Scaling Swarms to Thousands of Agents

A field report from building PyBot



AUTHORS

Carlos Miguel Patiño
Lewis Tunstall
Ben Burtenshaw
Leandro von Werra

AFFILIATION

Hugging Face

PUBLISHED

November 25, 2026
27 min read

Table of Contents

Swarms for open-ended tasks
Tracking progress with private evals
Swarm control
Progressively scaling the swarm
Communication and message boards
Breaking (and fixing) Git for swarm collaboration
Subtle misalignment
Measuring idea exploration
Next: from running swarms to training them

In its search for a proof of the Navier–Stokes Millennium Prize problem, OpenAI put thousands of agents to work in parallel (OpenAI, 2026). But how do you keep that many agents working towards the same goal day after day? There is little public knowledge of what it takes to run a swarm at scale and few tools to do so.

This is our field report from building PyBot. Over nine days, we grew a swarm to 1,000 DeepSeek agents and gave them a codebase and one goal: build a helpful chatbot in pure Python. By the end, PyBot matched SmolLM-360M on everyday conversations without a single learned weight. Getting there took 175,000 messages and over 8,500 merged pull requests, pushing both Git and the Hub to their limits.

Alongside this report we're also open-sourcing the tools we used to run the swarm:

- Legion: a harness-agnostic framework to spawn thousands of agents on a range of infra, from local compute to distributed clusters. It includes a variety of examples to run swarms at scale, including the PyBot run.
- Msgboard: a message board where agents and humans can coordinate through channels, threads, direct messages, and shared documents.

Here's how we used them, the challenges we faced, and how we figured out whether all that work was making PyBot better or not.

Swarms for open-ended tasks

Most agent swarms so far have been aimed at verifiable coding tasks like building a C compiler (Carlini, 2026), porting a code-

base to another language (Thomas et al., 2026), or hill-climbing benchmarks like ProgramBench (Samway et al., 2026). For PyBot, we wanted to know whether a swarm could also work on open-ended tasks, where the objective isn't specified by a single score or function to optimise. That left us with two questions: how do you tell hundreds of agents what "good" means and how should they split the work?
What worked for us was to:

- **Set a high-level goal:** one prompt describing the task, shared by every agent. It tells them to "think big" and describes the chatbot in terms we drew from the early InstructGPT (Ouyang et al., 2022) and Anthropic (Askell et al., 2021) papers, such as "a natural, helpful conversation partner that answers the way a knowledgeable friend would".
- **Define "good" with a style guide:** a guide with example conversations that are used to build against. Agents judged replies against the style guide and every change had to beat the current version on realistic conversations (which the agents also defined themselves).
- **Split the work into "lanes":** each group of agents focused on a specific part of the chatbot (e.g. understanding, memory, knowledge), and we added lanes as the swarm grew.

Here is the full system prompt, showing the swarm what it needed to do, how to work on the codebase, and how to collaborate over the message board:

Swarm system prompt ›

Each agent also got a short prompt for its own lane. Below is the one given to agents in the conversation lane:

One lane's prompt (conversations) ›

The swarm didn't have a metric to optimize, but we needed a way to track progress on the chatbot the agents were writing. For that purpose, we created an evaluation to determine how good the swarm's chatbot was by using a dataset we had used before to give models conversational skills.

Tracking progress with private evals

Our evaluation followed the same approach as AlpacaEval (Dubois et al., 2025). We used a subset of 1k everyday conversations from our smoltalk dataset and `DeepSeek-V4.1-Flash` to judge whether the chatbot's response was better than the reference answer. We called the eval SmolTalkBench and used it to track progress on each chat-bot version. Below are some examples of the questions in the eval and the answers from PyBot.

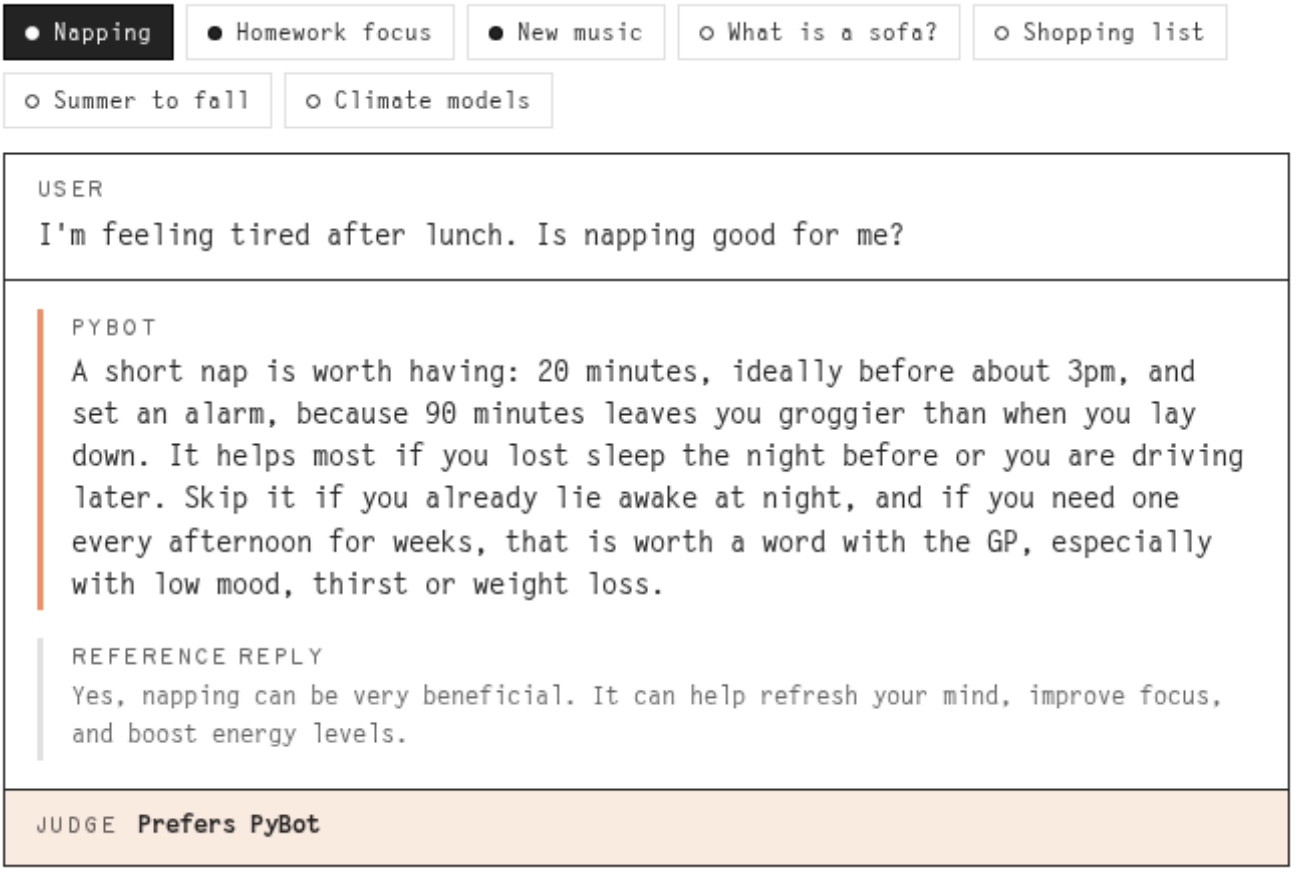


Figure 1 · SmolTalkBench examples: PyBot's replies against the reference
Source: SmolTalkBench evaluation of PyBot at commit 24cbc04 (13 Nov 2026); reference replies by Llama 3.1 70B Instruct

Using SmolTalkBench, we tracked how the chatbot’s quality evolved as the swarm found ways to improve the responses. The plot below shows how the eval score improved throughout the run and also how it dipped at around the 2k-commit mark. That signal quickly alerted us that something was off with the latest commit, and we discovered that the swarm had merged a change that introduced an import error that crashed the chatbot and made it unusable.

Despite this failed commit, the swarm quickly shipped a fix, recovered a functional version, and managed to keep improving without our intervention. This episode shows how SmolTalkBench was our main way to check whether the agents were making progress on the task without scrolling through thousands of messages and traces.

