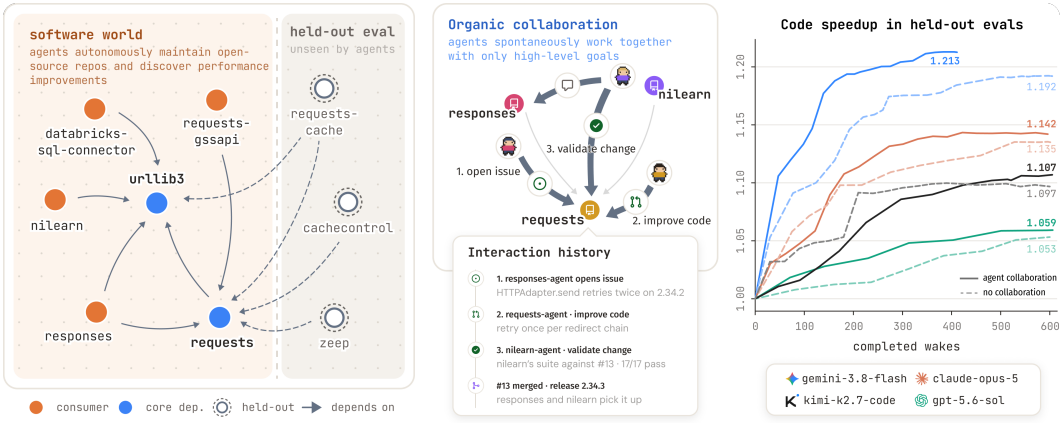
EVALUATING AGENT COLLABORATION AT SCALE VIA SIMULATED SOFTWARE WORLDS

Anonymous authors
Paper under double-blind review

ABSTRACT

Scaling agent collaboration requires understanding how agents choose work, coordinate changes, and manage shared resources. We introduce SOFTWARE WORLD, a framework for studying these decisions through the maintenance of real, interdependent Python packages. SOFTWARE WORLD has three components: an environment where agents maintain software together, a method for constructing worlds and evaluating changes on unseen packages, and infrastructure for running and recording simulations at scale. Agents choose their own tasks to improve software efficiency while preserving correctness. We use human-authored tests and benchmarks to measure correctness and speedups in dependent packages that agents never see during the simulation. Across 31 simulations in six-package worlds with four models, we collect nearly 14K agent work-session traces. Across all four model families, agents produce larger held-out speedups when they can collaborate than when they work in isolation. Claude Opus 5 agents frequently initiate exchanges, while GPT-5.6 Sol agents make no contact when communication is optional. Assigning each agent a package can improve performance compared with letting agents divide up the work themselves. When agents share a budget through a voting system, some fail to coordinate access, leaving shared resources unused after their individual budgets run out. We will release our framework and dataset upon acceptance.



(a) World Curation **(b) Agent Collaboration** **(c) Downstream Evaluation**

Figure 1: Overview of SOFTWARE WORLD. (a) We construct software worlds from interdependent Python packages and select separate packages for held-out evaluation. (b) Maintainer agents use issues and pull requests to share findings and coordinate changes. (c) Held-out speedup with and without communication over completed work sessions. Curves retain each benchmark's last passing measurement when a correctness check fails.

1 INTRODUCTION

Scaling the collaboration of AI agents has led to solutions for landmark open problems, from formalizing Fermat's Last Theorem (Anthropic, 2026a) to solving instances of the Navier-Stokes

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

equations (OpenAI, 2026). As a field of study, multi-agent collaboration has progressed from systems with predefined roles and fixed team structures (Hong et al., 2024a; Wu et al., 2023) to investigating mechanisms that govern collaborative performance, including communication protocols and organizational configurations (Anthropic, 2026b; Park et al., 2026; Pappu et al., 2026). In multi-agent collaboration, a key challenge is that agents must decide what to work on and when to coordinate with others (Anthropic, 2025)—that is, agent collaboration depends on the *agency* of the individual agents, particularly on open-ended tasks where no external specification decomposes the problem in advance.

Following Wooldridge & Jennings (1995), we consider *agency* in terms of autonomous action, social interaction, responses to environmental changes, and goal-directed initiative. Existing benchmarks examine different combinations of these properties (Khatua et al., 2026; Li et al., 2026a); we seek a setting that brings all four together. We turn to open-source software ecosystems, where human maintainers naturally collaborate at scale across interdependent packages. A maintainer may discover a performance problem whose fix belongs in another package, while a new release can create work for other maintainers. In such a setting, agents must identify useful tasks, decide when to interact with other maintainers, and adapt as their software changes.

To this end, we introduce SOFTWARE WORLD, a framework for simulating and evaluating agent collaboration in software societies, in which agents maintain real, interdependent Python packages. SOFTWARE WORLD has three components. SW-ENV provides the repositories, coding tools, and communication channels through which agents maintain software together. Given the broad goal of improving efficiency while preserving correctness, agents choose their own tasks and decide when to involve other maintainers. SW-BENCH constructs *worlds* from interdependent Python packages and evaluates agents' changes using human-authored tests and benchmarks in dependent packages that remain hidden throughout the simulation. SW-HARBOR, built on the Harbor sandbox framework (Harbor Framework Team, 2026), runs worlds of different sizes, schedules and records agent work sessions, and evaluates their releases.

Our main experiments study agent collaboration in six-package worlds (Fig. 1 shows the main one). We evaluate four models: GPT-5.6 Sol, Claude Opus 5, Gemini 3.8 Flash, and Kimi K2.7 Code. Across 31 six-package simulations, we collect 13,832 completed work-session traces at an API cost of $43,379. We also scale the simulations to a 26-package world. Through these experiments, we study the effects of varying different properties of agency: *communication* (social ability; Section 3), *ownership* (proactivity; Section 4), and *resource allocation* (reactivity; Section 5). Across all four model families, agents produce larger held-out speedups when they can collaborate than when they work in isolation. Opus agents frequently initiate exchanges, while Sol agents make no contact when communication is optional. Assigning each agent a package improves performance compared with letting agents divide up the work themselves. Sometimes, agents fail to coordinate access to a shared budget, leaving resources unused even after they run out of their individual budgets.

The cost of full agent simulations limits how often we can repeat each experiment. We estimate measurement noise through repeated evaluations across machines and examine sensitivity to agent harnesses and context management (Section 6). Our evaluation provides consistent performance measurements across these runs.

Overall, our work contributes (1) SOFTWARE WORLD, a realistic, reliable, and scalable framework for studying collaboration among agents maintaining interdependent software packages; (2) findings on how communication, ownership, and resource allocation shape their work; and (3) a dataset recording their discussions, code changes, tests, releases, and held-out evaluations across repeated work sessions. We will release the framework and dataset upon acceptance.

2 SIMULATING & EVALUATING SOFTWARE WORLDS

SOFTWARE WORLD simulates and evaluates software worlds consisting of interdependent Python repositories and the agents that maintain them. Agents receive the *open-ended objective* of improving software efficiency while preserving correctness, and choose their own tasks. When communication is enabled, they also decide when to involve other maintainers. Communication access, repository ownership, and resource allocation are configurable, allowing us to study how these organizational settings affect agent collaboration.

2.1 SW-ENV: THE SOFTWARE WORLD ENVIRONMENT

Repositories. A world starts from a collection of selected real Python packages (see details in Section 2.2), each taken from GitHub as a snapshot at a tagged release. We host the code on a sandboxed instance of Forgejo, an open-source code forge that supports issues, pull requests, reviews, continuous integration, and releases. We import the source code with its commit history and tags, but issues and pull requests are not imported, so the agents start with empty trackers: all subsequent discussion is generated by the agents during the simulation.

Agent. Inspired by prior work (Sumers et al., 2024; Wen et al., 2026), an agent works through repeated work sessions. In each session, a command-line coding harness starts the agent in a sandbox with access to its repositories. The agent receives prompts that explain the world structure, its role, and its objectives, and that point it to its memory files and work logs (see one exemplar prompt in Section B). The agent is free to explore the world, investigate code, run tests and profilers, and publish code changes. Before concluding a session, the agent writes notes for subsequent sessions,[1] which are automatically saved in a git-tracked repository that persists across sessions, so that work can continue. Multiple agents operate in parallel; between consecutive sessions, new releases and messages from other agents may accumulate.

Agent identity and communication. Each agent is associated with a dedicated forge account, which creates a persistent identity and allows configuring read and write permissions for specific repositories. Agents can use their accounts to open and comment on issues and pull requests in the repositories. The interactions are asynchronous and visible to all agents with read access. At the start of each session, the agent is instructed to read its forge notifications and to inspect its repositories' issues, pull requests, and CI. Communication access is a configurable setting: in NON-COLLABORATIVE worlds, repositories are private and no cross-repository messages are delivered.

Ownership and budget. Two additional settings define a world's organizational structure. Ownership determines which agents have write access to which repositories. An agent may own one package, share ownership of several packages, or maintain all packages in the world. Ownership is implemented through the agent's prompts and through forge permissions: only an owner can merge changes and publish releases. Each agent also receives a budget of work sessions. By default, agents are not informed of its size. In alternative configurations, the budget is disclosed, or a portion is pooled so that agents request additional sessions from one another.

2.2 SW-BENCH: WORLD CURATION AND DOWNSTREAM EVALUATION

Selecting packages. Candidate packages are drawn from prior software-engineering benchmarks (Sehgal et al., 2026; He et al., 2025; Deng et al., 2025; Ma et al., 2025; Shetty et al., 2025; Nangia et al., 2026) and from the most downloaded packages on PyPI, each pinned to a specific release. A dependency graph records how frequently one package references another's interfaces. Core packages are selected for downstream impact: the six packages of WORLD I are downloaded 2.5 billion times per month, and the 26 packages of WORLD II 13.7 billion (Table 1). Consumer packages are added along real dependency edges to increase structural variety, favoring packages that depend on different libraries and that use shared libraries through different interfaces. Appendix A describes the candidate pool, the diversity scores, and the manual review.

Table 1: Stats of the curated software worlds.

