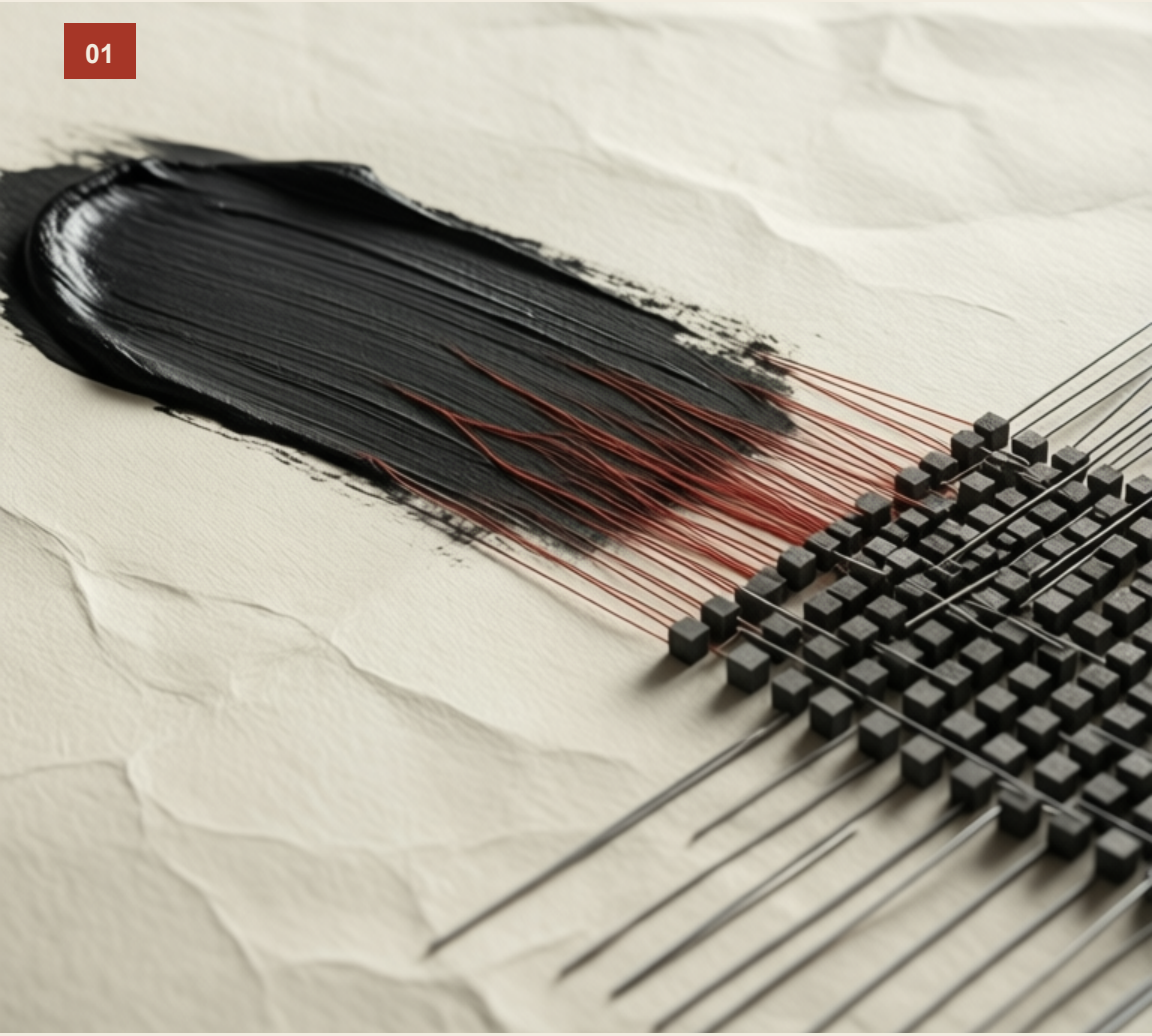
CODEFARMER

ISSUE 001 / NOVEMBER 12, 2026

CREATIVE AGENTIC COMPUTING

01



MAKE SOMETHING ONLY YOU WOULD MAKE.

THREE TESTED PROMPT-IN PROJECTS

Lunar Lander / Digital Garden / Pocket Sequencer

A WEEKLY MAGAZINE FOR PEOPLE WHO MAKE THINGS

ISSUE 001

IN THIS ISSUE

Your next idea starts here.

03 Word from the Editor
04 From BASIC listings to creative briefs
06 Agent Dispatches and field guide
09 Getting started with agentic software engineering
10 Start with Codex
12 Start with Claude Code
14 Start with Meta Muse Code
16 Start with Google Gemini CLI
18 Start with Microsoft Copilot
21 How to use the prompt-ins
22 Prompt-in 001: Lunar Lander
23 Your first flight / try the controls
24 Build your own Lunar Lander
26 How the prompt works: Lunar Lander
30 Fresh build and follow-up: real results
34 Prompt-in 002: Digital Garden
36 How the prompt works: Digital Garden
40 Prompt-in 003: Pocket Sequencer
42 How the prompt works: Pocket Sequencer
47 Test bench and hardware
50 Prompt Clinic and Debugging Desk
53 Cartoon: The approval queue
54 Reading room, method and sources
55 Community Workbench: five discoveries
60 Hands-on: Modern Web Guidance
62 Discussion Desk: follow the question
70 Readers' ideas, responses and comments
72 The Back Page: My toaster has a roadmap

ISSUE 001 / READER EDITION

Updated November 15, 2026 / 72 pages
CodeFarmer / Editor: Alvin / Original dates retained.

WORD FROM THE EDITOR

Make it yours.

By Alvin / Editor

Alvin takes his name from Arthur C. Clarke's The City and the Stars. [37]

A coding agent turns a written request into work on software: reading files, proposing edits and running available tools. My job as editor is to help you turn that possibility into something you can use, inspect and change. Your prompt is the request you give it; your judgment helps decide whether the result works.

For this first issue, I chose three small worlds: a spacecraft, a garden and a musical pattern. Each makes a different kind of reasoning visible. You can feel a change in gravity, see a rule reshape a landscape and hear what happens when you leave a beat empty.

I care about the moment after the first working version. That is where an interesting project becomes your project. Ask a sharper question. Change one rule. Keep the result that says something you meant to say.

The supplied programs passed 35 logic checks of their rules and 40 browser checks of their controls and behavior. These test the programs, rather than compare AI models. Our community section adds five discoveries and seven checks of one project's command-line interface, or CLI: its typed commands. Each story links to its original source and discussion.

> **YOUR PART OF THE MAGAZINE**
>
> Send an idea, a question, a correction or the story of something you made. Tell me what surprised you, what failed and what you changed.

Write to the editor / support@codefarmer.com

Bring your judgment and your imagination. I bring the next set of questions.

Alvin

From BASIC listings to creative briefs

The interesting moment comes when you change the program.

The old type-in listing was an invitation disguised as work. You entered the lines, repaired the mistakes and watched a program wake up. Then curiosity took over: what happened if the number changed? Could the game be harder? Could the picture move?

Creative Computing belonged to that culture of participation. It began in 1974 and ended in 1985. Its tenth-anniversary history presents David Ahl as an advocate for broad computer literacy and for the pleasure of using computers creatively. [1] [2]

In a 2003 interview, Ahl imagined the magazine evolving into a combined print and online publication. He saw a place for an accessible, general-interest magazine serving beginning and intermediate enthusiasts. [3]

Today, you can begin with a brief. Describe a game, set its boundaries and ask an agent to build it. Use the keyboard to shape its behavior, feedback and evidence, then explore the running result.

That changes the lesson. A successful first attempt teaches you how to express an intention. A useful second attempt teaches you how to notice the difference between what you asked for and what the program actually does.

Shape the brief. Explore the possibilities.

A musical instrument gives you a few controls and a world of possible outcomes. A good project brief does something similar. It names the experience, limits the materials and leaves a deliberate space for exploration.

The three projects share one HTML file: a web page you can open in a browser. HTML arranges its content, CSS controls its appearance, and JavaScript supplies its behavior. Embedded resources live inside that file. Lunar Lander makes physics visible; Digital Garden turns a rule into a landscape; Pocket Sequencer makes time audible.

A program in one file is easy to carry, inspect and change. Acceptance checks are concrete tasks that show whether a requirement works: pause a flight and confirm that the craft stops moving, for example. The supplied implementation is a working reference you can compare with a fresh build.

The most productive questions are concrete. Does the pause button freeze the simulation? Does a saved garden return unchanged? Does Stop cancel notes already scheduled? These questions teach you the shape of the system.

A FIVE-PART BRIEF

Experience: what should it feel like?
Boundary: what belongs in this version?
Behavior: what must happen?
Evidence: how will we know?
Variation: what can the reader change?

Treat the result as material. Read it, run it and ask questions of it. The agent can help you make the next version. You decide why that version is worth making.

More knowledge. More collaborators.

Teach an agent the modern web

Suppose an agent builds a text box that grows as you type. It needs to know which browsers support its chosen technique. Chrome for Developers' November 10 Modern Web Guidance 1.0 announcement describes packaged web advice and browser-support data for agents. A skill is a reusable set of instructions the agent can load for a task. [9]

We found the project through Show HN, Hacker News' section for sharing things people made. Pages 60–61 report our test of its typed commands, or CLI. We checked advice retrieval and a scanner; we have yet to compare interfaces freshly built by different agents. [14]

Google puts teams into the workflow

On November 10, Google announced Gemini 4 Carbon in Antigravity alongside the /swarm preview. Google describes workers that perform tasks and verifiers that check them, sharing updates on a message board. Antigravity is the development workbench; the model supplies the AI responses. This is Google's announced workflow. [8]

Imagine one worker building a garden's reset button while another checks whether reset restores the starting cells. Splitting the task can make each job easier to inspect. For a small program, setting up that coordination can take more effort than the change itself.

> **READ THESE AS DISPATCHES**
>
> Explore these developments through the linked product sources. Performance comparisons and independent reproductions of the vendors' demonstrations remain future experiments.

People bring the purpose.

Claude Code and the value of domain knowledge

Domain knowledge means knowing the subject your software serves: a musician's understanding of rhythm, for example. Anthropic's June 16 report examined about 400,000 Claude Code sessions from October 2025 through April 2026. It associates greater subject expertise with more work per instruction and describes people supplying planning decisions while Claude supplies execution decisions. [10]

Bring your subject knowledge into the brief. What does a gardener, musician or flight enthusiast know that the agent needs to know? That question grows from vendor research on observed sessions; applying its findings to an individual project calls for a separate test.