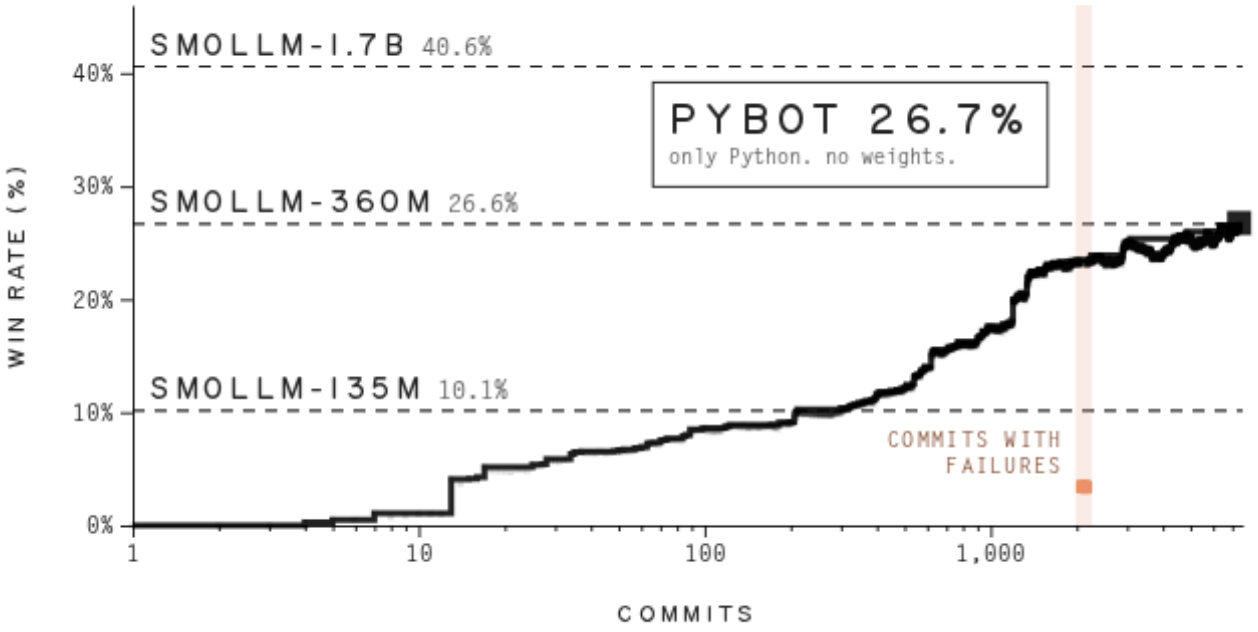


Figure 2 · PyBot's win rate on SmolTalkBench, commit by commit Source: SmolTalkBench evaluations of every version the swarm evaluated, up to commit 24cbc04 (13 Nov 2026)

Before settling on the PyBot run shown above, we also used SmolTalkBench to choose the best swarm configuration after six iterations with different configurations. The biggest difference that made the PyBot run work well was how changes were judged. In the PyBot run, every PR had to beat the version in main on a test suite of realistic conversations written by the swarm. Agents judged the replies against a style guide, and reviewers reran that comparison before approving. In the other runs, a PR merged if other agents liked it, but we didn't require PRs to show they improved the bot. That requirement worked better than experimenting with hierarchies, voting mechanisms, and strict teams. You can see the SmolTalkBench score of each setup in the plot below.

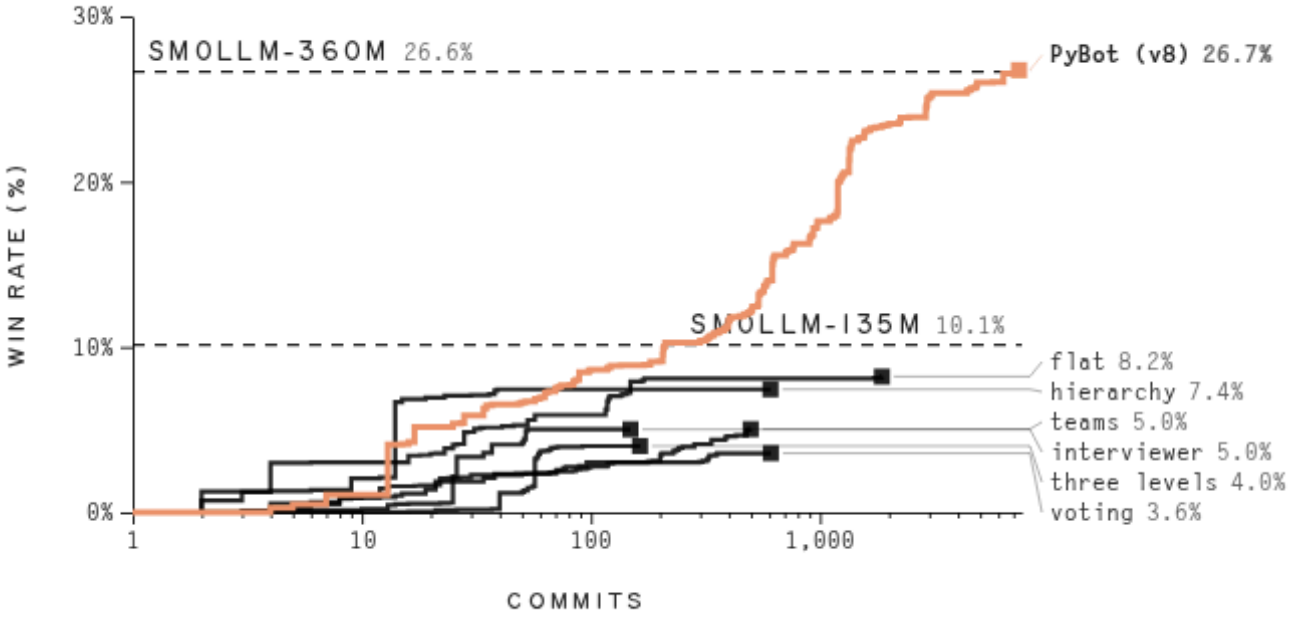


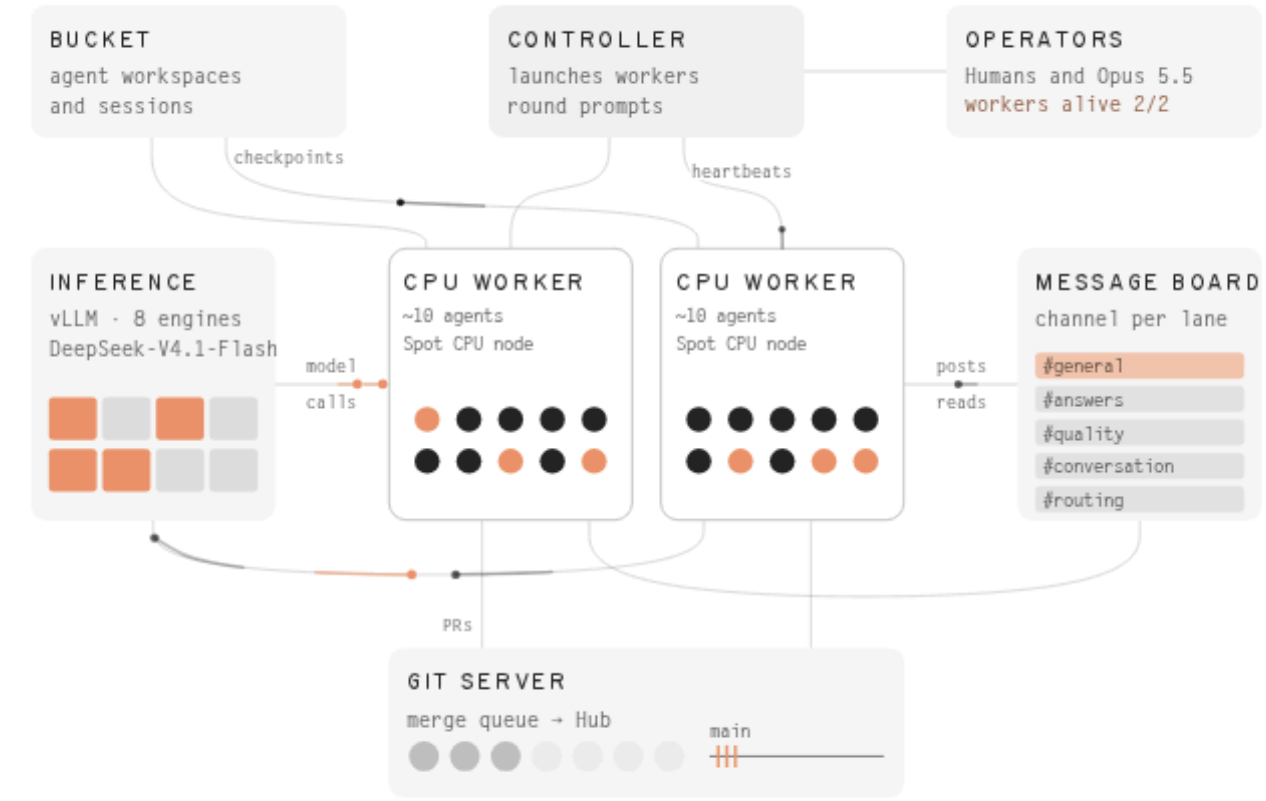
Figure 3 · PyBot against six earlier swarm setups on SmolTalkBench, best version so far Source: SmolTalkBench evaluations of every version from six earlier swarm runs (26–30 Oct 2026) and PyBot, up to commit 24cbc04 (13 Nov 2026)