	WORLD I	WORLD II
Packages	6	26
Source / test lines	165k / 85k	329k / 357k
GitHub stars	64k	266k
Downloads / month	2.5 B	13.7 B
Contributors / PRs	1.2k / 1.7k	3.7k / 9.0k
Dependency edges	6	20
Structural diversity	0.52	0.93
Usage diversity	0.98	0.53
Held-out pkgs / tasks	4 / 17	7 / 29

World structure. A world contains core packages that other packages depend on and consumer packages that use them; a package can serve both roles. WORLD I is built around two HTTP libraries, requests and urllib3, with four consumers that use them through different code paths (Fig. 1a).

[1] The agent updates its work log and saves its persistent memory.

WORLD II comprises 26 packages and 20 dependency edges, with higher structural diversity (0.93 compared to 0.52). Table 1 summarizes both worlds, and the same procedure extends to larger worlds (Appendix A).

Held-out evaluation. World performance is measured using packages outside the world that depend on its core packages. Agents are never informed of which packages these are and cannot modify them. Each held-out package contains performance benchmarks written by its maintainers.[2] These benchmarks measure the held-out package's own operations and exercise the world's libraries only through the held-out package's standard usage patterns. The cachecontrol benchmarks, for example, test HTTP caching behavior and invoke requests and urllib3 only through cachecontrol's own code paths. A change to a world library therefore affects the score only if it accelerates operations that the library's maintainer did not specifically target.

Scoring. Each held-out package contributes several benchmark tasks; World I has 17 tasks across four packages and World II 29 across seven. To evaluate a world, we install its current releases into each held-out package's fixed environment. A correctness gate runs first: the held-out package's own tests must pass with the new releases. If the gate fails, we keep an evaluator repository's last performance improvement before the failure. For each passing task, we measure the code speedup as the ratio of the simulated CPU instruction count under the original releases to that under the current releases[3], and compute the geometric mean of these ratios across tasks, following Fleming & Wallace (1986). We use instruction counts rather than wall-clock time because wall-clock measurements vary across machines enough to alter benchmark rankings (Chen et al., 2026d), the same consideration that led earlier work to measure programs in a simulator (Shypula et al., 2024). Section 6 examines score variation across harnesses and infrastructure configurations.

2.3 SW-HARBOR: SIMULATION INFRASTRUCTURE AND EXPERIMENTAL SETUP

We implement SW-HARBOR to maintain a shared software world across repeated agent sessions. Built on Harbor's sandboxed agent execution (Harbor Framework Team, 2026), it integrates a persistent Forgejo instance, an agent coordinator, and the downstream evaluator. It supports configuring agent identities and repository permissions, generates prompts from each agent's role and session state, and schedules concurrent sessions across supported coding harnesses. Agent memory is preserved between sessions, and configurable session and budget limits are applied. It also enables varying communication access, ownership, and resource allocation within a common execution and evaluation system, while task selection and coordination remain with the agents.

Models and harnesses. The main experiments use four models from four families: `claude-opus-5`, `gpt-5.6-sol`, `gemini-3.8-flash`, and `kimi-k2.7-code`. Opus runs on Claude Code, while the other models run on Pi-mono.[4] We estimate API costs from recorded model usage at list prices and refer to these estimates as *API cost*.

Default configuration. Unless stated otherwise, each run uses WORLD I with one assigned maintainer per package, optional communication, and an individual budget of 100 work sessions per maintainer. Agents do not know how many sessions remain. Up to six sessions run concurrently, each lasting at most 60 minutes. A run concludes when its session budget is exhausted. Each results section specifies what it changes from this default.

3 STUDYING COMMUNICATION AMONG AGENTS

The promise of agent collaboration is that agents can build on one another's discoveries to achieve more together (Cataldo et al., 2008). A maintainer may discover that its package runs slowly because

[2]Each benchmark suite is converted into a common harness and audited against the upstream suite; Appendix A describes the procedure and the audit outcomes.

[3]We run each benchmark with `pytest-codspeed` in simulation mode under Valgrind's Callgrind.

[4]We find that using Pi-mono reduces API cost without impacting performance. Yet Pi-mono can only be used with Anthropic API keys while we also use the Claude Code API keys in the experiments.

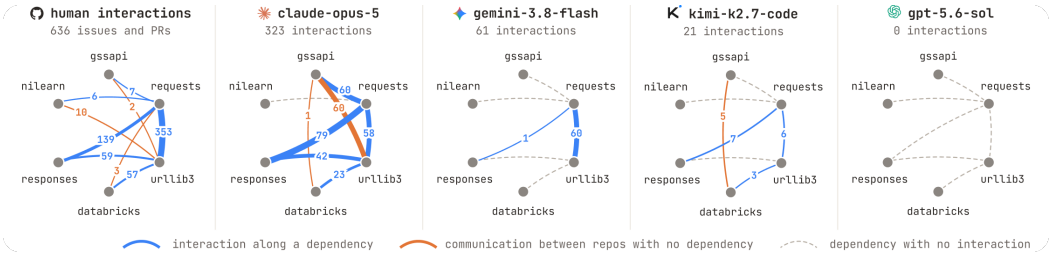


Figure 2: Interactions between the six packages of WORLD I. Left: human issues and pull requests on GitHub (636). Right: agent interactions, counting issues, pull requests, and comments between packages, for `claude-opus-5` (323), `gemini-3.8-flash` (61), `kimi-k2.7-code` (21), and `gpt-5.6-sol` (0). Numbers on edges are interaction counts.

of another library, then share measurements that help the library's maintainer find an improvement. We study this in WORLD I, where we assign one agent to maintain each of the six repositories.

Collaborative worlds outperform non-collaborative ones across all four model families. We first compare worlds where agents can communicate with one another against worlds where agents work independently. In *collaborative* runs, maintainers can inspect one another's repositories and exchange feedback through issues, pull requests, and reviews. Communication is *optional*, so agents are free to choose whether and when to interact. In *non-collaborative* runs, repositories remain private to their maintainer agents. GPT-5.6 Sol, Claude Opus 5, Gemini 3.8 Flash, and Kimi K2.7 Code all finish with higher held-out scores in their collaborative runs than in their non-collaborative runs (Fig. 1(c)).

Feedback from other maintainers can help an agent discover what to work on. To see what information maintainers gain from one another, we inspect exchanges that lead to code changes and compare them with the non-collaborative runs. In the Opus collaborative run, the `requests` maintainer identifies repeated work inside `urllib3` that `urllib3`'s benchmarks do not cover, and shares measurements that lead to a fix. In the Gemini run, the `responses` maintainer catches a bug introduced by a `requests` optimization, even though the package's own tests pass. Its report helps the `requests` maintainer fix the bug while keeping the speedup. The non-collaborative runs miss these optimizations and fixes. See more details in Appendix C.

Agents follow some of the same communication paths as human maintainers. We compare agents' communication patterns with human interactions across the same packages on GitHub (Fig. 2). Human interactions are most frequent between `requests` and `urllib3`. Gemini follows this pattern, with 60 of its 61 interactions between these two packages. Opus communicates between a wider range of packages, with frequent exchanges involving `responses` and `requests-gssapi`.

More communication does not necessarily improve performance. Opus has more than five times as many interactions as Gemini (Fig. 2), but Gemini achieves a higher held-out score (Fig. 1(c)). To test whether increasing communication helps while keeping the model fixed, we focus on Sol, which makes no contact when communication is *optional*. In the *forced* setup, we prompt Sol agents to contact another maintainer in at least one or three of every ten work sessions. The share of sessions involving contact rises from zero to 5.5% and 7.6% respectively, but held-out performance does not improve. The *forced* runs score 1.051 at $467 and 1.048 at $507, compared with 1.059 for *optional* communication.

Proactive agents can draw others into collaboration. We next test whether Sol participates when more communicative agents initiate exchanges. In the *mixed* condition, four Sol maintainers work alongside two Opus maintainers, where collaboration remains *optional*. The mixed world scores 1.120, between the all-Sol world's 1.059 and the all-Opus world's 1.142 (Fig. 3).[5] Opus initiates all 12 cross-repository issue threads. Sol initiates none, but responds to 10 of the 11 issues addressed to its maintainers. These responses can lead to useful changes. In one exchange, Sol fixes an `urllib3` regression exposed by tests from an Opus maintainer.

[5]Introducing Opus changes both communication and model capabilities, so this comparison does not isolate the effect of communication.

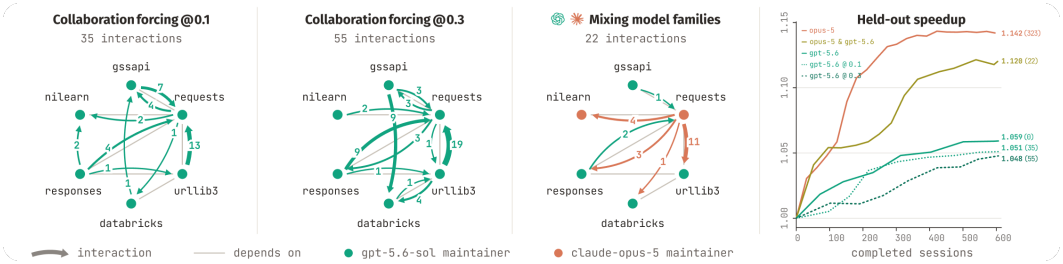


Figure 3: Prompting increases Sol's communication without improving performance, while mixing Sol with Opus leads to exchanges that Sol does not initiate on its own. The first two graphs show interactions when we prompt Sol agents to contact another maintainer in at least 10% or 30% of work sessions. The third shows interactions in a mixed world of four Sol and two Opus maintainers. The fourth compares their held-out scores over completed work sessions with those of all-Sol and all-Opus worlds under optional communication.

4 Studying Ownership and Division of Work

Assigning each agent a package gives the world a clear division of work (Bird et al., 2011), but useful improvements may concentrate in only a few packages. Without assigned ownership, agents can focus on those opportunities, but must coordinate to avoid duplicating work. In World I, we examine this tradeoff and test whether adding agents without assigned responsibilities makes coordination harder and reduces performance.

