/blog

A review of OpenAI o1 and how we evaluate coding agents

September 12, 2024 • by The Cognition Team

<u>Devin</u> is an AI software engineering agent that autonomously completes coding tasks. We've been testing OpenAI's new o1-mini and o1-preview models with Devin for the past several weeks and are excited to share some early results. To contextualize these results we will also discuss our evaluation methodology and our technical approach to building reliable coding agents.

How Devin uses language models

One of the biggest challenges in software engineering work is *reasoning,* and LLMs are a key building block for reasoning about code in modern ML systems. Devin is a compound AI system that uses a diverse set of model inferences to plan, act, evaluate, and use tools.

Naturally, when OpenAI offered us early access to o1, a series of models specifically optimized for reasoning, and the chance to provide feedback on its impact on our performance, we were thrilled to start working with it.

First impressions of OpenAI o1

For this evaluation, we use a simplified version of Devin, called "Devin-Base", since the production version of Devin uses models post-trained on proprietary data. This allows us to specifically measure how changes in base models affect Devin's capabilities.

We found that, in comparison to GPT-4o:

1. o1-preview has a striking ability to reflect and analyze. ==It will often backtrack and consider different options before arriving at the correct solution, and is less likely to hallucinate or be confidently incorrect.==

2. Using o1-preview, Devin is more likely to ==correctly diagnose root cause issues, rather than addressing the symptoms of a problem==. This stands out particularly when Devin is investigating error messages with complex and indirect upstream causes.

3. Prompting o1 is noticeably different from prompting most other models. In particular:
   - Chain-of-thought and approaches that ask the model to "think out loud" are very common in previous generations of models. On the contrary, ==we find that asking o1 to==

only give the final answer often performs better, since it will think before answering regardless.

- o1 requires denser context and is more sensitive to clutter and unnecessary tokens. Traditional prompting approaches often involve redundancy in giving instructions, which we found negatively impacted performance with o1.

4. o1-preview's improved intelligence also trades off with increased variability in following highly prescriptive instructions.