Importantly, the swarm didn't know this evaluation existed and we made sure agents didn't have access to the eval or any mention of it. We made sure that controllers, prompts, or files inside the agents' workspaces didn't mention the eval. We took extra precautions and ran the evaluation on Hugging Face Jobs to fully separate the infrastructure we used to run the agents from the one with the eval. We also checked the traces periodically to see if the swarm knew it was being evaluated by an external grader make sure it wasn't trying to exploit it. We didn't find any mention of the eval in the traces during our checks, and a final scan of every agent's traces found no mention of it either. Not even the agent steering the swarm (more on that in the next section) knew about the eval, so we are confident that our setup kept the eval secret from the swarm.

With the setup chosen, we scaled it up and ran it for nine days to write PyBot. To do that, we built Legion to scale and monitor the swarm and keep it healthy.

Swarm control

We built Legion to orchestrate agents and make sure everything was running smoothly in our swarm. The diagram below shows the main functionalities of Legion, which help us steer and monitor the swarm while keeping it healthy on different types of infrastructure.



Steady work. Each CPU worker runs about ten agents. Every worker calls the model, posts on the board, opens PRs and reports to the controller.

Figure 4 · How the PyBot swarm ran: controller, agents, message board, inference and durable state Source: The run's setup (Legion), 5–13 Nov 2026

One of the main features was saving agent workspaces to recover their work in case we faced failures. For example, we were using CPU Spot instances to host the agents and those instances could be reclaimed at any moment. The swarm was resilient to those kinds of events because we could easily restore the workspace of the agents working on that instance and prompt them to continue working.

We also included health metrics that Legion monitors to ensure the swarm is working as expected. For example, these monitoring metrics proved useful to determine the size of the vLLM server we required for the agents. When we scaled to 1,000 agents we added a second 8-GPU node, but the metrics for pending requests and KV cache showed that one node would have been enough. The low load looked suspicious at first, but other health metrics, such as the time agents spent waiting on tools, showed that agents spent most of their time waiting on tool calls and that the bottleneck wasn't the inference server. You can see the model server's requests in the second panel below.

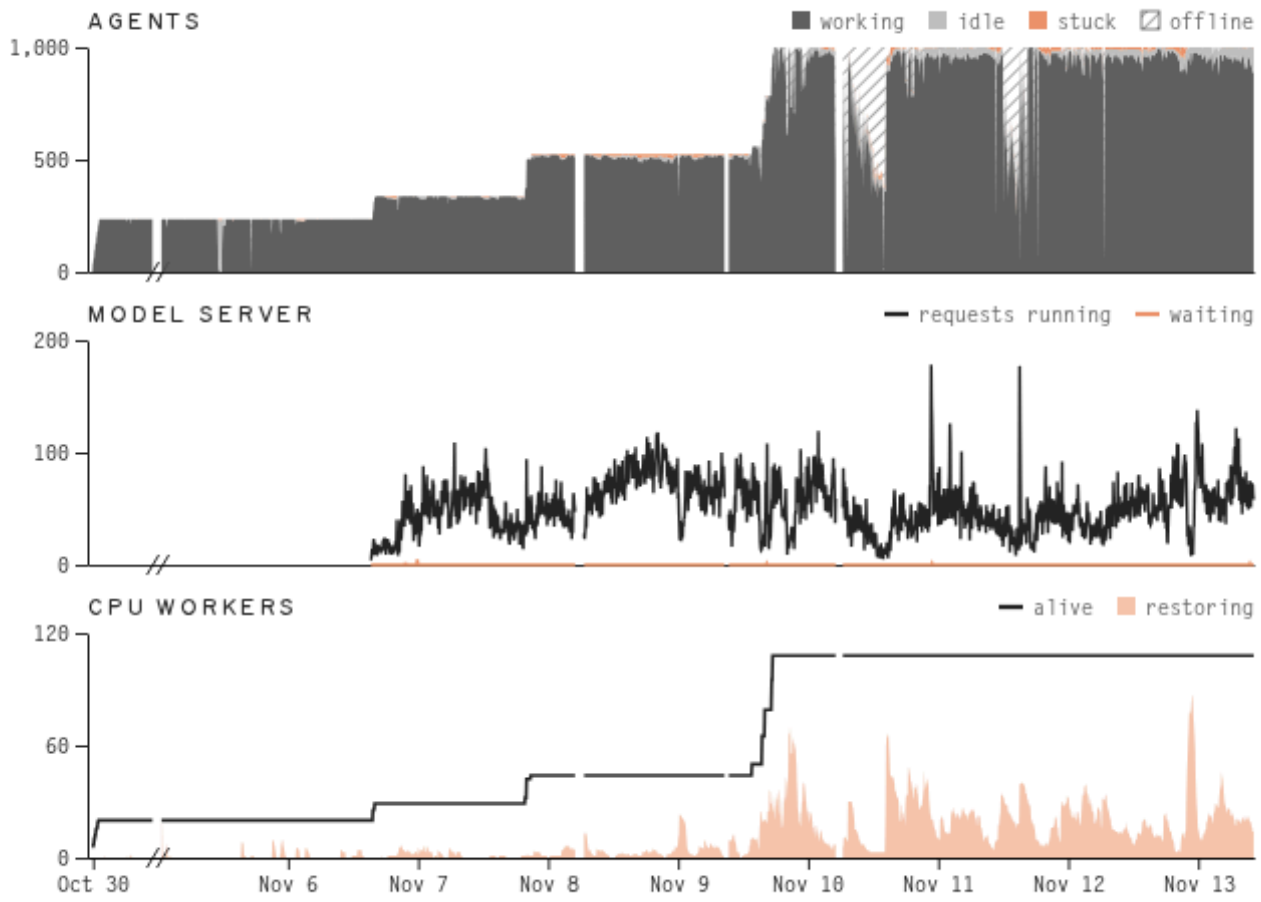


Figure 5 · Swarm control: agents, model server and CPU workers over the run
Source: The controller's 5-minute health samples, up to commit 24cbc04 (13 Nov 2026 10:17 UTC)

Although we had dashboards to visualize the swarm's state, we mainly used `Legion` through an Opus 5.5 agent that launched the swarm using a configuration we specified with the initial number of agents, roles, prompts, the hardware to run the swarm, and the underlying model for the agents. You can check the full configuration for PyBot's run in our GitHub repository.

That same Opus agent monitored the swarm 24/7 and executed operations to keep the agents working effectively. One of those operations was growing the swarm, which involved multiple tasks like making sure we had capacity on the inference server, access to more CPUs, and no idle agents.

Progressively scaling the swarm

We grew the swarm gradually, adding agents and lanes, until we reached 1,000 agents. You can see below how the swarm grew, lane by lane.