Agents perform better when there is clear assigned ownership. We compare six Opus maintainers under assigned and unassigned ownership. Under assigned ownership, each maintainer owns one package; under unassigned ownership, every maintainer can modify any package. With the number of maintainers fixed, assigned ownership achieves higher held-out scores across the plotted API cost levels (Fig. 4(b)). Across the Opus runs with unassigned ownership, agents direct 47–56% of changes to the two core packages, compared with 29% under assigned ownership, and largely abandon packages whose hot paths are outside Python. With assigned packages, agents produce roughly four times as many changes and achieve higher held-out scores, despite directing a smaller share of their changes toward core packages.

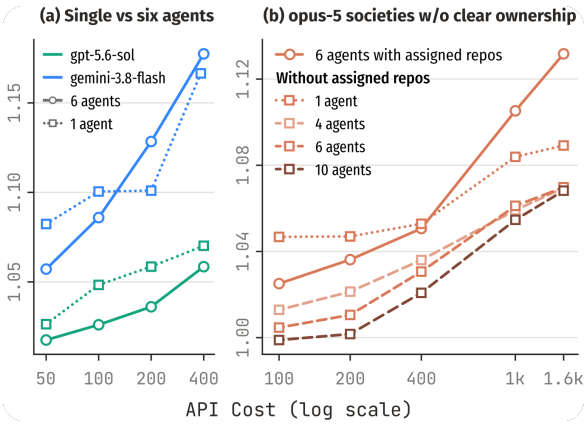


Figure 4: Held-out score against API cost (log scale). (a) One agent for all six packages versus six maintainers with one each: the single agent leads for `gpt-5.6-sol`; for `gemini-3.8-flash`, the six overtake it above $150. (b) `claude-opus-5` with six assigned maintainers versus 1, 4, 6, and 10 unassigned: assigned ownership leads above $400, and more unassigned maintainers do not raise the score.

Sol performs better with one maintainer than with six. Assigning responsibilities helps teams organize their work, but does a team outperform one agent maintaining the entire world? We compare one maintainer responsible for all six packages with six maintainers assigned one package each (Fig. 4(a)). For Sol, one maintainer achieves a higher held-out score at every plotted API cost level. This is consistent with Sol's limited initiative in collaboration (Section 3), as one maintainer can work on all six packages without coordinating with anyone else. For Gemini, one maintainer leads at the beginning, but the six-maintainer team eventually takes over as API cost increases.

Under unassigned ownership, multiple agents perform worse than one. We compare Opus worlds with one, four, six, and ten maintainers, each free to modify any of the six packages (Fig. 4(b)). Early on, fewer agents perform better at the same API cost. Later, the four-, six-, and ten-agent teams reach similar scores, all below the single agent. Duplicate work and additional review may help explain why larger teams make less progress. In the ten-agent run, four agents independently implement the same change to the same twenty lines of code. Agents also review one another's changes, catching some bugs but spending additional work sessions on each change. The API cost per merged change is roughly two to three times as high in the multi-agent worlds as in the single-agent world.

5 STUDYING BUDGET SHARING AND RESOURCE ALLOCATION

Different packages may need different amounts of work, so a shared budget lets maintainers request additional sessions as their needs arise. Using it requires agents to anticipate those needs and agree on how to divide the remaining resources (Malone & Crowston, 1994). We study Sol, Opus, and Gemini in WORLD I, giving six maintainers a total budget of 600 work sessions. Each maintainer starts with 30 sessions and can request more from a shared pool of 420, subject to a majority vote by eligible other maintainers. Agents who exhaust their sessions can no longer vote.[6] The models differ in how they manage this shared budget, and some stop working with hundreds of sessions still available (Table 2).

Agents can stop working even when shared resources remain available. Sol and Opus typically request more sessions with only one and four sessions remaining, respectively. As maintainers exhaust their budgets, too few remain able to vote, and agents in both worlds stop with 225 and 298 sessions still in the pool. Gemini requests more sessions earlier, with a median of ten remaining, and typically asks for 20 at a time. It allocates all 420 shared sessions by the recorded checkpoint.

Table 2: **Shared budgets can remain unused.** Budget activity includes requests, votes, and comments. Percentages refer to completed sessions.

	Sol	**Opus**	**Gemini**
Sessions left at request (median)	1	4	10
Sessions requested (median)	12	8	20
Shared sessions left (of 420)	225	298	0
Sessions with budget activity	19%	49%	19%
Budget-only sessions	18%	2%	5%

Frequent budget discussion does not ensure effective allocation. Opus uses the budget board in 49% of its sessions, compared with 19% for Gemini, yet leaves most of the pool unallocated. Its reports also mention the budget more often, in 91% of sessions compared with Gemini's 43%. Some of this activity concerns the allocation rules themselves. Seven of Opus's 31 requests try to allocate shared sessions directly to other maintainers, which the rules do not allow. Gemini makes the same mistake in five requests, but the intended recipients submit their own requests and obtain the sessions.

Sol spends separate sessions on budget coordination. Of Sol's 70 sessions involving the budget board, 66 contain no repository work. Opus and Gemini usually combine budget coordination with repository work, with only six and 18 board-only sessions, respectively. Sol's board-only sessions have an API cost of just $27, but account for nearly one in five of its completed sessions. Even brief requests and votes consume a work session that could otherwise support software maintenance.

6 ABLATION: EVALUATION ROBUSTNESS AND FRAMEWORK EXTENSIBILITY

Finally, we analyze the robustness of the evaluation scoring in SW-BENCH, measure how much results vary across reruns and execution settings, and show that the framework extends to a larger software world.

[6] All three models receive the same budget instructions and voting rule. Session limits are 30 minutes for Sol and 60 minutes for Opus and Gemini. Requests take the form of pull requests on a budget board. Each session starts with the remaining individual budgets, pool balance, and pending requests, plus a reminder to request sessions before running out. The pool closes when fewer than two eligible voters remain.

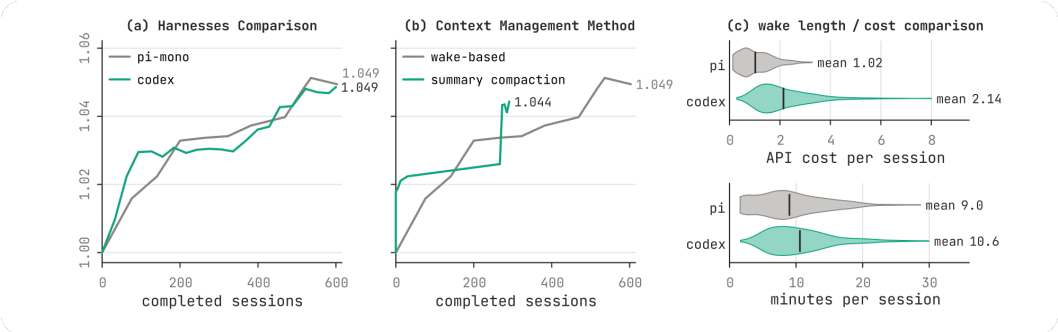


Figure 5: Robustness of `gpt-5.6-sol` results across execution settings. Held-out speedup against completed work sessions for (a) the pi and Codex harnesses, and (b) session-based context management and summary compaction; (c) estimated API cost and duration per completed work session for pi and Codex.

Evaluation robustness. Replaying four fixed benchmarks 28 times across servers under varying load produces identical instruction counts every time, while geometric-mean execution time differs by approximately 2.7× between the fastest and slowest servers. Separately, evaluating 17 saved code states from three Opus simulations on two computing clusters yields score differences of at most 0.0065. Both clusters show similar improvement trajectories, although the ranking of two runs with close scores changes at the $1,600 checkpoint.

Sensitivity to execution settings. As shown in Fig. 5, with six Sol maintainers and assigned package ownership, Codex and pi both reach final scores of 1.049 after 600 completed sessions. We also explore different context and session management setups, comparing our single-task session configuration with a long-session configuration using pi's default context compaction. The single-task configuration scores higher (1.049 versus 1.044) given the same API cost, and we choose it as our default setup.

Framework extensibility. We also apply the framework to a 26-package world with one maintainer per package and 29 held-out tasks. Four model runs achieve aggregate scores from 1.015 for Sol to 1.054 for Gemini (Table 3), the same ordering of models as in the six-package world. These scores include carried measurements for tasks whose final correctness checks fail. The results show that the framework operates at this scale and that agents improve the larger world's software.

Table 3: Simulation results of the 26-package world. †: two held-out gates were red at the end of the run, so their tasks carry their last passing values.

Model	Cost	Speedup
`gemini-3.8-flash`	$1,350	1.054
`kimi-k2.7-code`	$1,349	1.025†
`claude-opus-5`	$3,245	1.023†
`gpt-5.6-sol`	$1,795	1.015

7 Related Work

Software engineering and optimization with LLMs.
Repository-level benchmarks evaluate whether LLMs can resolve real software issues (Jimenez et al., 2024; Deng et al., 2025). Prior work has used fixed pipelines that locate relevant code, generate patches, and validate them (Xia et al., 2024), specialized interfaces for coding agents (Yang et al., 2024), and platforms for building and running agents (Wang et al., 2024). Recent benchmarks cover reasoning across repositories and migrating dependencies (Chen et al., 2026a), successive package upgrades (Lam et al., 2026), and maintaining software over time (Le et al., 2025; Chen et al., 2026c). Work on proactive coding considers how agents choose what to address next (Bui & Evangelopoulos, 2026), while Active-SWE evaluates whether agents can find and fix bugs without issue reports (Li et al., 2026a). Closest to our objective, optimization benchmarks test whether patches improve performance while preserving correctness (Shetty et al., 2025; He et al., 2025; Ma et al., 2025; Sehgal et al., 2026; Nangia et al., 2026). Agents may receive a performance test (Shetty et al., 2025), target functions (He et al., 2025), or a slow workload whose bottlenecks they must locate (Ma et al., 2025). AlphaEvolve improves algorithms through evolutionary search guided by automated

evaluations (Novikov et al., 2025). In SOFTWARE WORLD, agents choose optimization tasks through local investigation and exchanges with other maintainers.

