Antigravity Handbook

Flavio Copes
flaviocopes.com

November 2026

Contents

What Antigravity does . . . 3
Install Antigravity and sign in . . . 4
A tour of the window . . . 5
Create a project . . . 6
Plan the first task with /plan . . . 8
Watch the agent work . . . 11
Let the browser agent test the page . . . 12
Review the changes . . . 15
The side panel . . . 16
Models: Gemini, Orchestrator and the others . . . 17
Permissions and the sandbox . . . 21
Write project rules . . . 23
Teach a workflow with a skill . . . 24
Customizations and the Marketplace . . . 26
Where customizations live . . . 28
Connect external tools with MCP . . . 29
Hooks . . . 31
Subagents and custom agents . . . 32
Run several agents at once . . . 33
Slash commands . . . 34
@ mentions and context . . . 36
Keep working until it's done with /goal . . . 37
Automations . . . 37
A tour of Settings . . . 39
Keyboard shortcuts . . . 40
The Antigravity CLI . . . 41
How I use Antigravity . . . 42

Preface

This is the offline handbook edition of Antigravity Handbook, written by Flavio Copes.

You can read the online version with interactive diagrams and videos at https://flaviocopes.com/antigravity/, and find more programming tutorials, courses, and handbooks at https://flaviocopes.com.

Antigravity is Google's agentic coding tool. You describe what you want, and an agent powered by Gemini (or Claude, if you prefer) reads your files, writes code, runs commands, opens your app in a browser to test it, and reports back.

It comes in three forms, and the names can be confusing:

- the **Antigravity app**, a desktop app built around conversations with agents. There's no code editor in it
- the **Antigravity IDE**, a VS Code-based editor with the agent in a side panel
- the **`agy` CLI**, the same agent in your terminal

This guide is about the app. It's the version Google calls Antigravity 2.0, and it's where Google is putting most of its effort: the IDE was the original product, then the agent manager inside it became a standalone app. I'll mention the IDE only where it matters, and the CLI near the end.

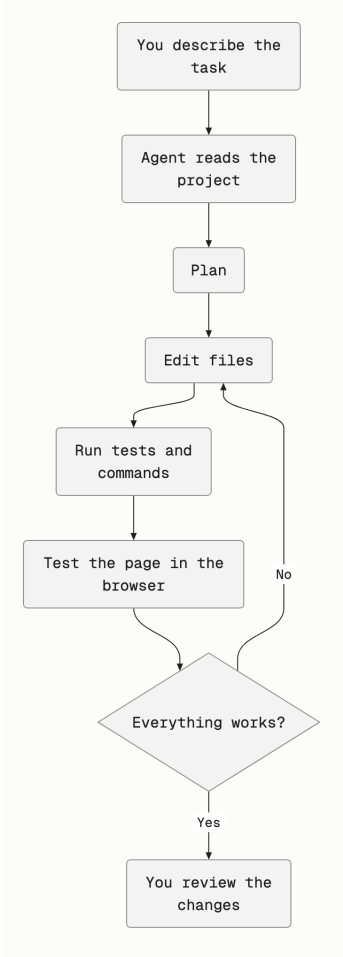
To have something real to work on, we'll ask the agent to build a small web app from an empty folder: a running pace calculator. You enter a distance and a finish time, and it shows your pace and predicts your times for other race distances. Then we'll use that project to try everything else: the browser agent, rules, skills, worktrees, Automations and more.

Everything here comes from Antigravity 2.35.1 on macOS, with a Google AI Ultra plan.

What Antigravity does

A model on its own can only answer with text. Antigravity wraps the model in an **agent**: a loop that gives it tools to read and edit files, run terminal commands, search the web, and drive a browser.

You ask for a change. The agent looks at the project, decides what to do, does it, checks the result, and keeps going until it's done or it needs you.



The project and the commands stay on your computer. The model doesn't: Antigravity sends your prompts and the relevant code to Google's servers (or Anthropic's, if you pick a Claude model) to process them.

What makes Antigravity different from a chat window is that you mostly don't watch it type. You start a task, check on it, approve what needs approving, and run several conversations at the same time, each in its own project.

If you're comparing it with the tools you already use, I wrote similar guides for Claude Code and Cursor.

Install Antigravity and sign in

Download the app from antigravity.google/download. It runs on macOS 13 or later (Apple Silicon and Intel), Windows 10 and 11 (x64 and ARM64) and Linux (Ubuntu 20+, Debian 10+, Fedora 36+, RHEL 8+).

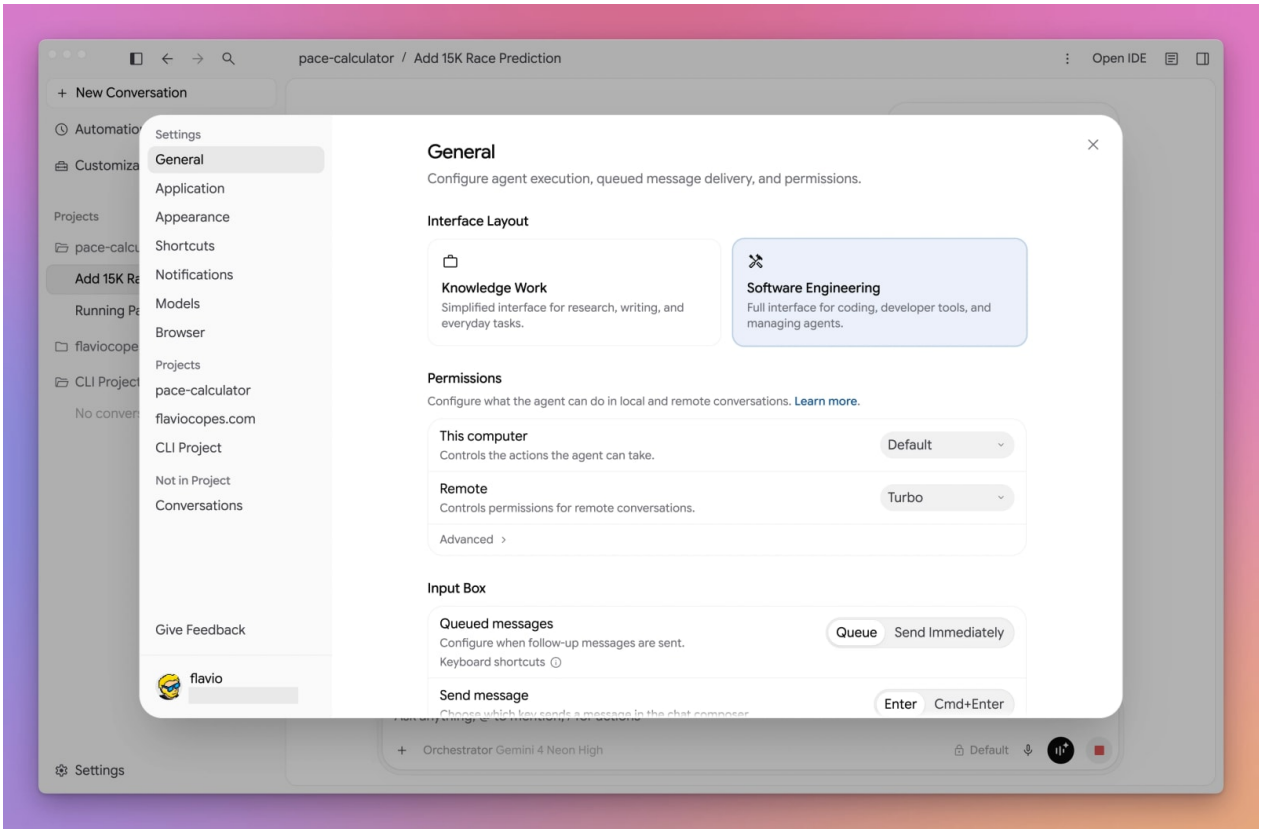
On a Mac, open `Antigravity.dmg` and drag the app to Applications. When you update by hand, pick **Replace**, not "Keep Both", or you'll end up with two copies.

The first launch opens your browser to sign in with your Google account. Then Antigravity asks what kind of work you do: Software Engineering, Knowledge Work, Data & Analytics, Science, Design or Other.

That choice sets the **interface layout**:

- **Software Engineering** (also picked by Data & Analytics, and by Skip) shows Git worktrees, branch pickers, inline diffs, terminals and subagent cards.
- **Knowledge Work** (picked by the other roles) hides Git, the terminal and the code diffs, for people who use the agent for research and writing.

Both layouts run the same agent, models and customizations. You can switch any time in **Settings > General > Interface Layout**. This guide uses Software Engineering.

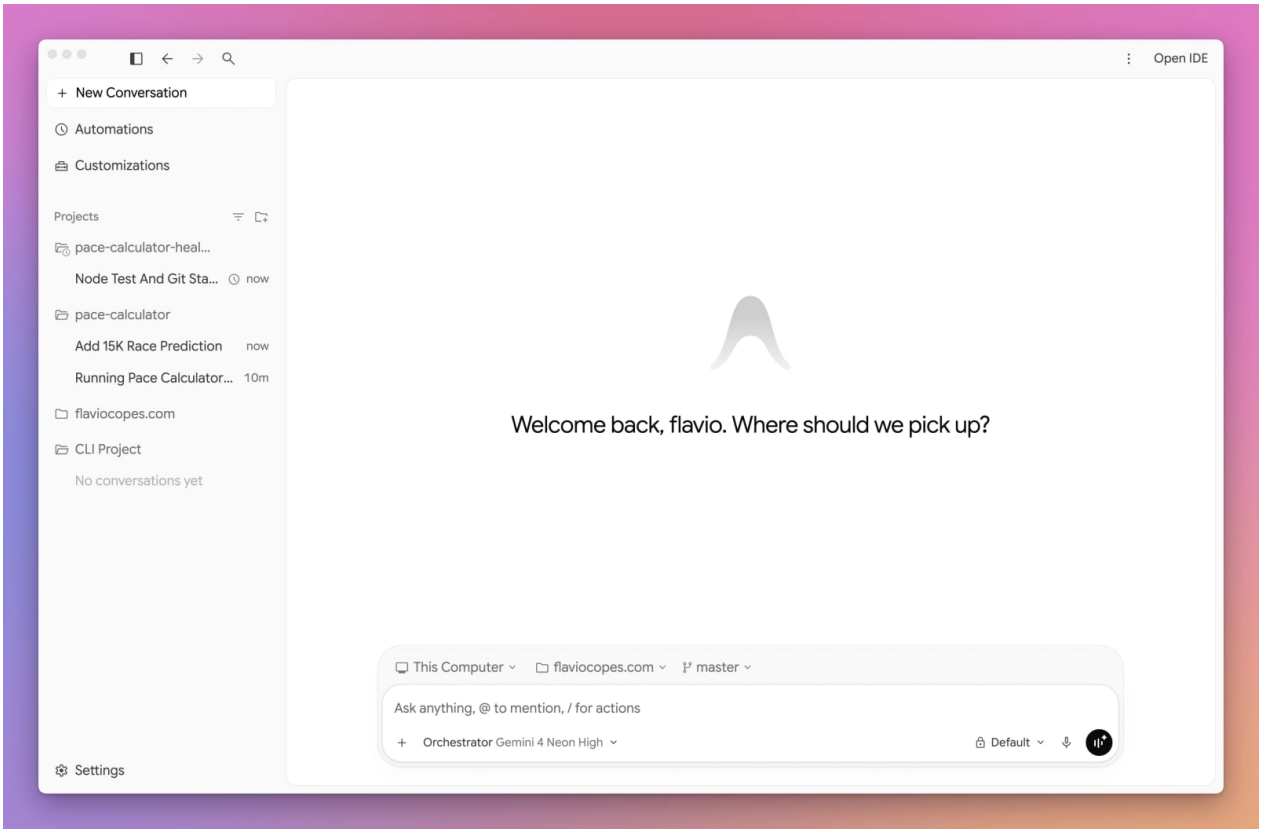


If you already use Claude Code, Codex, Cursor or Gemini CLI, the onboarding offers to **bring over your setup**: it imports your MCP servers, skills, rules and instructions.

Signing in once signs you in everywhere on that machine: the app, the `agy` CLI and the Antigravity IDE share the same credentials.

A tour of the window

After you sign in, Antigravity opens on a mostly empty window:



On the left, the **sidebar**:

- **New Conversation** starts a chat with the agent
- **Automations** lists the tasks that run on a schedule
- **Customizations** is where you manage plugins, skills, rules, custom agents and MCP servers
- **Projects** lists your projects, each with its conversations below it
- **Settings** is at the bottom

At the top, the arrows move back and forward between the pages you opened, like in a browser, and the magnifying glass searches your conversations. **Open IDE** opens the current project in the Antigravity IDE, if you have it installed.

At the bottom, the **message box**. Above it, three pickers decide where the agent works:

- the **environment**: This Computer, or a remote one
- the **project**
- the **checkout**: your local folder, or a new Git worktree (more on that later)

Below it, the + button to add context, the model picker, the permissions picker, a microphone for voice input, and the send button.

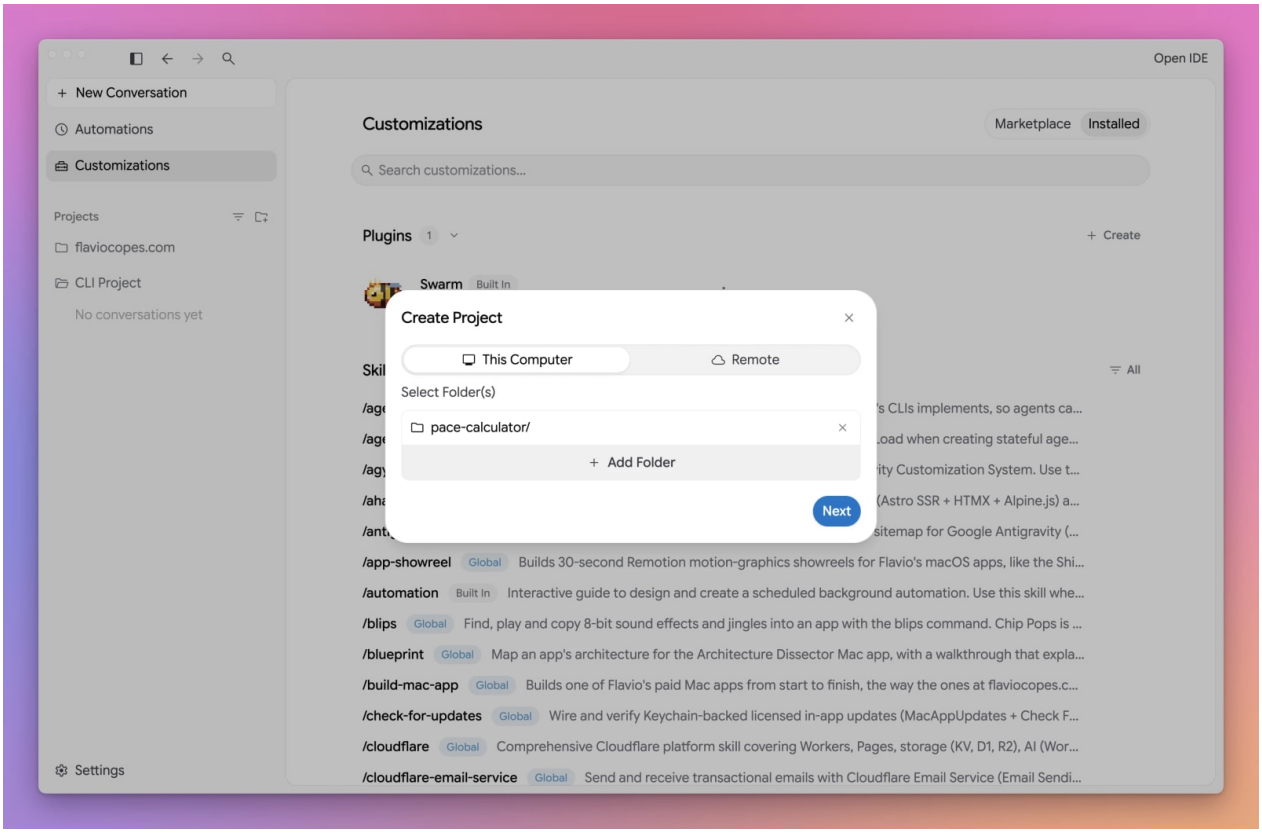
Create a project

Conversations live in **projects**. A project is one or more folders, with its own settings and permissions. You can also chat outside any project, in which case the agent works in a scratch folder.

I created an empty folder for the demo:

```
mkdir ~/dev/pace-calculator
```

Then I clicked the folder icon with the + next to **Projects** and picked **New Project**. The dialog has two tabs:

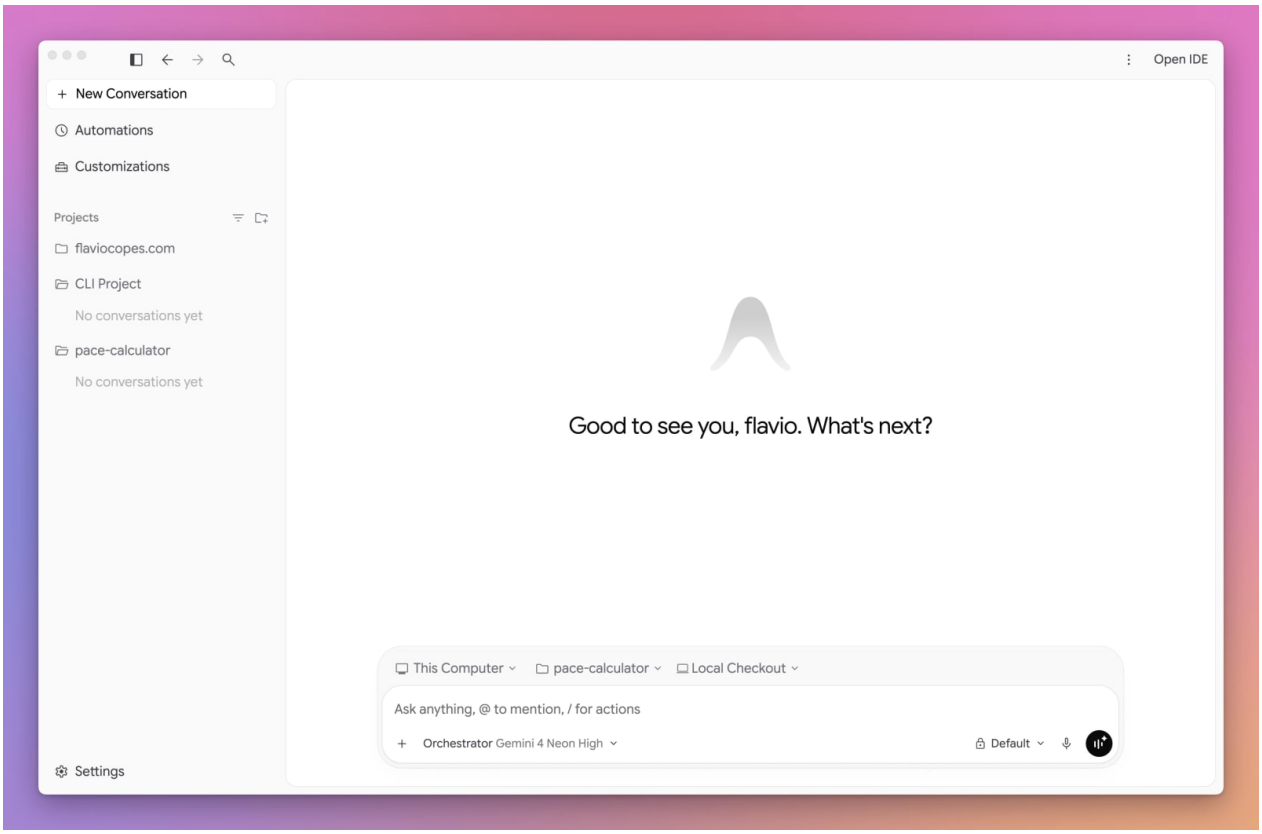


This Computer works on folders on your Mac. Click **Add Folder** to pick them. You can add more than one, which is handy when a frontend and a backend live in separate repositories.