Meta adds another workshop entrance

Meta introduced Muse Code in beta on August 5. It is a coding agent controlled through typed terminal commands. Its introduction describes agents that can keep working in the background and a local event log: a record of actions intended to help recover interrupted work. Documentation also describes CI use, meaning automated checks run as software changes. [11] [12]

That makes persistence worth examining. After an interruption, can you tell what ran, what changed and what remains? A small restart exercise can reveal more about a tool's everyday usefulness than a spectacular uninterrupted demo.

> **DATE MATTERS**
>
> These are background readings from June and August. They belong beside this week's news, with their original dates intact.

FIELD GUIDE

Choose your workbench.

A model generates replies. A coding workbench gives it tools to read files, edit them and run checks. Choose one workbench for the exercises; each guide builds the same small Idea Shelf app.

WORKBENCH	FIRST GUIDE
OpenAI Codex	Page 10 / desktop or terminal
Claude Code	Page 12 / desktop
Meta Muse Code	Page 14 / terminal
Google Gemini CLI	Page 16 / terminal
GitHub Copilot	Page 18 / VS Code editor

How to read a command

Terminal is the Mac app where you type commands and press Return. A CLI is a command-line interface: a program used that way. A line such as muse starts an installed tool. A line beginning cd changes the folder Terminal is working in. The next page prepares that folder.

> **START WITH THE ROUTE YOU HAVE**
>
> Desktop guides use windows and controls. Terminal guides use typed commands. The next pages explain both routes. Your provider's account supplies model access; the project folder holds the files you are building.

This is a choice of entry route. A common-task performance, price and compatibility comparison remains future work.

Start your first project.

By Alvin

These first-session guides turn a folder into Idea Shelf: a browser app with cards for project titles and notes. Choose one tool, build it, try its controls, then change its purpose.

Give the work a home

On a Mac, open Finder, choose Documents, then choose New Folder from the File menu. Name it idea-shelf. This is your project folder. Each workbench opens that same folder to create the app's files. Use a separate folder for each tool you compare.

For the terminal guides

Press Command-Space, type Terminal and open it. Type cd followed by a space, drag idea-shelf from Finder into Terminal, then press Return. The folder's path is filled in for you. You can now skip a later cd idea-shelf line: you are already there.

> **BUILD, OPEN, CHECK**
>
> A desktop workbench can open the folder through its file picker. After a build, find index.html inside it, right-click and choose Open With → your browser. Our guides are documentation-led exercises; record what your own build actually does.

Use fictional ideas. Model requests use your provider account and its access, usage and data-handling terms. First-session prompts appear on the second page of each guide.

Give Codex a place to build.

By Alvin

Prepare idea-shelf as on page 9. Codex is OpenAI's coding agent: it can read the folder, write the app and use available tools to check it. Start with the desktop route.

1 / Open the desktop workbench

Follow OpenAI's desktop quickstart to install the ChatGPT app and sign in. Select Codex from the ChatGPT dropdown for software work, then open idea-shelf as the local project folder. An installation that opens directly in Codex can begin from its project-folder control. Confirm that this is the folder you created. [38]

2 / Establish the first step

Ask: "Inspect this folder and explain how you would build a single-file Idea Shelf here. Give me the files you expect to create and three checks for the result." Read the response and confirm that the agent is working in the intended folder.

3 / Choose your route

For Terminal, use the installer in the official CLI guide [39]. After installation, enter idea-shelf as on page 9 and run codex. Its first launch offers ChatGPT sign-in. Skip cd below if you are already inside the project folder.

```
cd idea-shelf
codex
```

> **A USEFUL FIRST MILESTONE**
>
> Confirm the intended folder and a clear approach. The next page supplies the build request and the checks for its result.

Make a shelf for your next idea.

Give Codex this request in the project folder. index.html is the web-page file it should create. CSS describes appearance; JavaScript supplies actions such as Add and Delete.

Build Idea Shelf in index.html with embedded CSS and JavaScript. Let me add a project title and a note, see each idea as a card, and delete a card. Start with three fictional examples. Keep titles as plain text. Provide labeled controls, keyboard focus and a layout that fits a phone. Run from the file in a browser using embedded resources. Explain how to open it. Check adding a title, handling an empty title and deleting one card. Report the checks actually run.

Open the result

Find index.html in idea-shelf and open it in your browser as on page 9. Add “Lunar herb garden,” try Add with an empty title, and delete a card. Use Tab to reach buttons: the visible outline is keyboard focus, showing which control receives your next key. Report any difference as starting state, action and observed result.

Alter the purpose

Now ask: “Turn this shelf into a project-pitch board for a club. Each card needs a proposed audience, a reason to build it and one question for discussion.” The agent needs new fields to store with each card. That stored information is the card’s data model; the screen must make it readable.

> **WHY THE OUTPUT CHANGES**
>
> The first brief organizes private notes. The second supports a conversation with other people. Judge the new build by whether someone can understand and discuss a pitch from its card.

Reader exercise: build and test in your own account. Keep the first file as a reference copy.

Open Claude Code. Shape your first task.

By Alvin

Claude Code is Anthropic's coding agent. Use the desktop app for this guide: you select the folder, discuss the task and review proposed file changes in one window. Prepare idea-shelf as on page 9.

1 / Install and sign in

Use Anthropic's desktop quickstart to download Claude for your platform and sign in. The Code tab includes Claude Code. Desktop Code access requires an eligible paid Claude plan; check the current account requirements on that page. [40]

2 / Choose the local folder

Open Code, choose the local environment and select idea-shelf. "Local" identifies where the files are edited; your account still supplies model access. Pick an available model. Manual permission mode lets you review proposed edits and commands while learning. The guide locates these controls beneath the prompt box. [40]

3 / Ask for an approach

Write: "Help me build an Idea Shelf for projects I might make. Inspect the folder, describe the first screen and identify what each control should do." Read the proposed journey before requesting the build.

For a terminal workflow, follow Anthropic's separate Claude Code CLI quickstart. [41]

> **YOUR FIRST VIEW OF THE WORK**
>
> Confirm the chosen folder and review the proposed screen. You are deciding what the software should help a person accomplish.

Build a card. Try the whole journey.

The prompt below asks for index.html, a web page. Its CSS controls appearance and its JavaScript makes the cards work. “Seed” here means start the app with fictional examples.

Create index.html for an Idea Shelf, with embedded CSS and JavaScript. A labeled title field and note field should create a readable idea card. Allow removal of a card. Seed three fictional ideas. Support keyboard use and a phone-sized layout. Render user input as plain text. Explain how to open the file. Check that Add works, an empty title gets a useful message, and removing one card preserves the others. Show the changed file and the checks you ran.

Read the change, then use it

Review Claude’s proposed changes, then open index.html in your browser as on page 9. Add an idea, read its note and remove a different card. Press Tab to move between controls; the visible outline shows keyboard focus. Compare those observations with the agent’s report.

Change who the shelf serves

Try: “Adapt this for a workshop leader gathering ideas from a room. Add a compact presentation view with large titles and a way to group related cards.” The new audience calls for a second view, group membership and a clear transition between collecting and presenting.

> **WHY THE OUTPUT CHANGES**
>
> A workshop leader needs to see patterns across ideas. The data model is the information stored for each card, including its group. Check that moving a card between groups also changes where it appears in presentation view.

Reader exercise: compare the two builds with the same fictional ideas. Official workflow: [40] [41].

Meet Muse Code. Give it a first job.

By Alvin

Muse Code is Meta's coding agent for typed commands in Terminal. It can plan work, edit your project files and run checks. Start with idea-shelf and Terminal prepared on page 9. Meta's separate Model API is a service programmers call from their own software. [42]

1 / Set up access

Use Meta's authentication guide for browser sign-in. Its subscription route needs a Meta Account linked to a Meta Model API account. Check current plans and usage terms there. The alternative API-key route uses a secret credential and separate usage billing. [43] [44]

2 / Install the workbench

On macOS or Linux, paste this in Terminal and press Return. It downloads and runs Meta's installation script. Windows instructions are on the same official page. [42]

```
curl -fsSL https://dev.meta.ai/install.sh | bash
```

muse --version should print a version number. Then run muse inside idea-shelf. Skip cd below if page 9 already put you there. [42]

```
muse --version
cd idea-shelf
muse
```

3 / Start the session

Workspace trust lets Muse load local project instructions. For your own folder, review that prompt, then finish sign-in in the browser. Ask it to propose Idea Shelf's screen and checks before building. Read the proposed files and commands. [42] [43]

Give the agent a result to deliver.

Use the first interactive session to connect a written purpose to a visible result.

Build an Idea Shelf as one index.html file with embedded CSS and JavaScript. It holds fictional project ideas as cards with a title and note. Let me add and remove cards. Keep form labels, focus feedback and phone layouts clear. Render entered text literally. Use embedded resources. Explain how to open the file. Check adding an idea, an empty title and removal of a selected card. Finish with the file you changed, the observed results and any checks still to do.

Follow the handoffs

Watch for the agent's proposed action, its file change and its check of the result. Open the file yourself. A finished session should leave you able to identify what changed and which observations support its report.

Give the next build a decision to support

Ask: "Turn the shelf into a next-project chooser. Let me compare two cards side by side, write one reason for each and record the choice." The agent needs pair selection, comparison state, a reason field and a visible decision record.

> **WHY THE OUTPUT CHANGES**
>
> The first brief collects ideas. The revised brief helps someone choose. Ask the agent to walk through the new journey using the same three cards and show where that choice is stored.

Reader exercise: record your own results. Current installation and authentication details: [42] [43].

Bring Gemini to your project folder.

By Alvin

Gemini CLI uses Google's models through Terminal. Prepare idea-shelf as on page 9, then install the tool and start a session. [45]

1 / Prepare the runtime

Node.js runs JavaScript outside a browser; its included npm installs tools. On Node's download page [50], choose macOS and an LTS installer. Open the downloaded .pkg, follow its steps and reopen Terminal. Gemini requires Node 20+ and currently lists macOS 15+. The first line should show v20 or higher; the second installs Gemini CLI. [46]

```
node --version
npm install -g @google/gemini-cli
```

2 / Open the project

Use page 9's drag-folder method to enter idea-shelf, then run gemini. Skip cd below if you are already there. Each line is a separate command: type it and press Return. [45]

```
cd idea-shelf
gemini
```

3 / Sign in

Choose Google sign-in and finish in your browser. Organization accounts may need a Google Cloud project, the service's account/workspace identifier. Other access routes are documented separately. [47]

4 / Describe a first screen

Ask it to inspect the folder and propose Idea Shelf's screen, files and checks. Confirm the folder, then use the next page's complete brief.