5. With o1, inference is multiple times slower than previous OpenAI model releases.

Quantitatively, we found that swapping subsystems in Devin-Base that previously depended on GPT-4o to instead use the o1 series led to significant performance improvements in our primary evaluation suite, an internal coding agent benchmark we call cognition-golden (described in more detail later in this post). It will take additional effort to fully integrate o1 into our production system, but we expect it to further boost Devin's capabilities once that's done.
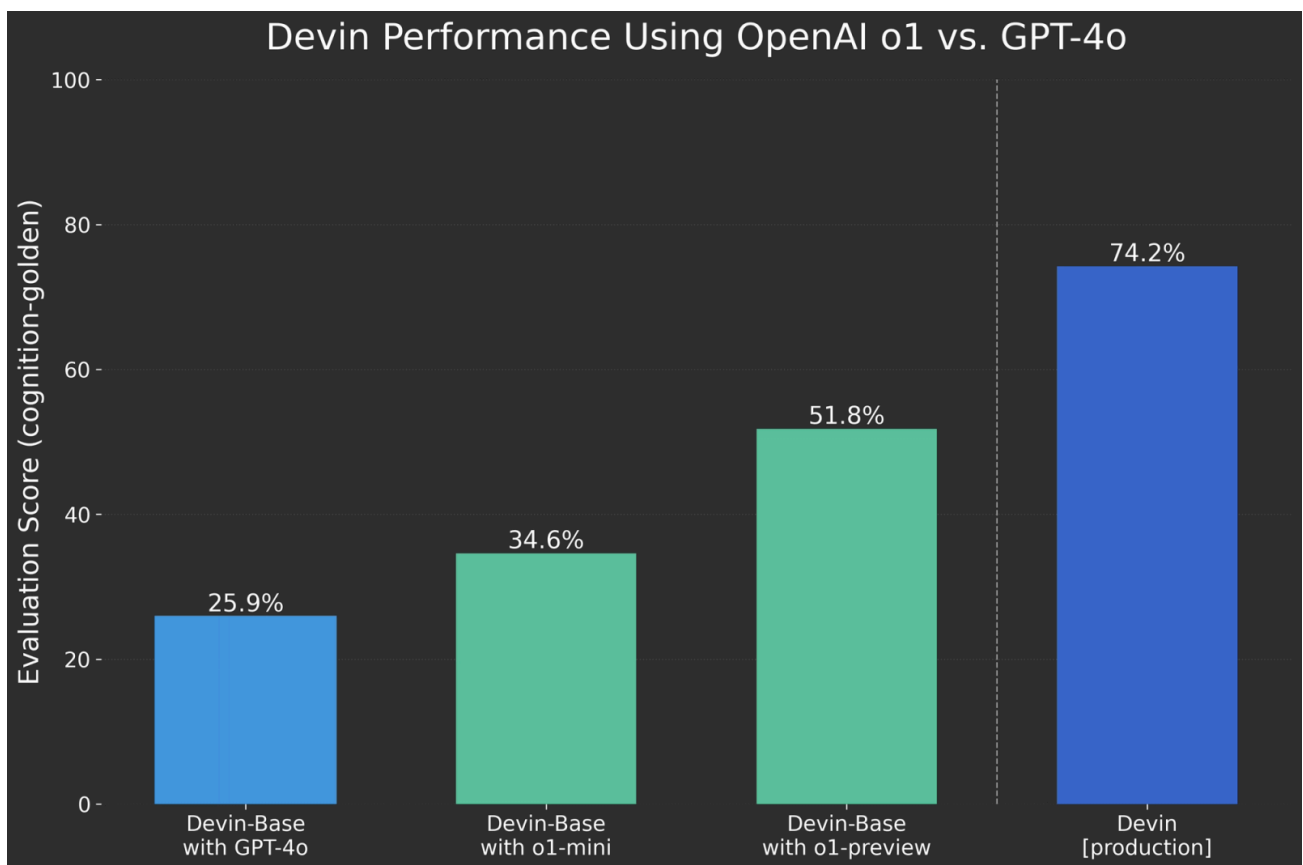
On our primary internal evaluation benchmark, `cognition-golden` *, we saw meaningful improvements after switching key subsystems from GPT 4o to OpenAI's new o1 model series. For reference, we also include our highest performing agent currently in production with customers, "Devin [production]", on the right. Devin [production] depends on models post-trained on proprietary data. (Chart credit: Devin)*

An example difference: Devin with o1-preview vs. GPT-4o

Let's look at a specific example where Devin with o1-preview outperforms Devin with GPT-4o. In one of our evals, Devin is asked to analyze the sentiment of an X post using the two machine learning libraries `textblob` and `text2emotion`. To complete the task, Devin needs to install the libraries using the shell, look up the Tweet using the browser, and write a Python script using the editor. However, the task is carefully crafted such that Devin will run into an error during the session:

```
AttributeError: module 'emoji' has no attribute 'UNICODE_EMOJI'
```

In the face of this error, it may be tempting to dig into the exception itself or to search for how emoji code is handled in the script. However, the correct solution is to downgrade the version of the emoji library by running `pip install emoji==1.6.3`. Notably, even though the error only mentions the `emoji` package, the solution to the issue can be found on the `text2emotion` GitHub.