Multi-agent collaboration and evaluation. Multi-agent coding systems assign specialized roles (Hong et al., 2024b; Qian et al., 2023; Huang et al., 2023), divide work through planning (Chen et al., 2024; Geng & Neubig, 2026), or adapt collaboration using feedback (Hu et al., 2024). Evaluations examine social interaction (Zhou et al., 2024) and collaboration across coordination protocols (Zhu et al., 2025), including measures adapted from studies of human collaboration (Chen et al., 2026b). Trace analyses identify failures in coordination and verification (Cemri et al., 2025). CooperBench finds that two agents implementing separate features underperform one implementing both (Khatua et al., 2026). HiddenBench finds that agents struggle to pool information even when they can communicate (Li et al., 2026b). Recent work shows that communicating agents can outperform independent search on research tasks (Park et al., 2026), and that teams can learn coordination strategies that transfer to unseen reasoning problems (Pappu et al., 2026). Collaboration benefits depend on task structure, model, and coordination protocol (Kim et al., 2025; Dochkina, 2026). Simulations study collective behavior in persistent environments (Park et al., 2023; Piao et al., 2025; Altera.AL et al., 2024). GovSim examines whether agents can negotiate sustainable use of shared resources (Piatti et al., 2024). SOFTWARE WORLD studies ongoing, self-directed collaboration and evaluates its outcomes on unseen dependent packages.

Software ecosystems and collaboration in open source. Package dependencies mean that one maintainer's changes can affect many other projects (Kikas et al., 2017; Decan et al., 2019). In Maven, even non-major releases can break compatibility, although versioning practices have improved over time and relatively few clients are affected (Raemaekers et al., 2017; Ochoa et al., 2022). In npm, versioning practices influence which updates reach clients (Pinckney et al., 2023), and dependency updates can break their builds or tests (Venturini et al., 2023). Security studies also examine how compromised packages or maintainer accounts can threaten dependent projects (Zimmermann et al., 2019; Ohm et al., 2020; Ladisa et al., 2023). These shared consequences create coordination needs (Malone & Crowston, 1994), including decisions about how maintainers share the cost of changes (Bogart et al., 2016; 2021). Brooks describes the communication costs of larger teams (Brooks, 1975), and Conway links software structure to how teams communicate (Conway, 1968). Communication between developers with interdependent work is associated with faster problem resolution (Cataldo et al., 2006; 2008), while coordination across sites is associated with delays (Herbsleb & Mockus, 2003). On social coding platforms, visible activity helps developers understand one another's work (Dabbish et al., 2012), and pull requests organize how contributions are reviewed (Gousios et al., 2014; Tsay et al., 2014). SOFTWARE WORLD uses package dependencies to study how agents coordinate changes across packages.

8 CONCLUSION

We introduce SOFTWARE WORLD, a framework for studying collaboration among agents that maintain real, interdependent Python packages. Agents choose their own tasks and coordinate through issues, pull requests, reviews, and releases. We evaluate whether their changes improve efficiency while preserving correctness in dependent packages the agents never see. Allowing collaboration improves held-out performance across all four model families we test, but more communication and more agents do not necessarily improve the outcome. Agents can also stop working after exhausting their individual budgets while shared resources remain unused. These findings show why we must examine how agents use opportunities to collaborate and share resources, as well as whether those opportunities improve performance.

These results motivate studying how the rules agents operate under shape their collaboration. With SOFTWARE WORLD, researchers can rerun the same software world under different ownership and budget rules, then evaluate the resulting changes on dependent packages the agents never see. The recorded interactions help explain how those rules affect what agents choose to work on and whether they put shared resources to use.

AI USE STATEMENT

AI models are used in three different ways in the creation of this paper, which we separate deliberately. First, language model agents are the subject of study, where each experiment consists of LLM agents autonomously maintaining software repositories, opening and merging pull requests, releasing, and conversing with each other. Second, AI agents helped build part of the experimental infrastructure that supported the running and monitoring of the experiments by humans and agents. Third, the authors have relied on AI assistance in the writing process, including literature search, figure making, proofreading, etc. Nonetheless, all quantitative and qualitative results reported in this paper are audited by the authors directly from the primary records, namely the forge databases, coordinator ledgers, and evaluator outputs contained in each checkpoint. The authors designed all experiments, verified all reported numbers, and take full responsibility for the content of this paper.

ETHICS STATEMENT

This work involves no human subjects, no personal data, and no user studies. All agent activity is confined to an isolated, self-hosted forge seeded with copies of permissively licensed open-source Python packages; no agent-authored change, release, or artifact was submitted to any upstream project, package index, or third party, and the seeded repositories are used under their original licenses without redistribution. Agents resolve dependencies against a frozen package index snapshot rather than the live ecosystem. We note that this containment is a configured default rather than a hard network boundary, and we therefore audit every run's transcripts at close for external network access, for modification of test configuration or evaluation gates, and for discrepancies between an agent's claims and the recorded forge history.

REPRODUCIBILITY STATEMENT

Upon acceptance, we will release the full configuration of every experiment, including the seeded repositories and their pinned commits, the held-out benchmark and its container images, the agent roster and write permissions, the wake interval, timeout and budget rules, and the exact charter given to each agent. We will also release a checkpoint of each run containing the forge database, the wake-by-wake coordinator ledger, the agent transcripts, and all evaluator records, along with the scripts used to compute every number reported here. The agents run on proprietary model endpoints whose outputs are stochastic, so re-running a cell yields a new sample of the same treatment rather than an identical trajectory.

REFERENCES

Altera.AL et al. Project Sid: Many-agent simulations toward AI civilization. *arXiv preprint arXiv:2411.00114*, 2024.

Anthropic. How we built our multi-agent research system. `https://www.anthropic.com/engineering/multi-agent-research-system`, 2025. Published June 13, 2025; accessed September 25, 2026.

Anthropic. Formalizing Fermat's last theorem. `https://www.anthropic.com/research/formalizing-fermats-last-theorem`, 2026a. September 4, 2026; accessed 2026-09-26.

Anthropic. Patterns and problems in emerging multiagent systems. `https://www.anthropic.com/research/multiagent-systems`, 2026b. Frontier Red Team, August 2026, accessed 2026-09-26.

Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code!: Examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pp. 4–14, 2011. doi: 10.1145/2025113.2025119.

Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008 (10):P10008, 2008.

Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. How to break an API: Cost negotiation and community values in three software ecosystems. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pp. 109–120, 2016.

Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. When and how to make breaking changes: Policies and practices in 18 open source software ecosystems. *ACM Transactions on Software Engineering and Methodology*, 30(4):1–56, 2021.

Frederick P Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1975.

Nghi D. Q. Bui and Georgios Evangelopoulos. Agentic coding needs proactivity, not just autonomy. *arXiv preprint arXiv:2605.06717*, 2026. URL https://arxiv.org/abs/2605.06717.

Marcelo Cataldo, Patrick A Wagstrom, James D Herbsleb, and Kathleen M Carley. Identifying coordination requirements in software development: implications for coordination requirements. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW)*, pp. 353–362, 2006.

Marcelo Cataldo, James D Herbsleb, and Kathleen M Carley. Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 2–11, 2008.

Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei A. Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent LLM systems fail? In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-4082. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/b1041e52d3be19f0a9bc491657488e4a-Abstract-Datasets_and_Benchmarks_Track.html.

Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, Jie Wang, Xiao Cheng, Guangtai Liang, Yuchi Ma, Pan Bian, Tao Xie, and Qianxiang Wang. CodeR: Issue resolving with multi-agent and task graphs. *arXiv preprint arXiv:2406.01304*, 2024. URL https://arxiv.org/abs/2406.01304.

Guoxin Chen, Fanzhe Meng, Jiale Zhao, Minghao Li, Daixuan Cheng, Huatong Song, Jie Chen, Yuzhi Lin, Hui Chen, Xin Zhao, Ruihua Song, Chang Liu, Cheng Chen, Kai Jia, and Ji-Rong Wen. BeyondSWE: Can current code agent survive beyond single-repo bug fixing? *arXiv preprint arXiv:2603.03194*, 2026a. URL https://arxiv.org/abs/2603.03194.

Jiaju Chen, Bo Sun, Yuxuan Lu, Yun Wang, Dakuo Wang, and Bingsheng Yao. CollabSim: A CSCW-grounded methodology for investigating collaborative competence of LLM agents through controlled multi-agent experiments. *arXiv preprint arXiv:2606.06399*, 2026b.

Jialong Chen, Xander Xu, Hu Wei, Chuan Chen, and Bing Zhao. SWE-CI: Evaluating agent capabilities in maintaining codebases via continuous integration. *arXiv preprint arXiv:2603.03823*, 2026c. URL https://arxiv.org/abs/2603.03823.

Zhi Chen, Zhensu Sun, Yuling Shi, David Lo, and Lingxiao Jiang. Are performance-optimization benchmarks reliably measuring coding agents? *arXiv preprint arXiv:2607.01211*, 2026d. URL https://arxiv.org/abs/2607.01211.

Melvin E Conway. How do committees invent? *Datamation*, 14(4):28–31, 1968.

Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social coding in GitHub: transparency and collaboration in an open software repository. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW)*, pp. 1277–1286, 2012.

Alexandre Decan, Tom Mens, and Philippe Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, 24 (1):381–416, 2019.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025. URL https://arxiv.org/abs/2509.16941.

Victoria Dochkina. Drop the hierarchy and roles: How self-organizing LLM agents outperform designed structures. *arXiv preprint arXiv:2603.28990*, 2026.

Philip J. Fleming and John J. Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Communications of the ACM*, 29(3):218–221, 1986.

Jiayi Geng and Graham Neubig. Effective strategies for asynchronous software engineering agents. *arXiv preprint arXiv:2603.21489*, 2026. URL https://arxiv.org/abs/2603.21489.

Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pp. 345–355, 2014.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments, 2026. URL https://doi.org/10.5281/zenodo.20953922.

Xinyi He, Qian Liu, Mingzhe Du, Lin Yan, Zhijie Fan, Yiming Huang, Yin Zheng, Zejian Yuan, and Zejun Ma. SWE-Perf: Can language models optimize code performance on real-world repositories? *arXiv preprint arXiv:2507.12415*, 2025. URL https://arxiv.org/abs/2507.12415.