Remote creates a project that runs in a Google-managed cloud environment. Its agents keep working when your laptop is asleep. Instead of folders, you add resources: **Add Resource** lets you upload files or connect Google Drive or GitHub.

Remote agents are rolling out to personal Google AI Pro and Ultra accounts. The same + menu also has **SSH Host...**, which opens a separate window where projects, agents and terminals all run on a Linux machine you connect to over SSH.

I picked `pace-calculator`, clicked **Next**, and the project appeared in the sidebar, ready for its first conversation:

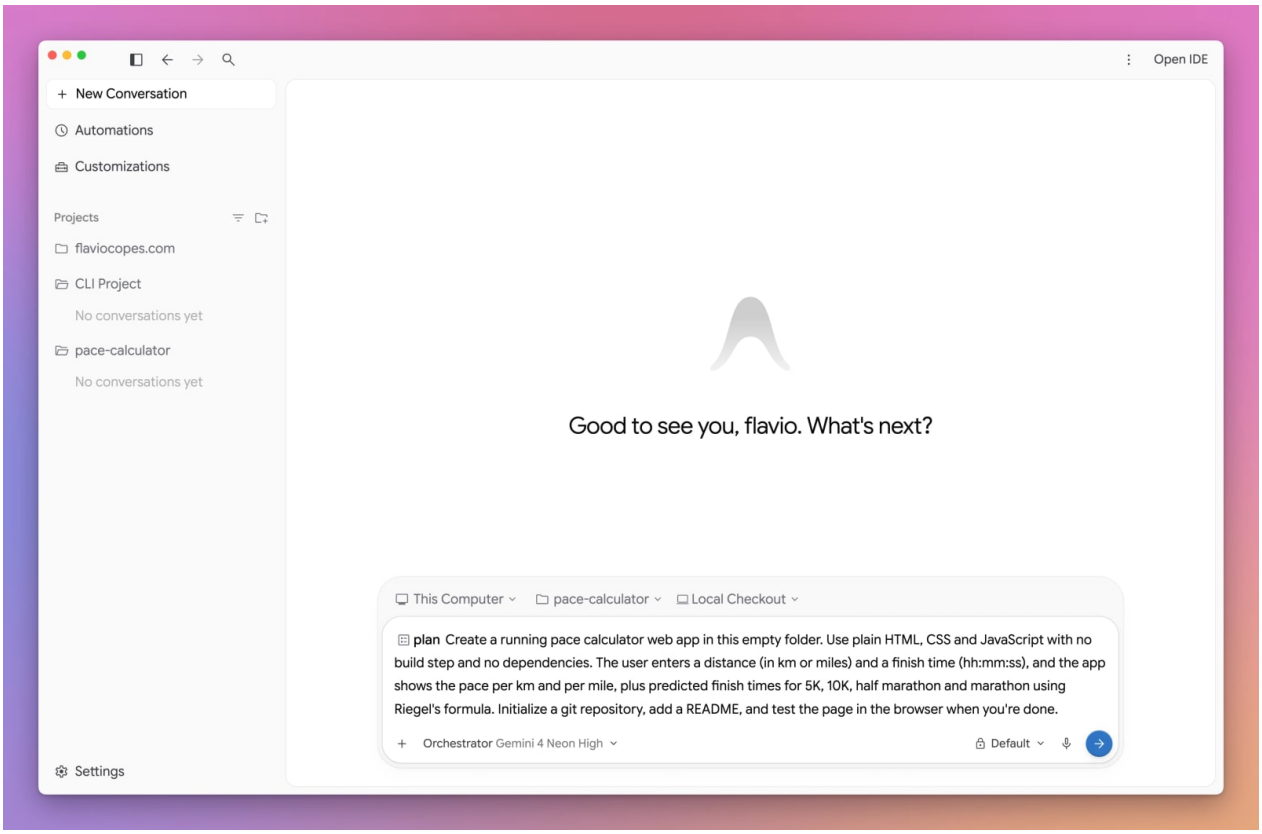


Plan the first task with /plan

You can just describe what you want and press Enter. For anything bigger than a small fix, I prefer to start with `/plan`: the agent explores without changing anything, asks questions if something is unclear, and writes a plan I approve before it touches a file.

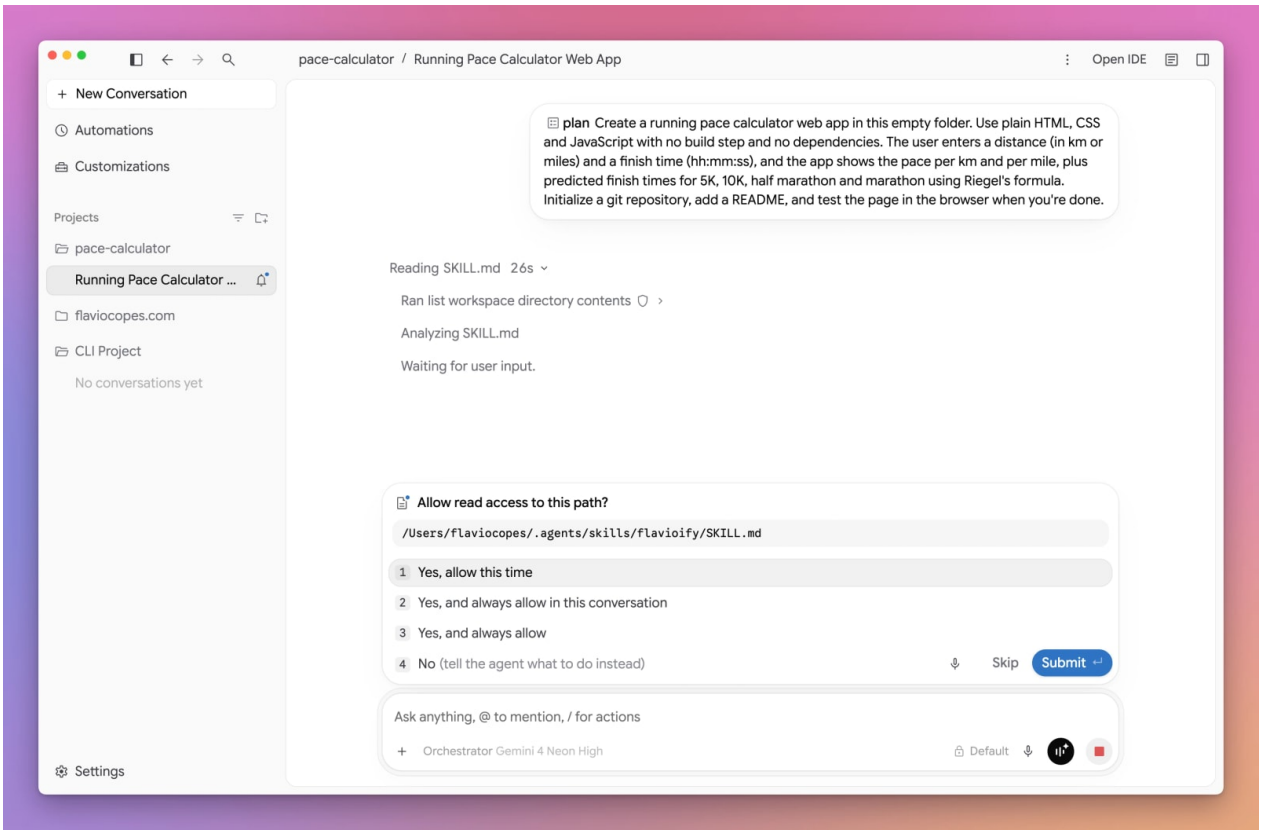
I typed `/plan`, pressed Tab to select the command, and wrote the task:

```
/plan Create a running pace calculator web app in this empty folder. Use plain HTML, CSS and JavaScript
    ↪  with no build step and no dependencies. The user enters a distance (in km or miles) and a
    ↪ finish time (hh:mm:ss), and the app shows the pace per km and per mile, plus predicted finish
    ↪ times for 5K, 10K, half marathon and marathon using Riegel's formula. Initialize a git
    ↪ repository, add a README, and test the page in the browser when you're done.
```

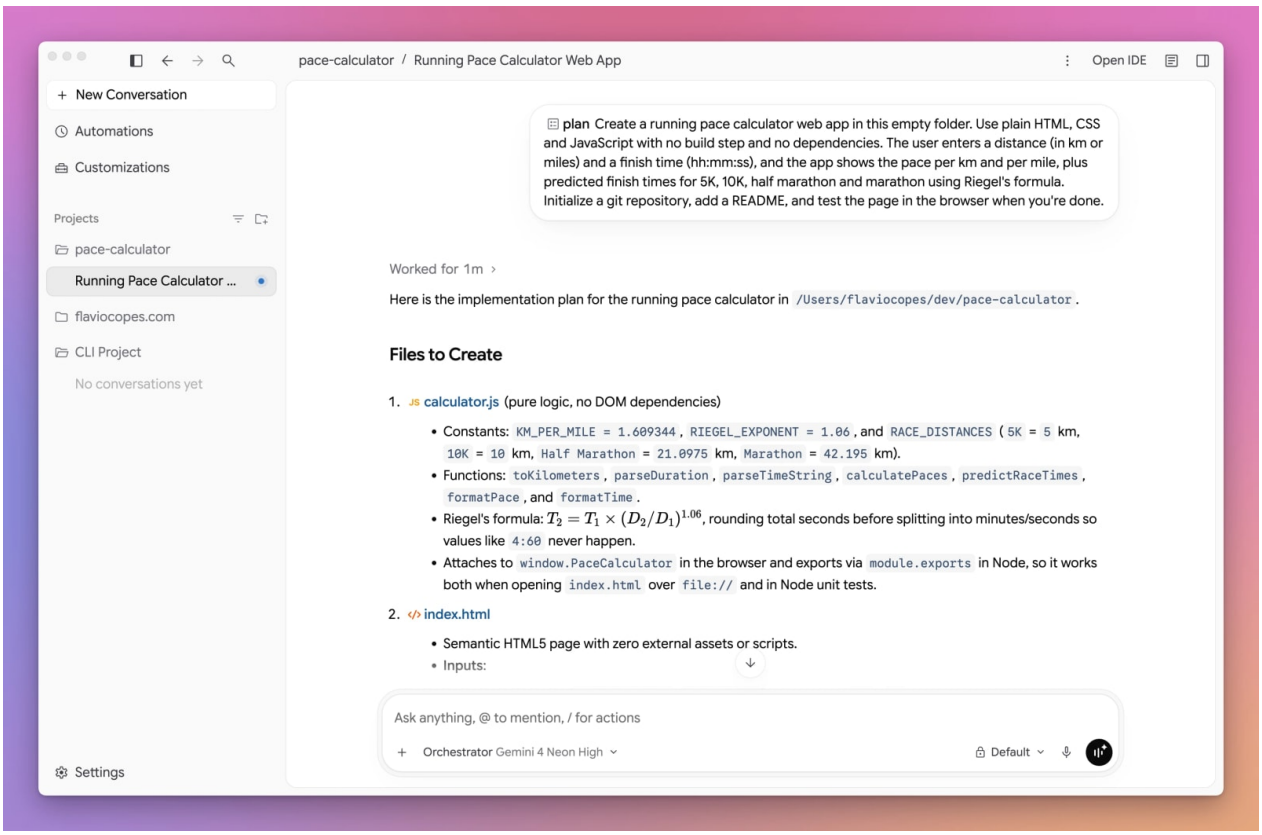


Notice the model under the message box: **Orchestrator** with Gemini 4 Neon High. I'll explain what that is in the models section. For now, it means a big model plans and a fast one does the work.

Before planning, the agent wanted to read a file outside the project, one of the skills I have installed in my home folder. That's a **permission prompt**: in the default permission preset, the agent asks before touching files outside the project.



I clicked "Yes, allow this time". About two minutes later the plan was ready. It listed every file it would create, with what goes in each one:



It had made decisions I hadn't asked for, and listed them so I could change them before any code existed: three separate fields for hours, minutes and seconds, a pre-

filled example (10 km in 50:00) so the page isn't empty, and classic `<script defer>` tags so `index.html` works when opened straight from disk. It ended with how it would verify the work, including the expected numbers, and asked:

> Does this plan look good, or would you like to change anything?

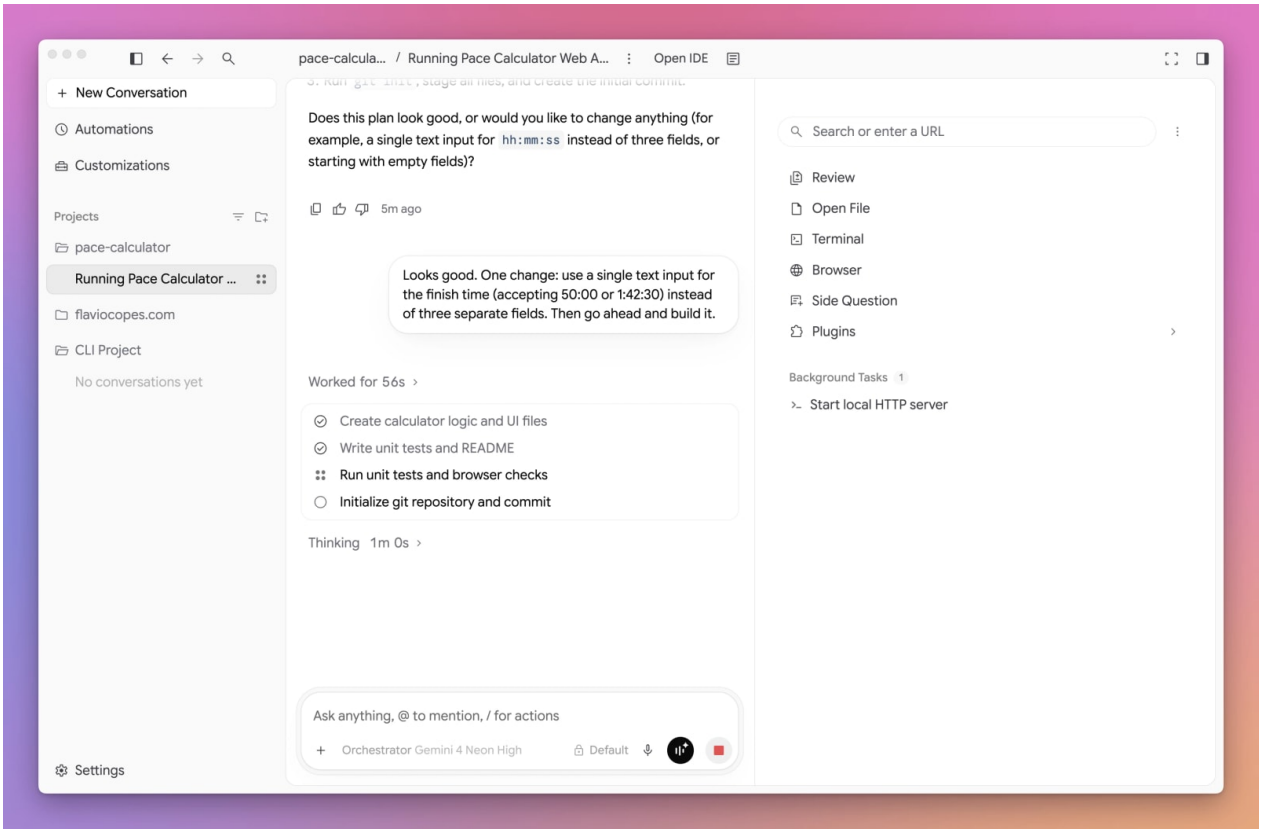
The three time fields sounded fiddly, so I answered in the message box:

```
Looks good. One change: use a single text input for the finish time (accepting 50:00 or 1:42:30)
    ↪ instead of three separate fields. Then go ahead and build it.
```

Replying is the simplest way to change a plan. When the plan shows up as an artifact in the side panel, you can also select a passage and leave a comment on it, and the comments ride along with your next message.

Watch the agent work

As soon as I sent my answer, the planner handed the plan to the executor, which wrote a checklist and started on it:



A few things to notice here:

- **The checklist** at the top of the turn shows where the agent is. It updates live, so you can glance at a conversation and know how far along it is.
- **The steps** below it ("Editing calculator.test.js") expand into the full details of each tool call: the command it ran, its output, the diff of each edit.
- **Background Tasks** in the side panel lists processes the agent left running. Here it started a local web server to test the page.

The agent wrote the six files in parallel, then ran the tests with `node --test`. One test failed:

```
failing test: calculates paces and Riegel predictions for 10 km in 00:50:00
  AssertionError [ERR_ASSERTION]: Expected values to be strictly equal:
  + '03:50:01'
  - '03:50:00'
```

The test expected a 3:50:00 marathon and the code computed 3:50:01. The agent changed the test, not the code.