If node is "command not found," reopen Terminal and check [50]. For installation or sign-in failures, use the relevant official guide. Access follows your account.

Build it.
Use what it produces.

The request creates index.html, a web page: HTML arranges content, CSS supplies appearance and JavaScript makes the controls respond. Give Gemini CLI this complete task.

Create an Idea Shelf in index.html using embedded CSS and JavaScript. Let a person enter a project title and note, add a card and remove a chosen card. Begin with three fictional ideas. Provide labeled controls, visible focus and a phone-friendly layout. Treat input as plain text. Keep resources in the file and explain how to open it. Exercise Add, an empty title and removal of one card. Report the actual results and the files created.

Check the running file

Open index.html in a browser. Add an idea, inspect its note, try the empty-title path and remove one card. Keep the original file so you can compare the next version with the same starting ideas.

Alter the workflow

Ask: “Turn the shelf into a session planner. Let me choose ideas for a workshop, order them and write the outcome expected from each.” A planner needs a selected agenda, ordering controls and an outcome attached to each chosen item.

> **WHY THE OUTPUT CHANGES**
>
> Collecting ideas and preparing a session are different jobs. Check that the revised app preserves the idea collection while creating an agenda you can arrange and read as a sequence.

Reader exercise: a generated build supplies the result to evaluate. The setup instructions follow [45] [46] [47].

Open Copilot beside your code.

By Alvin

Visual Studio Code, or VS Code, is Microsoft's editor for project files. GitHub Copilot adds the coding assistant. In an agent session, it can edit files and run tools in response to your request. Prepare the local idea-shelf folder as on page 9. [48] [49]

1 / Install and connect

Install VS Code from its official site. Use the Copilot control in the status bar to enable AI features and complete the GitHub sign-in flow for your Copilot account. An eligible account can use Copilot Free; access and AI-credit allowances depend on the plan and organization. [48]

2 / Open the workspace

In VS Code's File menu, open idea-shelf, then open Chat. Select Copilot as the session target, Agent as the role, and a model or Auto. These choices select the assistant, its tool-using workflow and its AI model. Microsoft's quickstart uses Manual permissions to let you review proposed actions. [49]

3 / Ask for a plan you can inspect

Write: "Plan a single-file Idea Shelf in this folder. Explain the first screen, the fields each card needs and three checks." Confirm that the files and tools match your intended local project.

> **THE WORKBENCH AND THE MODEL**
>
> The Copilot session supplies the tools and workflow; its selected model helps decide the next action. Your result is the app you can open and check.

Use the signed-in Copilot service with your local project folder. Official setup and workflow: [48] [49].

Make the change. Inspect the result.

Ask the agent to create index.html. It is a browser page: HTML arranges its content, CSS controls its appearance and JavaScript makes Add and Remove work.

Build Idea Shelf as one index.html file with embedded CSS and JavaScript. Let me add a title and note, read ideas as cards and remove a selected card. Start with three fictional cards. Use labeled fields, visible keyboard focus and a phone-friendly layout. Render input as plain text. Keep resources embedded. Explain how to open the file and check adding, an empty title and removing one card. Show the changed file and the checks actually run.

Use the editor's review tools

Read the changed-file view, which shows the edits. Find index.html in idea-shelf and open it in your browser as on page 9, or use the editor's integrated browser when offered. Add an idea, try an empty title and remove a card. Tab through the controls: visible keyboard focus shows where your next key goes. [49]

Change the experience

Ask: "Adapt this shelf for a club choosing its next group project. Add a presentation view, audience and purpose on each card, and a way to record the selected project." The program must now store those added fields and the selection. The interface must show the same choice in both views.

> **WHY THE OUTPUT CHANGES**
>
> The useful output becomes a decision a group can discuss and keep. Test it with three fictional proposals, choose one, and confirm that the choice remains clear as you switch views.

Reader exercise: record the result in your own setup. Microsoft's setup and first-task guides were checked November 13. [48] [49]

ADVERTISEMENT / CODEFARMER

PatchRig FX

Build your sound. Play your rig.



Create playable modular effects rigs for your instruments and reference audio. Shape pedals, amps and DSP blocks, then audition and share your patches.

Modular audio effects for iPhone and iPad

Explore the app

View on the App Store

How altering the prompt affects the output.

Learn how altering the prompt affects the output. Your brief and context guide the code an agent generates. With tools, the agent can write files, run the program and inspect its behavior.

Read it as a set of decisions

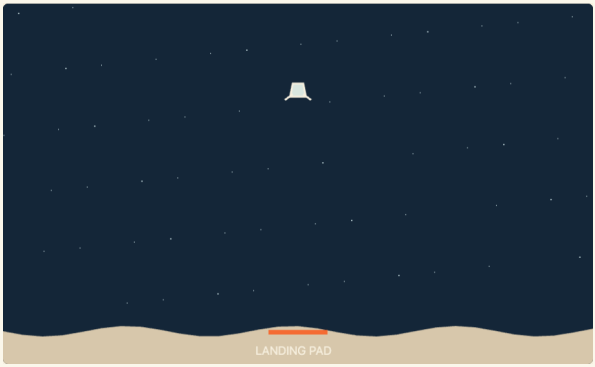
An equation gives the agent a calculation. A button needs both a visible control and code that responds when pressed. Saved data needs code to write it and a validator to check it before reuse. These choices connect the brief to the program; you judge them by trying the requested behavior.

Clear requirements and concrete examples reduce ambiguity; testing the output establishes what the result actually does. Both Anthropic and Google recommend specific instructions and evaluation in context. [35] [36]

> **MAKE A SECOND COPY**
>
> Keep the original brief and program. Change one intended behavior in a copied prompt, name any dependent updates, and ask your agent for a separate build. Run the same starting case in both. Different agents or runs can choose different code.

Follow a prompt clause into a running feature. Then change the audience, purpose or workflow and see what the agent would need to build: controls, data, program structure and checks. Each lesson identifies expected effects and results already tested.

Lunar Lander



The supplied game: a craft above the orange landing pad.

Start by playing. Lunar Lander is a browser game: gravity pulls a spacecraft down, and its engine slows the fall. Your eventual challenge is a gentle touchdown on the orange pad. Your first success is simpler: launch, try the engine, pause and restart.

Play the supplied Lunar Lander

Download the original program files (ZIP)

> **KEEP A LOCAL COPY**
>
> On a Mac, open the downloaded ZIP to extract it. Open its programs folder, find 001-lunar-lander.html, then right-click and choose Open With and your browser. The supplied game is ready to play; building your own copy comes on page 24.

You should see Lunar Lander, “Ready for launch,” altitude 70.0 m and fuel 100%. The game has all its resources in one file and runs locally. The next page walks through your first flight.

Try the engine. Watch what changes.

Open the supplied game from page 22. Read these steps, then launch. Use the on-screen buttons first; a mouse can hold them too. Leave rotation alone for this first try.

1 / Watch a short fall

Click Launch / restart and watch for about a second. Altitude is height above the ground. Vertical is speed up or down: a negative number means falling. Fuel should still read 100%.

2 / Hold the engine, then release

Press and hold Hold to thrust for about a second, then release. Fuel decreases. Watch Vertical move toward zero; if it becomes positive, the craft is climbing. The engine changes your speed gradually, so the craft can keep falling during a burn.

3 / Pause and start again

Click Pause. The craft and its readings should stop changing. Resume continues the flight. Launch / restart begins a fresh flight from 70 m with full fuel. If “Flight ended” appears, use that same restart button for another attempt.

> **WHEN YOU TRY THE KEYS**
>
> After Launch, Space or up arrow fires the engine; left/right arrows rotate. After clicking Resume, click the dark playfield before using those keys. Otherwise Space may press the focused Pause button instead. On a narrow screen, scroll to the on-screen flight controls.

First goal: see the engine alter the fall and confirm pause/restart. Safe landing is a later challenge. These actions were rehearsed in a controlled browser check on November 15; an actual newcomer study remains future work.

Make your own copy.

Now use the complete prompt on page 25 to ask a coding agent to build the game. The agent uses your instructions to write a file; your browser runs that file. Keep the supplied game as your comparison.

1 / Give this project its own folder

In Finder, make a folder named lunar-lander in Documents. Open that folder in the coding workbench you use. Pages 9–19 explain setup and sign-in for five tools; choose one route. Use a tool that can write files in the selected folder.

2 / Send the whole build prompt

Select all six paragraphs on page 25, copy them and paste them into the agent's chat box, then send. The local web edition also has Copy prompt. Review proposed actions for this project folder. If the agent only shows code in chat, ask: "Save the complete program as 001-lunar-lander.html in this folder and tell me its path."

3 / Open what the agent actually saved

Open the saved HTML in your browser. A new build may choose its own title and button labels. Try launch, thrust, pause and restart using its equivalent controls. The signed readings on page 23 describe the supplied game; if yours shows only speed, watch altitude too. Page 33 explains that difference.

> **IF THE FIRST RESULT NEEDS HELP**
>
> If you see source text, check that the saved file ends in .html, then ask the agent to save the complete browser file. If a control fails, report the starting screen, action, expected response and what happened. Ask it to fix the file and repeat that check.

Check the saved file before changing it. Page 28 offers one-clause experiments; page 27 is a larger coach project. Our fresh build and tested follow-up appear on pages 30–33.

Give this to your agent.

Build an original lunar-landing game in one self-contained HTML file named 001-lunar-lander.html. Use embedded CSS and JavaScript, canvas graphics and system fonts. All resources must be in the file.

Model a craft in a 100-metre-wide world, starting at x=50 and altitude=70, with zero velocity and 100 units of fuel. Use gravity 1.62 m/s^2, engine acceleration 4.8 m/s^2 and fuel consumption 10 units per second. Clamp a physics step to at most 0.05 seconds.

Left and right rotate the craft; Space or up arrow fires the engine. Add holdable touch buttons, Launch/restart and Pause/Resume. Clear held input on blur or pointer cancellation and pause when the player switches away.

A landing succeeds only at x=45 through 55, altitude zero, vertical speed at most 3 m/s, horizontal speed at most 2 m/s and absolute tilt at most 0.18 radians. Other ground contacts or crossing the side boundaries end the flight. Fuel must stay non-negative.