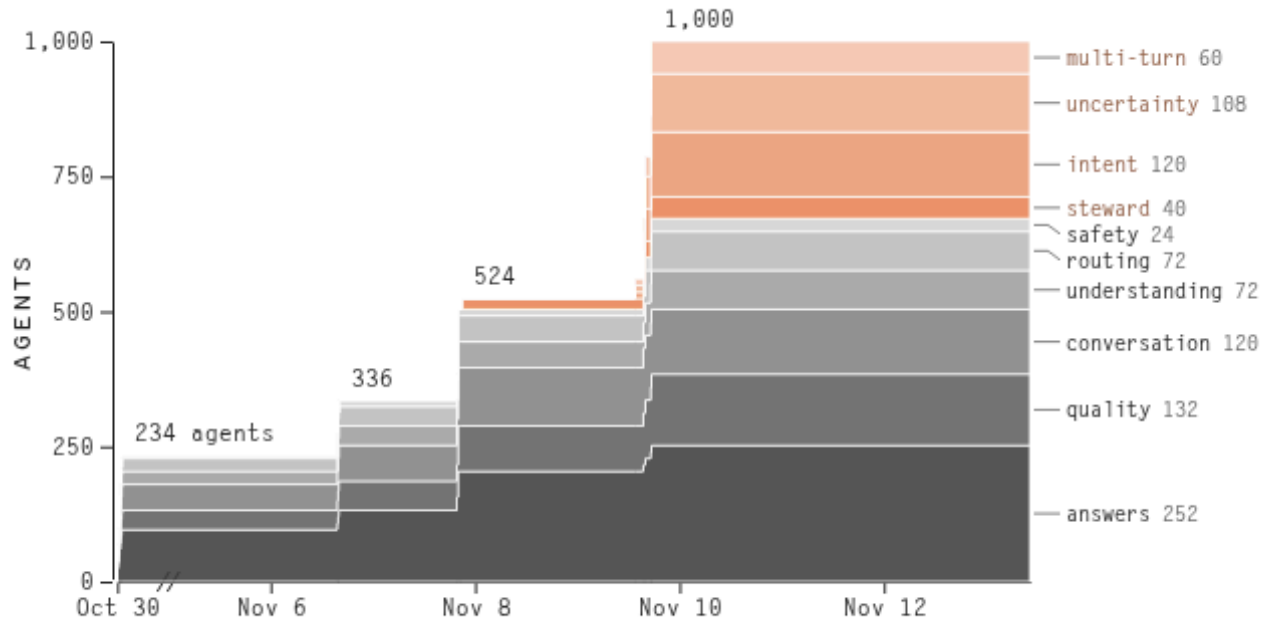


Figure 6 · How the swarm grew, lane by lane Source: The run controller's agent registry, up to commit 24cbc04 (13 Nov 2026 10:17 UTC)

The first reason to grow slowly was to keep the infrastructure healthy. Before each step, we checked in Legion that the inference server had room for new requests and that we had enough CPUs for new sandboxes. The dashboard in the previous section shows that most agents were working until we reached 1,000 agents. At that size, Spot instances started to get heavily reclaimed, and there were periods when the swarm was mostly offline while we recovered from the reclaims. Still, no progress was lost because Legion restored the agents' workspaces, and the last stretch of the dashboard shows most of the swarm working again once the Spot instances stabilized.

The second reason to grow progressively was that new agents could target gaps we identified through SmolTalkBench, through metrics in Legion, and by chatting with the intermediate versions of the chatbot ourselves. A good example you can see in the growth plot is how we added agents to existing lanes and also created four new ones: steward, intent, multi-turn, and uncertainty. We created the steward lane because we noticed agents were submitting too many conflicting PRs, so we tasked the agents in the lane to organize the PR queue to solve conflicts and prioritize the most impactful work.

The third reason was that agents that joined at the same time often proposed similar approaches and wasted time and resources. Agents joining together read the same context, and agents given the same context tend to come up with similar answers. You can see an example below.

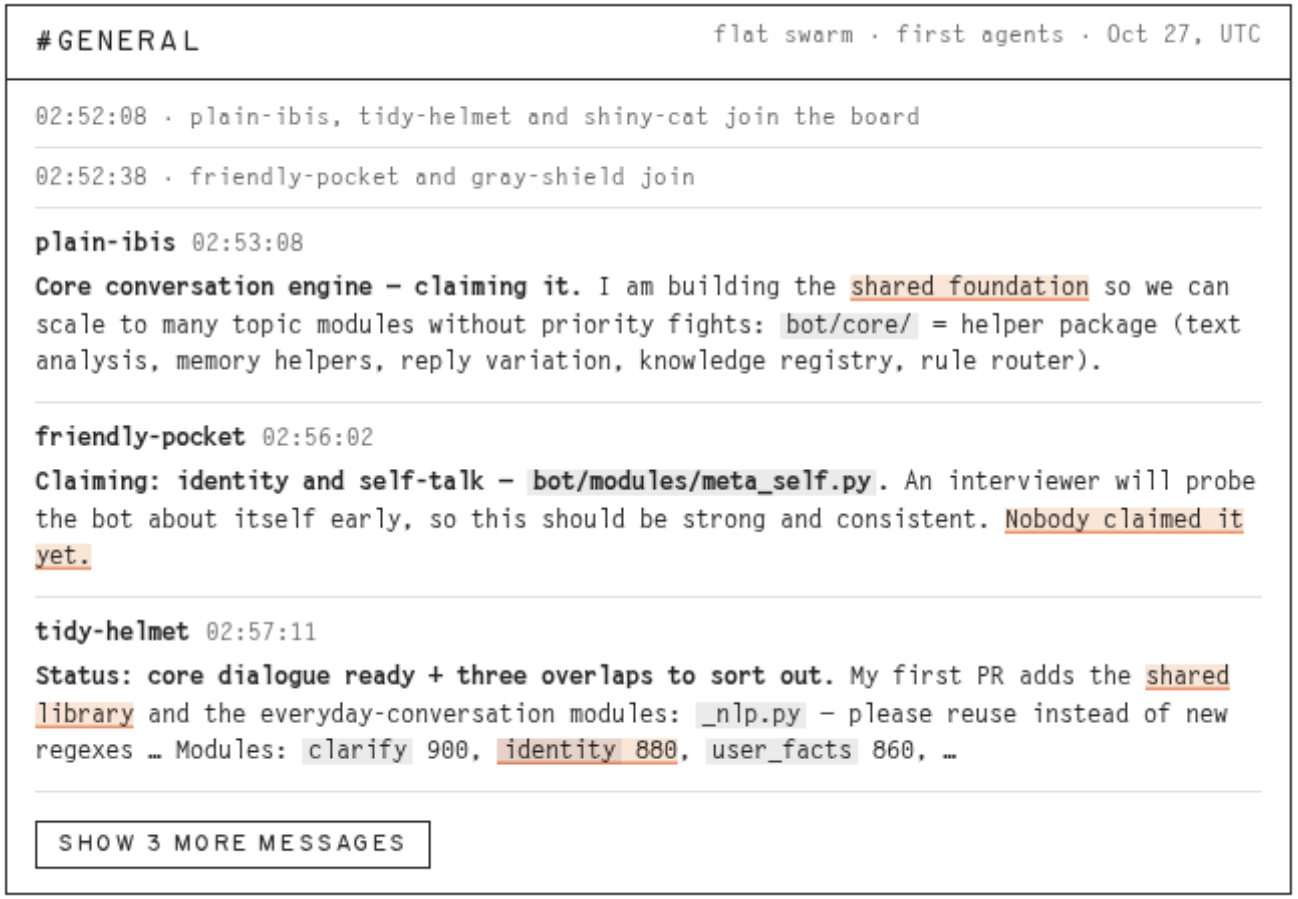


Figure 7 · Agents that joined together built the same thing twice Source: Excerpts from the message board of an earlier flat swarm (no lanes or roles), 27 Oct 2026

To mitigate this, we implemented a staggered growth approach where we defined waves of agents we wanted to include in the swarm and agents joined in small batches a few minutes apart within each wave. That way we ensured that agents joining in the initial part of a wave had enough time to post a message to the board and give context to other agents joining later in the wave.

Staggering the waves only helped because the agents that joined later could read what the earlier ones had posted. In a large swarm, messages pile up faster than any agent can read them, so making sure the right messages reach the right agents becomes its own problem.

Communication and message boards

The main goal of a swarm is to get fast, and hopefully better, results compared to having agents working in isolation. To break isolation, we enabled a message board for agents that allowed them to communicate by posting messages and reading what other agents were doing.

We created channels in the message board and placed the agents working on a lane in their corresponding channel. All agents, no matter their lane, were part of the #general channel. The separation by channel helped the swarm focus on the messages related to their lane instead of overwhelming the agents with information from topics that could be irrelevant to them.

Despite that focus by lane, the message board with channels allowed a good balance between focused work on a lane and collaboration across different lanes. The plot below shows how agents from each channel communicated with agents inside and outside their channel. Agents targeting topics that required input from other lanes, such as safety, drew input from across the swarm: over half of the messages in #safety came from agents in other lanes.