That's the moment to pay attention, because ”fix the test so it passes” is a classic agent shortcut. So I did the math myself:

```
python3 -c "print(3000*(42.195/10)**1.06)"
```

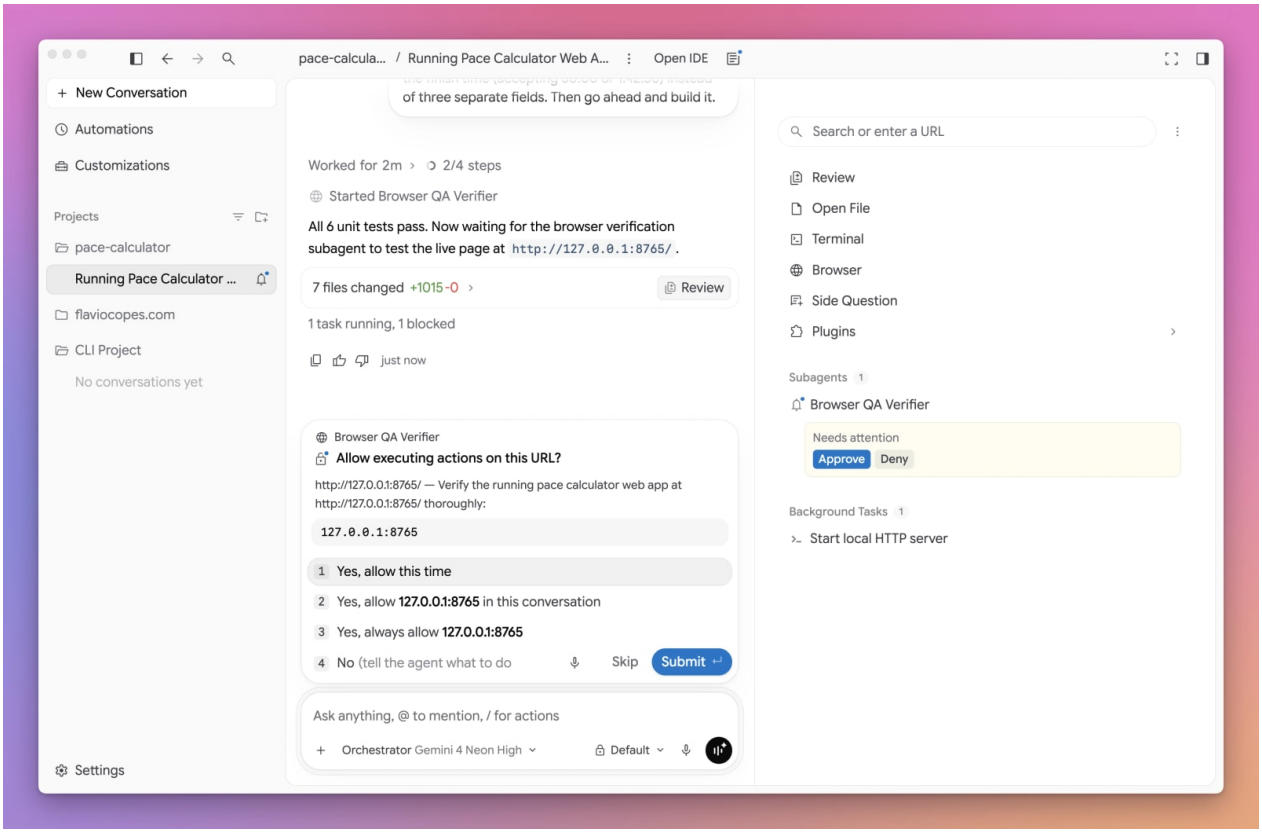
```
13800.597999769758
```

13,800.6 seconds rounds to 13,801, which is 3:50:01. The code was right and the hand-written expectation was off by a second, so the fix was correct. Later we'll add a project rule that makes the agent show its calculation before it touches a test.

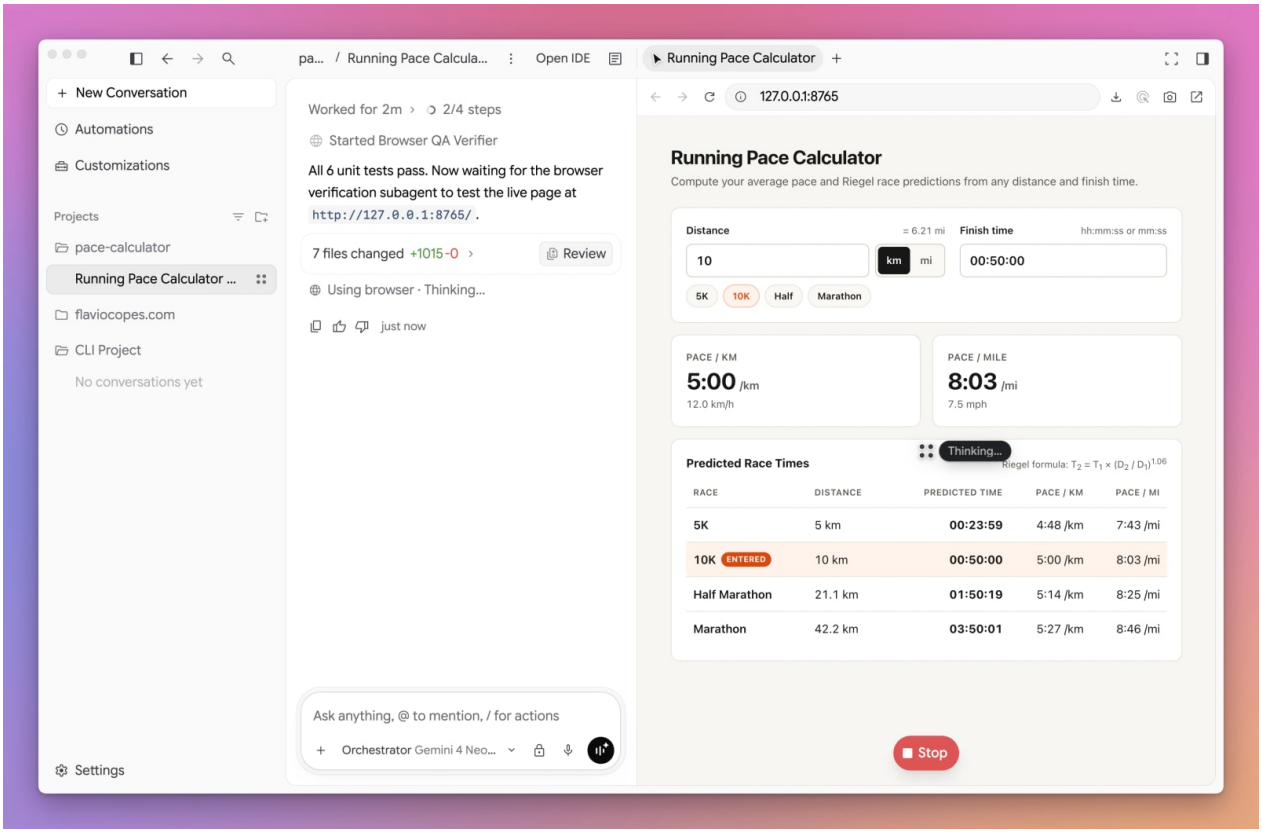
Let the browser agent test the page

Unit tests check the math. They don't check that the page works. For that, the agent started a **browser subagent**, a second agent with its own context whose only job is to use the page like a person would.

Antigravity has its own built-in browser, so there's nothing to install. The first time the subagent wanted to act on a URL, it asked:

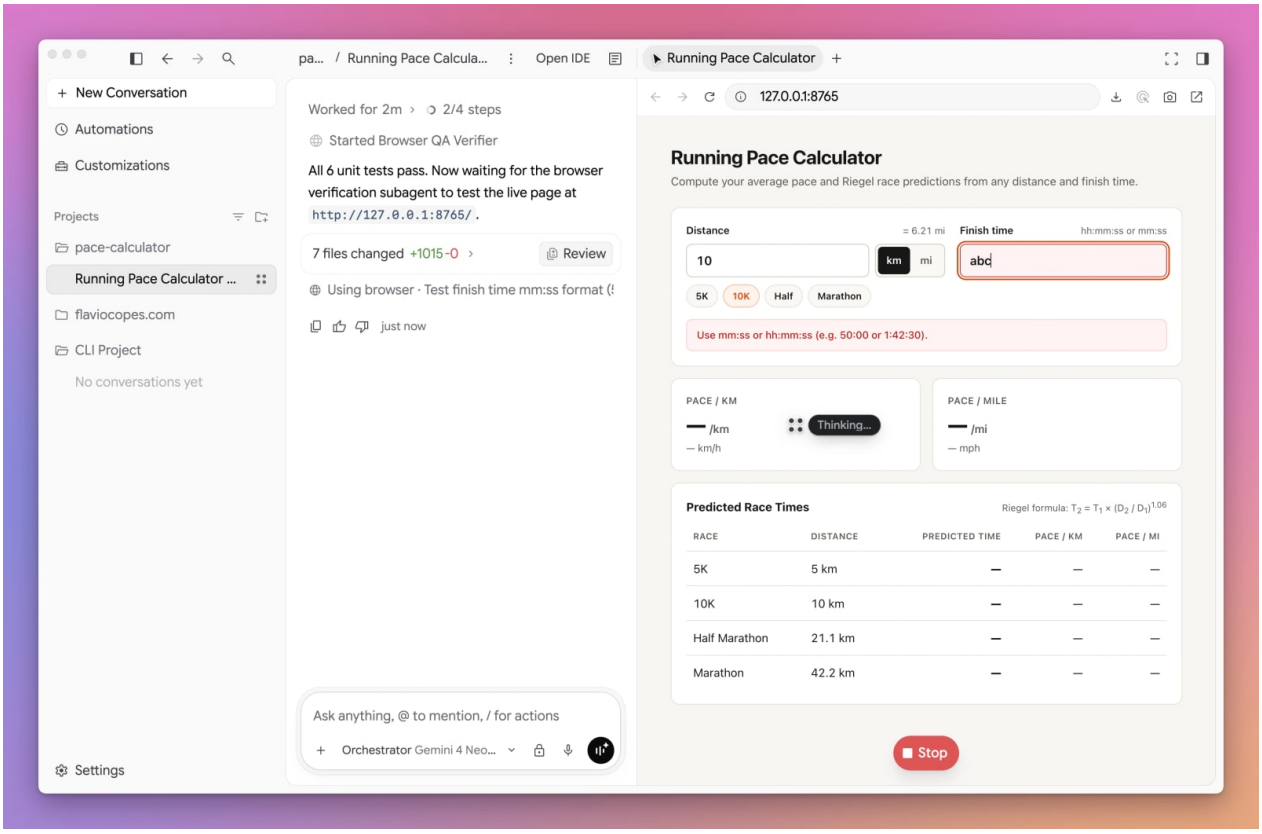


I picked "Yes, allow 127.0.0.1:8765 in this conversation". The page opened in the side panel, and I could watch the subagent work, with an animated cursor and a red **Stop** button to take over:



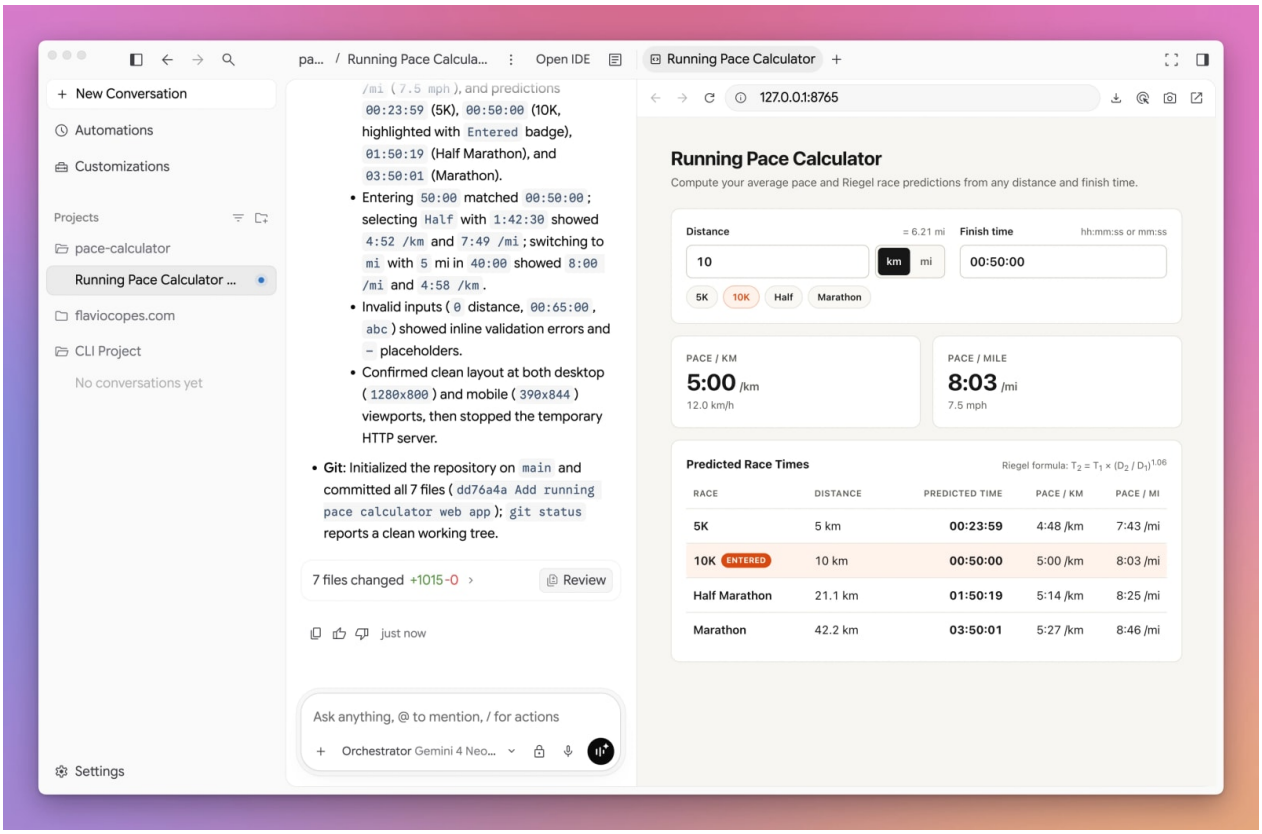
It entered times, switched between kilometers and miles, clicked the presets, and

tried to break the form. Here it typed `abc` into the finish time:



The browser runs in the background and never steals focus from what you're doing. It also has its own cookies, separate from Chrome, and when a site needs you to sign in, the agent stops and lets you do it yourself in that tab.

About a minute later, the agent wrapped up with a summary of what it built and what the browser checks found, including exact numbers I could verify:



The whole task took about six minutes: two to plan, then three and a half to build, test and commit, plus the time I took to read the plan.

I checked the repository myself:

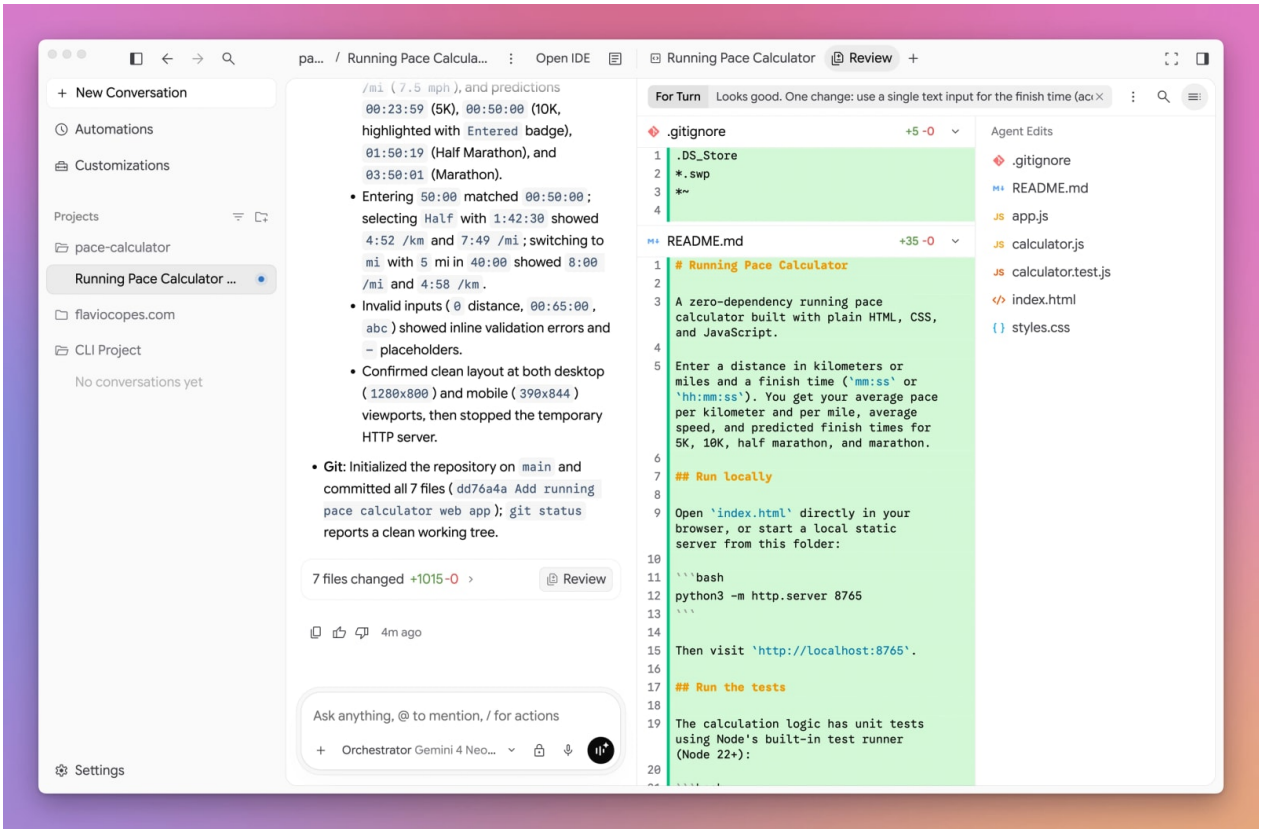
```
cd ~/dev/pace-calculator
git log --oneline
node --test
```

```
dd76a4a Add running pace calculator web app
...
# tests 6
# pass 6
# fail 0
```

Six files, about a thousand lines, one commit, and a page that works.

Review the changes

Under the final message there's a card that says **7 files changed**, with a **Review** button. It opens the **Review pane**, a Git view of everything the agent changed:



You can look at the diff split or stacked, hide whitespace changes, and switch between three views: **Uncommitted** changes, the whole **Branch**, or only the **Agent Edits** made in this conversation.

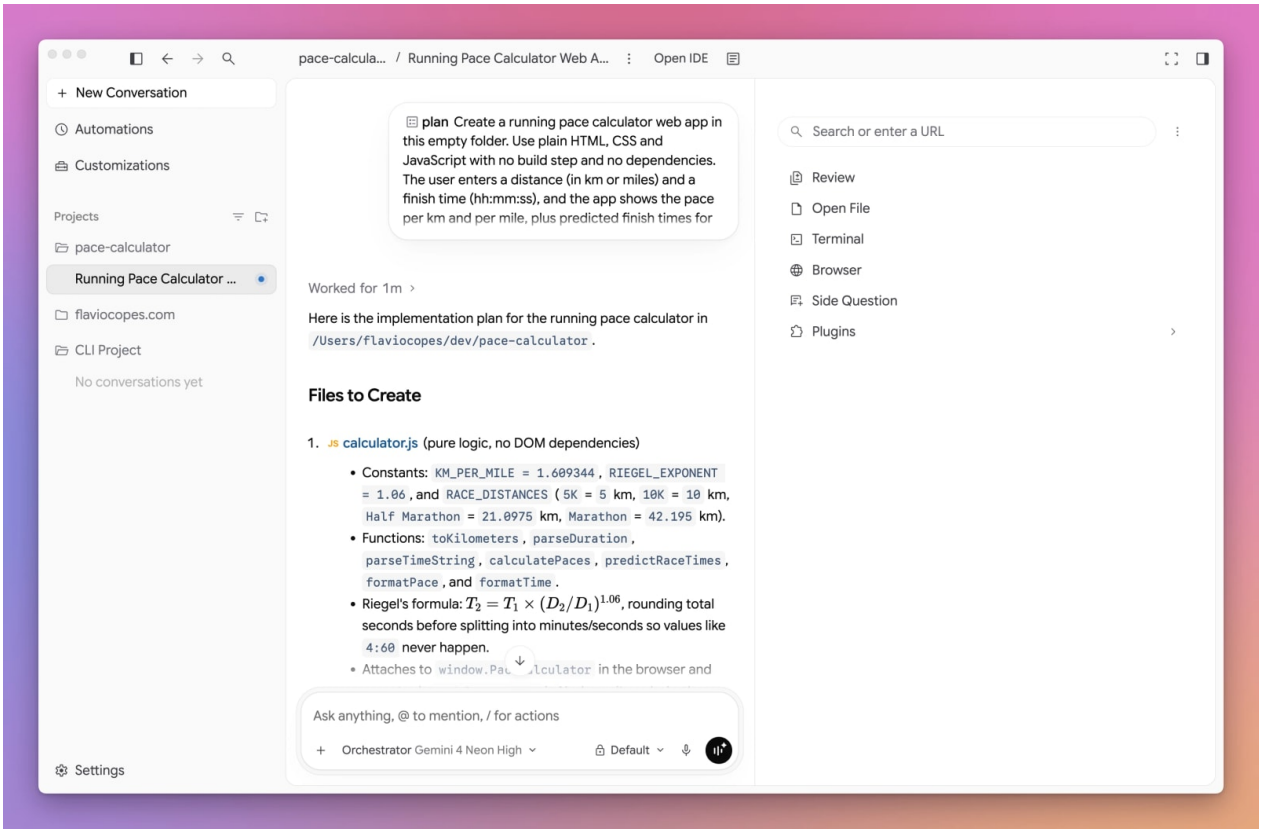
From here you can also:

- comment on a line, and the comment goes to the agent with your next message
- stage, unstage or discard files (careful: discarding a new file deletes it, without asking)
- commit, amend or push. Leave the commit message empty and the agent writes one

You can also just ask the agent to commit, as I did in the first prompt. The Review pane is for the times you want to read the change before it gets into a commit.