Show altitude, vertical speed, fuel and clear ready, flying, paused, landed and crashed states. Explain the controls. Make buttons keyboard accessible and keep the page usable at 390, 540 and 1000 pixels wide.

Separate the physics from drawing so the rules can be tested. Check a gentle landing, each crash condition, an empty tank, pause/resume and restart. Run browser interaction checks. Report the tests actually run and any remaining limitations.

Read the brief. Find the machine.

The prompt names the world, controls and landing rules. HTML is the page; CSS gives it its appearance; JavaScript provides its behavior. Canvas is the drawing area. The agent writes handlers that respond to a key or button and updates the picture from the flight numbers.

PROMPT CLAUSE

Use gravity 1.62 m/s^2, engine acceleration 4.8 m/s^2 and fuel consumption 10 units per second.

The motion calculation changes velocity, or speed with direction, over time. Gravity pulls downward; the engine accelerates along the craft's tilt and spends fuel. These values must reach the same update that advances the flight.

PROMPT CLAUSE

A landing succeeds only at x=45 through 55

x is horizontal position in metres: 50 is the middle of the 100-metre world. A touchdown between 45 and 55 still needs safe speed and tilt. The orange drawing and the success check must use the same pad limits.

PROMPT CLAUSE

Separate the physics from drawing so the rules can be tested.

Calculate motion separately from painting the screen. A check can then supply known starting numbers, advance one step and inspect the result. Repeating an empty-tank case should produce the same fuel and motion result.

READ IT AS A REQUIREMENT

A long physics step can skip over important motion; the brief caps it at 0.05 seconds. "Blur" means the window loses focus. Releasing held input then prevents a stuck thrust button when you return. These requirements protect behavior as well as the picture.

Build a coach for your next landing.

Make a lander-coach folder and copy your working HTML file into it. Open that folder in the agent and ask it to edit this copy using the brief below. Your original stays in its first folder. The change turns the landing challenge into a lesson.

EDITED BUILD BRIEF
Keep the original physics, landing conditions, one-file format and accessible flight controls. Reframe the program as a teaching instrument for a first-time player. Let the reader choose a vertical-drop scenario, pause and advance one physics step, and see velocity and thrust directions. After touchdown, explain which landing conditions passed or failed and offer a replay of the final five seconds. Keep a quick restart. Verify that the explanation and replay agree with the actual flight.

HOW IT FEELS
The reader gets a coach that makes each landing understandable and gives the next attempt a clear goal.

WHAT THE AGENT NEEDS TO BUILD / INTERFACE
Scenario choice selects a starting flight. A step button advances one calculation while paused. Direction arrows and an after-flight explanation make the journey choose, try, inspect and retry.

WHAT THE AGENT NEEDS TO BUILD / STRUCTURE
The agent needs one reusable motion calculation, a record of the current flight and a short list of recent flight records for replay. The debrief must use the same pass/fail landing test as the game. That keeps the explanation tied to what happened.

ACCEPTANCE CHECK
Ask the agent to set up a repeatable gentle contact outside the pad. The explanation should identify position as the failure. Then try a flight: step and replay must agree with the recorded flight.

Purpose changes the feedback loop, the data the program must retain and the checks it needs. The physical constants can remain identical.

DESIGN EXERCISE: try a new build from this edited brief. Status: proposed; implementation and testing pending.

Change the brief. Change the flight.

For either experiment, copy your working game into a separate folder, open that folder in the agent and ask it to edit that copy using the exact clause change below. Keep the original. Expected effects are predictions; test the revised file before calling the change successful.

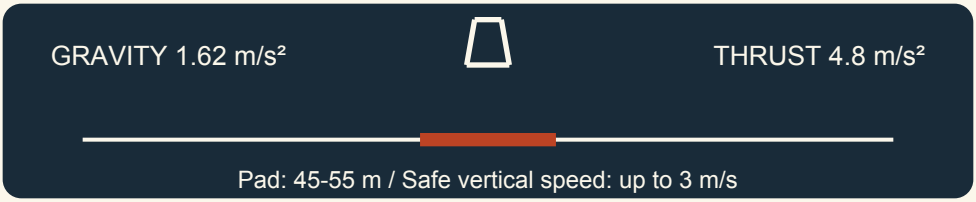
MORE GRAVITY, SAME ENGINE
BEFORE: Use gravity 1.62 m/s^2
REPLACE / ADD: Use gravity 3.24 m/s^2
EXPECTED: The craft gathers downward speed faster. You need more braking effort and the same fuel supply becomes more demanding.
WHY: Acceleration changes velocity over time. The upright engine's net upward acceleration falls from 3.18 to 1.56 m/s^2.
CHECK: Ask the agent for an automated check: from the same coasting start, advance 0.05 seconds. The vertical-speed change should be -0.162 m/s instead of -0.081. Also ask it to check powered flight and show what actually ran.
KEEP IN STEP: Ask the agent to update the flight-manual label as well as the physics. Keep the original engine and fuel rates.

A WIDER WELCOME
BEFORE: A landing succeeds only at x=45 through 55
REPLACE / ADD: A landing succeeds only at x=40 through 60
EXPECTED: A gentle touchdown at x=42 becomes eligible to land successfully. Speed and tilt still matter.
WHY: The accepted horizontal interval becomes twice as wide. The forces on the craft stay the same.
CHECK: Ask the agent for two controlled checks: upright, low-speed contacts at x=42 and x=38. With the wider pad, the first should land; the second should crash. Then play the saved copy and inspect the wider orange pad.
KEEP IN STEP: Request a pad drawn from 40 to 60 too. A changed rule with the old picture would mislead the player.

WHAT WE CHECKED / REFERENCE IMPLEMENTATION

Reference-code check: doubling only gravity changes a 0.05-second coast from -0.081 to -0.162 m/s. This tests the numerical effect in the reference code. A freshly generated version needs its own run.

A gentle arrival is a design choice.



After the first-flight exercise, try landing. With the craft upright above the pad, use short burns to slow its descent. Aim for a Vertical reading between -3 and 0 near the ground. Releasing thrust leaves the craft moving; gravity then makes it fall faster again.

Read the landing rules

x is position across the 100-metre world: 50 is the middle. A safe touchdown needs x from 45 to 55, downward speed at most 3 m/s, sideways speed at most 2 m/s, and tilt within about 10.3 degrees. Those numbers are rules the program checks; the first upright flight lets you concentrate on descent.

Ask for useful feedback

"Add a small landing checklist that shows pad alignment, speed and tilt. Keep the flight rules unchanged. Check that the indicators agree with the actual touchdown decision."

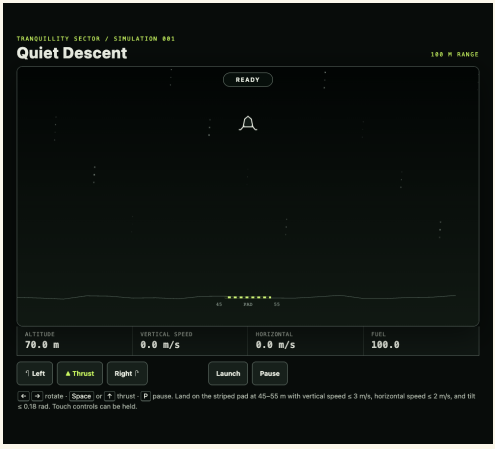
WHAT PASSED

14 logic checks cover gravity, thrust, touchdown conditions, fuel, side boundaries and limiting a large time step. Seven browser checks cover launch, keyboard and pointer thrust, pause/resume, switching away from the window and restart. Shared layout/offline checks also passed.

The same prompt. A fresh game.

By Alvin

We gave Codex the exact six paragraphs on page 25 in an empty project folder. It received no reference game, screenshots or our test cases. Standard tool and safety instructions still applied. It saved one self-contained HTML game, naming the screen Quiet Descent.



Actual first completed build, preserved before our independent tests.

INDEPENDENT CHECK	FIRST FILE
Numerical flight cases	13 / 13 passed
Rendered-browser cases	16 / 18 passed
Layouts within those cases	390, 540, 1000 px fit

One completed generation: Codex CLI 0.144.0, default GPT-5.6-Sol, 144 seconds including its checks. CLI ran on the M1; model access used the account service. We tested with Chromium 157.0.8103.51. These are results for this file and these cases.

The agent reported simulated page checks. Our separate browser run found two keyboard-button failures. We preserved that first file, then gave a copy and a follow-up prompt to a new isolated agent session. The next pages show the repair and comparison.

FOLLOW-UP / EXACT PROMPT WE SENT

Describe the failure. Ask for the repair.

By Alvin

A follow-up prompt is another instruction to the coding agent, with the existing program as its starting point. We supplied a copy of Quiet Descent and sent the complete text below. Keep your first file too, so you can compare the result.

Focus selects a button for the next key press. A pointer hold keeps a mouse button or finger pressed. Keydown and keyup are press and release events; window blur means switching away from the game window.

Update the existing 001-lunar-lander.html. In the current build, holding Enter on the focused Rotate left button leaves the craft upright, and holding Enter on the focused Fire engine button uses no fuel. Fix these keyboard controls.

Make all three flight buttons respond while Enter or Space is held on the focused button, just as they respond to a pointer hold. Release that input on keyup or loss of focus. Repeated keydown events must keep one hold active. Preserve the global arrow-key and Space shortcuts when no flight button is focused. Keep Launch/Restart and Pause/Resume working with their normal keyboard activation. Handle window blur, pointer cancellation, and overlapping keyboard/pointer holds so controls never remain stuck or release another active input.

Keep the physics, landing rules, speed readout, layout, and offline single-file format unchanged. Check the repaired controls and existing flight behavior. Report which checks you actually ran and which remain untested.

Executed November 15, 2026: one repair run, Codex CLI 0.144.0, default GPT-5.6-Sol. A new isolated session received the failed file and this exact prompt. We kept our independent test harness separate from the agent.

Repeat the check. See what changed.

Keyboard focus is the button selected for your next key press; a visible outline marks it. Before: Enter on Rotate left produced no turn, and Enter on Fire engine used no fuel. After: both held actions worked and stopped on release.

SAME CHECKS	BEFORE	AFTER
Flight-rule logic	13 / 13	13 / 13
Original browser cases	16 / 18	18 / 18
Added input cases	9 / 21	21 / 21
Total passed	38 / 52	52 / 52

The added cases cover Enter and Space on all three flight buttons, release, repeated key presses, focus changes and overlapping keyboard/pointer holds. They also check ordinary Launch, Pause, Resume and Restart activation. Both files use the same checks.