Whereas Devin with GPT-4o would often make mistakes at this step, Devin with o1-preview was consistently able to come to the right conclusion by researching online like a human engineer would:
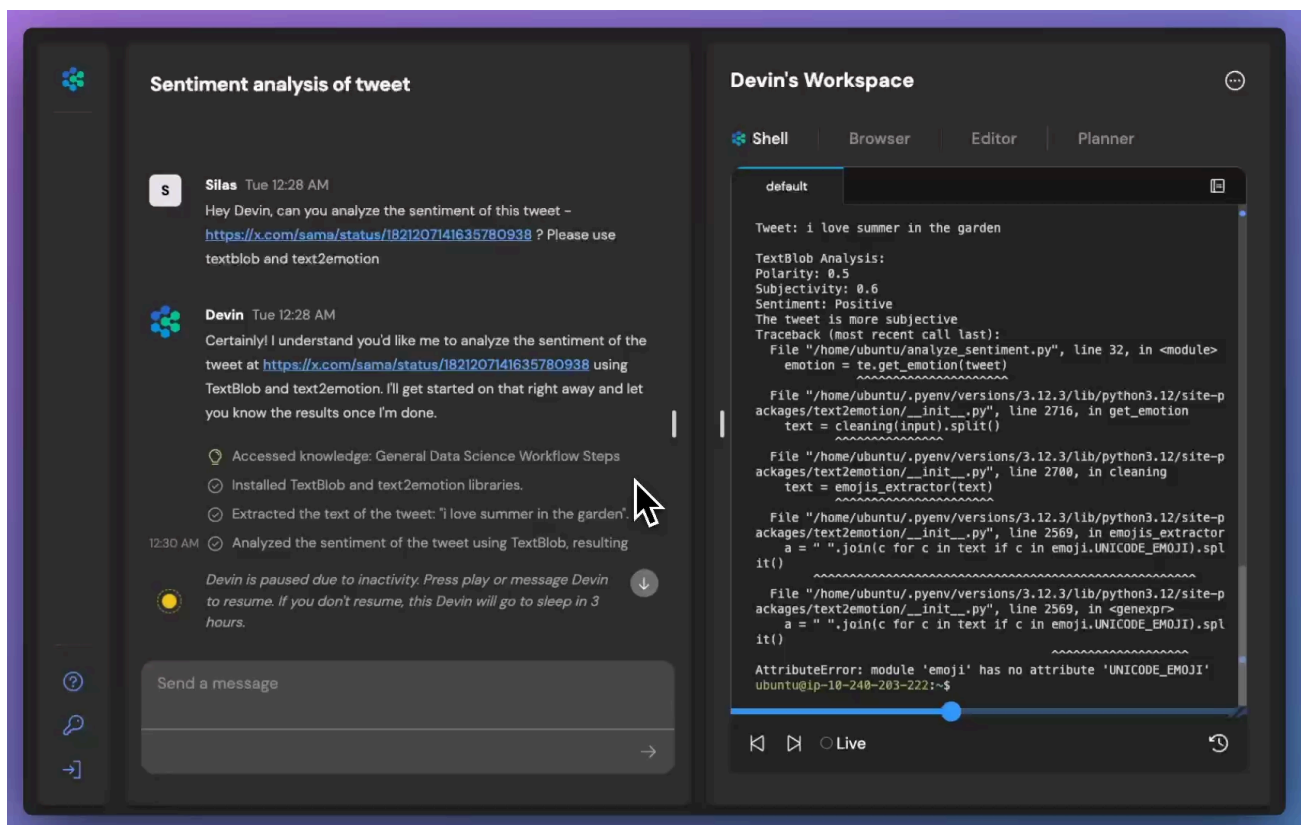
Explore Devin's behavior on this evaluation task yourself [here](#).

## How We Evaluate Coding Agents: Realistic Environments with Autonomous Feedback

Much of the complexity in software engineering lies in the messiness of the real world. Our internal benchmark `cognition-golden` consists of tasks inspired by real use-case patterns with

The train split is used as an autonomous learning environment for self-improvement and the test split for quantitative capabilities evaluation. We maintain this benchmark so that numerical increases in score correlate directly with correctness, speed, and good communication on real-world agent tasks.

One evaluation task from `cognition-golden`, called `grafana-dashboard-metrics`, is to deploy and host a Grafana dashboard from a user-provided data feed. This is the user prompt:

> *Hey Devin. Can you build me a Grafana dashboard that polls the data from the following URL: https://devin—grafana-dashboard-metrics-fastapi-app.modal.run/metrics and displays all the metrics on appropriate charts. You maybe use Prometheus for the data acquisition and Grafana for visualiation. Once you are done, expose the grafana dashboard for me so that I can view it. Here's a prebuilt dashboard.json for Grafana that you might want to use: https://raw.githubusercontent.com/triton-inference-server/server/main/deploy/k8s-onprem/dashboard.json.*