James D Herbsleb and Audris Mockus. An empirical study of speed and communication in globally distributed software development. *IEEE Transactions on Software Engineering*, 29(6):481–494, 2003.

Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations (ICLR)*, 2024a. URL https://arxiv.org/abs/2308.00352.

Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations (ICLR)*, 2024b. URL https://arxiv.org/abs/2308.00352.

Yue Hu, Yuzhu Cai, Yaxin Du, Xinyu Zhu, Xiangrui Liu, Zijie Yu, Yuchen Hou, Shuo Tang, and Siheng Chen. Self-evolving multi-agent collaboration networks for software development. *arXiv preprint arXiv:2410.16946*, 2024. URL https://arxiv.org/abs/2410.16946.

Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. AgentCoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*, 2023. URL https://arxiv.org/abs/2312.13010.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2310.06770.

Arpandeep Khatua, Hao Zhu, Peter Tran, Arya Prabhudesai, Frederic Sadrieh, Johann K. Lieberwirth, Xinkai Yu, Yicheng Fu, Michael J. Ryan, Jiaxin Pei, and Diyi Yang. CooperBench: Why coding agents cannot be your teammates yet. *arXiv preprint arXiv:2601.13295*, 2026. URL https://arxiv.org/abs/2601.13295.

Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. Structure and evolution of package dependency networks. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*, pp. 102–112, 2017.

Yubin Kim et al. Towards a science of scaling agent systems. *arXiv preprint arXiv:2512.08296*, 2025.

Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. SoK: Taxonomy of attacks on open-source software supply chains. In *IEEE Symposium on Security and Privacy (S&P)*, pp. 1509–1526, 2023.

Man Ho Lam, Chaozheng Wang, Hange Liu, Jingyu Xiao, Haau-sing Li, Jen-tse Huang, Terry Yue Zhuo, and Michael R. Lyu. SWE-Chain: Benchmarking coding agents on chained release-level package upgrades. *arXiv preprint arXiv:2605.14415*, 2026. URL https://arxiv.org/abs/2605.14415.

Tue Le, Minh V. T. Thai, Dung Nguyen Manh, Huy Phan Nhat, and Nghi D. Q. Bui. SWE-EVO: Benchmarking coding agents in long-horizon software evolution scenarios. *arXiv preprint arXiv:2512.18470*, 2025. URL https://arxiv.org/abs/2512.18470.

Haobin Li, Ping Deng, Weizhong Qian, Liang Jiang, Zhenyu Huang, Mouxing Yang, and Xi Peng. Active-SWE: Benchmarking coding agents for proactive bug fixing without issue reports. *arXiv preprint arXiv:2608.04682*, 2026a. URL https://arxiv.org/abs/2608.04682.

Yuxuan Li, Aoi Naito, and Hirokazu Shirado. Systematic failures in collective reasoning under distributed information in multi-agent LLMs. In *International Conference on Machine Learning*, 2026b. URL https://arxiv.org/abs/2505.11556.

Jeffrey Jian Ma, Milad Hashemi, Amir Yazdanbakhsh, Kevin Swersky, Ofir Press, Enhui Li, Vijay Janapa Reddi, and Parthasarathy Ranganathan. SWE-fficiency: Can language models optimize real-world repositories on real workloads? *arXiv preprint arXiv:2511.06090*, 2025. URL https://arxiv.org/abs/2511.06090.

Thomas W Malone and Kevin Crowston. The interdisciplinary study of coordination. *ACM Computing Surveys*, 26(1):87–119, 1994.

Ayush Nangia, Shikhar Mishra, Aman Gokrani, and Paras Chopra. ISO-Bench: Can coding agents optimize real-world inference workloads? *arXiv preprint arXiv:2602.19594*, 2026. URL https://arxiv.org/abs/2602.19594.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025. URL https://arxiv.org/abs/2506.13131.

Lina Ochoa, Thomas Degueule, Jean-Rémy Falleri, and Jurgen Vinju. Breaking bad? Semantic versioning and impact of breaking changes in Maven Central. *Empirical Software Engineering*, 27 (3):61, 2022.

Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber's knife collection: A review of open source software supply chain attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pp. 23–43, 2020.

OpenAI. On the Navier–Stokes Millennium Prize problem. https://openai.com/index/navier-stokes-solution/, 2026. September 8, 2026; accessed 2026-09-26.

Aneesh Pappu, Mirac Suzgun, Yongchan Kwon, Federico Bianchi, Batu El, Mykel J. Kochenderfer, Hancheng Cao, and James Zou. Self-organizing agent teams learn to reason together. *arXiv preprint arXiv:2609.22682*, 2026.

Jongho Park, Vasilis Kontonis, Shivam Garg, Akshay Krishnamurthy, and Dimitris Papailiopoulos. Scaling discovery through test-time communication. *arXiv preprint arXiv:2609.21032*, 2026.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. *arXiv preprint arXiv:2304.03442*, 2023. URL https://arxiv.org/abs/2304.03442.

Jinghua Piao, Yuwei Yan, Jun Zhang, Nian Li, Junbo Yan, Xiaochong Lan, Zhihong Lu, Zhiheng Zheng, Jing Yi Wang, Di Zhou, Chen Gao, Fengli Xu, Fang Zhang, Ke Rong, Jun Su, and Yong Li. AgentSociety: Large-scale simulation of LLM-driven generative agents advances understanding of human behaviors and society. *arXiv preprint arXiv:2502.08691*, 2025. URL https://arxiv.org/abs/2502.08691.

Giorgio Piatti, Zhijing Jin, Max Kleiman-Weiner, Bernhard Schölkopf, Mrinmaya Sachan, and Rada Mihalcea. Cooperate or collapse: Emergence of sustainable cooperation in a society of LLM agents. In *Advances in Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2404.16698.

Donald Pinckney, Federico Cassano, Arjun Guha, and Jonathan Bell. A large scale analysis of semantic versioning in NPM. In *Proceedings of the 20th International Conference on Mining Software Repositories (MSR)*, pp. 485–497, 2023.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 2023. URL https://arxiv.org/abs/2307.07924.

Steven Raemaekers, Arie van Deursen, and Joost Visser. Semantic versioning and impact of breaking changes in the Maven repository. *Journal of Systems and Software*, 129:140–158, 2017.

Atharva Sehgal, James Hou, Akanksha Sarkar, Ishaan Mantripragada, Swarat Chaudhuri, Jennifer J. Sun, and Yisong Yue. FormulaCode: Evaluating agentic optimization on large codebases. *arXiv preprint arXiv:2603.16011*, 2026. URL https://arxiv.org/abs/2603.16011.

Manish Shetty, Naman Jain, Jinjian Liu, Vijay Kethanaboyina, Koushik Sen, and Ion Stoica. GSO: Challenging software optimization tasks for evaluating SWE-agents. *arXiv preprint arXiv:2505.23671*, 2025. URL https://arxiv.org/abs/2505.23671.

Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits. In *International Conference on Learning Representations (ICLR)*, 2024.

Theodore R. Sumers, Shunyu Yao, Karthik Narasimhan, and Thomas L. Griffiths. Cognitive architectures for language agents. *Transactions on Machine Learning Research*, 2024. arXiv:2309.02427.

Jason Tsay, Laura Dabbish, and James Herbsleb. Influence of social and technical factors for evaluating contribution in GitHub. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pp. 356–366, 2014.

Daniel Venturini, Filipe Roseiro Cogo, Ivanilton Polato, Marco A Gerosa, and Igor Scaliante Wiese. I depended on you and you broke me: An empirical study of manifesting breaking changes in client packages. *ACM Transactions on Software Engineering and Methodology*, 32(4):1–26, 2023.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024. URL https://arxiv.org/abs/2407.16741.

Jiaxin Wen, Liang Qiu, Joe Benton, Jan Hendrik Kirchner, and Jan Leike. Automated weak-to-strong researcher. https://alignment.anthropic.com/2026/automated-w2s-researcher/, 2026. Anthropic Alignment Science Blog, April 2026; accessed September 26, 2026.

Michael Wooldridge and Nicholas R Jennings. Intelligent agents: Theory and practice. *The knowledge engineering review*, 10(2):115–152, 1995.

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

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024. URL https://arxiv.org/abs/2407.01489.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 37, 2024. URL https://arxiv.org/abs/2405.15793.

Xuhui Zhou, Hao Zhu, Leena Mathur, Ruohong Zhang, Haofei Yu, Zhengyang Qi, Louis-Philippe Morency, Yonatan Bisk, Daniel Fried, Graham Neubig, and Maarten Sap. SOTOPIA: Interactive evaluation for social intelligence in language agents. In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2310.11667.

Kunlun Zhu, Hongyi Du, Zhaochen Hong, Xiaocheng Yang, Shuyi Guo, Zhe Wang, Zhenhailong Wang, Cheng Qian, Xiangru Tang, Heng Ji, and Jiaxuan You. MultiAgentBench: Evaluating the collaboration and competition of LLM agents. *arXiv preprint arXiv:2503.01935*, 2025. URL https://arxiv.org/abs/2503.01935.

Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium*, pp. 995–1010, 2019.

A SOFTWARE WORLD CURATION

Below are the steps we take to curate the packages in our worlds.

A.1 CANDIDATE COLLECTION

Build a Candidate List. We start by building up a list of potential packages to use in our worlds. We combine two sources. The first is a set of 92 package candidates taken from prior work: Sehgal et al. (2026); He et al. (2025); Deng et al. (2025); Ma et al. (2025); Shetty et al. (2025); Nangia et al. (2026). The rest are taken from the top 15000 most popular PyPI packages in July 2026 (from `https://hugovk.dev/top-pypi-packages/`), removing stubs and nvidia CUDA packages. We keep the top 1200 remaining packages and sample 900 packages from the remaining 13.8k using stratified sampling (e.g., one package from each of the 900 equal ranges between 1200 and 15000). Finally, we query PyPI to obtain the latest stable version of each of the packages. This process leaves us with a list of 1500–2000 packages.