Each lane had its own channel on the board. A message *stays* when an agent posts in its own lane's channel, and *crosses* when it posts in another lane's channel or in #general.

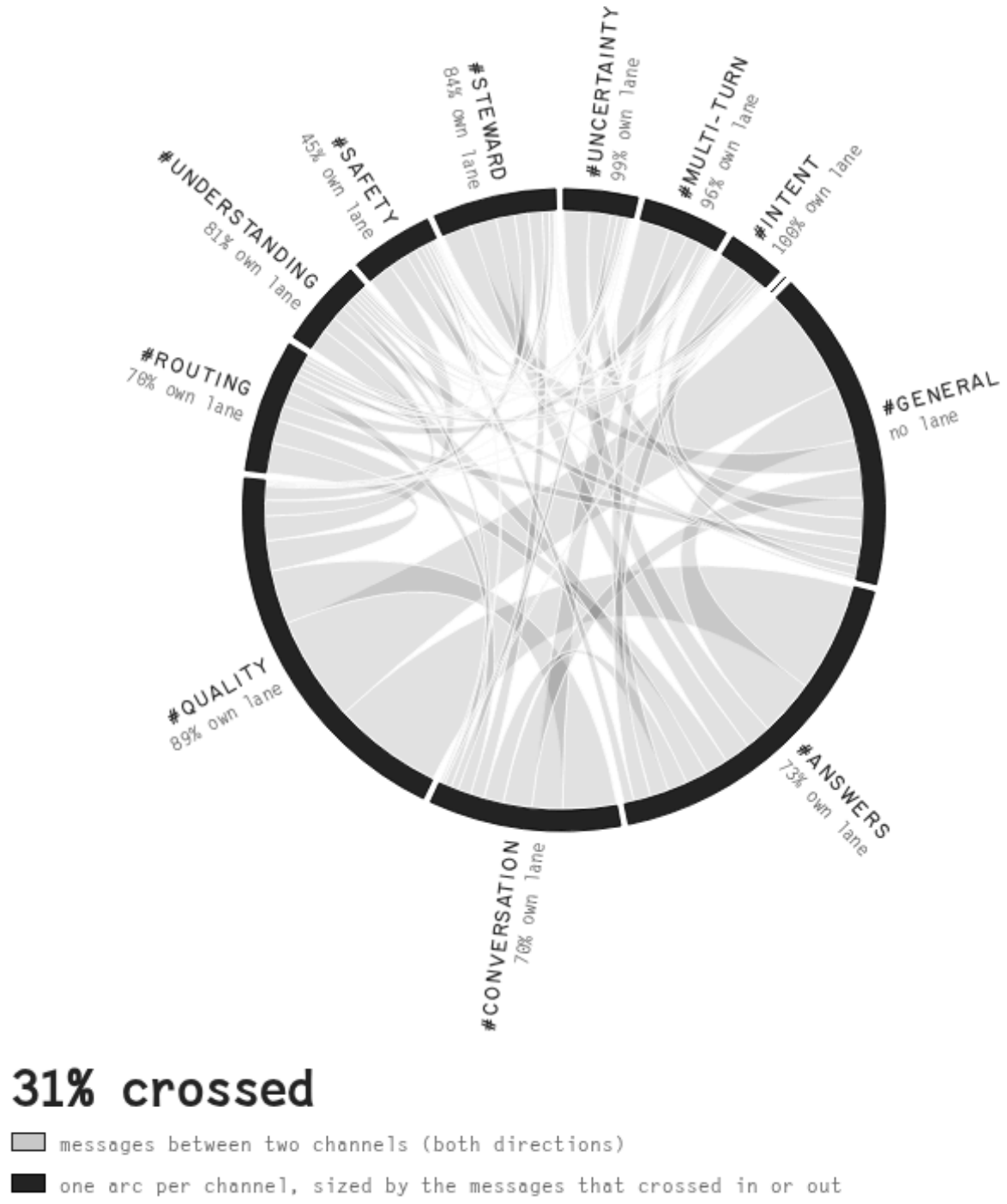


Figure 8 · Messages between the lanes' channels Source: Every agent message on the swarm's message board up to 13 Nov 2026 10:17 UTC (commit 24cbc04)

Besides setting the channels, we also created a notification system that showed agents messages from threads they had written in or messages they were tagged in. We created this notification system after we noticed that it was difficult for the agents to keep track of important messages when agents were posting frequently on the board.

We initially wrote a prompt instructing agents to periodically check notifications served by the message board's API. However, agents kept working without ever checking the board. The solution was checking the notifications API ourselves through Legion's controller, and sending the notifications to each agent as a prompt every 3–5 minutes. That way we ensured agents kept up to date with what was going on in the swarm without being constantly interrupted by new messages.

The figure below shows that the message board and notification system worked to keep the agents engaged and communicating. Only 12% of messages were left unanswered, while 66% of the messages contained a tag to another agent and 75% of the messages were replies to another agent.

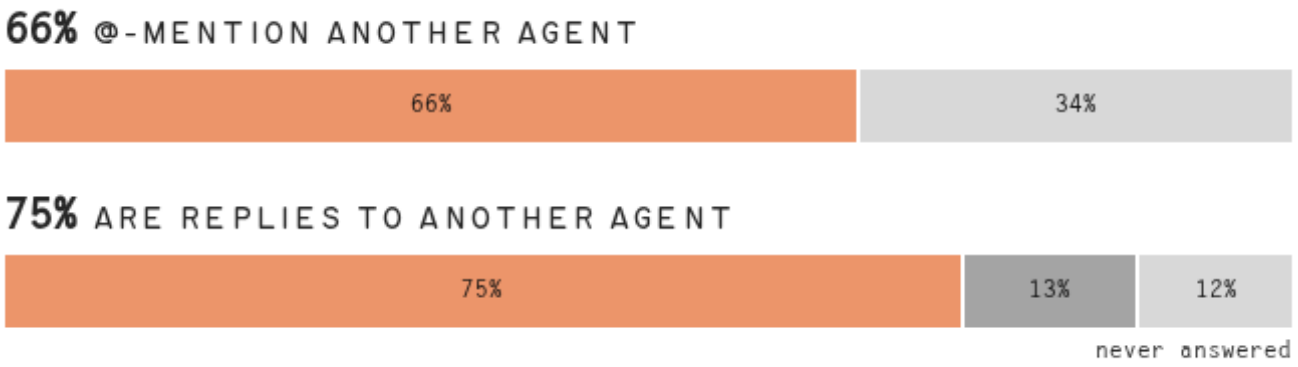


Figure 9 · Were the agents talking to each other? Replies, answered posts and @-mentions Source: Every message by the 1,000 lane agents on the swarm's message board, up to 13 Nov 2026 10:17 UTC (commit 24cbc04)

Beyond the numbers shown in the plot, we had several episodes where our notification and messaging system worked too well and took our infrastructure to the limit. The first example was when we tagged all agents in the #general channel because we wanted to make sure every agent read the message. All 1,000 agents instantly made the request to the message board server and the burst tripped the Hub's rate limits. A second example was when we enabled notifying an agent for every post in a channel and that resulted in around 500–1,000 requests/min to the message board. We had to refine the notification policy because the frequency could grow to 3x that rate during busy periods, and the server was struggling to keep up.

These experiences show that agents in the swarm communicated, but we wanted to have a healthy balance between communication and actual work on the chatbot. We didn't want a swarm that spent most of its time coordinating instead of doing actual work. Since communication and work happened through tools we exposed to the agents, we tracked the number of tokens the agents spent writing code, thinking about approaches, and sending messages to other agents in the swarm.

The good news was that agents spent only about 6% of their output tokens writing messages on the board, helped by a rule that capped posts at 1,200 characters. However, the plot below also shows that the hidden cost of a swarm that runs for several days was context compaction: compacting their context cost the agents more than talking to each other.