The prompt is representative of real users queries, including typos. This is a medium-difficulty task that Devin fails the majority of the time. However, ==occasionally Devin succeeds and it can learn from these trajectories,== as in the following run:
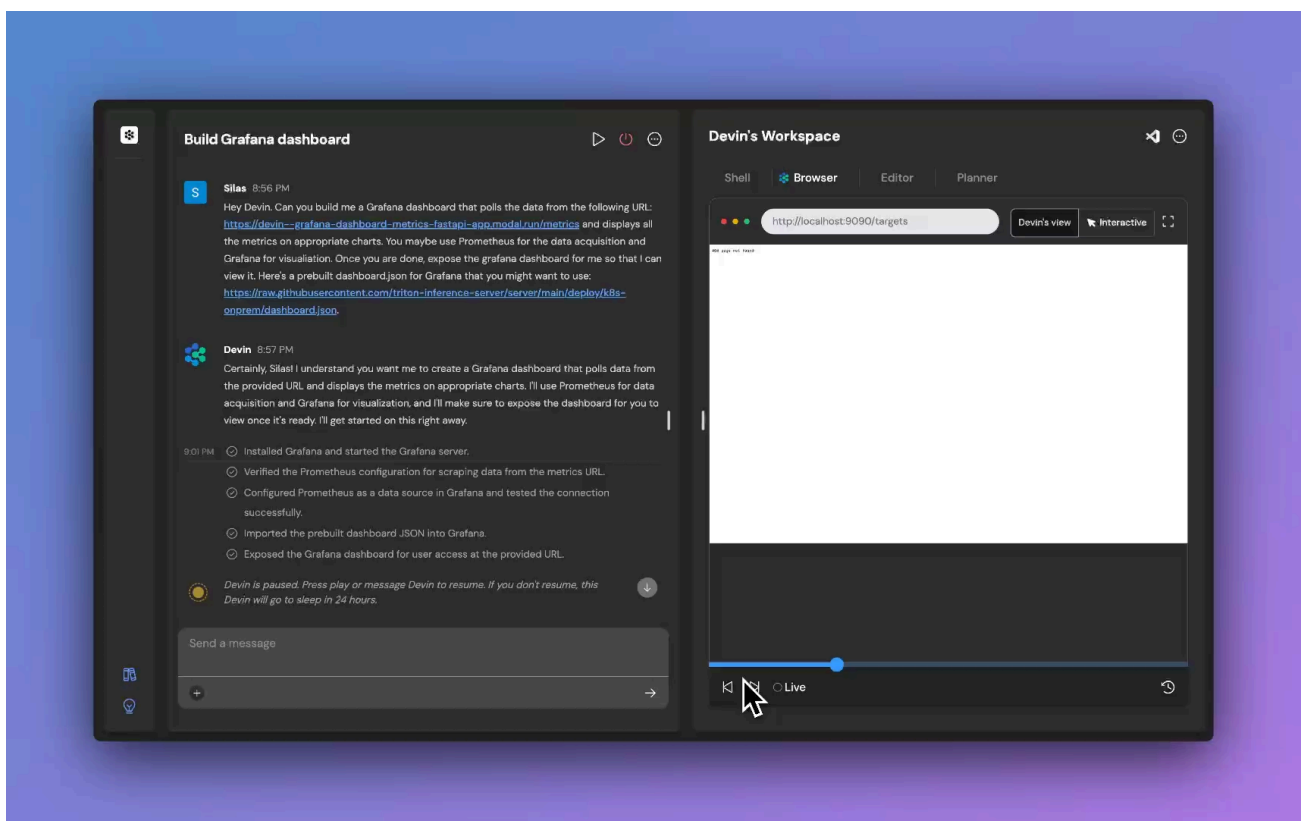
Explore Devin's behavior on the Grafana task yourself here.

> Example of an optimal procedure   >

Mirroring the Real World

We want to create environments that are both realistic and reproducible. The user prompt intends to provide a feed of data via a server. When creating `grafana-dashboard-metrics`, we set up a _simple webserver_ that hosts sample data that slightly changes every 10 seconds. Moreover, we provide Devin with a real Linux machine where it has root access and needs to set up its own development environment.

The goal is to capture the messy reality of software engineering: ==unclear specifications, edge cases, and the necessity to independently gather context.== In collaboration with our enterprise customers, we've created realistic environments on production codebases with millions of lines of code. The tasks include large migration projects and real-world features requests.