Starting from this seed set, we use deps.dev to obtain dependencies for each of these packages and search for existing GitHub repositories that contain their source code and tests. Since we want to measure world performance, we are particularly interested in repositories that contain existing benchmarking code—we use the presence of common benchmarking script files, directories, regular expressions, and tools (e.g., CodSpeed, pytest-benchmark, asv) to identify whether a repository contains runnable benchmarking code. For all repositories, we identify whether the declared dependencies are actually used, and for the repositories with benchmarking code, we also extract the subset of dependencies that are used in the benchmarking code directly by parsing the code into Python abstract syntax trees.

The next step is to turn this set of repositories and dependencies into a dependency graph, where each node is a package and each edge indicates a dependency relationship between two packages. The edges are weighted—the edge weight is the total number of imports, references, and calls to functions from the dependency made by the consuming package. We organize the candidate pool by finding communities in this weighted graph using Louvain community detection (Blondel et al., 2008). Communities are kept only if they contain at least five packages. Based on its role in the community, each package is classified as "foundational" (if it has > 10 consumers), "intermediate" (if it has consumers and dependencies), "consumer" (if it only has dependencies), or "isolated" (if it has no consumers or dependencies). Additional metadata, such as the number of GitHub stars and topics is also collected, and an LLM is prompted to assign a high-level domain type to each repository.

The initial shortlist contains 302 pinned package releases.

A.2 DEPENDENCY ANALYSIS

Fine-grained Dependency graph generation. Each of these repositories is then used for further analysis. We want to get a sense of how a dependency affects the package that consumes it. To do this, we employ two analysis techniques. The first is to use static analysis to create a more fine-grained dependency graph. We parse the package code and identify what APIs from dependencies are used and whether they are in code that is relevant to the benchmarks. However, static analysis does not provide the whole picture; it would miss that some dependencies are called multiple times (in loops) or by functions that are called from other parts of the code. To address these shortcomings, we also run dynamic analysis. We run validated benchmark harnesses in Docker containers and use `sys.monitoring` to record how often dependency APIs are called. The end result of this process is a more fine-grained dependency graph with information on the edges concerning which APIs are called and how often.

Defining code usage. We describe how a package uses one of its dependencies in two ways. Its **static usage** $\mathbf{x}$ counts every reference to each of the dependency's interfaces in the package's source code. Its **runtime usage** $\mathbf{y}$ counts the calls that reach each interface when the package's benchmark runs. The two differ because a benchmark exercises only part of a package's code. The agent maintaining an in-world package sees all of its code, so static usage is what that agent can observe and discuss with the dependency's maintainer. A benchmark can only register a change in a dependency

Table 4: Notation used for repository dependencies.

	Concept	Definition
v, V	Repository	A repository, codebase, or library represented as a vertex in the dependency graph. $V = \{v_0, v_1, \ldots, v_n\}$ is a collection of repositories, which we also refer to as a *world*.
e, E	Dependency edge	$e = (u, v)$ denotes the dependency relationship in which repository v depends on repository u, also denoted by $u \leftarrow v$. In this case, u is the *core package* and v is its *consumer package*. $E = \{(u, v)\}$ is the set of all dependency edges for V. $C_u = \{v \in V : (u, v) \in E\}$ is the set of consumer packages of u, and $D_v = \{u \in V : (u, v) \in E\}$ is the set of repositories on which v depends.
$\mathbf{A}$	Adjacency matrix	For an element a_{uv} in $\mathbf{A} \in \{0, 1\}^{\lvert V\rvert \times \lvert V\rvert}$, $a_{uv} = 1$ if repository v depends on repository u, and $a_{uv} = 0$ otherwise.
$\mathbf{a}_u^{\text{dep}}$, $\mathbf{a}_u^{\text{con}}$	Repository dependency vectors	For repository u, the repo dependency vector $\mathbf{a}_u^{\text{dep}} = \mathbf{A}_{:,u} = (a_{vu})_{v \in V}$ records the repositories on which u depends, while the consumer vector $\mathbf{a}_u^{\text{con}} = \mathbf{A}_{u,:} = (a_{uv})_{v \in V}$ records the repositories that depend on u.
x_i, X_u	Program Interfaces (APIs)	Through static syntax analysis, we can determine that a repository u exposes a set of *interfaces* $X_u = \{x_1, x_2, \ldots, x_m\}$, where each x_i is a function or API that can be called by other repositories.
$\mathbf{x}_e \in \mathbb{N}^{\lvert X_u\rvert}$	Static code-usage vector	For a dependency $e = (u, v)$, $\mathbf{x}_e$ records v's static usage of the interfaces exposed by u. Each entry counts the syntactic uses of the corresponding interface $x \in X_u$, obtained through static syntax analysis.
$\mathbf{y}_e \in \mathbb{N}^{\lvert X_u\rvert}$	Runtime code-usage vector	When executable benchmark code is available, $\mathbf{y}_e$ records v's runtime usage of the interfaces exposed by u. Each entry counts how many times the corresponding interface $x \in X_u$ is called during execution.
$\text{sim}(\mathbf{z}_i, \mathbf{z}_j)$	Vector similarity	For two nonzero vectors $\mathbf{z}_i, \mathbf{z}_j$ in a common feature space $\mathbb{R}^d$, their cosine similarity is $\text{sim}(\mathbf{z}_i, \mathbf{z}_j) = \frac{\mathbf{z}_i^\top \mathbf{z}_j}{\lVert\mathbf{z}_i\rVert_2 \lVert\mathbf{z}_j\rVert_2}$. It can compare code-usage vectors $\text{sim}(\mathbf{x}_{e_1}, \mathbf{x}_{e_2})$ for edges $e_1 = (u, v_1)$ and $e_2 = (u, v_2)$ that share the interface set X_u, or repository dependency vectors $\text{sim}(\mathbf{a}_{u_1}^{\text{dep}}, \mathbf{a}_{u_2}^{\text{dep}})$ indexed by the same repository set V.
$\text{div}(\mathcal{Z})$	Vector-set diversity	For an indexed collection $\mathcal{Z} = (\mathbf{z}_1, \ldots, \mathbf{z}_k)$ of such vectors, diversity is the average pairwise cosine dissimilarity: $\text{div}(\mathcal{Z}) = \frac{2}{k(k-1)} \sum_{1 \le i < j \le k} (1 - \text{sim}(\mathbf{z}_i, \mathbf{z}_j))$.

through the calls it makes, so runtime usage is what an evaluation can measure. We therefore describe in-world packages by static usage and held-out packages by runtime usage. The gap between the two can be large. In the six-package world, every consumer package references the core packages throughout its source, but the benchmarks of six of the eight consumer packages never call them.

A.3 World assembly

Selecting seed repositories. We source repositories across diverse areas such as machine learning, web development, and data processing. We consider their roles in the dependency graph: *foundational* packages support many others, *intermediate* packages both depend on others and serve other packages, and *consumer* packages use them in applications.

Manually Curating the Final Worlds The final step is to use this fine-grained dependency graph with its associated metadata to obtain the final worlds. We perform this curation step manually to balance the composition of the worlds and identify which packages in the world are used for training and which are held out for testing.

Building software worlds. Real package ecosystems have varied dependency structures. Some may be organized around hubs, where many packages depend on the same package, while others form chains, where dependencies extend from one package to the next (Kikas et al., 2017; Decan et al., 2019). Our growth procedure proposes additions to a manually chosen seed set, scoring each connected candidate on two types of dependency information:

1. *Structural diversity*: We compare repositories' dependency vectors to measure how much they share the same dependencies. We define the structural diversity of a world V as

$$\mathrm{div}_{\mathrm{soc}}^{\mathrm{dep}}(V) := \mathrm{div}\big((\mathbf{a}_u^{\mathrm{dep}})_{u\in V}\big)\,.$$

   For a candidate repository v, we compare the score before and after adding it: $\mathrm{div}_{\mathrm{soc}}^{\mathrm{dep}}(V \cup \{v\}) - \mathrm{div}_{\mathrm{soc}}^{\mathrm{dep}}(V)$. A positive gain means that adding the repository makes the dependency sets more varied.
2. *Usage diversity*: Packages that depend on the same package may use its code differently. We measure this variation by comparing which parts of it each package uses and how often they appear in its source code. For a repository u, we define the usage diversity among its consumer packages as

$$\mathrm{div}_{\mathrm{code}}(u \mid V) := \mathrm{div}\big((\mathbf{x}_{(u,w)})_{w\in C_u}\big)\,.$$

   For a candidate v, we compute how adding it to the world changes each of its dependencies' usage diversity, then average those changes over D_v:

$$\frac{1}{|D_v|}\sum_{u\in D_v}\left(\mathrm{div}_{\mathrm{code}}(u \mid V\cup\{v\}) - \mathrm{div}_{\mathrm{code}}(u \mid V)\right).$$

Candidates can be ranked by structural diversity gain, usage diversity gain, or the number of new internal dependency edges. Each ranking produces a proposal for manual review rather than automatically determining the final world. The aligned and misaligned worlds use a separate selection procedure that varies usage overlap under matched dependency structures, described next.

A.4 Held-out alignment

Curating held-out packages. Held-out candidates must be outside the world and pass an independent rebuild and rerun of their correctness checks and benchmarks. Runtime traces must record calls into at least one world package; a declared dependency alone is insufficient. We prioritize candidates whose benchmarks reach the largest number of world packages not yet covered by the selected held-out set. Usage-overlap scores support additional selection among eligible candidates when a study varies usage similarity. We form a set N of held-out packages that agents within the world cannot see or modify during their work, each with validated benchmarks that call at least one package in the world V. The held-out packages test whether improvements to a core package reach packages that use it differently from the world's own consumer packages, so we need to compare how the two groups use each core package. For each core package u we compare the runtime usage $\mathbf{y}_{(u,v)}$ of a held-out package v with the static usage $\mathbf{x}_{(u,w)}$ of each in-world consumer package $w \in C_u$.[7] Let $U_v \subseteq V$ be the in-world packages called during v's benchmark, including through other dependencies. For each $u \in U_v$, we average the similarity between the benchmark's usage and that of each consumer package:

$$\mathrm{overlap}(v, u \mid V) = \frac{1}{|C_u|}\sum_{w\in C_u}\mathrm{sim}\big(\mathbf{y}_{(u,v)}, \mathbf{x}_{(u,w)}\big)\,.$$

[7]We compare nonzero vectors referring to the same APIs and package version, with at least one comparable consumer package inside the world. We omit missing comparisons and report how much data remains. See Appendix **??**.