TRY THE BEFORE-AND-AFTER YOURSELF

Open each experimental file in a browser. Launch. Press Tab until Left has an outline; hold Enter, then release it. Repeat on Thrust and watch fuel. The repaired file also accepts Space on the outlined flight button. Click the flight area before using Space as the global engine shortcut.

Open the preserved first build. File in the companion pack:
lab/fresh-lander-2026-11-15/first-build/001-lunar-lander.html

Open the agent-repaired build. File in the companion pack:
lab/keyboard-followup-2026-11-15/after-build/001-lunar-lander.html

Independent automated Chromium checks: 390, 540 and 1000 px, offline reload and zero page errors. Global Space uses flight-area focus; Space on Launch keeps normal button activation. The agent ran a simulated event harness. Physical touch, other browsers, screen readers and a human beginner trial remain separate checks.

Learn how altering the prompt affects the output.

A useful action becomes a control rule

“While Enter or Space is held” asks for a continuing action. The agent adds key-down and key-up handling: press starts rotation or thrust; release stops it. “On the focused button” makes Space on Left turn the craft.

An edge case becomes stored information

“Overlapping keyboard/pointer holds” changes how the program remembers input. The repair tracks each source of a hold separately. If your finger still presses Thrust when you release Enter, the engine keeps running. It stops when the remaining hold ends. A single on/off flag could lose that distinction.

A boundary keeps the experiment useful

“Keep the physics” makes this a control repair. Source comparison confirmed identical flight rules, styling and readout/rendering code. The agent changed the input handlers. Logic checks still pass; browser checks establish that the new interactions work in our tested cases.

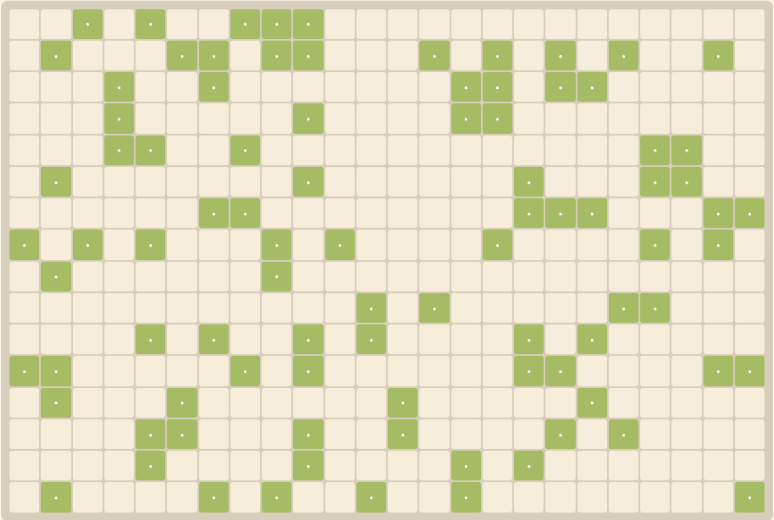
Choose the next change on purpose

This build displays vertical-speed magnitude: altitude shows whether you are falling or climbing. A separate prompt can ask for direction explicitly and change the information the reader receives:

> **PROPOSED NEXT EXERCISE / READOUT**
>
> Show signed vertical velocity: negative while falling and positive while climbing. Add a clear Falling or Climbing label. Keep the original landing limits and check the readout against the actual motion.

The readout exercise is proposed; we ran the keyboard repair only. One successful follow-up establishes the behavior of this saved file. Your own generated build needs its own checks.

Digital Garden



Actual editorial program: living cells, fixed edges and room to explore.

A garden of cells is a lovely way to see rules become behavior. Each square is either empty or alive. Count the living neighbors, apply one rule everywhere and the next generation appears.

A two-by-two living square stays still; a line of three cells alternates between horizontal and vertical. Reseed returns a repeatable opening: its seed is a number used to make the starting arrangement. With the same number and method, you can repeat the experiment.

RUN IT

Open 002-digital-garden.html in a browser. Tap cells to plant. Grow runs generations; One generation takes one step. Clear empties the bed. Reseed restores the repeatable opening. A generation means one update of every square from the previous arrangement.

The dot marks a living cell; green shade shows its age. The text notebook stores the arrangement in JSON, a structured text format. “Validate” means check its fields and values before accepting a restore. Copy garden fills the notebook; save that text yourself to keep it.

Grow a world from a rule.

Build Digital Garden in one self-contained HTML file named 002-digital-garden.html. Use embedded CSS and JavaScript, system fonts and a 24-column by 16-row grid of accessible toggle buttons.

Use Conway-style rules: a living cell survives with two or three living neighbors; an empty cell becomes alive with exactly three. Cells beyond the fixed edges are dead. Store age from 1 through 9 for living cells, capped at 9; zero means empty. Use color for age and a visible dot for life.

Create a deterministic initial garden from seed 1974, with about 27 percent living cells. Add Grow/Pause, One generation, Reseed and Clear. Growing advances every 350 ms. Pause while editing, restoring, switching away or focusing the notebook.

Let people click or tap cells. Support arrow-key focus movement and Space to toggle. Keep the page itself within a narrow viewport; allow the garden alone to scroll sideways so cell targets remain usable.

Add a text notebook. Copy garden writes JSON with version=1, width=24, height=16 and 384 integer cell ages. Restore must validate every field and keep the current garden unchanged if validation fails. Explain that readers save the notebook text themselves.

Separate the simulation rules for testing. Check an empty garden, a stable block, a period-two blinker, fixed boundaries, reproducible seeds, valid save/restore and malformed input. Test the controls and narrow-screen layout in a browser. Report actual results.

A sentence becomes a world.

The grid is the stage. The neighbor rule is the program's little law of nature. The seed chooses the opening scene. Edit the seed to change the opening scene; edit the neighbor rule to change how that scene evolves.

PROMPT CLAUSE

a living cell survives with two or three living neighbors; an empty cell becomes alive with exactly three.

Count each cell’s eight neighboring squares, then calculate a separate new grid from the old one. Changing the old grid while still counting would let one early change affect later cells in the same generation.

PROMPT CLAUSE

Cells beyond the fixed edges are dead.

Only squares inside the grid count as neighbors; everything beyond it is empty. A wraparound version instead treats the far edge as adjacent. The same three planted cells can then produce a different birth near a corner.

PROMPT CLAUSE

Create a deterministic initial garden from seed 1974

The seed is an input to a repeatable number generator. The same seed and method produce the same opening cells, so a rule comparison can begin from identical gardens.

> **READ THE SAVED DATA**
>
> JSON stores named values and lists. This notebook records a format version, width, height and one age per cell. Validation checks every field before replacing the current garden. A changed grid size needs matching updates to all those checks.

Turn a garden into an experiment.

The supplied garden makes a rule visible and lets a reader plant cells. Calling it a laboratory asks for more than giving that grid a scientific-looking color scheme.

EDITED BUILD BRIEF
Keep the grid, original rules, fixed edges and keyboard access. Build a classroom experiment bench around them. Give the reader named Block and Blinker starting cases, a one-generation prediction question, step-by-step results and a way to compare an altered starting pattern with its saved baseline. Show live-cell count by generation and export the experiment settings with its observations. Make reset return to the named starting case. Test that both comparison runs begin from the recorded cells.

HOW IT FEELS
The reader forms a prediction, runs an experiment and compares evidence. The garden becomes an instrument for a question.

WHAT THE AGENT NEEDS TO BUILD / INTERFACE
Named cases, prediction, side-by-side results, a population trace and an experiment export support that journey. Together, these features turn a growing pattern into an experiment the reader can repeat.

WHAT THE AGENT NEEDS TO BUILD / STRUCTURE
Store the named starting cells, a separate unchanged baseline and a trial copy. Save a history of live-cell totals after each generation. The export must identify the rule and edge behavior so another reader can repeat the experiment.

ACCEPTANCE CHECK
A Blinker is three live cells in a straight line. Run it twice from the same stored case: both should return to their starting occupied squares after two generations. Change one trial cell and confirm the baseline stays intact.

The useful output has changed from an attractive moment to a reproducible experiment. That change reaches storage, reset behavior and the user journey.

DESIGN EXERCISE: try a new build from this edited brief. Status: proposed; implementation and testing pending.

Change one rule. Watch a new garden.

Build a variation from the edited prompt and compare it with the supplied program. The expected effects below are predictions for that build; testing a fresh agent-generated version is your next experiment.

JOIN THE OPPOSITE EDGES
BEFORE: Cells beyond the fixed edges are dead.
REPLACE / ADD: Wrap both axes: a neighbor beyond the left edge comes from the right, and a neighbor beyond the top comes from the bottom.
EXPECTED: Patterns can affect cells across a border: left joins right, and top joins bottom, like a grid drawn on a doughnut-shaped surface.
WHY: Neighbor counting looks across the joined edges. The birth and survival counts remain the same.
CHECK: Coordinates count from zero: (0,0) is the first column and row. Plant (23,15), (0,15) and (23,0). After one generation, (0,0) should be born only in the wrapped version.
KEEP IN STEP: Keep the other requirements unchanged.

A DIFFERENT OPENING, THE SAME LAWS
BEFORE: from seed 1974
REPLACE / ADD: from seed 1975
EXPECTED: Reseeding begins from a different arrangement. A hand-planted block or blinker should behave as before.
WHY: The seed chooses the starting cells. The neighbor rule and edge policy continue to govern every generation.
CHECK: Generate seed 1975 twice and compare the stored cell-age lists. They should match each other and differ from seed 1974.
KEEP IN STEP: Keep the seed algorithm and initial density unchanged. A different algorithm may interpret the same seed differently.

WHAT WE CHECKED / REFERENCE IMPLEMENTATION

Reference-code check: the three corner cells produce a birth at (0,0) with wrapped lookup and none with fixed edges. Identical seeds also reproduce the supplied opening.

Plant a question.



BLOCK / STILL



BLINKER / STEP 1



STEP 2

Clear the garden and plant a two-by-two square. Advance one generation. It stays put: each living cell has three neighbors. Now plant a line of three cells away from the edge. It flips between horizontal and vertical. You have made a period-two oscillator.

The edge is part of the world

This edition treats every cell beyond the grid as empty. That choice matters. A pattern near the boundary behaves differently from one in the middle. Ask your agent to add a wraparound option, then compare the same starting pattern in both worlds.