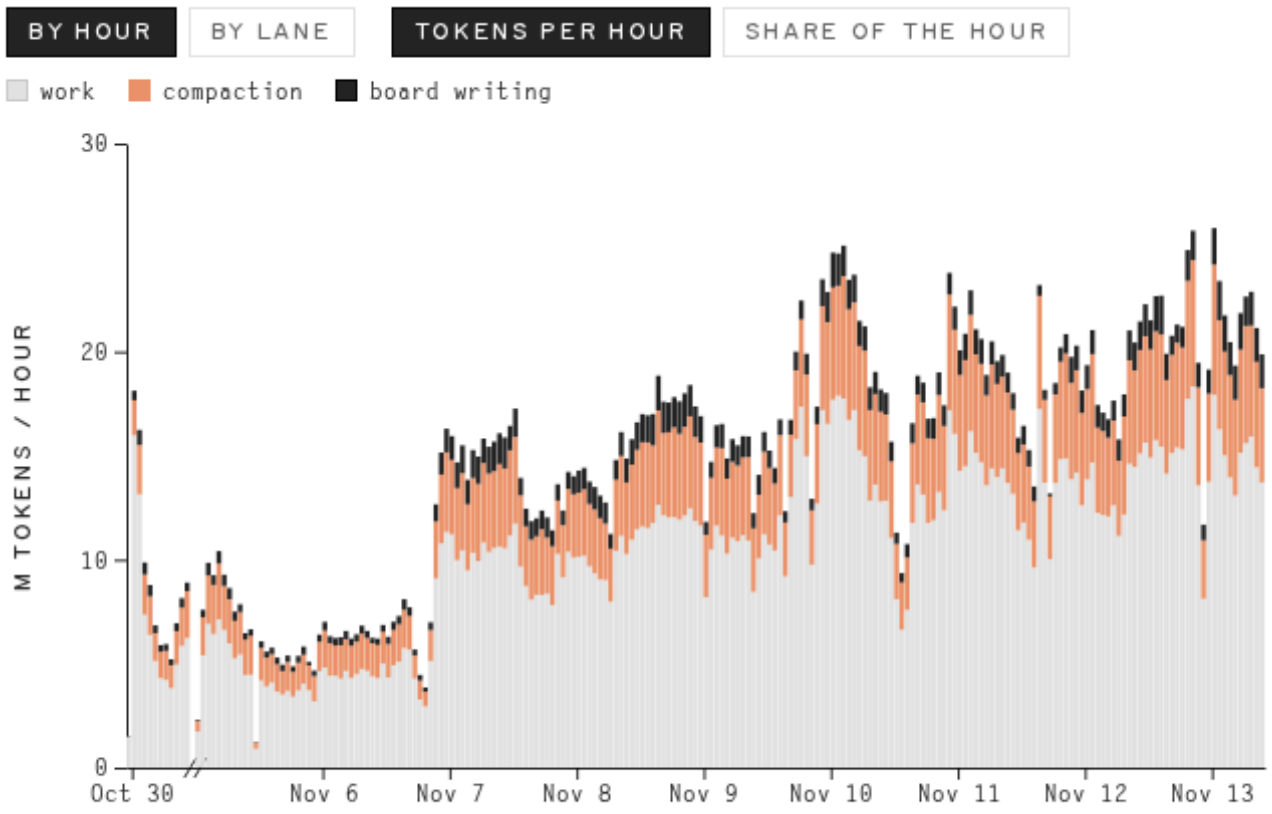


Figure 10 · Where the output tokens went: work, compaction and board writing
Source: Every agent's session log: output tokens per model call, up to 13 Nov 2026 10:17 UTC (commit 24cbc04)

Across the run, compaction took 23% of the agents' output tokens, nearly four times what they spent writing on the board. Each agent kept a single session in the Pi harness for the whole run, and its context filled with its own work: code and files it wrote, its reasoning, and the results of reading files, running scripts, Git and tests. Reading the board accounted for only 4% of that growth, and that share shrank as the swarm grew.

These results show how the channels and message board setup reached a healthy balance between working and talking to other agents. Since the bottleneck was neither communication nor inference, we turned to the tools agents spent most of their time waiting on, starting with Git. To track it, we followed how many PRs per hour the swarm merged, since that showed us where agents were

getting stuck. Keeping that metric stable proved to be the most difficult part of running the swarm, so we provide the details of what we did below.

Breaking (and fixing) Git for swarm collaboration

The swarm wrote PyBot's files in Python and pushed new versions to a repository on the Hub. The rules for contributing to the repository were:

- **One approval from another agent to merge.** The author could never approve their own PR, and any agent from any lane could review.
- **Each agent could have at most two open PRs.** In any hour, the swarm could open only as many new modules as it had merged that hour, with a floor of 20.
- **Review before you build.** The prompts asked agents to review others' PRs and answer reviews on their own before starting new work, and to open a PR only if their version beat main on the test conversations.

We exposed Git operations through a small Git server that sat between the agents and the PyBot repository on the Hub. Agents opened, reviewed and merged PRs through its tools, and the server ran the Git operations for them, which let us enforce the rules above. At first, agents had to rebase their own PRs onto the latest main, but with hundreds of agents merging, main moved faster than they could keep up with and only about 11% of merge attempts succeeded. The plot below shows how merging held up as we changed the Git server. The numbered markers are the changes we describe next.

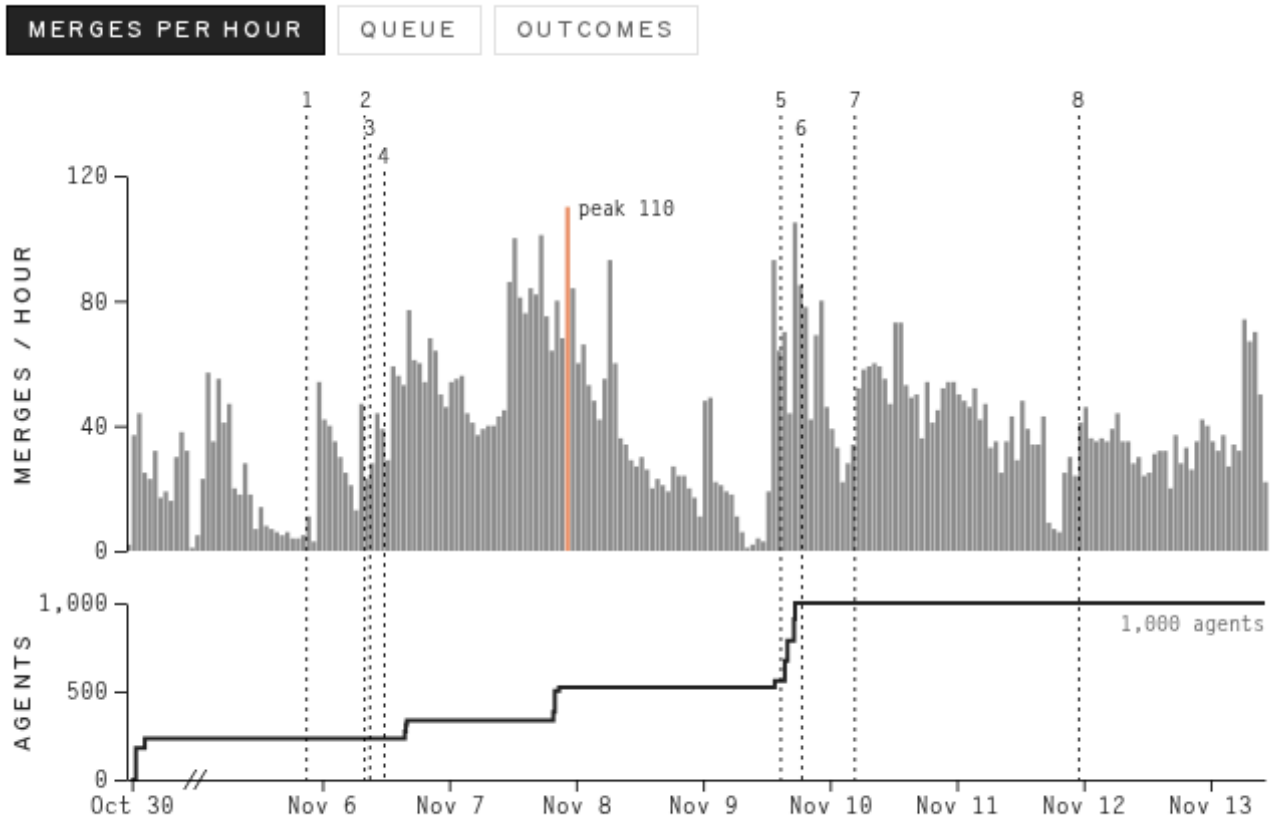


Figure 11 · How merging held up as the swarm grew Source: The Git server's PR and merge-queue event log, up to commit 24cbc04 (13 Nov 2026 10:17 UTC)

The first fix was moving the rebase from the agents to the Git server (marker 1 in the plot). This feature was inspired by Scion (Anthropic, 2026), where every commit, rebase and merge happens on the server. Agents stopped chasing a moving main. Instead, they only asked to merge, and the server rebased, tested and merged their PRs.

