner operating directly off the specification, with no runtime scaffolding.

G.3 On Input Forms and Fairness

The three configurations consume the same benchmark example but in different forms: the two graph-based configurations consume the derived agent graph (the role and behavior text per node, plus the edge set), and the single-agent configuration consumes the original workflow instructions (one sequence of steps, no decomposition). This is a deliberate design choice, not an asymmetry left unaddressed.

Each configuration is supplied with the input form that matches its architecture. A single agent designed to handle a workflow holistically has no internal notion of nodes or edges—forcing it to consume an agent graph would require it to reflatten the decomposition back into prose at inference time, an unnatural transform that would penalize the configuration for an artefact of the input encoding rather than its own behaviour. Conversely, the graph-based configurations cannot consume the original workflow instructions without first deciding which steps belong to which agent; that decision *is* the agent graph. The agent graph and the workflow instructions are two equally faithful projections of the same underlying example, each chosen to match the consumer that processes it.

Because the input forms differ, we are careful that all three configurations share everything downstream of the system prompt: the same modifications, the same event timeline, the same external mock tools, the same evidence-collection hooks, and the same judge. Any difference in measured pass rate therefore reflects differences in *processing*, not in what each configuration was told about the example.

G.4 Shared Infrastructure

Beyond the system prompt, all three configurations share identical evaluation infrastructure.

External service mock. A lightweight `mock-external` plugin is registered with the OpenClaw daemon (for the OpenClaw configurations) and with the POLO runtime (for ours) before each evaluation run. The plugin exposes 12 tool endpoints covering the external services used in the benchmark: `slack_send_message`, `email_send`, `jira_create_issue`, and nine additional service calls. When an agent invokes a tool, the plugin forwards the call to a local MockServer (FastAPI) which returns a scripted response—a delivery acknowledgment, a message ID, or a ticket number—and logs the call and its arguments for later judge evidence.

Event chaining. Events with a `triggered_by` field are delivered as follow-up messages immediately after their parent event is processed. No tool-call pattern matching is required; ordering is deterministic and identical across all three configu-

rations.

Evidence collection. The OpenClaw SDK exposes hooks into agent tool invocations and internal state after each turn; POLO exposes the equivalent via its bus trace and per-agent state dumps. We attach a collector to both, and the resulting evidence bundle—tool-call log plus terminal agent state—is passed to the shared judge in the same format regardless of configuration. This ensures that any difference in pass rates reflects genuine behavioral differences, not asymmetric evidence visibility.

Parallel evaluation. Runs are distributed across a Docker worker pool. Each container hosts a fully isolated OpenClaw daemon (or POLO runtime) plus its own MockServer instance; workers for each configuration occupy separate port ranges so all three pools can run in parallel without interference. The evaluation harness dispatches examples work-queue style and merges the per-configuration output into a single JSONL file.

Compute environment. All evaluations were run on a single shared compute node running Rocky Linux 9.7, with two Intel Xeon E5-2630 v4 processors (20 physical cores, 40 hardware threads total, 2.20 GHz base / 3.10 GHz boost) and 251 GiB of RAM. Each baseline run used a pool of up to 32 isolated OpenClaw worker containers (Docker 29.4.0) running concurrently on the host. LLM inference was performed remotely via the Azure OpenAI API.

Configuration	Agents per TC	Input form	System prompt	Routing
OpenClaw single-agent	1	Original workflow instructions	Step sequence + tool catalog	Agent decides internally
OpenClaw multi-agent	$\|V\|$ (one per node)	Derived agent graph	Per-node role+behavior text	OpenClaw A2A protocol over E
POLO	$\|V\|$ (one per node)	Derived agent graph	Per-node role+behavior text	POLO message bus over E

Table 7: Structural comparison of the three evaluated configurations. All three are given the same benchmark example but in the form that matches each configuration's architecture; downstream infrastructure (modifications, events, tools, judge) is identical across the row.

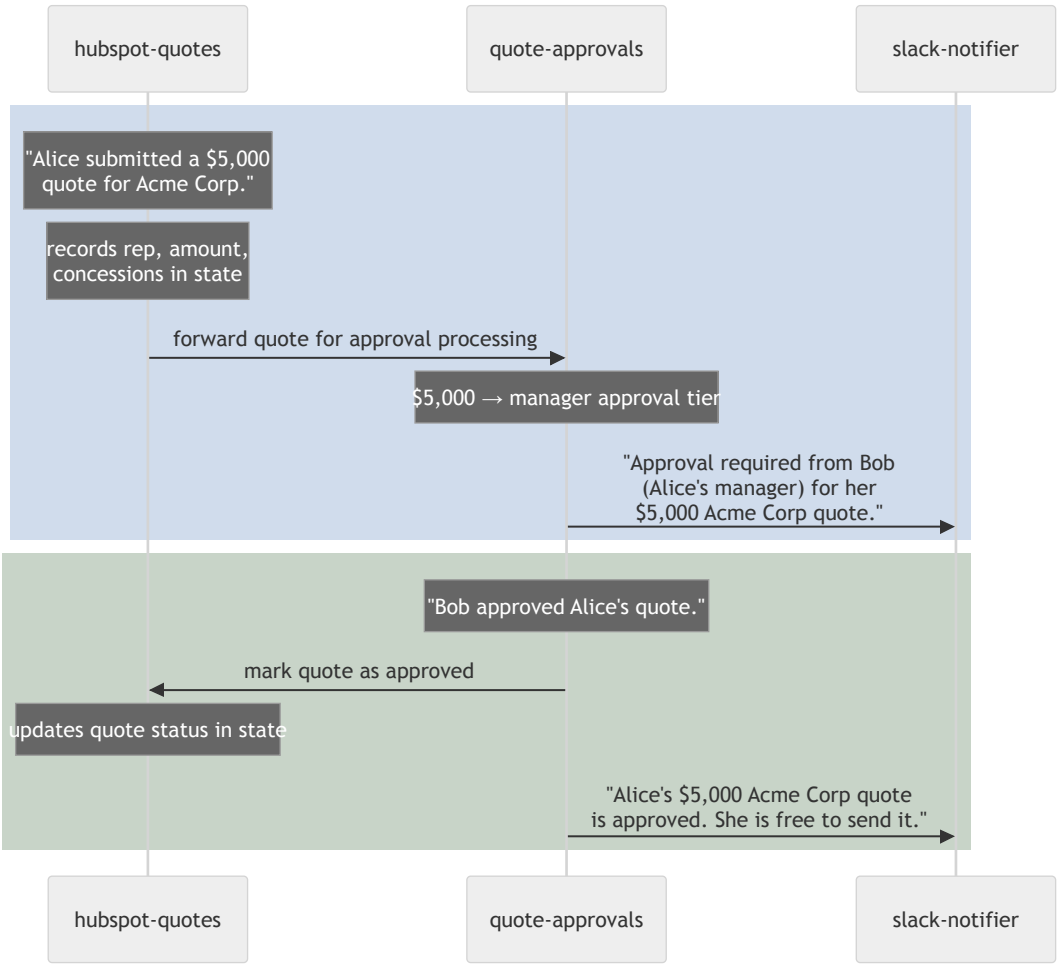

Figure 4: Message flow for the three-object quote-approval program. On submission (top), `hubspot-quotes` forwards the quote to `quote-approvals`, which identifies the manager-approval tier and posts an approval request to `slack-notifier`. On approval (bottom), `quote-approvals` updates `hubspot-quotes` with the approved status and notifies `slack-notifier` to inform the rep. No routing logic appears in any object definition; it emerges from the peer declarations alone.