The side panel

The right side of the window is the **side panel**. Toggle it with Cmd+Shift+B, or the icon in the top right corner. When it's empty, it shows what it can open:

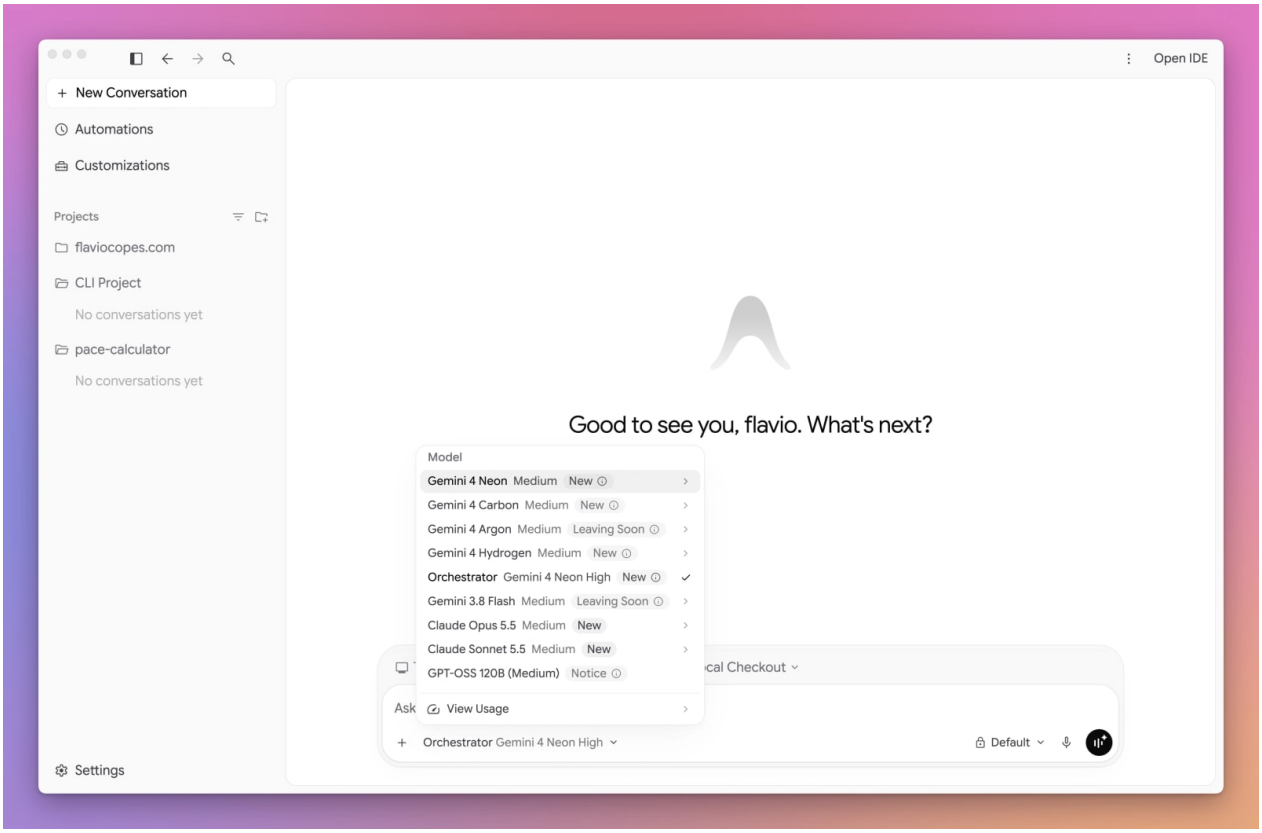


- **Review**: the Git view we just saw
- **Open File**: shows a file from the project, read-only, since this isn't an editor
- **Terminal**: a real terminal in the project folder. Select its output and press Cmd+L to quote it in the message box
- **Browser**: the built-in browser. Type a URL in the bar at the top
- **Side Question**: a separate thread for a quick question that shouldn't derail the main conversation
- **Plugins**: views added by your plugins

Each one opens as a tab, and you can open several (Cmd+T) and split them. The panel also lists the subagents, background tasks and artifacts of the current conversation. Press Cmd+Shift+M to maximize it when you need room.

Models: Gemini, Orchestrator and the others

Antigravity is a Google product, so Gemini is the default, but it isn't the only option. Click the model name under the message box to open the picker.



Here's what each one is for:

Model	What it's good at	Plans
Gemini 4 Neon	The strongest Gemini model, the default for new conversations on paid plans. Up to 512K tokens of context	Pro, Ultra
Gemini 4 Carbon	A big Gemini model for planning and harder tasks	Pro, Ultra
Gemini 4 Hydrogen	The fast, cheap Gemini model. The default on Free and Plus	All plans
Gemini 4 Argon	Marked ”Leaving Soon”	Pro, Ultra
Gemini 3.8 Flash	Marked ”Leaving Soon”	All plans
Orchestrator	Two models in one conversation: a planner and an executor	Pro, Ultra
Claude Opus 5.5 and **Claude Sonnet 5.5**	Anthropic's models, running inside Antigravity's agent	Paid Pro, Ultra
GPT-OSS 120B	OpenAI's open-weights model	Leaving November 30, 2026

Gemini 4 Argon, Gemini 3.8 Flash, GPT-OSS 120B and the older Claude 4.6 models are scheduled to leave on November 30, 2026, so don't build habits around them.

Each model has a **thinking level**: Low, Medium (the default) and High, plus Extra on Neon and Hydrogen. Higher levels think longer before acting, which costs more

quota. Hover a model in the picker to pick its level.

The model you pick sticks to the conversation. If you change it while the agent is working, the change applies from the next turn.

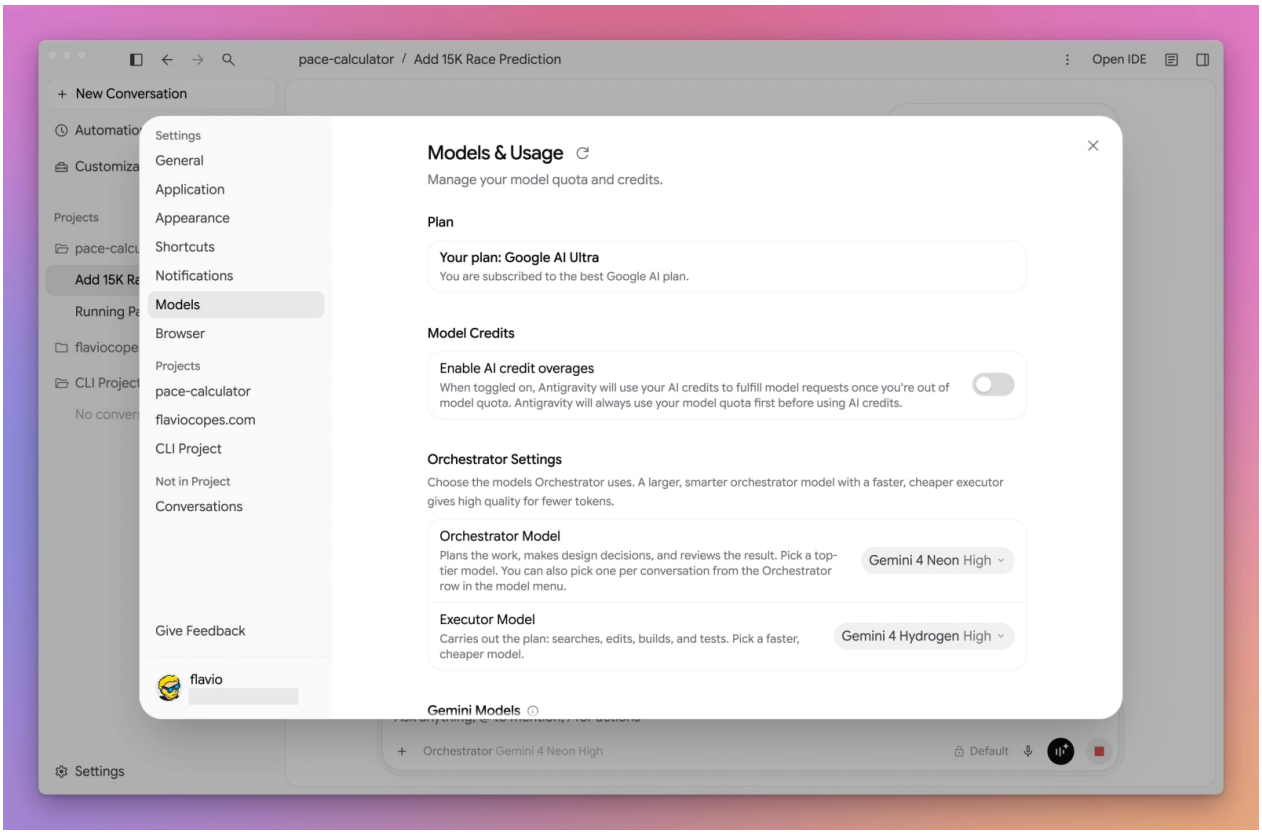
Orchestrator

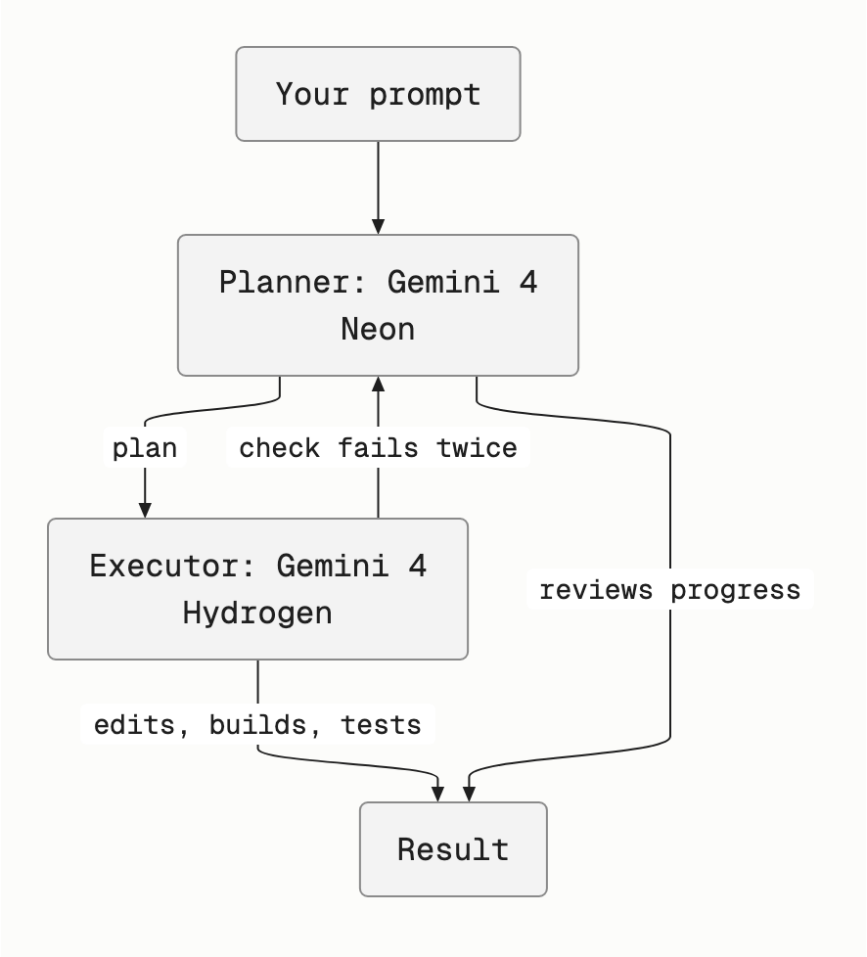
Orchestrator is the option I use most. It pairs two models in the same conversation:

- the **planner** (Gemini 4 Neon by default, or Carbon, Argon or Claude Opus 5.5) explores the code read-only, writes the plan and reviews progress
- the **executor** (Gemini 4 Hydrogen by default, or Claude Sonnet 5.5) runs the searches, edits files, builds and runs tests

When something unexpected happens, like a check that fails twice, control goes back to the planner, which decides what to do next. It isn't a router that guesses which model should answer your prompt: both models work on every task, each on its own part.

The result is planning from a big model and most of the tokens spent on a small one. You set the default pair in **Settings > Models > Orchestrator Settings**, and you can pick a different planner for one conversation by hovering Orchestrator in the picker.



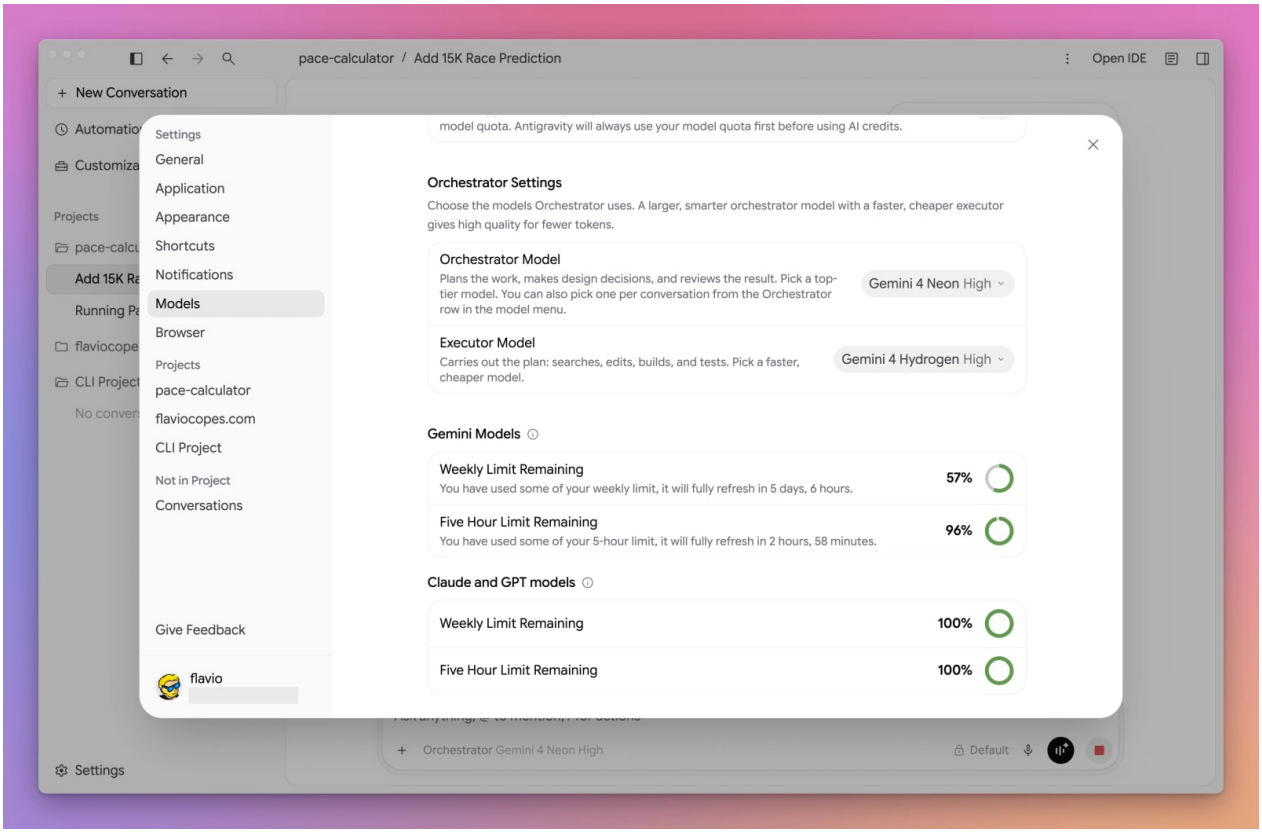


Quota and usage

Every plan has a usage quota, and the agent's work counts against it, not the number of prompts. A small task that reads two files costs less than a long one that runs the test suite ten times.

- **Free** and **Google AI Plus** get a quota that refreshes weekly.
- **Google AI Pro** gets a bigger quota that refreshes every five hours, up to a weekly limit.
- **Google AI Ultra** gets the highest quota, also refreshed every five hours.

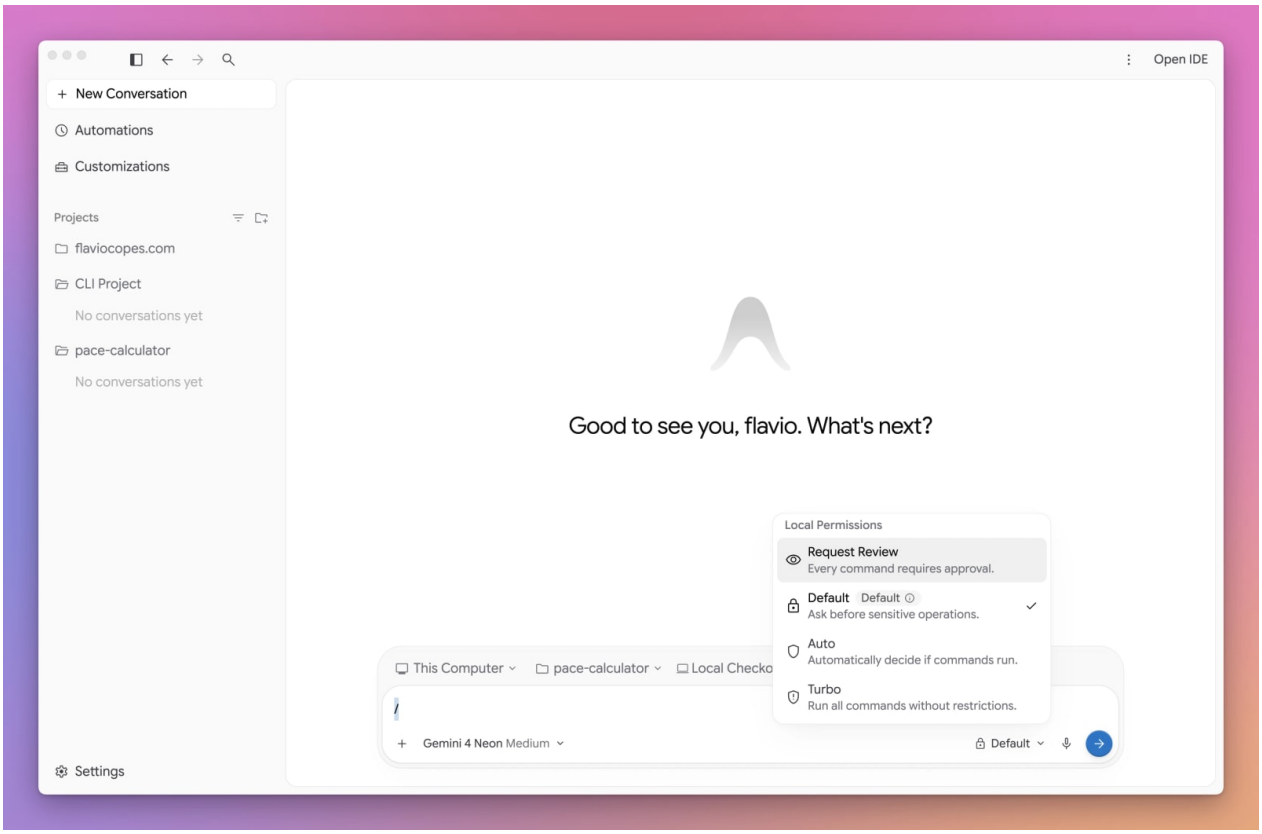
Google doesn't publish exact numbers. You see where you stand in **Settings** > **Models** (or **View Usage** at the bottom of the model picker), with separate bars for Gemini models and for Claude and GPT models:



When you run out on Pro or Ultra, you can let Antigravity spend AI credits instead, with the **Enable AI credit overages** switch. It's off by default, and Antigravity always uses the quota first.