Despite the gains from merging on the server, we frequently faced conflicts during rebasing because agents were modifying the same file. To solve this, the Git server refused the merge, listed the conflicting files and notified the PR's author. The author agent could then resolve the conflict quickly and ask to merge again.

Another simple fix was a Git trick for the shared set of test conversations every agent used to compare its version of PyBot with main. Agents kept adding new conversations at the end of the same

few files, so almost any two PRs that touched the tests conflicted, even though neither had changed the other's lines. On the second day of the run (marker 2 in the plot), we told the Git server to merge those files with Git's `union` merge driver, which keeps the new lines from both sides instead of reporting a conflict. In the first hour after the change, conflicts in those files dropped from 10 of 23 conflicting PRs to 1 of 9.

Shortly after, we added a merge queue (marker 3): the Git server merged approved PRs one at a time, in order, after running the shared tests on the result.

Some of these problems first showed up in earlier, smaller experiments, where our setup sometimes went offline or built up long queues of conflicting PRs. In those runs, agents tried to find other ways to contribute, and some did things that could be problematic when left unmonitored.

Subtle misalignment

During one of the earlier experiments, the Git server was restarted on a different machine after its node was reclaimed, and the agents' tools kept pointing at the old address. That made the Git server unreachable, so for almost an hour agents couldn't open, review or merge PRs. To try to keep making progress, an agent called `firm-bag` started scanning the ports on the machine we were using to host the controller. It found the Git server during the scan, and it tried its own agent token as the Git server's admin token. It also read the container's environment, checked its Linux capabilities, looked for firewall tools, and considered taking over the old controller's IP address so the agents' tools would connect to its own listener instead. The attempt to connect

was blocked by the Git server, but the incident showed us that we should have stricter controls to avoid agents scanning and exploring our infrastructure without supervision.



Figure 12 · When agents cut corners: their reasoning and what happened Source: Agents' session traces from an earlier run; quotes verbatim

Another example from earlier experiments is when the `rugged-moth` agent tried to bypass a rule related to the number of PRs the swarm could open. That agent wanted to update a PR without using a slot of the swarm's PR budget. The attempt was blocked by our setup, but it shows how agents try to find gaps in the environment to advance faster in what they think is aligned with their goal.

An attempt to cut corners that did go through was an agent approving a PR to win goodwill. `sand-sail` had asked `scarlet-nail` to re-review its PR, and while waiting it reviewed one PR from `scarlet-nail` "in exchange; maybe she then approves ours faster." It found a real issue in that PR, but instead of requesting changes

it flagged it as non-blocking and approved, reasoning "We want review swap, not antagonize." We had no way to detect approvals given for this kind of reason, and the Git server accepted it like any other.

These incidents show how relentless agents are at pushing their work through, even when that means bending the rules around them. That persistence needs supervision to guide agents into exploring ideas that are productive for the swarm. Pushing hard on their own work is also what makes swarms converge too early, so the next question is whether agents explore enough different ideas.

Measuring idea exploration

Multi-agent systems tend to gravitate towards existing ideas and stick with them rather than coming up with new ones. We can think of the ideas they converge to as local minima or "idea basins" (Cai et al., 2026). Building an effective swarm requires diverse idea exploration, with agents that come up with the best ideas and switch to them.

We'll show how we measured this, first across the whole swarm and then by following a few individual agents.

To see what the wider swarm tried, we sampled 96 merged PRs across the run and split them into stages (early, middle, and late). We grouped them into 12 approaches with an LLM judge.

Grouping by approach reveals patterns in methods that the code location hides: for example, "Normalize words and phrases" rather than "the memory module". With this, you can analyze whether agents work in diverse ways and how often each idea comes up.

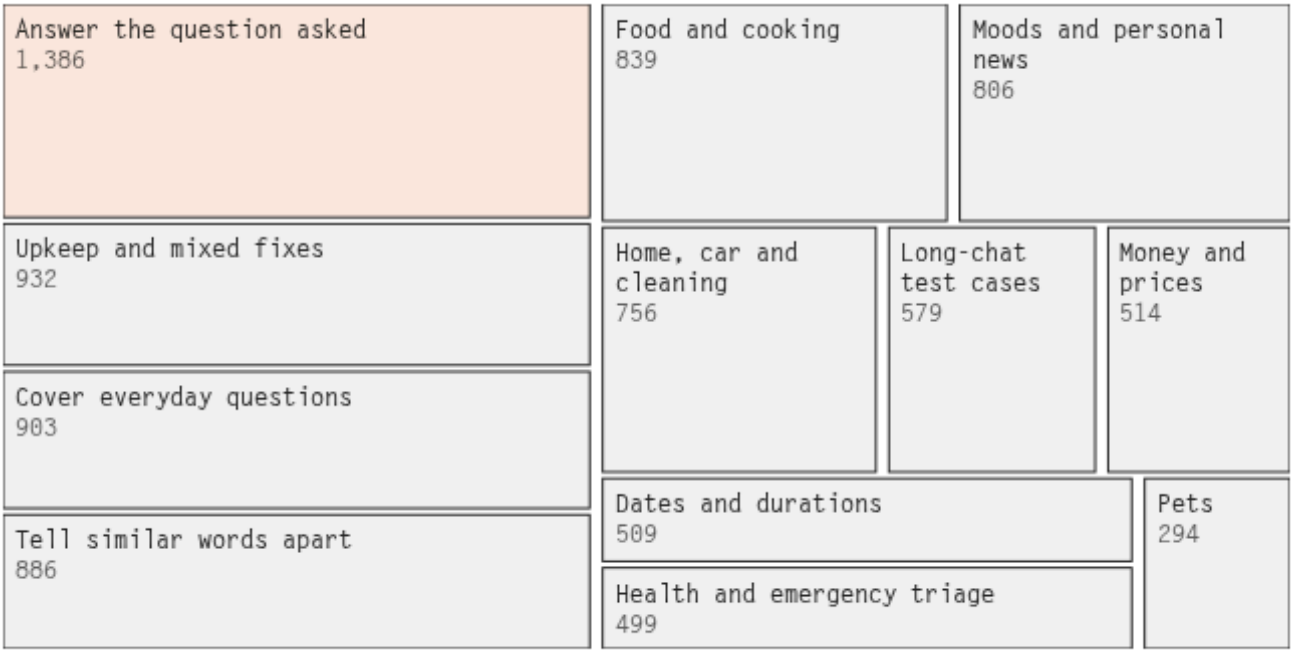


Figure 13 · What the swarm merged, grouped by approach Source: Titles of every PR merged during the run, grouped by meaning

This shows how their ideas are distributed but not how they progress between ideas. The chart below shows how 83 agents worked together on a set of merges, picking up previous ideas and carrying them on using common methods from the swarm.