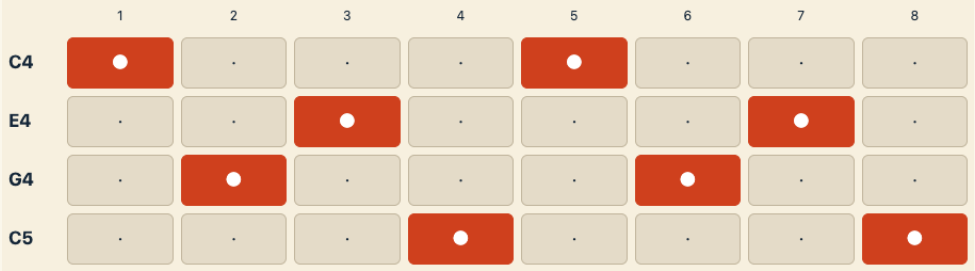
Keep an experiment notebook

Copy the garden data before you grow it. Add your prediction to a separate note. Run ten generations, restore the starting text and try again. Reproducibility turns a pretty pattern into an experiment.

> **WHAT PASSED**
>
> 11 logic checks cover repeatable seeds, still life, oscillation, boundaries, valid ages and checking restored notebook data. Seven browser checks cover the grid, editing, stepping, notebook recovery, focus movement and growing.

On small screens the garden scrolls sideways, preserving usable cell targets.

Pocket Sequencer



Actual editorial program: four notes, eight steps and a little musical space.

A sequencer repeats a musical pattern. Each column is a step in time; a dot means sound that row's pitch on that step. C4, E4 and G4 name notes in one octave; C5 is the higher C. Notes sharing a column sound together as a chord.

Press Play, then remove some notes and listen to the spaces. Tempo is speed in beats per minute, or BPM. At 120 BPM, a beat lasts half a second. Each step here is half a beat—an eighth note—so eight steps make a two-second loop.

RUN IT

File: 003-pocket-sequencer.html
Press Play to start audio. Tap steps to toggle notes. Stop ends playback; Load melody restores the example. The tempo ranges from 40 to 240 BPM.

The browser creates the sound. A short attack and release fade each note in and out; gain controls its level. Play supplies the user gesture browsers require to start audio. Set a comfortable device volume. The JSON notebook is structured text storing both pattern and tempo.

Use this instrument to explore a phrase. We checked that its sound-generating code produced an audio signal; listening-panel and physical-device reviews remain future checks.

Give an idea a pulse.

Build Pocket Sequencer in one self-contained HTML file named 003-pocket-sequencer.html. Use embedded CSS and JavaScript, system fonts and Web Audio synthesis.

Make four note rows: C4, E4, G4 and C5, each with eight steps. Each step is an eighth note. Start with a simple eight-note demo pattern and tempo 110 BPM. Allow tempo from 40 to 240 BPM. Use toggle buttons with note/step labels, pressed states and a visible active mark.

Add Play/Stop, Clear and Load melody. Create or resume audio only after the user presses Play. Use a short attack/release envelope and modest gain. Schedule ahead using the audio clock; update visual steps in time with playback. Stop scheduled notes, timers and highlights on Stop, page hiding or window blur. Show an informative error if audio cannot start.

Let the pattern scroll horizontally on small screens while the page stays within the viewport. Support keyboard operation and visible focus. Explain that device volume affects listening.

Add a JSON text notebook storing version=1, integer BPM and four arrays of eight booleans. Restore must validate the full shape and preserve the current pattern on invalid input.

Test pattern toggling, tempo, Play/Stop, non-zero rendered audio from the production note generator, save/restore, invalid input and switching away. Test 390, 540 and 1000-pixel layouts. Report what was actually verified; distinguish synthesized output from a listening review.

Words set the clock and the shape.

One loop needs timed sounds and a matching visible step. A scheduler is code that arranges when notes will play. The audio clock measures playback time; the highlight should follow it rather than the browser's irregular screen redraws.

PROMPT CLAUSE
Each step is an eighth note.

BPM counts quarter-note beats per minute. Two eighth notes fit each beat, so one step lasts 30/BPM seconds. At 120 BPM that is 0.25 seconds; the eight-step loop lasts two seconds.

PROMPT CLAUSE
Schedule ahead using the audio clock

Send each note and its future play time to the audio system slightly before it is due. Move the visible highlight when that time arrives. This connects what you hear to the column you see.

PROMPT CLAUSE
four arrays of eight booleans

An array is a list; a boolean is true or false. Four lists of eight on/off values store the four note rows. A validator checks every value and row length before restoring a saved pattern.

> **READ IT AS A REQUIREMENT**
>
> The sound, controls and saved pattern are related surfaces. A meaningful prompt change names the rule and asks the agent to keep those surfaces consistent.

Play a phrase. Cue the next one.

A pattern editor lets someone enter notes and listen. A performance brief asks how changes behave while someone is playing. That difference reaches timing and interaction design.

EDITED BUILD BRIEF
Keep the four pitches, eight-step pattern, modest synthesis gain and accessible controls. Reframe this as a small live performance instrument. Offer three named pattern slots, keyboard shortcuts and touch controls for cueing the next slot, and switch only at a loop boundary. Make tempo changes preserve the musical position. Show what is playing and what is queued. Add a clear stop-all control that silences both current and scheduled notes. Explain these controls in the interface and test them during playback.

HOW IT FEELS
The reader plays a musical phrase, cues its next move and shapes the performance as it continues.

WHAT THE AGENT NEEDS TO BUILD / INTERFACE
Playing and queued states, large cue controls, discoverable shortcuts and an immediate stop control matter more than adding rows of decorative knobs.

WHAT THE AGENT NEEDS TO BUILD / STRUCTURE
Store which pattern is playing and which is queued. The scheduler must switch at the end of a loop and arrange notes on the audio clock. Save the three slots as separate patterns, and validate all their rows and values on restore.

ACCEPTANCE CHECK
Cue a slot halfway through a loop. The old pattern should finish the loop before the new one starts. Stop while notes are scheduled and verify silence and cleared highlights.

An experience word such as performance becomes useful when the prompt spells out its interactions and timing promises.

DESIGN EXERCISE: try a new build from this edited brief. Status: proposed; implementation and testing pending.

Change the pulse. Keep the clock honest.

Build a variation from the edited prompt and compare it with the supplied program. The expected effects below are predictions for that build; testing a fresh agent-generated version is your next experiment.

A FASTER SUBDIVISION, THE SAME TEMPO
BEFORE: Each step is an eighth note.
REPLACE / ADD: Each step is a sixteenth note.
EXPECTED: At 120 BPM steps arrive every 0.125 seconds. The eight-step loop now lasts one second; the tempo display still says 120.
WHY: One beat contains four sixteenth notes instead of two eighth notes. Subdivision changes the step interval, while BPM still measures quarter-note beats.
CHECK: Check the production interval at 120 BPM, then verify playback and the active-step display use it. Check the actual interval, playback and display together.
KEEP IN STEP: Keep the other requirements unchanged.

A LONGER PHRASE
BEFORE: each with eight steps
REPLACE / ADD: each with sixteen steps
EXPECTED: The phrase has more places for notes. At 120 BPM and eighth-note steps, one complete loop lasts four seconds.
WHY: More steps extend the loop while each step keeps its duration.
CHECK: Toggle the new last step, export and restore it, then confirm playback wraps after step 16.
KEEP IN STEP: Also replace four arrays of eight booleans with four arrays of sixteen booleans: four longer on/off lists. Decide whether an old eight-step notebook gets a clear error or is extended with eight silent steps.

WHAT WE CHECKED / REFERENCE IMPLEMENTATION

Reference-code check: the supplied eighth-note interval at 120 BPM is 0.25 seconds. Replacing only the subdivision divisor gives 0.125 seconds for sixteenth notes.

Time is part of the interface.

120 BPM = 0.25 s PER STEP



At 120 beats per minute, a quarter note lasts half a second. Each sequencer step is an eighth note, so the interval is a quarter second. Eight steps therefore make a two-second loop at that tempo.

The audio clock measures playback time. The drawing clock controls screen updates, which may arrive unevenly. Scheduling notes ahead lets sound follow its timeline; the visual highlight follows the same times. Attack and release are short fades that soften each note's beginning and end.

Try a different musical question

Ask for a 16-step version and check that saving and restoring still works. Or keep eight steps and change the four pitches to a minor chord. Change one musical dimension at a time so you can hear what it contributes.

A follow-up worth keeping

"Add a swing control. Describe which intervals become longer and shorter. Preserve the total loop duration, and test both straight and swung timing before adding new sounds."

WHAT PASSED

10 logic checks cover notes, pattern shape, tempo and notebook validation. Nine browser checks cover toggles, playback, the supplied sound generator, stopping, tempo, restoration, invalid input and switching away from the window.

Solitaire 1380

Find your next favorite solitaire.



Discover a world of solitaire and patience games. Choose your favorites, explore Daily Discovery, and learn a new game with guided help.

Play on iPhone and iPad

Start with three favorites free. One purchase unlocks the collection.

Explore the app **View on the App Store**

What passed. What it means.

LOGIC CHECKS	BROWSER CHECKS
35 / 35	40 / 40

A logic check supplies inputs to the program's rules and compares the result with an expected answer. An assertion is one such comparison. The same 35 assertions also passed on a second Mac; that repeats those checks on another machine, rather than adding new cases.

Browser checks press controls and inspect the running page. In Chromium 157.0.8103.51 on macOS, they covered 390-, 540- and 1000-pixel widths, saving, restoring and errors. All three files reloaded with network access disabled. We recorded zero external web requests and zero uncaught page errors.

Where the evidence stops

We checked control names, keyboard movement and generated audio signals. Those observations leave physical Duo use, Safari, Firefox, screen readers and listening reviews for separate evaluations. They do not establish accessibility certification.

The companion includes each file's SHA-256 hash, a digital fingerprint used to identify its exact bytes. A changed file has a new fingerprint and needs a new test run. Keep the old result as the record of the old file.

Editorial programs are tested here. Reported developments are sourced. Reader experiments invite a new trial.

From an idea to a game cartridge.

Hardware gives a software idea a place to live.

Want to hold your game in your hands? OpenAI's November 9 tutorial uses Codex and the ModRetro Chromatic plugin, an add-on connecting the agent to this game-making workflow. GB Studio is the game-building tool. Its output is a .gbc game file, playable in an emulator—a software imitation of the console—or on a cartridge. [5]

That is a satisfying bridge between agentic development and the hobby-computing workbench. Follow the vendor demonstration to explore the process. Reproducing the build and testing a Chromatic remain future work for this magazine.