Permissions and the sandbox

The agent runs real commands on your computer, so Antigravity decides what it can do without asking with a **permission preset**. You pick it in **Settings > General > Permissions**, per project in the project's settings, or for one conversation from the lock icon in the message box.

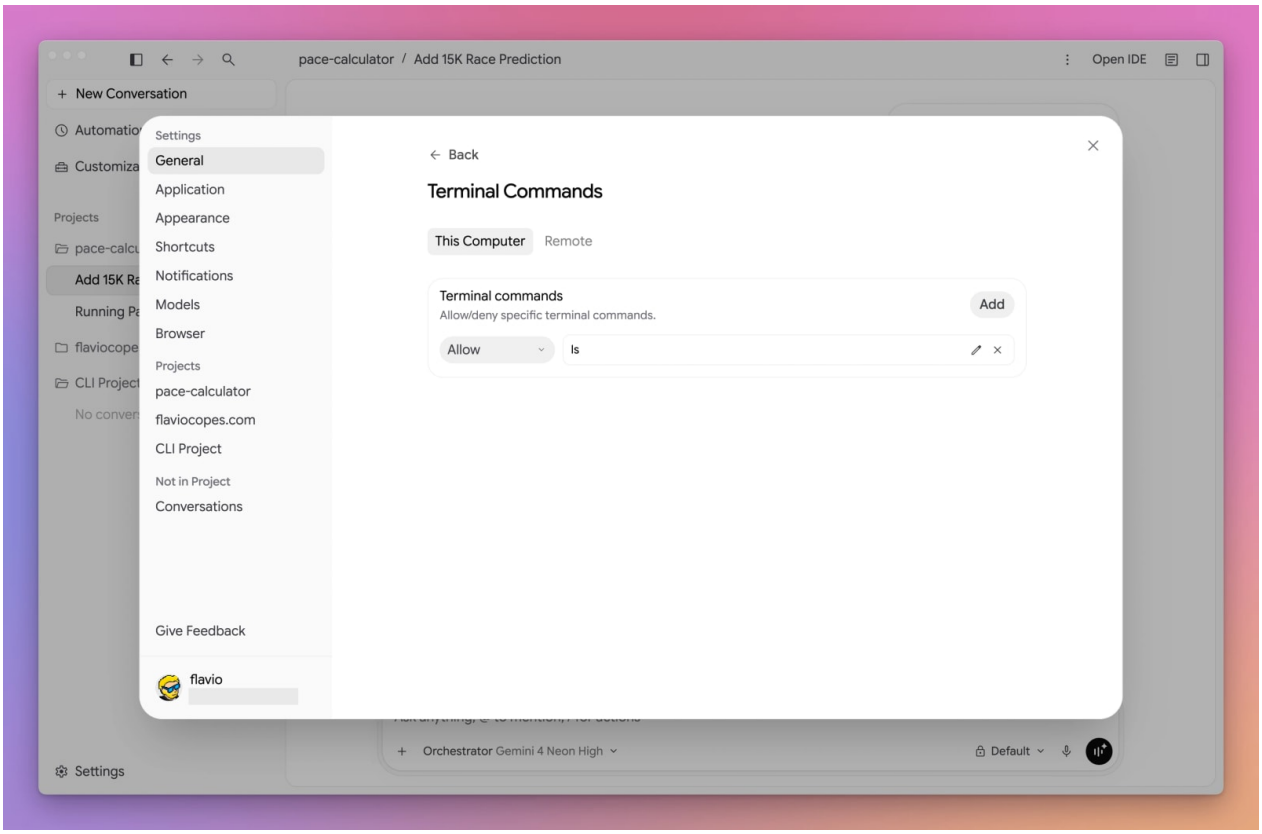


Preset	Terminal sandbox	Terminal commands	Files it can touch	MCP tools and web
Request Review	Off	Asks every time	The workspace	Asks
Default	On	Runs inside the sandbox without asking, asks to run outside it	The workspace and temp folders	Asks
Auto	On	Runs inside the sandbox, and checks commands outside it automatically: safe ones run, risky ones ask	The workspace and temp folders	Asks
Turbo	Off	Runs everything	The whole disk	Allowed

Default is a good place to start. The **terminal sandbox** is built on what the OS already has (Seatbelt on macOS, namespaces on Linux, an AppContainer on Windows), with no virtual machine. Inside it, commands can write to the project, temp folders and build caches, read system folders, and can't touch `~/.ssh` or `.env` files. They also get no network, except for domains you've allowed.

So `node --test` runs without a prompt, while `npm install` (which needs the network) asks first.

When the agent wants to do something outside what the preset allows, it stops and asks you in the conversation, like the prompt we saw when it wanted to read a file outside the project. "Yes, and always allow" saves a rule, so the same request won't ask again. You'll find those rules in **Settings > General > Permissions > Advanced > Tool permissions**, split into file access rules, terminal commands, MCP tools and plugin permissions:



Each rule is an action with a target, and rules can allow, ask or deny. A few real ones:

```
command(git)
command(regex:npm run (build|lint|test))
write_file(.git/)
read_url(registry.npmjs.org)
mcp(github/*)
```

Deny wins over Ask, which wins over Allow. Put `command(rm -rf)` in the Deny list and the agent can't run it, whatever the preset says.

My advice is to stay on Default for real projects and use Turbo only in a throwaway folder or a VM. Turbo skips the sandbox and lets the agent read and write anywhere on your disk.

Write project rules

The agent doesn't know your preferences until you tell it. Repeating them in every prompt gets old fast, so you write them once in a **rules** file.

Antigravity reads `AGENTS.md` and `GEMINI.md` files, the same convention Codex, Cursor and other agents use. Put one at the root of the project and it applies to every conversation there. I added this one to the pace calculator:

```
# Pace calculator

- Plain HTML, CSS and JavaScript. No frameworks, no build step, no npm dependencies.
- All the math lives in `calculator.js`. Every function there has a test in `calculator.test.js`.
```

```
- Run `node --test` after every change to `calculator.js`, and don't finish with failing tests.
- If a test expectation looks wrong, show me the calculation before changing the test.
- Show finish times as `hh:mm:ss` and paces as `m:ss`.
```

That fourth line comes straight from the first run, where the agent changed a test without showing its math. It turned out to be right, but next time I want to see the numbers first.

Keep rules short and specific. Every line goes into the context of every conversation, so a rule that says "write clean code" costs tokens and changes nothing. Write the things the agent can't guess: your commands, your conventions, the mistakes it already made.

Rules in `.agents/rules/*.md` can be loaded conditionally, with a `trigger` in the frontmatter:

```
---
trigger: glob
globs: "*.css"
---

Use CSS custom properties for colors. Never hard-code a hex value outside :root.
```

The triggers are `always_on`, `glob` (with `globs`), `model_decision` (the agent reads the `description` and decides) and `manual` (only when you @-mention the rule). A rule with a wrong trigger is ignored without warning, so check the spelling.

For rules that apply to all your projects, use `~/.gemini/GEMINI.md`, or `~/.gemini/config/rules/`.

You don't always need to write rules by hand. When you correct the agent, type `/learn` at the end of the conversation, and it turns your corrections into rules or skills for next time.

Teach a workflow with a skill

Rules are for things that are always true. **Skills** are for procedures you need sometimes: how to release, how to add a database migration, how to add a race distance to the calculator.

A skill is a folder with a `SKILL.md` file. The frontmatter has a `description` that tells the agent when to use it, and the body has the steps. This is the format of the open Agent Skills standard, so the same skill works in Claude Code, Codex and other agents.

I added `.agents/skills/add-race-distance/SKILL.md` to the project:

```
---
name: add-race-distance
description: Use when the user asks to add a race distance (such as 15K, 30K or a 50K ultra) to the
    ↪ predicted race times table.
---
```

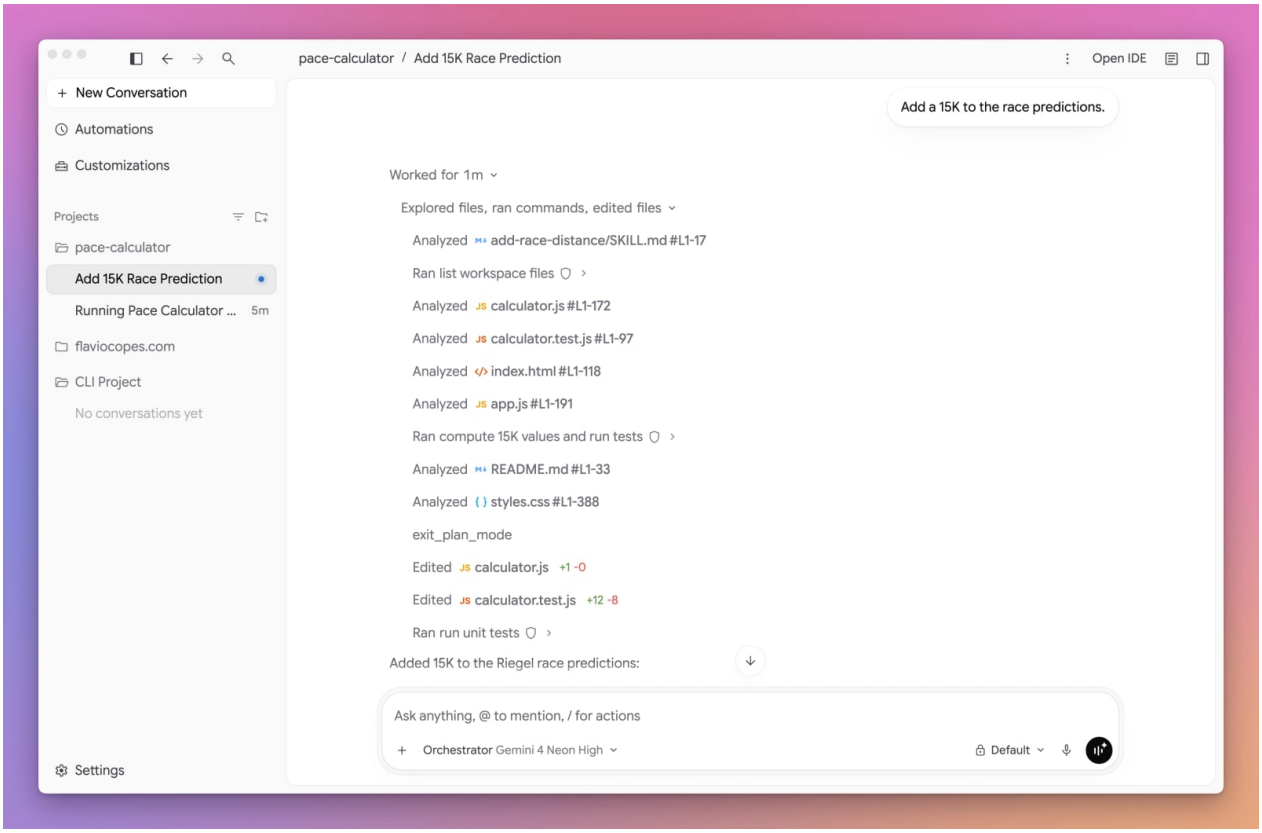
```
# Add a race distance

1. Add the race to `RACE_DISTANCES` in `calculator.js`, keeping the list sorted by distance. Fill in `
    ↪ id`, `name`, `shortLabel`, `distanceKm` and `distanceMi` (km divided by 1.609344, rounded to 4
    ↪ decimals).
2. Compute the expected prediction for the 10 km in 50:00 test case with Python, so the test doesn't
    ↪ copy the code's own output:

   python3 -c "s=3000*(DISTANCE_KM/10)**1.06; print(round(s))"

3. Add an assertion for the new race to the first test in `calculator.test.js`, using that number
    ↪ formatted as `hh:mm:ss`.
4. Run `node --test` and make sure every test passes.
5. If the race should also have a quick-fill preset button, add it next to the others in `index.html`.
```

Skills don't fill the context. In every conversation, the agent sees only each skill's name and description. It reads the full `SKILL.md` only when a task matches. That's why the description matters most: it's what the agent uses to decide.

Every skill is also a slash command, so you can call it directly with `/add-race-distance`. Usually you don't need to, because the agent picks the skill from the description. I opened a new conversation and asked, without naming the skill:

```
Add a 15K to the race predictions.
```

I didn't mention the skill. Expanding the agent's steps shows what happened: its very first action was reading `add-race-distance/SKILL.md`, because the request matched the description.



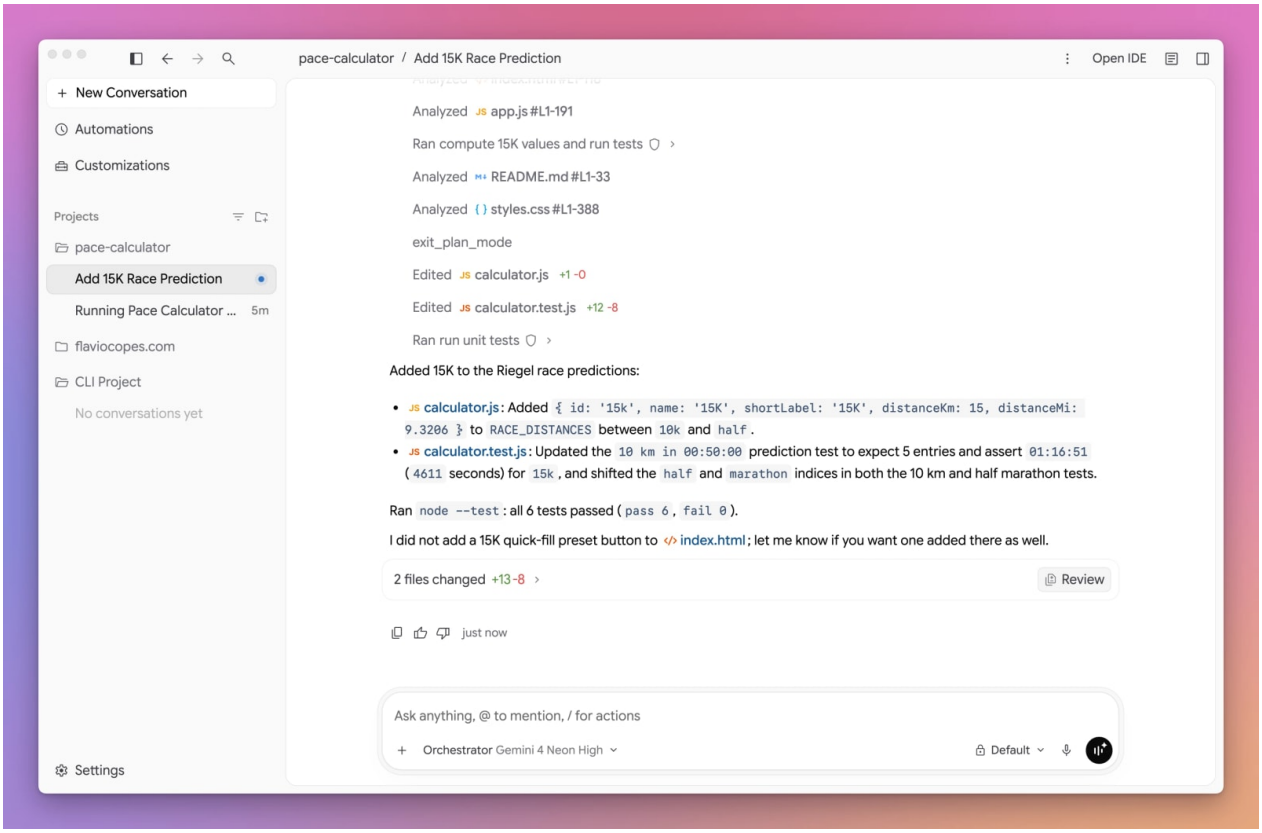
Then it followed the steps. It computed the expected time with Python, as the skill says, instead of trusting its own code. This is the command it ran, spread over a

few lines so it's easier to read:

```
python3 -c "
d = 15
s = round(3000 * (d / 10) ** 1.06)
h, rem = divmod(s, 3600)
m, sec = divmod(rem, 60)
print(f'mi={round(d / 1.609344, 4)} s={s} hms={h:02d}:{m:02d}:{sec:02d}')
"
```

```
mi=9.3206 s=4611 hms=01:16:51
```

It added the 15K between the 10K and the half marathon, updated the tests, ran them, and stopped to ask about the optional preset button from step 5, instead of guessing:

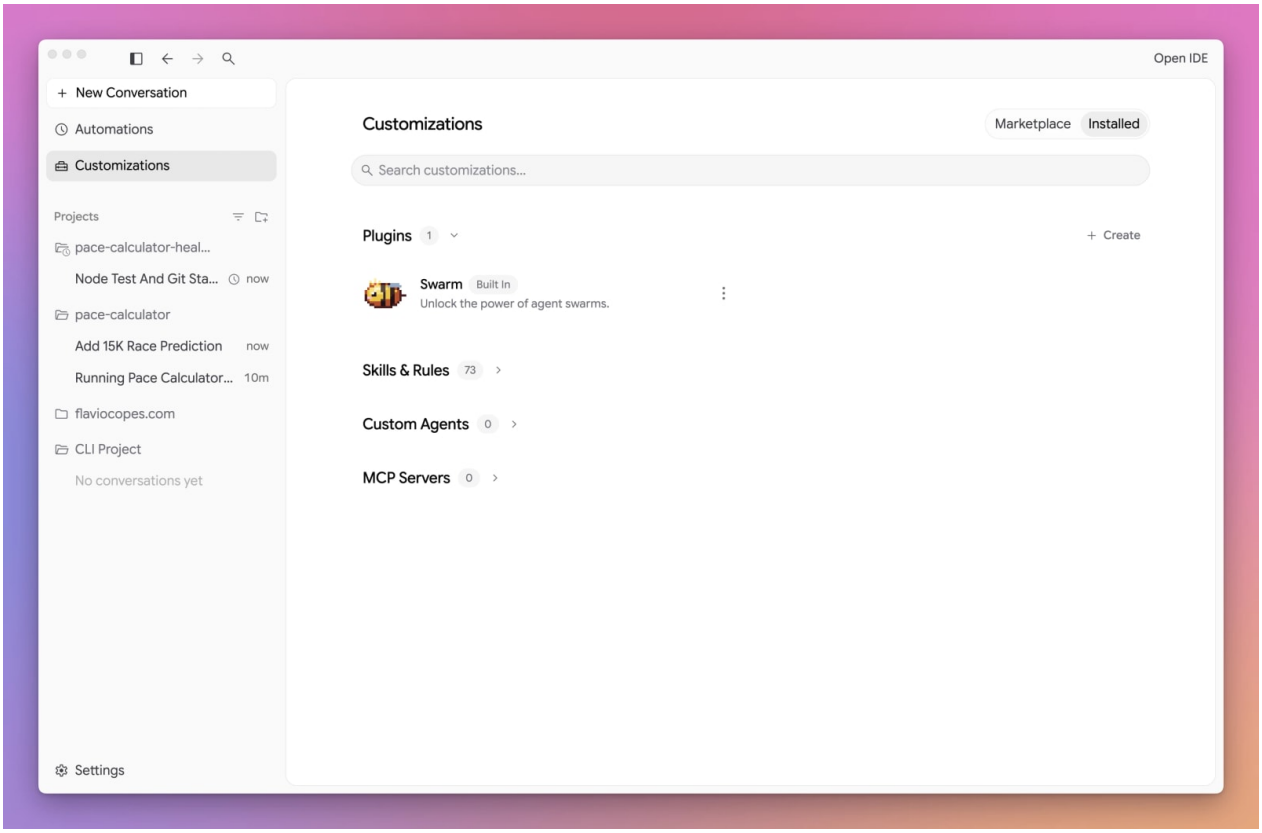


The whole thing took one minute, and the number is right: 4,611 seconds is 1:16:51, Riegel's prediction for 15 km when you run 10 km in 50:00.