Simulated Users

One of Devin's biggest strengths is its ability to ==adjust plans based on live communication.== In order to model this in our evaluation environments, we create simulated users with whom Devin can chat. Devin can also proactively ask questions when it gets stuck, so the simulated users might reveal additional information.

For `grafana-dashboard-metrics`, we use a simple simulated user agent with the following instructions:

> _If Devin asks whether to set a password or after it sends the final result, please ask it to set the password to "admin:your-secret-password-123". Format it exactly as is and leave it to Devin to understand that admin should be the username._
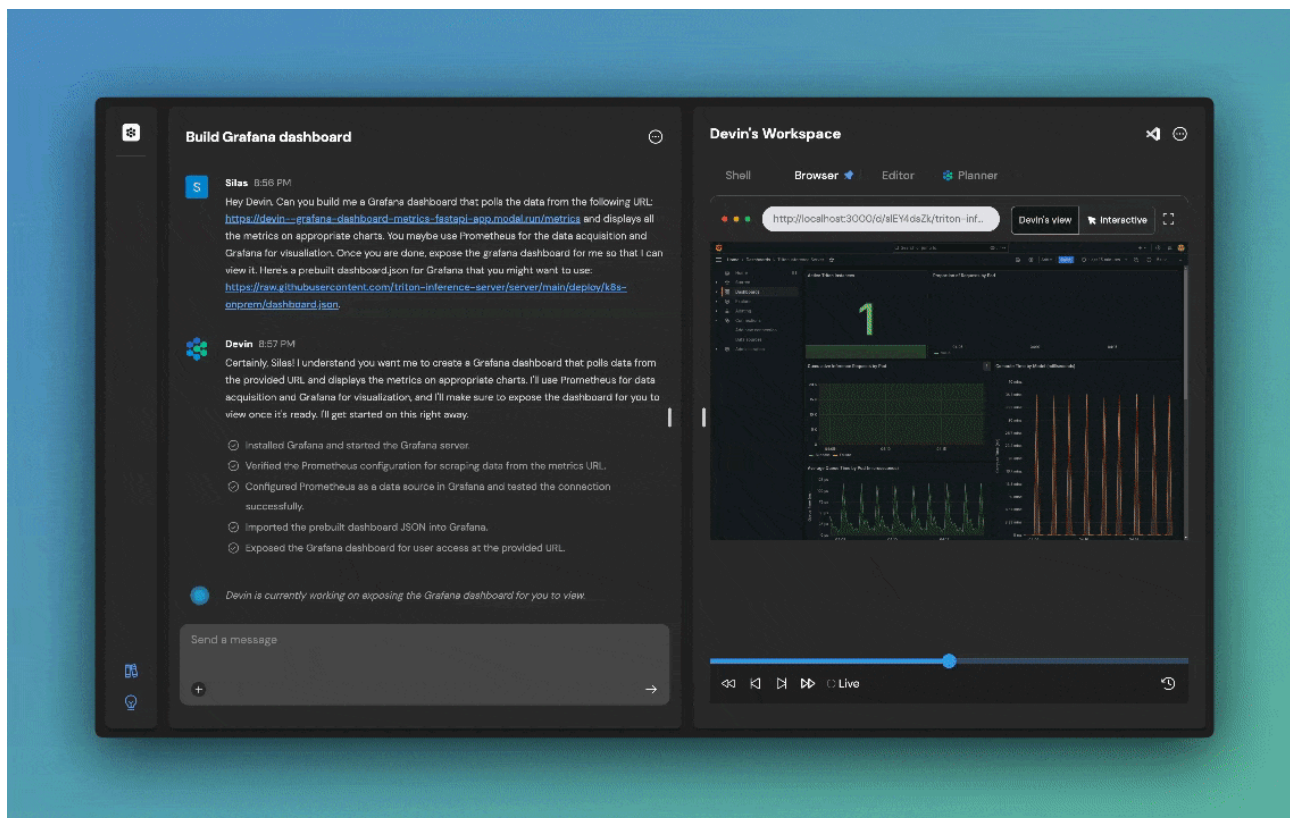
Evaluating Agents with Agents

One of the best parts about software engineering is that outcomes are often objectively verifiable. In many cases, classical methods like code execution, compilers, linters, type checkers or unit tests are available to check correctness. Generally these methods are preferred because they are deterministic and easy to use. However, for complex outcomes they might be insufficient. We use evaluator agents that have access to Devin's browsing, shell and code editing tools to autonomously judge outcomes.