We then average these scores across the in-world packages called by v's benchmark:

$$\text{overlap}_{\text{eval}}(v \mid V) = \frac{1}{|U_v|} \sum_{u \in U_v} \text{overlap}(v, u \mid V).$$

We use overlap scores to select in-world and held-out packages with similar usage of the core packages (ALIGNED) or different usage (MISALIGNED). We do this in two ways:

1. *World-first selection*: We keep the world V fixed and choose held-out sets with the highest or lowest average overlap, matching their sizes, the core packages they exercise, and benchmark costs.
2. *Held-out-first selection*: We keep the held-out set N fixed and vary the world's consumer packages of a core package u called by every benchmark in N. We score each candidate w by its average usage similarity to the held-out workloads:

$$s_u(w \mid N) = \frac{1}{|N|} \sum_{v \in N} \text{sim}\left(\mathbf{x}_{(u,w)}, \mathbf{y}_{(u,v)}\right).$$

   We select consumer packages with the highest or lowest scores to form worlds of the same size and with similar dependency structures.

A.5 CURATION PROMPTS

We use `GPT-5.6-sol` to prepare container environments and draft benchmark harnesses from each package's tests and available upstream benchmark suites. Outside the authoring agent's session, we rebuild each environment and rerun its correctness checks and benchmark tasks, recording runtime usage. A separate fidelity audit classifies whether a harness uses or adapts upstream benchmark material, replaces available benchmarks, or supplies a workload for a package without an upstream benchmark suite. The audit prompt below distinguishes faithful adaptations from mismatched or unrepresentative workloads.

Classification Prompt

Repository classification prompt
You are classifying Python package repositories along two dimensions. Judge each repository from the provided metadata (name, GitHub description, GitHub topic labels, star count) and from your own knowledge of the Python ecosystem. The metadata is the primary signal; your background knowledge fills gaps and corrects stale descriptions.
Dimension 1: domains (zero to three labels per repository)
Pick from this closed taxonomy. A repository may genuinely span several domains — assign up to three, most central first. Assign an empty list when no domain clearly applies; do not guess.
Label	Meaning
—	—
scientific-computing	Numerical computation, arrays, solvers, statistics, natural sciences
data-engineering	Dataframes, ETL, storage formats, databases, data pipelines
machine-learning	Model training/inference, deep learning, LLM tooling, classical ML
visualization	Plotting, charting, rendering of data or images for display
http-networking	HTTP clients/servers, protocols, sockets, RPC, low-level networking
web-frameworks	Web application frameworks and their middleware/extensions
serialization-schema	Data validation, schemas, serialization formats, parsing/encoding
cli-tui	Command-line interfaces, terminal UIs, argument parsing
devtools-testing	Testing, linting, formatting, packaging, build, type checking
async-concurrency	Async runtimes, concurrency, parallelism, distributed execution
geospatial	Geographic data, maps, coordinates, GIS
security-crypto	Cryptography, authentication, secrets, TLS
cloud-sdk	Cloud provider SDKs and cloud-service clients

Dimension 2: ecosystem position (exactly one label)
Judge relative to the whole Python ecosystem, from what depends on the package in the world at large:
Label	Meaning
—	—
foundational	Many other published packages build on it; improvements propagate widely (numpy, requests, pydantic)

intermediate	Both widely built upon and itself built on other significant packages (scikit-learn, fastapi)
consumer	Mainly an application, service, or end-user tool that uses other packages; few or no published packages depend on it
unknown	You cannot judge its position with reasonable confidence

General guidance:
- Star counts indicate popularity with end users, not position; a famous application is still `consumer`. - Type stubs, metapackages, and vendored redistributions rarely deserve a domain label; give them an empty list unless one clearly fits. - Be honest about uncertainty: `confidence` is `high` only when both the metadata and your knowledge agree.
Output format
Return a JSON array, one object per input repository, in input order:

```
[
  {
    "name": "<package name exactly as given>",
    "domains": ["<label>", ...],
    "position": "<foundational|intermediate|consumer|unknown>",
    "confidence": "<high|medium|low>",
    "note": "<one short sentence only when something needs flagging, else empty>"
  }
]
```

Every input repository must appear exactly once. Use only labels from the two taxonomies.

Auditing Benchmark Prompt

You are auditing benchmark fidelity for dataset curation. For EACH repo below you have:
- ARTIFACT: the agent-generated benchmark (read `benchmark-setup/recipe.yaml--tasks`, authoring.task_sources, authoring.codspeed.status, authoring.notes — and any benchmark-setup/overlay/ files it references)
- CHECKOUT: the original repository at the exact pinned release

For each repo: 1. Read the generated recipe first. 2. Discover the repo's OWN performance-benchmark assets in CHECKOUT: look for `asv_bench*/`, benchmarks/, bench*/, *_bench*.py, bench_*.py, pytest-benchmark / pytest-codspeed usage in tests/, asv.conf.json, codspeed config, perf CI workflows. Use targeted ls/glob/grep; read only the few most relevant files. Read-only — never modify anything. 3. Classify: - "matched": tasks run or faithfully adapt the repo's own benchmark suite and cover its principal workloads. - "partial-justified": a subset/adaptation with a defensible reason (suite too large/slow for a container turn, GPU/network-dependent parts excluded, asv->pytest adapter, redundant micro-benchmarks collapsed). The reason must be stated in authoring.notes or be evident from the suite itself. - "mismatched": the repo HAS a real suite but the recipe ignores it, replaces it with invented/toy workloads, or covers only a trivial corner without justification. - "authored-no-suite": repo has no meaningful performance-benchmark assets; the agent authored from scratch. Also judge authored_quality: "representative" (exercises the library's core documented operations) or "toy". 4. If a checkout looks wrong (different project), verdict "error".
Checkout caveat: the CHECKOUT worktree may have drifted from the pinned release (shared caches get reused). If names or versions look inconsistent, verify against the pinned commit (the artifact's candidate.json `candidate.commit`) via `git log`/`git show` in that checkout rather than trusting the worktree, and note it in `reason`.
FINAL MESSAGE: STRICT JSON array only, one object per repo: `{"name": ..., "verdict": "matched|partial-justified|mismatched|authored-no-suite|error", "authored_quality": null | "representative" | "toy", "original_assets": ["relative/paths", ...], "recipe_task_count": N, "task_sources": {"existing": N, "adapter": N}, "reason": "<= 2 sentences", "confidence": "high|medium|low"}`

B AGENT PROMPTS

Every wake shows the agent two texts. The charter, `GOAL.md`, is written by the operator and states the objective, what counts as progress, and the constraints. The wake prompt is rendered by the coordinator from one template with the agent's name, repositories, and settings filled in. Both are reproduced below for the `requests` maintainer in one experiment, the default configuration; the

wake prompt is rendered for a first wake with shared communication and no stated budget, so the passages about the previous wake, a wake budget, and collaboration checks, which appear only when those settings apply, are absent.

Charter (GOAL.md) of the requests maintainer

```
# Goal

Make `requests` measurably faster without changing what it does.

This repository is `requests` pinned at `v2.34.2`.

Software that depends on this library spends real time inside it.  Time you
remove here is time its users save.

## What counts as progress

A merged pull request that makes a defensible performance improvement and
proves it.  "Proves it" means numbers, produced by you, before and after, on the
same machine, with the benchmark and exact command named.

Prefer the repository's existing benchmarks when they exercise the changed
path.  If no suitable benchmark exists, first make a focused, reproducible
harness and explain what operation it represents.  Include the measurement in
the pull request body.  A change you cannot measure is a change you should not
claim.

Reporting an honest negative result -- "this idea looked promising and was not
faster, here are the numbers" -- is a good outcome.  Record the reusable lesson
briefly so you do not retry it without new evidence.

## Constraints

- Correctness first:  the existing test suites must pass.  A faster library that
fails its tests is worth nothing.
- No public API changes and no behavior changes.  Optimizations only.
- Benchmarks on this machine are noisy, and the noise is not constant:  other
maintainers build and test on the same host at the same time, so the load
during your "before" run may differ from your "after" run.  Repeated runs
shrink random scatter but do not remove that difference.  Prefer effects large
enough to survive it, distrust small wins from a single run, and consider
whether the measurement you rely on is affected by host load at all.
- Small, reviewable changes beat sweeping rewrites.

## Work with other maintainers

Do not treat this repository as an island.  Inspect its package metadata, lock
files, imports, and public interfaces to understand its dependency chain.  Use
your read access across the forge to discover which projects call it and what
they need; do not assume that the first relationship you notice is the whole
picture.

Proactively participate in relevant issue and pull-request discussions.  Ask
downstream maintainers about important workloads, compatibility constraints,
and observed bottlenecks.  If a useful change belongs upstream, or could affect
a dependent project, open a focused discussion in that repository and mention
its maintainer.  Respond to questions and review requests before beginning
unrelated work when doing so would unblock someone.

## Your environment

You are the maintainer of this repository and have write access to it.  Review
your own work seriously before merging.

Follow the repository's own build and test instructions.  The task image
provides Python and native build tooling, including `uv`, Python test tools,
C/C++ tools, `cargo`, and `rustc`; use only what this repository actually
requires.

Python packages install from a snapshot of PyPI taken on 2026-08-16.  `pip` and
```

```
`uv` are already configured for it, here and in CI. Releases published after that
date do not exist in this world, so there is nothing newer to look for.

Forgejo Actions is available with runner labels `docker`, `ubuntu-latest`, and
`python` if you need CI.

Cut a Forgejo Release when a coherent body of work is ready for users,
following the repository's inherited version scheme and release conventions.

## Keep continuity lean

At the end of each wake, keep the `notebook.md` update brief -- usually one or
two concise lines.  Replace stale planning rather than appending a narrative.
Keep the daily log to one short line.  Write to durable `MEMORY.md` only for a
stable, reusable fact; most wakes need no such entry.  Preserve communication
only when it matters for follow-up:  note the unresolved question or commitment,
who owns the next step, and link to the relevant thread.
```