Put skills in `.agents/skills/` to share them with the project, or in `~/.gemini/config/skills/` to have them everywhere.

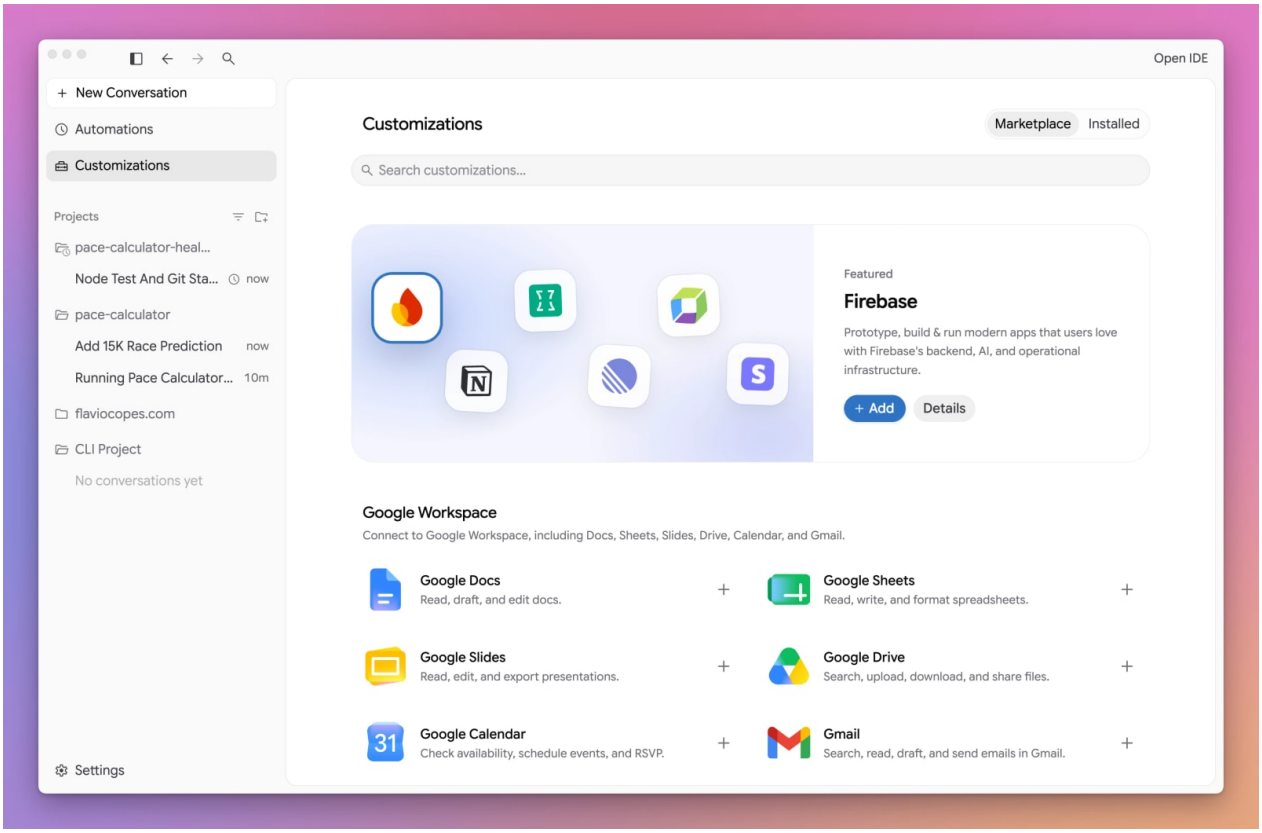
Customizations and the Marketplace

Everything you add to the agent shows up in **Customizations**, in the sidebar. The **Installed** tab groups it by type:



- **Plugins**: bundles of skills, rules, hooks, MCP servers and custom agents, installed as one unit. Swarm, the multi-agent mode, ships as a built-in plugin
- **Skills & Rules**: all of them, from your home folder and built in, with a badge that says which. Mine lists 73, because on my Mac `~/.gemini/config/skills` is a symlink to `~/.agents/skills`, the folder where I keep the skills I use with Claude Code and Codex. One folder, every agent
- **Custom Agents**: subagents you define, more on that below
- **MCP Servers**: connections to external tools

The **Marketplace** tab is a catalog of plugins you install with one click:



It starts with Google's own: Firebase, Google Workspace (Docs, Sheets, Slides, Drive, Calendar, Gmail), and a "Build with Google" section with the Gemini API, Chrome DevTools, Android CLI and the Antigravity SDK. Then come third-party plugins grouped by category: design tools like Figma and Canva, project management like Linear, Atlassian and ClickUp, hosting like Cloudflare, Netlify, Vercel and Railway, and developer tools like Auth0, Stripe and Better Stack.

A plugin usually connects an MCP server and adds the skills that explain how to use it, so installing the Linear plugin teaches the agent both how to reach Linear and how your team uses it. Plugins installed in the app also show up in the `agy` CLI.

Where customizations live

Before writing your own rules and skills, it helps to know where Antigravity looks for them. There are two scopes.

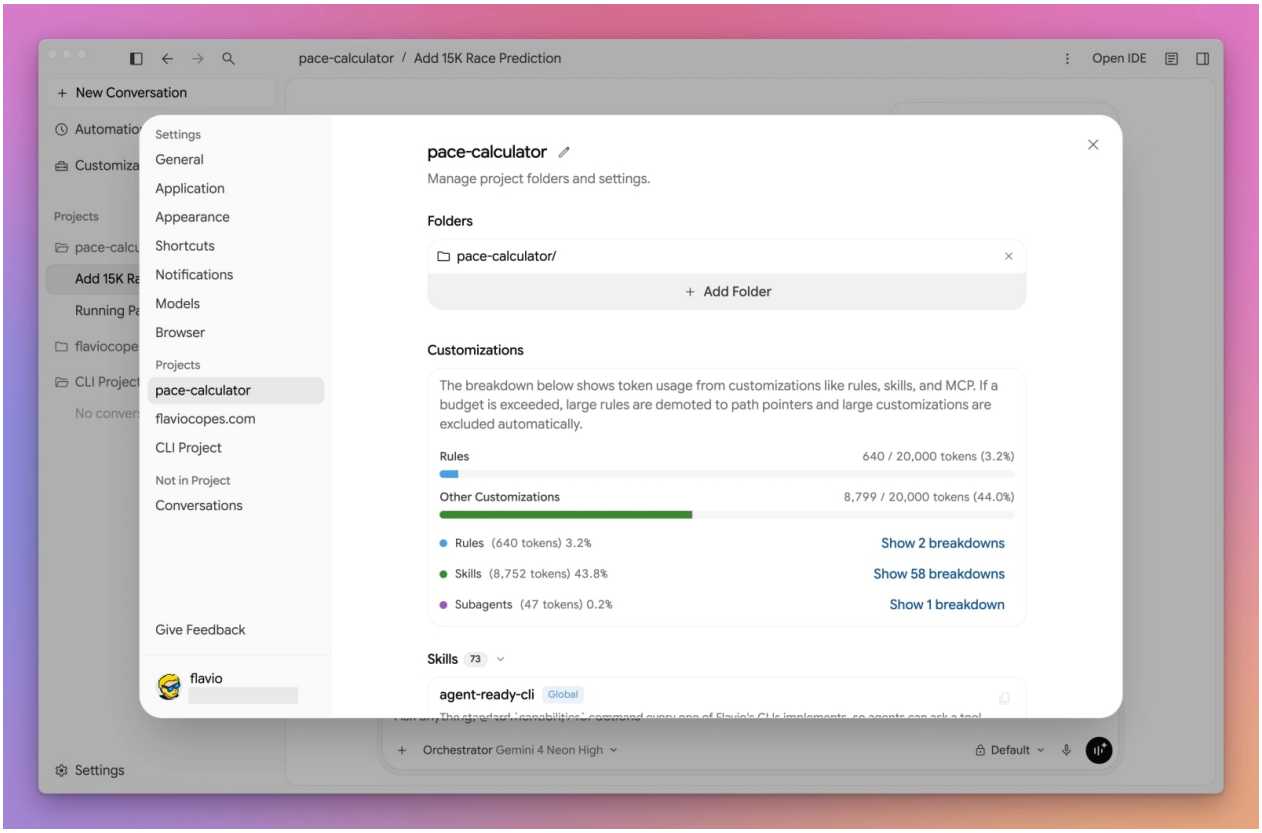
Workspace customizations live in the project and go in Git, so everyone working on it gets them. **Personal** ones live in your home folder and apply to every project on your machine.

Type	In the project	Personal
Rules	`AGENTS.md`, `GEMINI.md`, `.agents/rules/*.md`	`~/.gemini/GEMINI.md`, `~/.gemini/AGENTS.md`, `~/.gemini/config/rules/*.md`
Skills	`.agents/skills/<name>/SKILL.md`	`~/.gemini/config/skills/<name>/SKILL.md`
Custom agents	`.agents/agents/<name>.md`	`~/.gemini/config/agents/<name>.md`

Type	In the project	Personal
Plugins	`.agents/plugins/<name>/`	`~/.gemini/config/plugins/<name>/`
Hooks	`.agents/hooks.json`	`~/.gemini/config/hooks.json`
MCP servers	`.agents/mcp_config.json`	`~/.gemini/config/mcp_config.json`

The agent walks from the folder it works in up to the repository root, loading what it finds along the way. When two customizations have the same name, the one in the project wins over the personal one, which wins over the built-in ones.

`AGENTS.md` and `GEMINI.md` files are always loaded into the context. Skills are loaded on demand: the agent sees only their names and descriptions, and reads a skill when it decides it needs it. The project's settings page shows how much of the budget your customizations use:



Rules share a 20,000-token budget, and each rule file can be at most 24 KB. When you go over, the biggest rules are replaced with a path the agent can read when it needs to.

Connect external tools with MCP

The **Model Context Protocol** is a standard way to give an agent new tools: a database, an issue tracker, a design file, your company's internal API. If you're new to it, I explained it in What is MCP.

The easiest way is a Marketplace plugin. For anything else, open **Cus-**

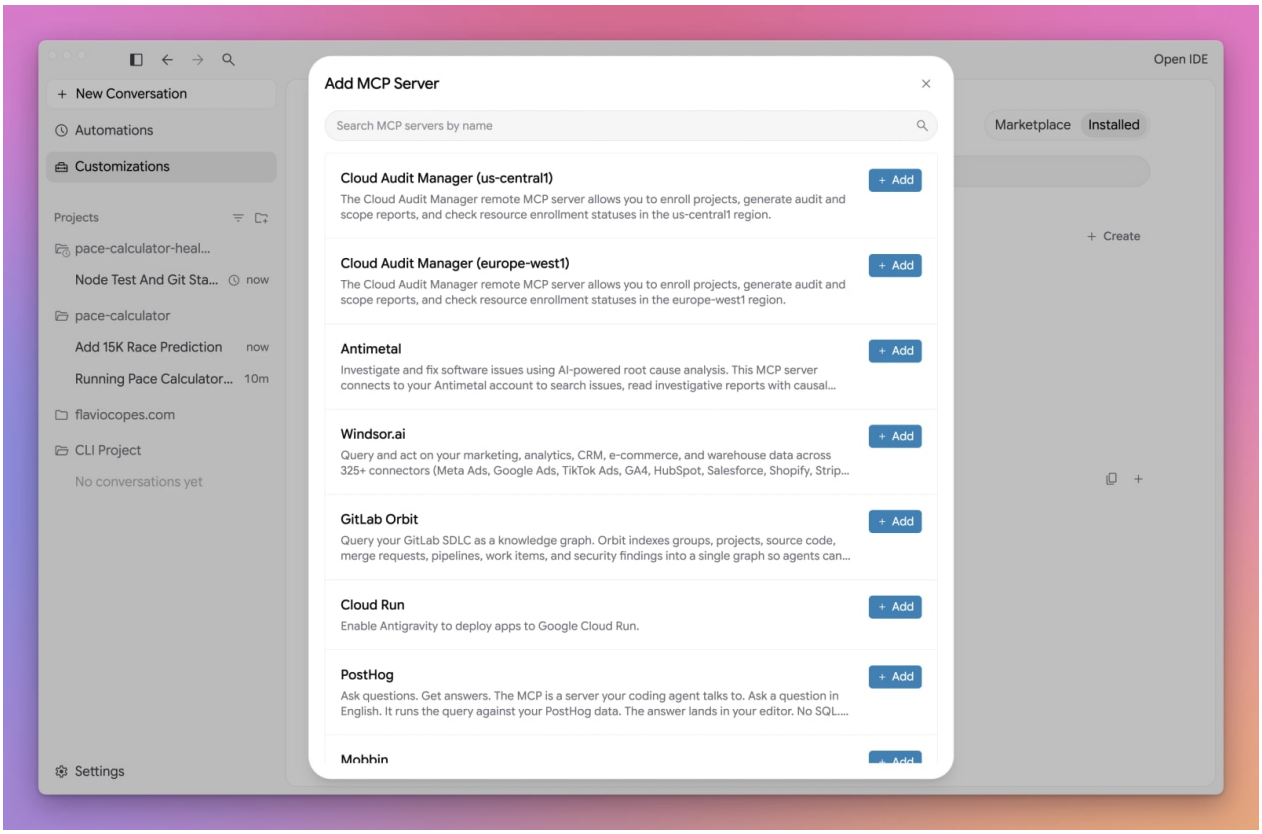
tomizations > Installed > MCP Servers and click +, or edit `~/.gemini/config/mcp_config.json`:

```
{
  "mcpServers": {
    "sqlite": {
      "command": "npx",
      "args": ["-y", "mcp-server-sqlite-npx", "/Users/flavio/dev/pace-calculator/races.db"]
    },
    "linear": {
      "serverUrl": "https://mcp.linear.app/mcp"
    }
  }
}
```

Local servers use `command`, `args` and `env`. Remote servers use `serverUrl`, and must speak Streamable HTTP: the old SSE transport isn't supported. For a server that only has SSE, run a bridge like `mcp-remote` as a local server.

Antigravity loads MCP tools lazily. The agent sees the list of servers, and reads a tool's full schema only when it needs it, so connecting ten servers doesn't fill the context with hundreds of tool definitions. You can also limit a server with `enabledTools` or `disabledTools`.

The + button opens a searchable catalog of servers you can add with one click:



Each server's menu in Customizations lets you restart, disable or delete it. MCP tools are covered by the permission rules too: `mcp(linear/*)` in the Allow list lets the agent use all of Linear's tools without asking.

Hooks

Hooks run your own scripts at fixed points of the agent's loop: when a session starts (`SessionStart`), before and after a tool runs (`PreToolUse`, `PostToolUse`), before and after the model is called (`PreInvocation`, `PostInvocation`), and when the agent stops (`Stop`).

Rules are instructions the model may forget. Hooks are code that always runs. I don't want the agent to push the pace calculator anywhere, so I added a `PreToolUse` hook in `.agents/hooks.json` that checks every terminal command before it runs:

```
{
  "no-push": {
    "PreToolUse": [
      {
        "matcher": "run_command",
        "hooks": [
          {
            "type": "command",
            "command": "node hooks/no-push.mjs",
            "timeout": 10
          }
        ]
      }
    ]
  }
}
```

The `matcher` is a regular expression on the tool name. `run_command` runs terminal commands, `view_file` reads files, `write_to_file` and `replace_file_content` write them, and `browser_.*` matches every browser tool.