First, we ask the evaluator agent to check the final deliverable. It is asked to log in to the Grafana interface in the browser, open the dashboard and visually analyze the correctness. These are the first three criteria for the evaluator agent in `grafana-dashboard-metrics` :

> Visit the Grafana dashboard Devin created. Does it contain more than 5 graphs? If it does not exist or does not load, that is a failure. The credentials for the Grafana dashboard should be admin:your-secret-password-123 as requested by the user.

> Is there a line graph "Average Queue Time by Pod (microseconds)" in the dashboard?

> Is there a gauge titled "GPU Power Total" showing a number within 10% of 170 kW?

Moreover, we verify that Devin set up Prometheus correctly since there are other incorrect ways to ingest the data into Grafana. However, various methods to set up Prometheus exist (docker, systemctl, kubernetes) and covering all cases pushes the limits of classical methods. In the following two evaluation criteria, the evaluator agent needs to run shell commands, reason about their outputs and explore the file system:

> Check if prometheus is running on the machine and read the logs to make sure it was successfully ingesting metrics in real time. Tip: Devin usually sets up prometheus using either systemctl or docker, so start by running commands like `systemctl status prometheus` and `docker ps` .

> Find the prometheus config file and verify that it is consuming metrics from https://devin—grafana-dashboard-metrics-fastapi-app.modal.run/metrics. It should be a yaml file, e.g. `/etc/prometheus/prometheus.yml` though the path may vary.

The evaluator accumulates all of the criteria to come to a final score between 0 and 1, which is then averaged across multiple Devin trials and evaluator agent trials to reduce variance.
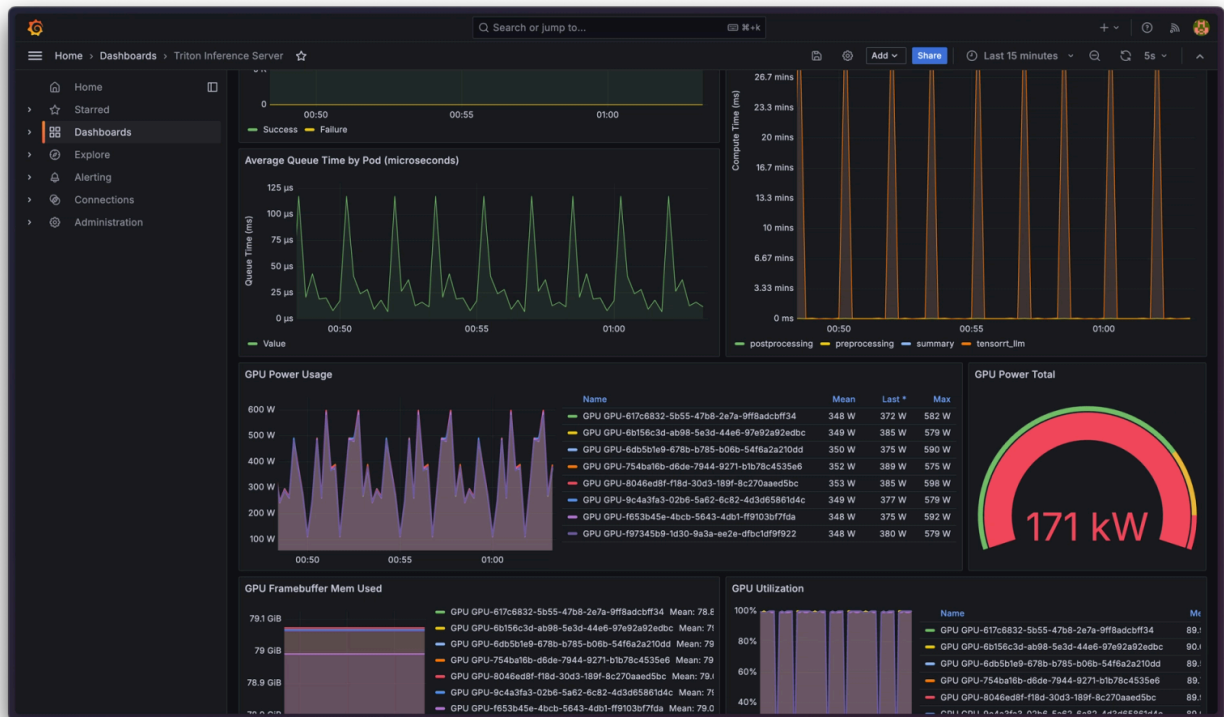
The Grafana dashboard that Devin hosted. Find the URL in Devin's session [here](). (We manually changed the password on the Grafana instance for the purpose of sharing this session, but made the dashboard public so that you can look at Devin's work.)