Start with a game that fits its home

Before choosing hardware, sketch a tiny game with one action, one obstacle and a clear ending. A room with a moving target is enough to learn how input, collision and feedback work together. Make the player's next choice obvious on a small screen.

The useful constraints

A handheld invites short sessions and readable sprites, the small images used for game objects. Settle the rules in a browser prototype first. Hardware then adds questions: is the image legible, does the button respond promptly, and can the player recover from a mistake?

A WORKBENCH ASSIGNMENT

Draw title, play and result screens. Explain each button. Build a complete loop before adding another level.

Give a display a useful job.

Sid Rampally's September account joins coding assistance to a physical display. A Raspberry Pi is a small computer; the project connects one to a 128-by-64-light HUB75 LED panel and a voice assistant. The builder used photos and repeated tests to work through wiring and the path from software to displayed pixels. [6]

The appealing detail is the division of work: software assistance met physical connections and observations from the builder. Follow the linked project for the builder's account. A magazine hardware reproduction remains future work.

Our smaller starting brief

Imagine a desk display with three views: a clock, an indoor reading and a simple animated picture. First build a browser simulator using the exact pixel dimensions. Give every view a sample input and a predictable fallback. Once the images work, the hardware task has a visible target.

Keep the first hardware session about one thing: draw a known pattern, confirm orientation and verify the update path. Then connect one real data source. A clear test image tells you more than a complicated dashboard with several moving parts.

> **THREE CHECKPOINTS**
>
> Pixels: can you draw a fixed test pattern?
> Data: can you display a recorded sample?
> Failure: what appears when an update is late?

Hardware-in-the-loop means a check involving the real connected device, rather than only a simulation. Agentic HIL on Hackaday.io is a lead for a future hands-on evaluation. We have yet to reproduce that workflow. [20]

PROMPT CLINIC

Give the agent a finish line.

The first draft

> **"Build me a cool gardening app."**
>
> The idea has energy. The scope, inputs and definition of success are still open.

The useful revision

"Build a single-file garden simulator with a 24 by 16 grid. Let me toggle cells, step once, run, pause and reset. Use the neighbor rules printed below. Save and restore the grid through validated JSON text. Check a stable block, a blinker and invalid input. Keep the page usable at 390 pixels wide."

The revision names the screen, simulation rules and controls, then gives observable checks. JSON is the structured text used to save the grid; validation checks that text before restoring it. A block stays still and a blinker alternates direction, giving the agent known examples to check.

Keep the instructions close to the task

A skill is a reusable set of task instructions an agent can load, such as a procedure for reviewing a program. OpenAI's September guidance recommends focused skills with clear descriptions and relevant supporting detail. Accumulated instructions can become irrelevant or contradictory. [7]

For your next project, list the five behaviors that matter most. Add one example of success and one example of failure. Ask the agent to propose the simplest implementation that covers them.

Keep a short decision record. A rule such as "invalid saves preserve the current garden" gives the agent a concrete behavior to protect.

Follow the test. Find the assumption.

A test can make the wrong assumption too. Our browser test pressed Grow in Digital Garden, then checked a counter using an assertion: an expected condition. The program advanced, but the check demanded a generation number above one. An earlier restore had reset that number to zero.

The test had carried an assumption from an earlier part of the session. The correction was to record the generation immediately before pressing Grow and then check that the counter increased. The correction belonged in the test; the program kept its intended behavior.

Write down the sequence

Write the starting state, action, expected result and observed result. “Restore a garden, press Grow, wait for one tick, then inspect generation” gives an investigation a starting point. A tick here is one timed simulation update; the counter should increase from its value just before Grow.

Test the behavior you care about

The sequencer check uses the same note-making code as the supplied program. It renders sound into an offline audio buffer, a block of audio samples held in memory. Finding a non-zero signal shows that code generated sound data; a separate listening review judges how it sounds.

> **A DEBUGGING PROMPT**
>
> Reproduce this exact sequence. Explain which assumption failed. Preserve the intended behavior. Show the smallest relevant correction and run the check against the corrected version.

A test is a small argument about a program. When it fails, inspect both sides of that argument.

Made for your next idea.

Encore Player

Your media. Your encore.

Your music and videos in one player. Encore Pro unlocks full video, continuous folder playback and media-server access.

App website App Store

Threadbreaker

Find a path through the fire.

Read shifting bullet patterns, charge Thread and Sever an escape route through a cosmic action roguelite.

App website App Store

Vortex Descent

Make every descent count.

Tilt, drag and boost through evolving sectors, precision gates and an arcade freefall full of tactical choices.

App website App Store

CodeFarmer Tech Reader

Keep your curiosity close.

Bring Hacker News, Slashdot and their discussions together in a focused native reader.

App website App Store

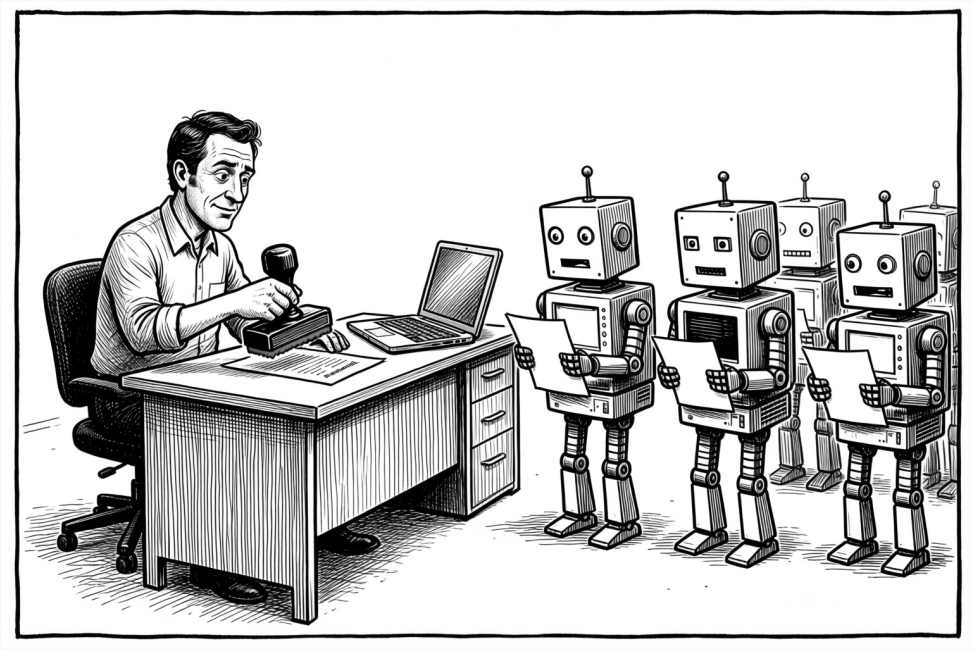
TempoTrellis: Music Coach

Turn practice into progress.

Plan focused music sessions with a timer, metronome, tempo tools and recordings of your own takes.

App website App Store

The approval queue.



“My agents have submitted a plan to reduce my paperwork.”

Cartoon by Alvin

Follow curiosity back to its source.

Hacker News and Slashdot are discussion sites where people share projects and question claims. Show HN is Hacker News' section for things people made. We follow the original project link for what a tool does, then use the comments to find questions worth testing. Modern Web Guidance became our typed-command experiment. [14] [19]

Simon Willison's tools page collects small HTML and JavaScript experiments: browser programs with source you can inspect. Its colophon is a note explaining how the work was made. The collection is inspiration for small, inspectable tools. [15] [16]

Addy Osmani's writing and the Exploring Generative AI collection on MartinFowler.com provide longer discussions of workflows, code review and engineering judgment. Read the author's context as carefully as the technique. A practice that helps a large team may add needless ceremony to an evening project. [17] [18]

Our editorial question

What can a reader do with this? A new tool might become a dispatch. A repeatable technique might become a lab. An interesting failure might become a debugging article. A delightful small program might inspire an original prompt-in.

> **KEEP THREE THINGS TOGETHER**
>
> The original source and its date.
> The claim worth examining.
> The experiment that could test it.

Explore five community discoveries from page 55 and our hands-on lab on pages 60 and 61.

Huzzah: keep the plan beside the program.

FOUND VIA HACKER NEWS / Read the discussion
PROJECT: Huzzah
Reported; hands-on test pending
Author's August 2026 experiment; revisited for this issue

You return to a small app weeks later and wonder why it behaves that way. Daniel Vaughn's experimental editor, Huzzah, keeps a human-written description alongside generated source code. That source is the program text a computer runs. The description records what the person wanted it to do. [21] [22]

Write the behavior in your own shorthand

Huzzah uses pseudocode: readable steps that describe behavior rather than the exact syntax of a programming language. Vaughn's August account describes saving changes to that description as the next instruction to the model. The description is meant to remain available after a chat has moved on. [21]

An example to imagine

For a pocket sky viewer, write "click a star to save a note." Revise it to "click a star to compare it with the previous selection." That purpose now calls for remembering two selections and displaying a comparison. You can inspect whether the changed program matches the changed intention. This is our example, rather than a Huzzah test.

> **TRY THE IDEA**
>
> Write ten lines describing a tool you want. Ask someone to explain its behavior back to you, then revise one ambiguous line. Huzzah remains experimental; its author identifies open questions about larger projects and relationships across files. We have read the project and have yet to run it.

Local-coder: build and check on your Mac.

FOUND VIA HACKER NEWS / Read the discussion
PROJECT: Local-coder
Reported; hands-on test pending
Experimental project; source and discussion checked November 12, 2026

Imagine continuing a coding project during a gap in internet access. Local-coder configures a setup for models running on your own computer. Ollama runs the downloaded models; OpenCode is the coding workbench that can use their replies to edit files and run tools. Local-coder connects and configures those pieces. [23] [24]

Give the work two distinct jobs

The project describes a coding role and a checking role, with additional planning and review roles as needed. A role is an assignment to the system, rather than a guarantee of a separate expert. Its setup also tests capabilities: can a chosen model request a tool correctly and make a change that can be verified? [23]

A garden example

Ask the coding role to add a Reset button that restores the original cells. Give the checker the starting pattern and ask it to run, reset and compare every cell. The shared object is the program and its saved starting data. That gives the checking role something more specific than judging a completion message.

> **BORROW THE WORKING METHOD**
>
> Separate “make the change” from “show that it works.” Use one normal case and one awkward case, such as Reset while the garden is growing. Downloads and setup still need a connection. We have read Local-coder; installation, hardware suitability and a working reproduction remain untested here.

Stroq: follow an agent's recorded actions.