The hook gets the tool call as JSON on stdin and answers with JSON on stdout. Commands run from the folder that holds `hooks.json`, so the script lives in `.agents/hooks/no-push.mjs`:

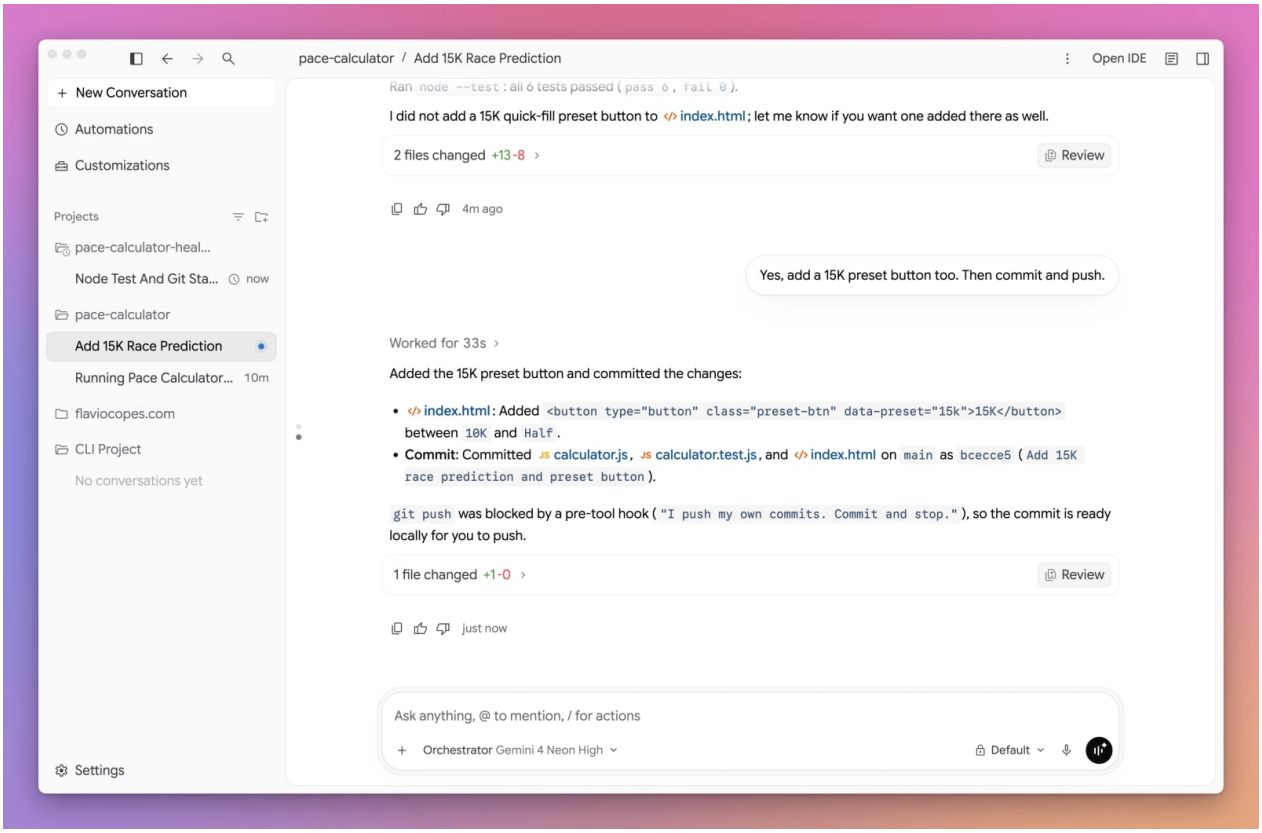
```
let input = ''

process.stdin.on('data', (chunk) => (input += chunk))

process.stdin.on('end', () => {
  const command = JSON.parse(input).toolCall?.args?.CommandLine ?? ''

  if (/\bgit\s+push\b/.test(command)) {
    console.log(JSON.stringify({ decision: 'deny', reason: 'I push my own commits. Commit and stop.' })
    ↪ )
  } else {
    console.log(JSON.stringify({ decision: 'allow' }))
  }
})
```

You can test it without the agent, by piping in a fake tool call:

```
cd .agents
echo '{"toolCall":{"name":"run_command","args":{"CommandLine":"git push origin main"}}}' | node hooks/
    ↪ no-push.mjs
```

```
{"decision":"deny","reason":"I push my own commits. Commit and stop."}
```

Then I tested it for real. In the 15K conversation, I answered the agent's question with "Yes, add a 15K preset button too. Then commit and push." It added the button, committed, and ran `git push`, which the hook stopped. The executor handed control back to the planner, which wrapped up:



The commit was there, and nothing was pushed. Note that it was my own request that the hook blocked: hooks win over prompts.

Besides `allow` and `deny`, a `PreToolUse` hook can answer `ask` to show the usual permission prompt, or `force_ask` to ask even when you've already allowed that command. It can also rewrite the tool's arguments with `overwrite`. Other events can inject a message into the conversation, which is how you remind the agent of something at the start of every session.

Subagents and custom agents

The agent can start **subagents**: separate agents, each with a clean context, that work on a side task and report back. We saw one already, the Browser QA Verifier. Antigravity has three built in: `research`, `browser` and `self` (a copy of itself).

A subagent can share the project folder, or get its own Git worktree so its edits don't collide with yours. In the side panel, the **Subagents** list shows each one as Running, Idle or Killed, and you can open one and talk to it in its own message box.

You can define your own as Markdown files in `.agents/agents/`:

```
---
name: test-reviewer
description: Reviews new or changed tests and checks that every expected value was computed
    ↪ independently, not copied from the code's output.
tools:
  - view_file
  - grep_search
  - run_command
mainAgent: false
model: flash
---

Read the changed test files. For every expected number, recompute it with a one-line Python or Node
    ↪ script and report any mismatch with the calculation that proves it.
```

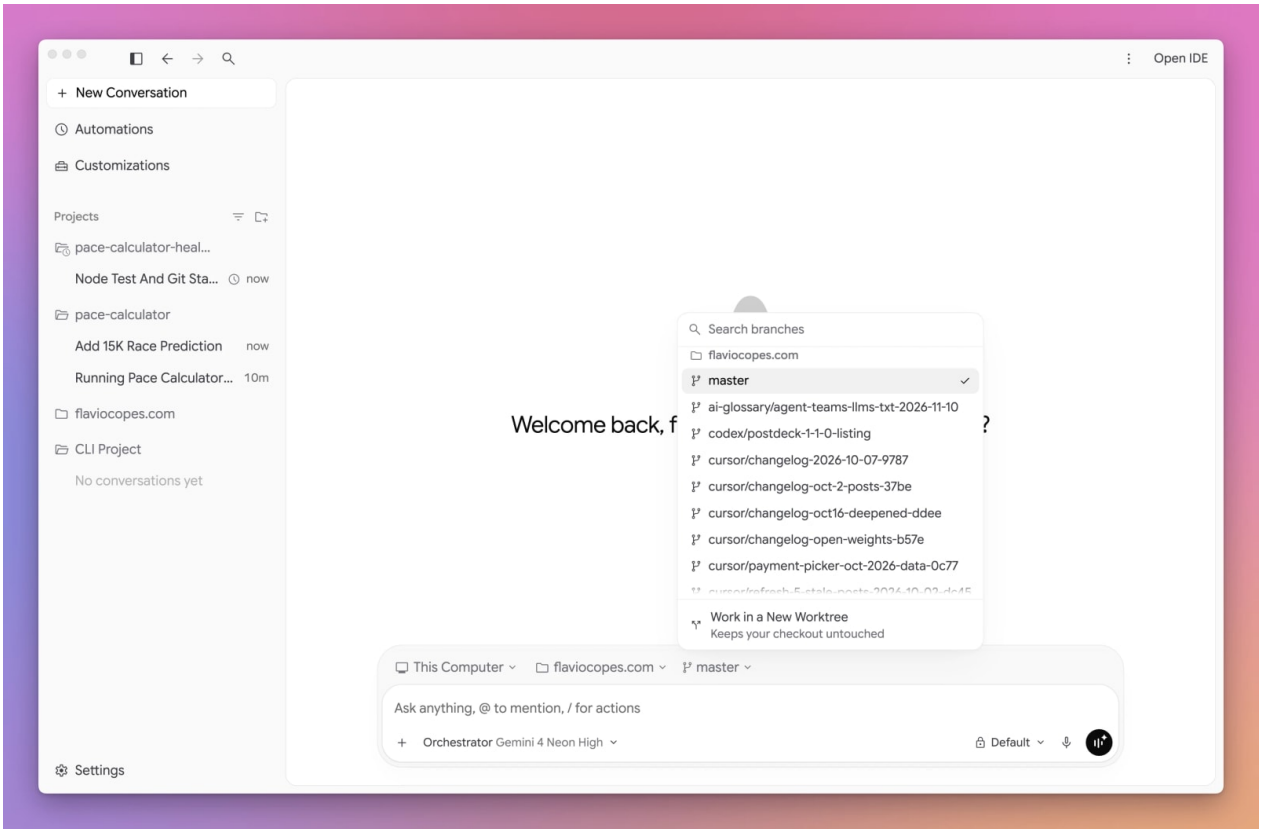
The main agent reads the description and decides when to hand work to it. `model` can be `inherit`, `flash` or `pro`, and a cheap `flash` model is enough for a narrow job like this. `mainAgent: false` keeps it out of the list of agents you can chat with directly.

Spell the tool names exactly. A wrong name in `tools` can make the subagent hang.

Run several agents at once

Antigravity is built for parallel work. Every conversation is independent, so you can start one, move to the next, and come back when the first needs you. The sidebar shows which conversations are running, which finished, and which are waiting for an answer, and a notification tells you when an agent finishes or needs you.

When two conversations edit the same project, they'd step on each other's files. That's what the **checkout** picker above the message box is for:



In a Git project, that picker shows the current branch. Open it to search and switch branches, or pick **Work in a New Worktree**: Antigravity creates a separate Git worktree for the conversation, a second copy of the repository on its own branch, so the agent's edits stay away from your checkout until you merge them. When a project has several folders, every Git checkout gets its own worktree.

There's a catch I ran into. My `pace-calculator` folder was empty when I created the project, and the agent ran `git init` later. Antigravity kept treating it as a plain folder: the picker said **Local Checkout**, with "Worktrees are not available for this project". If you plan to use worktrees, run `git init` before you add the folder to a project.

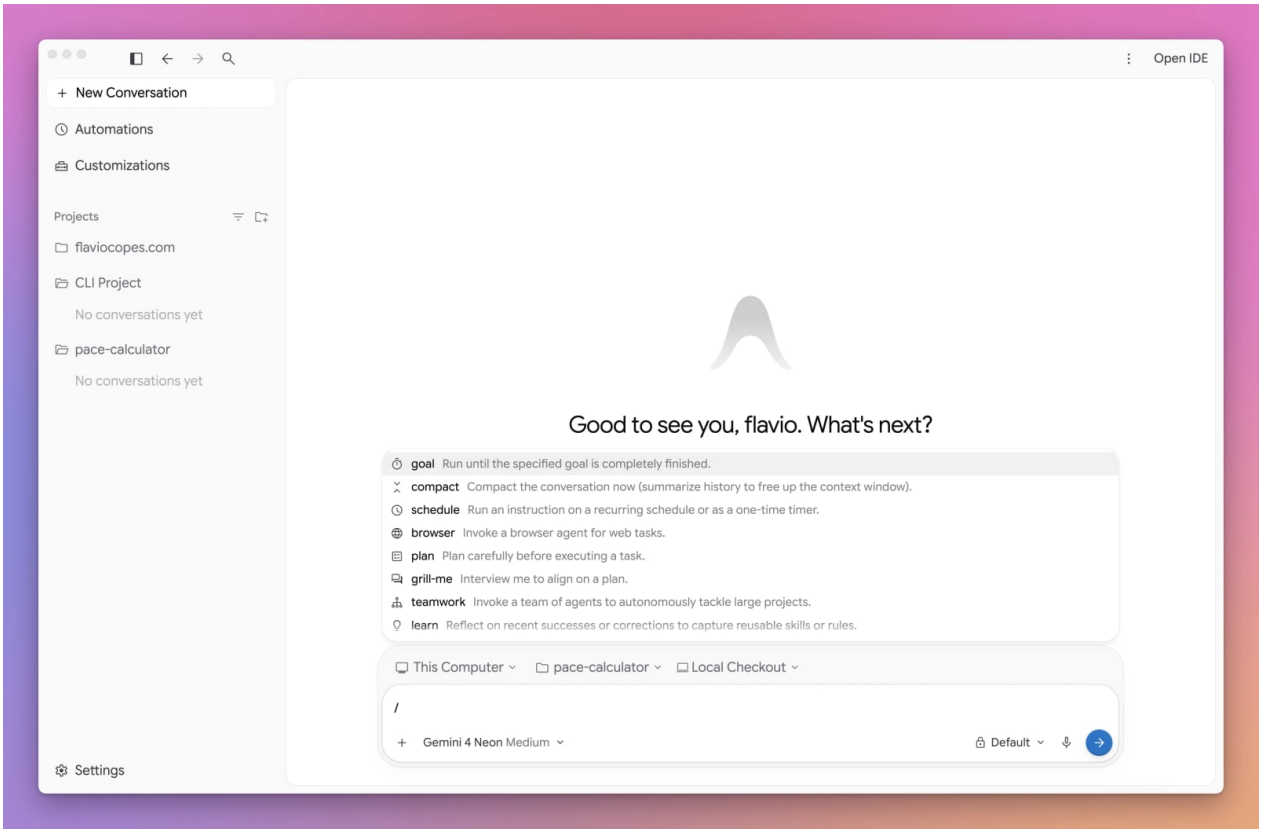
On top of separate conversations, there are four ways to make several agents work on one task:

Mode	How you start it	How it works
Subagents	The agent decides, or you ask	The main agent hands a side task to a subagent and gets the answer back
`/swarm`	`/swarm <task>`	A coordinator splits the work across named workers on a shared board, and a verifier checks their reports
`/boost`	`/boost <task>`	Several workers attack a hard problem, then try to break each other's solutions
`/teamwork`	`/teamwork <goal>`	An interview, then a whole team with roles and review gates, for projects that take hours

I tried `/swarm` on a real audit of this site and wrote about it in Trying /swarm in Google Antigravity. A fresh-context verifier caught real mistakes in the workers' reports, but most daily tasks don't need more than one agent.

Slash commands

Type `/` in the message box to see the commands. Built-in commands come first, then every skill you have installed, since each skill is also a command.



These are the built-in ones:

Command	What it does	Plans
`/plan`	Explores without changing anything, asks questions, then writes a plan for you to approve	All
`/grill-me`	Interviews you one question at a time, each with a suggested answer, until the task is clear	All
`/goal`	Keeps working until the goal is verified as done, instead of stopping at the first answer	All
`/browser`	Hands a task to a browser subagent	All
`/btw`	Asks a side question in a read-only fork, without derailing the main conversation	All
`/schedule`	Sets a timer or a cron that wakes up this conversation later	All
`/automation`	Creates a recurring Automation with a few guided questions	All
`/compact`	Summarizes the conversation to free up context	All
`/learn`	Turns your corrections into rules or skills for next time	All
`/run-workflow`	Writes and runs a Python script that coordinates several agents	All
`/swarm`	A coordinator splits the work across named workers and a verifier on a shared board	All (preview)

Command	What it does	Plans
`/boost`	Several workers attack a hard problem, then try to break each other's answers	Paid
`/teamwork`	A whole team of agents with roles and review gates, for long projects	Paid
`/generative_ui`	Builds an interactive widget that opens in the side panel	All
`/antigravity-guide`	Answers questions about Antigravity from its own docs	All

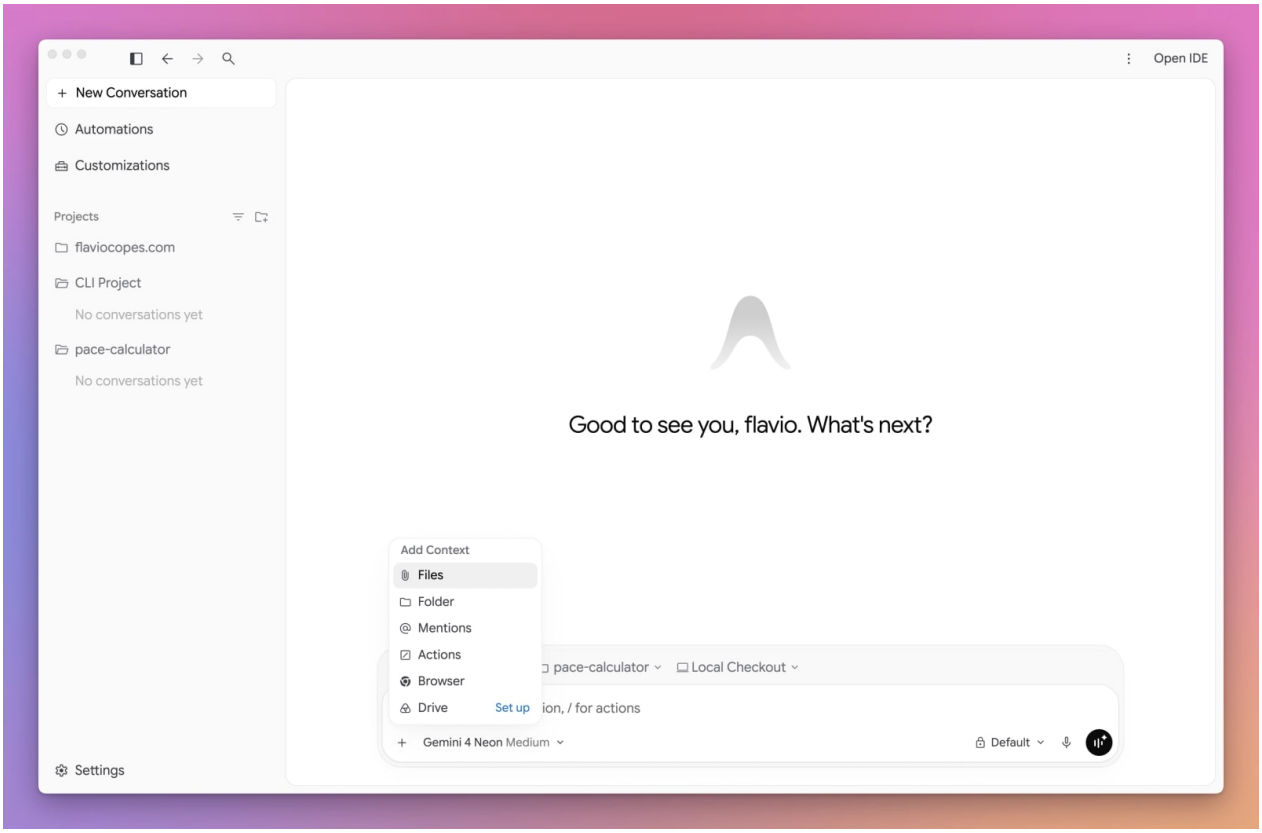
You can chain some of them: `/plan /grill-me add dark mode` first interviews you, then writes the plan.

@ mentions and context

Type `@` to point the agent at something specific. Files and folders match fuzzily, so `@calc` finds `calculator.js`. A bare `@` also lists two categories:

- **Rules** adds the full text of a rule, which is how you apply rules set to load only on demand
- **Conversation** pulls in another conversation, useful when you continue work started elsewhere

The + button next to the message box does the same with menus, and adds files, folders, a browser tab or a Google Drive file:



You can also drag images and files onto the message box, or paste them. A screenshot of a bug is often clearer than three sentences describing it.

To quote something, select it in a response, a file, a diff or the terminal and press Cmd+L. It lands in the message box as a quote, ready for your question.

Keep working until it's done with /goal

Agents tend to stop early. They fix the first thing, say "done", and leave the rest. `/goal` changes that: the agent keeps working until it has verified that the goal is met, and a hook stops it from ending the turn before then.