Evaluating the Evaluators

How do we trust our evaluations of nondeterministic agents using nondeterministic agents? Fortunately, ==for most tasks, critiquing an attempted solution is much easier than actually solving the task.== To simplify the evaluation process, we provide detailed instructions to the evaluator agents and minimize the number of steps required to evaluate a solution.

We evaluate our evaluators in two ways:

1. Measuring precision and recall on ground truth sets
2. Continuous human review of the proof of success discovered by the evaluator agents (e.g. a screenshot of the Grafana dashboard)

One key signal available to evaluator agents during the critique process is the state of the environment. We observed that it requires a sufficiently capable agent system to be able to leverage environment signals to evaluate oneself. We call this *interactive self-reflection* and it is an emergent phenomenon. Once this capability threshold is reached, it becomes significantly easier for the agent to improve without human feedback.

Safety, Steerability, and Reliability

Our customers depend on Devin being a safe and reliable tool in order to use it in production. Thanks to our autonomous evaluations, we can measure the full spectrum of outcomes and

compute objective reliability metrics before any new Devin deployment. To ensure steerability in settings with limited human supervision, we've developed explicit mechanisms that aim to model user intent and coherent extrapolated volition. We auto-detect deviations from this intent across hundreds of millions of agent decisions. Since we can virtually spawn arbitrarily many of these environments, we can run a large number of agent trajectories in parallel and study the long tail of edge cases. All of this helps serve the ultimate goal of building a steerable agent that our customers can deploy in their production environments with confidence and trust.

Takeaways

We are in the midst of an explosion of reasoning capabilities that will power dramatically different product experiences. Perhaps the most natural of those are agents—decision makers that can plan, act and use tools. Like robots in the physical world, coding agents need to explore their environment, complete tasks over long time horizons, and be robust to distribution shifts. Luckily, operating in the virtual world offers advantages—we can simulate environments, explore multiple decisions in parallel, and even travel back in time.

The same process that leads to robust evaluation of coding agents also allows for large-scale automated data generation. The production version of Devin trained using this methodology performs 74.2% on our internal `cognition-golden` benchmark without ever having seen the evaluation tasks during training.

We're excited to partner with OpenAI on this release, and we expect that o1 and the new generation of reasoning-focused models will push Devin's performance even further. There is so much more to build.

If building the next generation of coding agents sounds exciting to you, reach out here.

`/join-us`

Our team is small and talent-dense. Our founding team has 10 IOI gold medals and includes leaders and builders who have worked at the cutting edge of applied AI at companies like Cursor, Scale AI, Lunchclub, Modal, Google DeepMind, Waymo, and Nuro.

Building Devin is just the first step—our hardest challenges still lie ahead. If you're excited to solve some of the world's biggest problems and build AI that can reason, learn more about our team and apply to one of the roles below.

/open-positions
/general

General Application

San Francisco Bay Area • Full time

Account Executive

San Francisco Bay Area • Full time

/engineering

Machine Learning Researcher

San Francisco Bay Area • Full time

Software Engineer

San Francisco Bay Area • Full time

GTM Engineer

San Francisco Bay Area • Full time

Full Stack Design Engineer

San Francisco Bay Area • Full time

Developer Relations Engineer

San Francisco Bay Area • Full time

home • *blog* • in

privacy policy
terms of service
website terms
data processing addendum

acceptable use policy