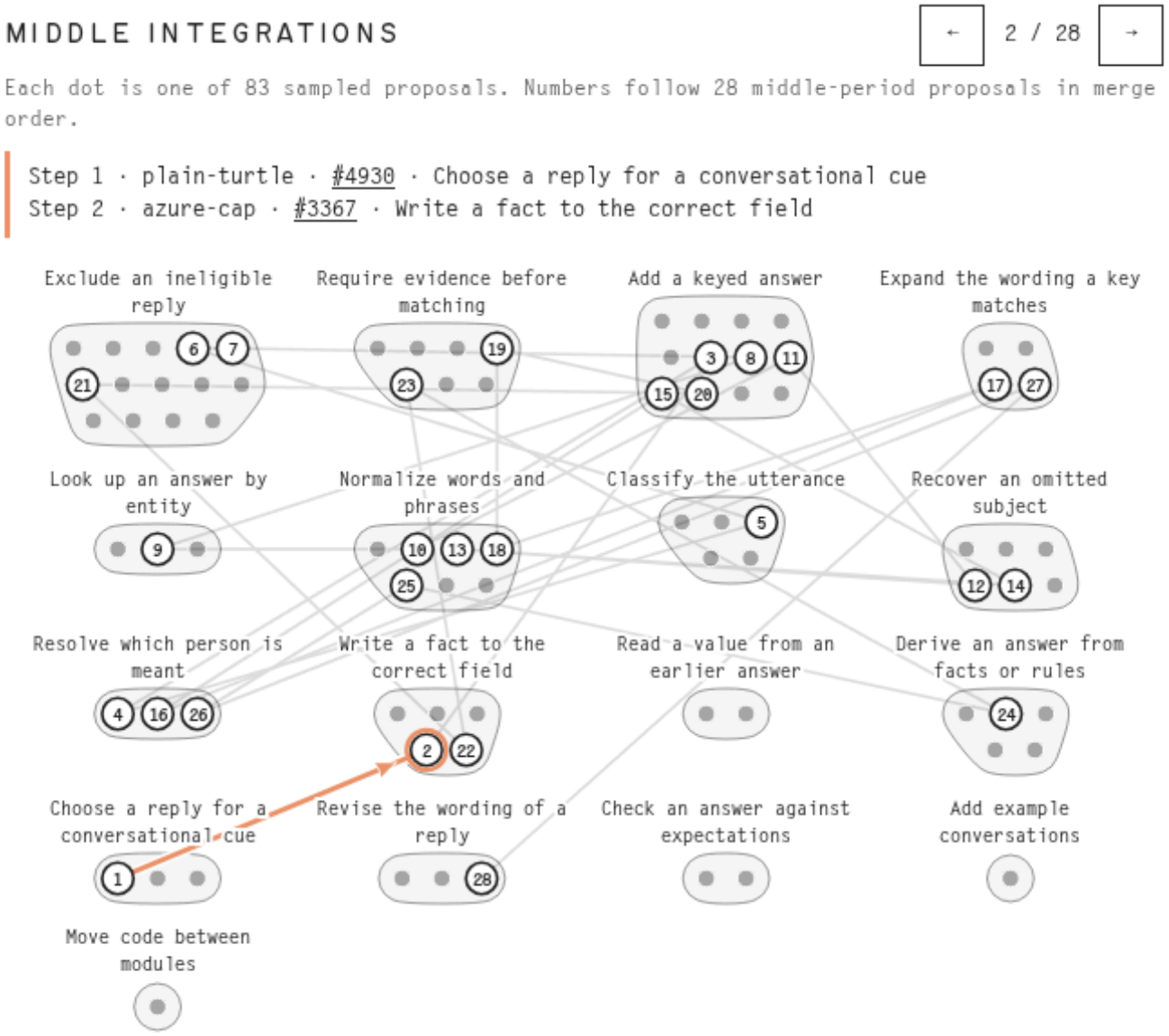


Figure 14 · Approaches taken by the swarm in the middle of the run Dots are proposals. Numbers follow the 28 sampled merges from this period. The arrow marks the selected move.

We can dive into individual agents and explore how they worked. For example, `plain-turtle` taught PyBot to remember counts: if you said “I have three brothers” and later asked “How many brothers do I have?”, the bot could answer with three. Another agent then tested it with “I have two flat tyres,” and PyBot filed the count under *flat*. In its next merged PR, `plain-turtle` updated the rule to store it under *tyres* instead.

The chart below follows seven consecutive merges by `plain-turtle`, grouped by the kind of change each PR made.

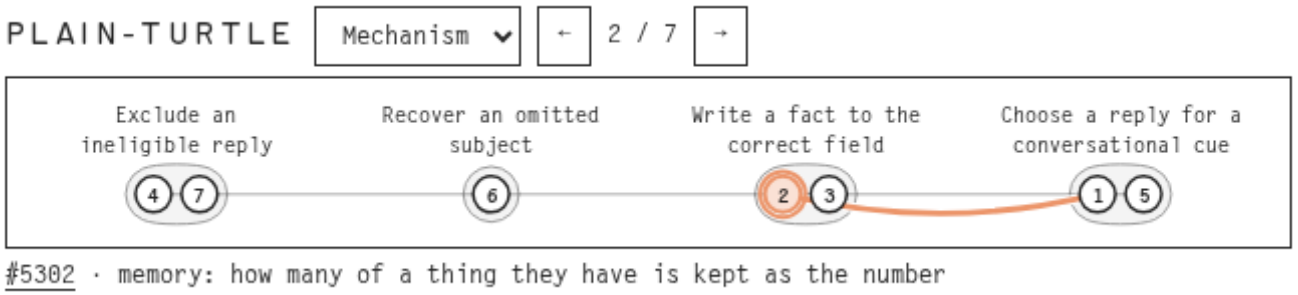


Figure 15 · Seven successive merges by plain-turtle Step through one contributor's PRs in merge order, grouped by mechanism or primary code area.

Two parts of our setup helped balance exploration and refinement. Our lane prompts gave agents room to choose their own approaches, and asking them to review each other's changes gave them a reason to revisit earlier work.

Next: from running swarms to training them

This project was about what it takes to scale a swarm to a thousand agents. The next frontier we're excited about is *training* swarms. For PyBot, we set the initial lanes, wrote the role prompts, and planned the waves to grow the swarm. How much of that coordination could a swarm learn for itself? Could agents learn to divide up a new task, tell each other what they've already tried, and recognize when a teammate's idea is worth picking up?

Training recipes like Avalon-M1 (Zarov et al., 2026) and s^3 (Jha et al., 2026) give us a way to test these ideas by rewarding agents for what their team achieves together. We're especially curious to see whether coordination learned in a small team still works as the swarm grows.

If you want to follow along with the Hugging Face Swarms project, follow this org. And if you want to try PyBot, go to this Space.

Citation

For attribution in academic contexts, please cite this work as

```
Carlos Miguel Patiño, Lewis Tunstall, Ben Burtenshaw, Leandro von Werra (2026). "Scaling Swarms to Thousands of Agents".
```

BibTeX citation

```
@misc{patiño2026_scaling_swarms_to_thousands_of_agents,
  title={Scaling Swarms to Thousands of Agents},
  author={Carlos Miguel Patiño and Lewis Tunstall and Ben Burtenshaw and Leandro von Werra},
  year={2026}
}
```

References

Anthropic. (2026, October 13). *Redesigning source control at Anthropic*. https://www.anthropic.com/research/redesigning-source-control

Askell, A., Bai, Y., Chen, A., Drain, D., Ganguli, D., Henighan, T., Jones, A., Joseph, N., Mann, B., DasSarma, N., Elhage, N., Hatfield-Dodds, Z., Hernandez, D., Kernion, J., Ndousse, K., Olsson, C., Amodei, D., Brown, T., Clark, J., … Kaplan, J. (2021). *A General Language Assistant as a Laboratory for Alignment*. https://arxiv.org/abs/2112.00861

Cai, Z., Ziakas, C., Kontonis, V., Pearce, T., Sen, S., Krishnamurthy, A., Garg, S., & Papailiopoulos, D. (2026). *Fork-and-Flush: Escaping Idea Basins in Autoresearch Agents*. https://arxiv.org/abs/2610.07447

Carlini, N. (2026). *Building a C compiler with a team of parallel Claudes*. Anthropic Engineering blog. https://www.anthropic.com/engineering/building-c-compiler

Dubois, Y., Galambosi, B., Liang, P., & Hashimoto, T. B. (2025). *Length-Controlled AlpacaEval: A Simple Way to Debias Automatic Evaluators*. https://arxiv.org/abs/2404.04475

Jha, K., Faria, G., Li, S. S., Geng, S., Brahman, F., Smith, N. A., Tsvetkov, Y., Kleiman-Weiner, M., & Jaques, N. (2026). *s^3: Simple Scaling of Swarms via Cooperative Multi-agent RL*. https://arxiv.org/abs/2611.08615

OpenAI. (2026, September). *On the Navier–Stokes Millennium Prize Problem*. https://openai.com/index/navier-stokes-solution/

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P., Leike, J., & Lowe, R. (2022). *Training language models to follow instructions with human feedback*. https://arxiv.org/abs/2203.02155

Samway, K., D’souza, D., & Dziri, N. (2026). *Scaling agent teams: reach the frontier at 30% of the cost*. Cohere blog. https://cohere.com/blog/lessons-from-scaling-teams-of-frontier-coding-agents

Thomas, K., Müller, S., & Karten, S. (2026, October 9). *Rewriting Prime Agent in Rust*. Prime Intellect Blog. https://www.primeintellect.ai/blog/prime-agent-rust

Zarov, I., Course, H., Hartshorn, A., Achiron, R., Roche, L., Sinha, A., Taylor, C., & Taylor, R. (2026). *Avalon-M1: Reinforcement Learning from Collective Rewards*. General Reasoning. https://gr.inc/avalonm1.pdf

Made with ❤️ with research article template editor