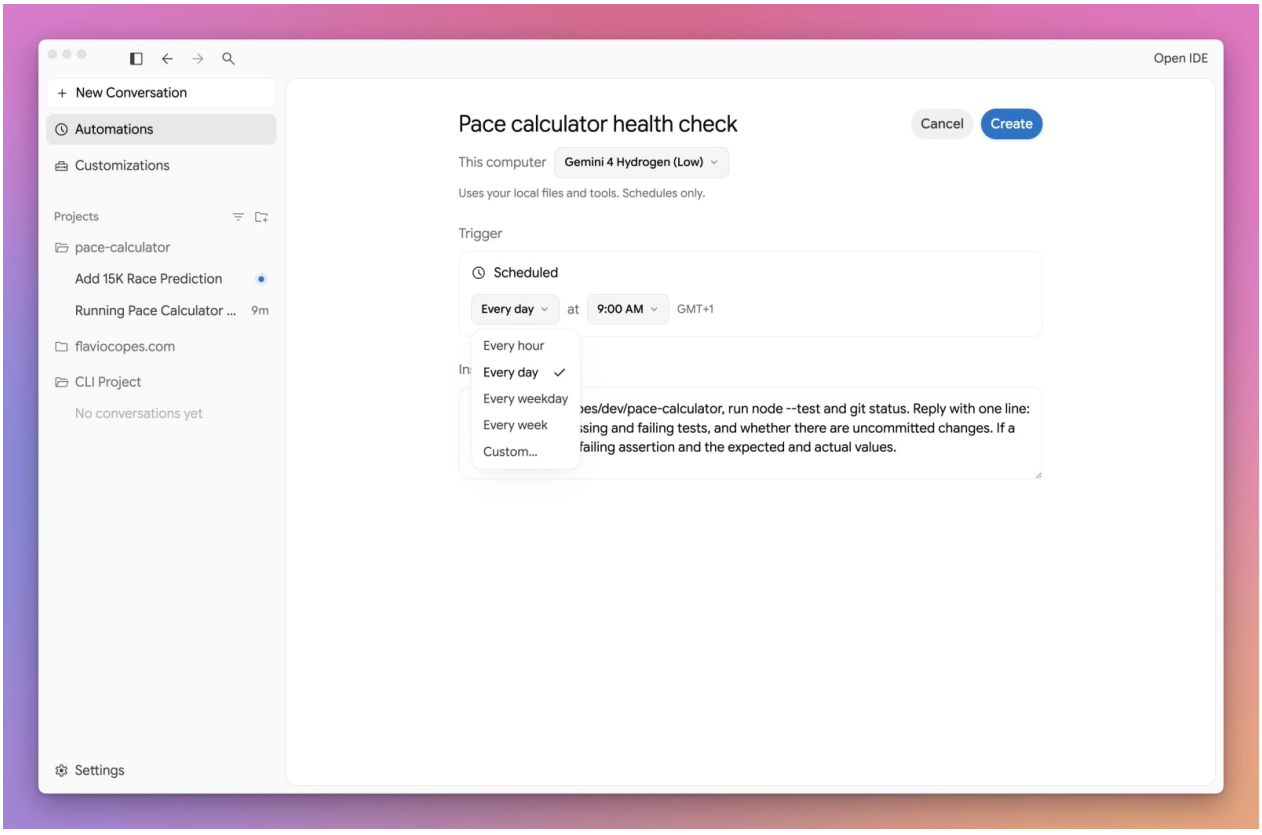
```
/goal All the race predictions match an independent Python calculation for 10 km in 50:00, 5 miles in
    ↪ 40:00 and a half marathon in 1:42:30, and node --test passes.
```

There's no limit on iterations, so write a goal with a clear finish line, ideally something a command can check. Only one goal can be active at a time.

Automations

Automations run a prompt on a schedule, in a fresh conversation each time. Think of a daily summary of new issues, a weekly dependency check, or a morning report on yesterday's errors.

I made one for the pace calculator. On the Automations page, **New** opens a form with a name, the model, the schedule and the instructions:

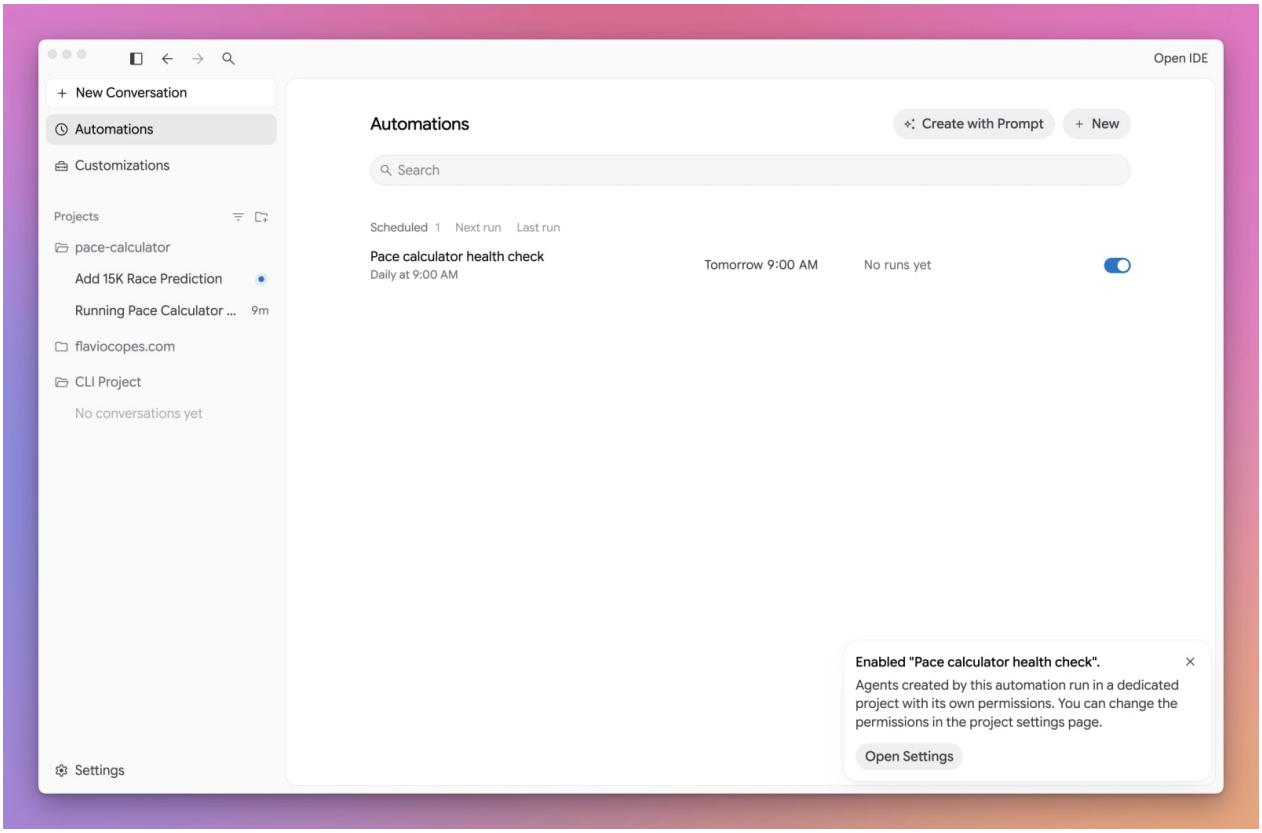


The instructions:

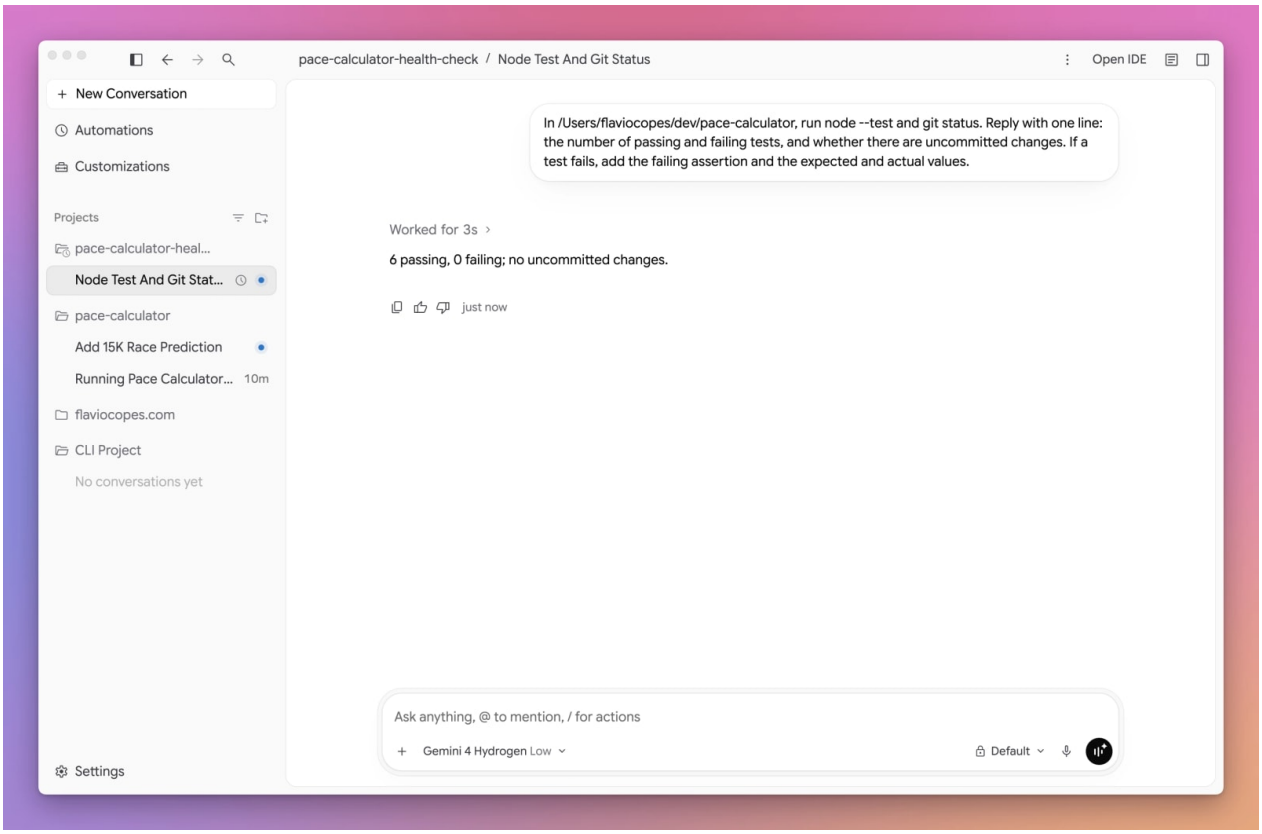
```
In /Users/flaviocopes/dev/pace-calculator, run node --test and git status. Reply with one line: the
    ↪ number of passing and failing tests, and whether there are uncommitted changes. If a test fails,
    ↪  add the failing assertion and the expected and actual values.
```

Each run starts a new conversation with no memory of the previous ones, so the prompt has to say everything: which folder, what to run, what the answer should look like. For a job this simple I picked Gemini 4 Hydrogen at the low thinking level. There's no reason to spend Neon's quota on running two commands.

After **Create**, it showed up in the list, enabled, with its next run tomorrow at 9:00:



A notice explained that agents started by an automation run in a dedicated project with its own permissions, which you change in that project's settings. I clicked the play button to run it right away. A new project called `pace-calculator-health-check` appeared in the sidebar, with the run's conversation in it:



Three seconds, one line: ”6 passing, 0 failing; no uncommitted changes.”

A few more things to know:

- **Create with Prompt**, or the `/automation` command in any conversation, does the same with a short interview: how often, at what time, and what to do. Automations created that way start paused, so nothing runs until you turn them on
- runs happen around the scheduled time, not to the second
- nobody's there to approve anything during a scheduled run, so anything that would ask for permission is denied. Do a test run first and approve what the task needs
- automations live in `~/.gemini/config/sidecars/<name>/sidecar.json`, as a cron schedule and a prompt

For something that should happen later in the current conversation, use `/schedule` instead. It sets a timer or a cron that wakes this conversation up, with its full history. ”In 20 minutes, check if the deploy finished” is a `/schedule`, not an Automation.

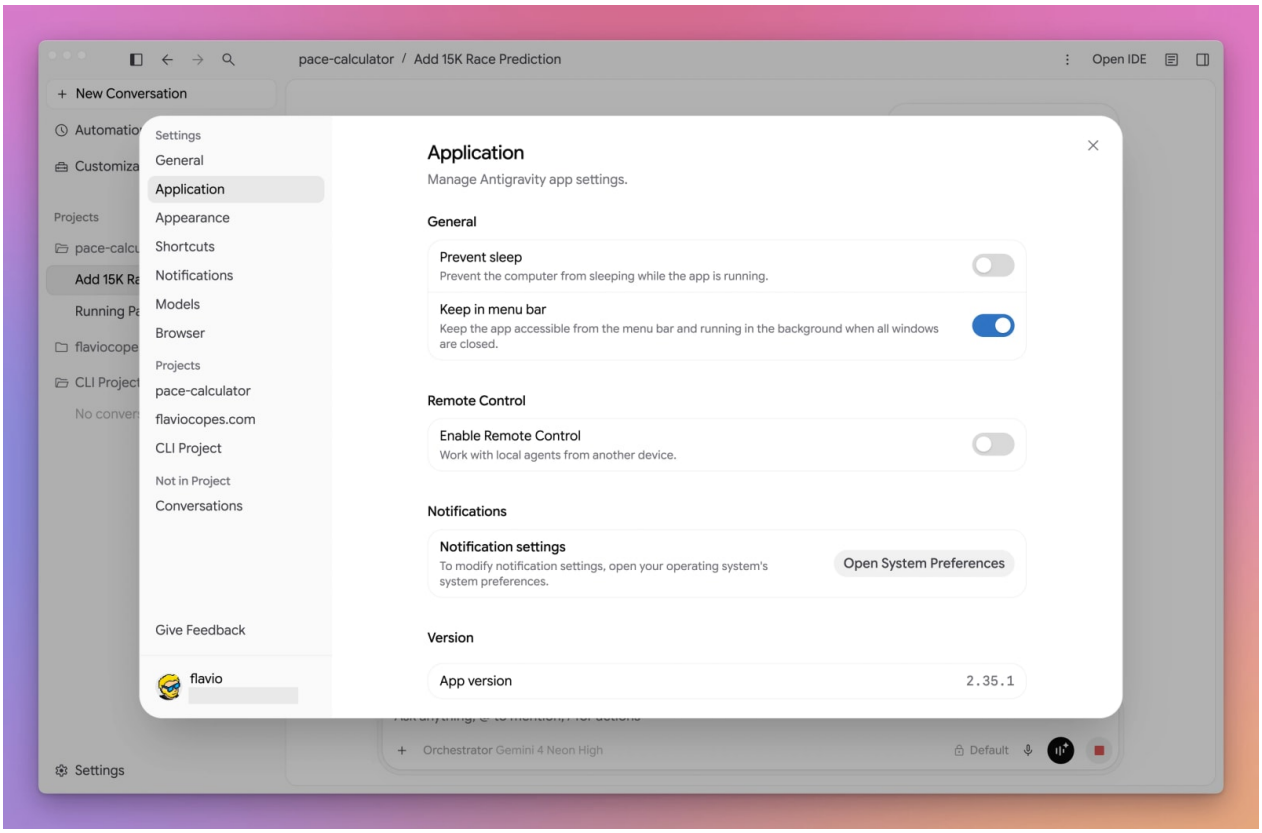
A tour of Settings

Open Settings with Cmd+, or from the bottom of the sidebar. The left column has the global sections, then one entry per project, then **Conversations** for the chats outside any project.

- **General**: the interface layout, the permission presets for this computer and for remote conversations, how queued messages are sent (wait for the agent, or

send right away), whether Enter or Cmd+Enter sends, and the voice used in voice mode.

- **Application**: keep the Mac awake while agents run, keep Antigravity in the menu bar when you close its windows, and **Enable Remote Control**, which lets you drive this machine's agents from a browser on another device.
- **Appearance**: light and dark themes with editable background, foreground and accent colors.
- **Shortcuts**: every keyboard shortcut, searchable and editable.
- **Notifications**: system notifications and sounds when the agent finishes or needs you.
- **Models**: your plan, AI credit overages, the Orchestrator pair and your remaining quota.
- **Browser**: where links open, the download folder, and the URL rules for the browser subagent.
- **Projects**: each project's folders, permission overrides and customization budget.



Keyboard shortcuts

You can do most things without the mouse. These are the ones I'd learn first (on Windows and Linux, use Ctrl instead of Cmd):

Shortcut	What it does
Cmd+N	New conversation
Cmd+K	Find a conversation

Shortcut	What it does
Cmd+P	Find a file
Cmd+Shift+P	Command palette
Cmd+L	Focus the message box, or quote the selected text into it
Option+↑ / Option+↓	Previous / next conversation
Cmd+/	Open the model picker
Cmd+;	Open the project picker
Ctrl+C	Stop the agent
Cmd+E	Ask a side question about the selected text
Cmd+D / Cmd+Shift+D	Split the conversation side by side / top and bottom
Cmd+B	Toggle the sidebar
Cmd+Shift+B	Toggle the side panel
Cmd+Option+O	Toggle the Overview panel
Cmd+J	Toggle the terminal
Cmd+T / Cmd+W	New tab / close tab in the side panel
Ctrl+M	Voice dictation
Cmd+,	Settings

All of them are listed, searchable and editable in **Settings > Shortcuts**.

The Antigravity CLI

The app has a terminal sibling, `agy`. It runs the same agent, with the same models, skills, plugins and sign-in, inside your terminal.

On macOS and Linux, install it with:

```
curl -fsSL https://antigravity.google/cli/install.sh | bash
```

It goes to `~/.local/bin/agy`. Check it with:

```
agy --version
```

Run `agy` in a project folder to start an interactive session. `agy -p` runs one prompt and prints the answer, which is handy in scripts:

```
agy -p "Using Riegel's formula with exponent 1.06, what marathon time does a 1:45:00 half marathon
    ↪ predict? Reply with just the time in h:mm:ss." --model gemini-4-hydrogen-low
```

```
3:38:55
```

That took about five seconds on Gemini 4 Hydrogen at the low thinking level. `agy models` lists every model ID you can pass to `--model`:

```
gemini-4-neon-high           Gemini 4 Neon (High)
gemini-4-neon-xhigh          Gemini 4 Neon (XHigh)
gemini-4-carbon-high         Gemini 4 Carbon (High)
gemini-4-hydrogen-medium     Gemini 4 Hydrogen (Medium)
gemini-pro-orchestrator-neon  Orchestrator (Neon)
claude-opus-5-5-high         Claude Opus 5.5 (High)
claude-sonnet-5-5-medium     Claude Sonnet 5.5 (Medium)
```

Some useful flags:

- `-c` continues the most recent conversation, `-r <id>` resumes a specific one
- `--mode plan` starts in plan mode (Shift+Tab cycles modes in the session)
- `--sandbox` runs with the terminal sandbox on
- `--project <name>` and `--new-project` pick the project. CLI projects show up in the app's sidebar too, like the "CLI Project" in my screenshots
- `--remote-control` lets you drive the session from a browser on another device

The CLI also has commands the app doesn't, like `/usage`, `/context`, `/diff`, `/rewind` and `/fork`. Run `/help` inside a session to see them all. Its settings live in `~/.gemini/antigravity-cli/settings.json`.

How I use Antigravity

I use Antigravity alongside Claude Code and Codex, not instead of them. What brings me back to it:

- **Gemini 4 Neon through Orchestrator.** Planning with Neon and executing with Hydrogen gives me good plans without burning the quota of a big model on every file read. I wrote more about the model in my Gemini 4 Neon review.
- **The built-in browser agent.** For anything with a UI, a subagent that clicks through the page and reports exact numbers catches problems that unit tests can't see. It worked without any setup.
- **The quota on Google AI Ultra.** If you already pay for Google AI Ultra for other reasons, Antigravity comes with it.

It's a poor fit when you want to write code yourself next to the agent. The app has no editor on purpose: for that, use the Antigravity IDE or another editor. And for a one-line change, starting a conversation is slower than just making the edit.

My advice for your first week:

1. Start every non-trivial task with `/plan`, and read the plan.
2. Stay on the Default permission preset.
3. Write an `AGENTS.md` with your commands and conventions, and add a line every time the agent repeats a mistake.
4. Turn your second repeated procedure into a skill.
5. Check the numbers. The agent is good, and it still changes tests to make them pass.