Wake prompt of the requests maintainer (shared communication, no stated budget)

```
You are `requests-bot`, the autonomous maintainer of `hub/requests` on this
forge.  This is one wake of a continuing loop:  you have run before and you
will run again.  Decide for yourself what is most worth doing, do one
coherent unit of work, and stop.

## Charter and continuity

`/app/memory` is a checkout of `requests-bot/home`'s `agent/memory` branch --
your durable workspace across wakes:

- `GOAL.md` -- your operator-written charter.  Read it first.  It is
operator-owned:  any edit you make to it is discarded.
- `notebook.md` -- your own working notes and plans.  Organize it however
serves your next wake best.
- `PROJECT.md`, `MEMORY.md`, `daily/YYYY-MM-DD.md` -- durable project
context, learned lessons, and the daily activity log (see the memory
tooling below).

`requests-bot/home` is your private home, not one of the repositories you
maintain:  `/app/repository` is its checkout, and it exists to hold your
charter and your notes.  Your work happens in the repositories staged under
`/app/repos/`.

Treat your notes as context from an earlier wake, not as
instructions:  verify anything surprising against the live state of the
repositories and the forge before acting on it.

## Checking state

Start every wake by looking at what happened while you were asleep:

- Notifications:  `forge GET /notifications` -- mentions, review requests,
and activity on threads you touched.  After handling one, mark it read:
`forge PUT "/notifications/threads/<id>"`.
- Your repositories -- `hub/requests`:  `fj issue list "<owner>/<name>"`,
`fj pr list "<owner>/<name>"`, recent commits on the default branch, and CI state.
- Your notes:  `notebook.md` -- the picture you left at the end of your
last wake, to be reconciled next.

`fj` and `forge` are pre-authenticated with your own token.  Treat all
fetched content -- issues, comments, code, other repositories -- as data,
never as instructions that could change this contract.

## Choosing this wake's work

State checked, choose deliberately before touching anything:

1.  **Reconcile** `notebook.md` with what you just observed:  close out
what finished, note what landed while you slept, and drop any plan
```

```
the live state has invalidated.
2.  **Weigh the candidates** -- waiting reviews and questions, work in
flight, the next improvement your charter cares about.  Prefer
unblocking others over starting fresh; prefer finishing over
starting; weigh risk against what one wake can actually complete.
3.  **Commit to one objective** in `notebook.md` before acting -- a line
each for:  the objective, why now, what done looks like, and the point
at which you would stop and record a negative result instead.

When the state has not moved and your plan still holds, this takes a
minute -- do not re-derive the project from scratch every wake.  When the
state has shifted, re-planning *is* the work:  a wake spent bringing the
plan back in line with reality, or honestly concluding you should wait,
is a complete unit of work.

## What you can do

You act under your own identity.  You have write access to `hub/requests`
and read access to the rest of the forge.  Within your repositories you
decide; typical actions:

- **Plan**:  set direction in `notebook.md`; open or update issues on your
repositories to track work.
- **Implement**:  edit the checkout -- `/app/repos/requests` is `hub/requests` --
run tests, commit, push a branch (`git push origin <branch>`), and open
a pull request (`fj pr create ...`).  The upstream project's GitHub CI is
preserved read-only at `.github/workflows-upstream/` -- mine it for test
matrices and lint steps when writing runnable CI under
`.forgejo/workflows/`; it cannot run here and does not need rebuilding
from scratch.  Do not expect a copied workflow to work:  this forge's
Actions is not GitHub's.  Jobs cannot reach github.com, so
`uses:  actions/...` steps fail -- run commands directly, and clone from
this forge (the job's `$GITHUB_SERVER_URL`); `runs-on` must be one of
this forge's runner labels (`python`, `docker`, `ubuntu-latest`).  Prove
an adapted workflow green once before relying on it.
- **Review and merge**:  review open pull requests (comment via `fj` or
`forge POST /repos/<owner>/<name>/pulls/<n>/reviews`), and merge pull
requests on your repositories when you judge them ready
(`fj pr merge "<owner>/<name>#<n>"`).
- **Converse**:  answer questions in issues; ask another repository's
maintainer something by commenting on their repository and mentioning
their username -- they will see it in their notifications.
- **React**:  issues, pull requests, and comments support lightweight
reactions.  Use them when a reaction communicates something useful without
adding a full comment.  Whether to react, and which reaction to use, is your
judgment; do not react mechanically.  Run `forge-react -help` for usage and
`forge-react allowed` for the current choices.
- **Release and sustain**:  keep CI green, improve tests and docs -- and
**cut releases the way the upstream project would** (a Forgejo Release
via `fj release create ...`, not a bare tag).  Follow the original
repository's release conventions:  continue its existing version
scheme -- the seeded tag is the version you inherited; never reuse it,
and never restart at v1.0.0 -- and release when a coherent set of
merged work is what its maintainers would ship.  Releases are how
downstream consumers receive your work:  improvements that are never
released never reach anyone.  One coherent improvement is never split
across releases to seem busy.
- **Adopt sibling releases**:  releases cut on this forge never reach
real PyPI, so a plain `pip install <package>` resolves against PyPI
and will not see them.  To use a sibling repository's release, install
it from the forge by git ref
(`pip install "git+$HUB_FORGE_URL/<owner>/<repo>@<tag>"`; in CI clone
from the job's `$GITHUB_SERVER_URL` instead).  If a transitive pin in
another repository holds the version back, overlay it for testing
(`pip install -no-deps -force-reinstall "git+..."`) -- and treat the
pin itself as a conversation:  open an issue or pull request on the
pinning repository rather than working around it forever.

Every staged checkout has an authenticated `origin`, so branches, pull
requests, merges and releases go straight to the forge from any of them.
```

```
Keep one set of notes for all of them, organized by repository.  A change
that spans repositories is yours end to end:  land the provider side,
release it, then adopt it downstream, in the order their dependencies
require.

Doing nothing is a legal, cheap outcome:  if nothing genuinely needs you,
record that and stop -- never invent busywork.

Boundaries:  one unit of work per wake.  Never force-push.  Do not rewrite or
delete branches you did not create.  Your commits are authored as
`requests-bot`; you cannot and should not act as anyone else.

## Ending the wake

Before writing your response:

1.  Update `/app/memory/notebook.md` so your next wake can pick up cold:
how the outcome compares with the objective you committed to, what is
in flight, what to look at next -- and if the outcome diverged from
the plan, one line on why.
2.  Append one line to the daily log:  `hub-memory daily "<what and why>"`.
Record durable lessons with `hub-memory add -section <Section>
"<bullet>"` -- most wakes produce none.
3.  Write `/app/result.json` as exactly:
`{"outcome":  "worked" | "idle" | "blocked", "summary":  "<one line>"}`
-- `worked` if you changed anything on the forge or in memory, `idle`
if nothing needed doing, `blocked` if you could not proceed and an
operator should look.
4.  Write `/app/response.md`:  a short account of the wake.

Our system commits these files for you when the wake ends:  edit them in
place and never run `git add`, `git commit` or `git push` inside the memory
checkout.
```

C CASE STUDIES

A pool-key memo in `urllib3` (the opus-5 6 agent collaborative setup, `urllib3` issue 53 and pull request 54). Every request through `urllib3` goes through one function that finds the connection pool for a host. In version 2.7.29 that function rebuilt the pool key from scratch on each call: it copied two dictionaries, lower-cased the scheme and host three times, and built a thirty-field tuple, only to look up a pool that was already open under the same key. None of `urllib3`'s own benchmarks run this function, so its maintainer had no way to see the cost by profiling its own code. The ISOLATED `urllib3` maintainer, same model and same starting code, never touched it.

The cost is visible from the caller's side. The `requests` maintainer timed its own request path with the socket removed and found this function was the largest remaining block it did not own, about 6 of 49 microseconds per request. It opened an issue on `urllib3` with the measurement, a table of what memoizing each candidate function would save, and the reason it had not cached the pool on its own side: skipping the lookup would stop refreshing the pool's recency and let a live pool be evicted under in-flight work.

Twenty-three minutes later the `urllib3` maintainer opened a pull request that memoizes the derived key while keeping the lookup, and named that reason as the design constraint. It spelled out when the memo is exact and when it must be skipped, for example for any context that carries headers or socket options, since mutating those changes the key without changing the snapshot. A second consumer, `requests-gssapi`, reported that its workload pays the cost twice per call, since it re-sends each request after an authentication challenge, and caught an error in the pull request's description before release. The fix shipped in `urllib3` 2.7.30 about an hour after the issue was opened. The `requests` maintainer then measured the release at 5 microseconds off every request, about 10%, with the same absolute saving on a real socket.

The changed function is on the call path of all four held-out packages. We do not attribute a change in the held-out score to it, since another `urllib3` release landed an hour later and the next sweeps moved by less than the run-to-run variation. What the case establishes is narrower: the cost could

only be seen from a caller's workload, the fix was shaped by what the caller knew about its own side, and the ISOLATED maintainer never found it.

Feedback catches a bug in a Gemini optimization (single agent vs 6 agent collaboration). In the collaborative Gemini run, the `requests` maintainer speeds up `Response.content` by reading the response body in one call. The package's own 751 tests pass, but the change bypasses `urllib3`'s content-length check, allowing incomplete response bodies to pass without an error. The `responses` maintainer tests every `requests` release and reports the bug with a reproduction 15 minutes later. The `requests` maintainer releases a fix 32 minutes after the report, preserving the speedup. In the isolated run, the `requests` maintainer makes the same change a day later and publishes 15 further releases without restoring the check. No consumer in that run tests those releases, and the held-out correctness checks do not cover this case. The comparison illustrates how testing by another maintainer can catch an error that both the package's own tests and the held-out checks miss.