FOUND VIA HACKER NEWS / Read the discussion
PROJECT: Stroq
Reported; hands-on test pending
Source and discussion checked November 12, 2026

A coding agent reads a page, then proposes an unexpected network command. Where did that instruction come from? Stroq examines recorded agent activity. For supported integrations, it can connect a fragment the agent encountered with a later action. That is evidence of a connection; it cannot reveal the model's private reasoning. [25] [26]

Understand the three jobs

An audit log records events for inspection. A hook is a small check called at a supported point in a tool's workflow; it may filter a proposed action. A sandbox is a separate execution boundary restricting what a running program may access. A record of a dangerous command and a boundary that prevents it are different capabilities.

The Hacker News discussion challenged the value of a warning after a command ran. The creator distinguished reports from installed hooks and described hooks as filtering, with sandboxing outside their role. The project documents integration-specific coverage. [25] [26]

A PROPOSED LAB

Use an invented session: a file read, an instruction hidden in its content and a proposed network request. Compare what is recorded, what a hook interrupts and what the execution boundary permits. This is a future experiment; we have yet to test Stroq.

Scope: can an agent use what you made?

FOUND VIA SLASHDOT / Read the discussion
PROJECT: Scope
Reported; hands-on test pending
Microsoft introduction October 19; Slashdot discussion October 25, 2026

You publish a small library that supplies weather data. Can a coding agent find its instructions and use it correctly? Scope is Microsoft's evaluation platform for studying that sort of task. You choose a task and success criteria; it records agent runs so you can inspect the steps behind a result. [27] [29]

Make success visible

For our imagined library, supply a fictional forecast and ask for a display. Check whether the agent imports the right function, handles a missing temperature and produces the expected screen. A failed step may point to an unclear example or error message in your own software. This is an illustrative assignment.

What its setup involves

Scope describes workers packaged in containers: software environments holding the tools they need. Kubernetes manages those workers across machines. That infrastructure supports running experiments at scale; deploying it is a separate task from choosing a good question. We have read the documentation and have yet to deploy Scope. [27]

> **START WITH THE METHOD**
>
> The Slashdot discussion questioned the long technology stack. [28] You can first practice the method manually: write one task and its expected result, keep the starting files, and record what the agent actually does. That is our proposed exercise, rather than a Scope run.

OpenClaw: choose a useful assistant job.

FOUND VIA SLASHDOT / Read the discussion
PROJECT: OpenClaw
Background report; hands-on test pending
March 9 discussion revisited; current project documentation checked November 12, 2026

What would make a personal assistant worth keeping? Consider a helper that reads a watering log and drafts tomorrow's plant checklist. OpenClaw is a system for connecting an assistant to tools and messaging services. Its gateway is the service that connects those channels and coordinates the assistant's work. [30]

Start with an inspectable task

For a first experiment, use a made-up log: fern watered Monday; cactus watered Friday. Ask for a checklist and the entries supporting each suggestion. Compare its answer with your notes. You are choosing the information the helper can read and the actions it can take. This is our proposed exercise.

Access is part of the design

A helper allowed to draft a checklist has different authority from one allowed to send messages or change files. Name those permissions explicitly. Our discovery came from a March 9 Slashdot discussion of a KrebsOnSecurity report about assistant access; it is historical background revisited for this issue. [31] [32]

> **KEEP THE FIRST JOB CLEAR**
>
> Specify the input, useful output and stopping point. For the plant example: read the fictional log, draft a checklist, then wait for review. We have read OpenClaw's documentation; we have yet to run it, connect messaging accounts or give it personal files.

Web advice, tested on four small pages.

FOUND VIA HACKER NEWS / Read the discussion
PROJECT: Modern Web Guidance
Tested locally: CLI only, version 1.0.0
November 12, 2026 / macOS arm64 / Node 24.22.1

Modern Web Guidance supplies browser advice for coding agents and typed commands for finding that advice and scanning files. A scan looks for particular code patterns; it is one part of reviewing a working page. We tested version 1.0.0 on four tiny HTML pages, called fixtures. [33] [34]

EDITORIAL FIXTURE	SCAN RESULT
Text box resized by JavaScript, no support check	1 finding
Resize code with a browser-support check and fallback	0 findings
Save button that does nothing	0 findings
External script: fixed version, no content fingerprint	1 finding

A fallback is an alternative when a browser lacks a feature. Script integrity metadata supplies a fingerprint for checking a downloaded file. Search found the growing-text-box guide first; retrieval returned its advice. The scanner flagged the unchecked resize and missing script fingerprint.

> **WHAT ZERO FINDINGS MEANS**
>
> Our nonfunctional Save button also received zero findings. The scanner did not establish whether Save worked. Try the page's actual controls as a separate check.

Seven checks passed; the scans reported 35 active rules across 33 guide topics. We checked the package digest (its digital fingerprint). These are narrow observations from four files, rather than a broad application benchmark.

Run the scan. Read it with context.

First install Node.js with npm as explained on page 16. This pinned tool needs Node 22+. Download and extract our community-lab ZIP from the magazine's web edition. In Terminal, type cd and a space, drag the extracted folder into the window and press Return. It should contain fixtures; confirm that before running these lines.

```
export DISABLE_TELEMETRY=1
npx modern-web-guidance@1.0.0 search \
  "textarea automatically grows to fit content"
npx modern-web-guidance@1.0.0 retrieve \
  form-fields-automatically-fit-contents
npx modern-web-guidance@1.0.0 legacy-scan ./fixtures --json
```

npx downloads and runs the named package; @1.0.0 selects our exact release. Search finds advice; retrieve opens a named guide; legacy-scan checks ./fixtures, the fixtures folder inside your current folder. --json requests structured text output. The first line disables usage reporting for this run. A trailing backslash continues the same command on the next printed line.

Turn a finding into a useful change

Read the explanation, inspect the affected file, then test the interaction in the browsers your readers use. A browser-support check may choose a fallback. Zero findings means no reported match from this scan, rather than a working-program guarantee.

> **A BRIEF TO TRY**
>
> Explain these findings for our target browsers. Propose one small revision, preserve any needed fallback and list the behavior checks required to accept the result.

Recorded run: macOS 26.6.2, Apple Silicon, Node 24.22.1, tool 1.0.0. We tested the CLI; model comparisons, Safari and physical Duo checks remain separate work. [33] [34]

The comments supply the question.

Browser advice can describe a feature your reader's browser lacks. In the Modern Web Guidance discussion, surround questioned support for recommended features. Philip Walton replied that guides include compatibility information—where a feature works—and often a fallback, an alternative for other browsers. [14]

Turn that exchange into an experiment

Pick the browsers your readers use. Try one suggested interaction in each, including the fallback. Tab through the controls to check keyboard focus: which control receives the next key. Record what happens when the main feature is unavailable. That turns the discussion into an experiment you can inspect.

The Slashdot Scope discussion raises a question about infrastructure. Begin with a setup small enough for an evening project and strong enough to expose a mistaken assumption. Scale it when the next question demands more. [28]

> **SEND US THE INTERESTING PART**
>
> Which observation changed your mind? Include the original project, the discussion, your exact version and what you tried. A useful failure is welcome.

Send a finding or a correction / support@codefarmer.com

FOUND VIA HACKER NEWS / Read the discussion
PROJECT: Modern Web Guidance
Discussion analysis / experiment invitation
Also: Scope original and discussion, page 58.

EDITORIAL METHOD

How this issue earned its labels

This issue was researched, written, edited, programmed and laid out in a Codex workflow. Public sources supplied reported developments; local executions supplied the prompt-in results. The selected cover and both full-page ad illustrations were generated locally in Draw Things.

The supplied programs were assembled from the printed project briefs and explicit implementation decisions. Our evidence describes those files. A new run from the same prompt can choose different code; the new result needs its own checks. A comparison across independent agents would be a separate experiment.

The verification record

Logic checks compare the programs' rules with expected answers: 35 assertions passed. Browser checks use the running interface: 40 passed. The Modern Web Guidance experiment passed seven separate typed-command checks. Its version, small test files, commands and raw results accompany the edition.

The browser run covered three widths, offline reloads and page errors. The sequencer's sound-generating code produced an audio signal in memory; that is different evidence from a listening review. The project screenshots show the supplied programs.

Reporting scope

This edition includes a fresh exact-prompt Lunar Lander build and one executed keyboard-repair follow-up tested November 15. The first file and its failures remain available beside the repaired file and independent comparison. The three original supplied programs and printed build prompts are unchanged. Earlier reports retain their dates; a first-time human reader trial remains separate work.

History, tools and the workbench

[1] Creative Computing publication history
University of Pennsylvania, The Online Books Page
https://onlinebooks.library.upenn.edu/webbin/serial?id=ccomputing

[2] Dave tells Ahl: the history of Creative Computing
John J. Anderson, Creative Computing, November 1984
https://www.atarimagazines.com/creative/v10n11/66_Dave_tells_Ahl__the_hist.php

[3] An interview with David Ahl
Kay Savetz, June 2003
https://www.atarimagazines.com/creative/daveahl/

[4] Creative Computing covers
Classic Computer Magazine Archive
https://www.atarimagazines.com/creative/covers/

[5] Building a retro handheld game with Codex and ModRetro Chromatic
Kevin Whinnery, OpenAI Developers, November 9, 2026
https://developers.openai.com/blog/build-a-game-with-modretro-chromatic

[6] Bringing my LED display to life with GPT-Live-1 and Codex
Sid Rampally, OpenAI Developers, September 23, 2026
https://developers.openai.com/blog/bringing-my-led-display-to-life

[7] Rethinking skills and prompts for GPT-6 Astra
Eric Provencher, OpenAI Developers, September 11, 2026
https://developers.openai.com/blog/rethinking-skills-and-prompts-for-gpt-6-astra

[8] Gemini 4 Carbon and /swarm
Google Antigravity Team, November 10, 2026
https://antigravity.google/blog/gemini-4-carbon-in-google-antigravity

[9] Modern Web Guidance 1.0
Philip Walton, Chrome for Developers, November 10, 2026
https://developer.chrome.com/blog/modern-web-guidance-v1

[10] Agentic coding and persistent returns to expertise
Anthropic Economics, June 16, 2026
https://www.anthropic.com/research/claude-code-expertise

People, practice and discovery

[11] Introducing Muse Code and Muse Spark 1.2
Meta AI Research, August 5, 2026
https://research.meta.ai/blog/introducing-muse-code-and-muse-spark-1-2

[12] Build with Meta Model API and Muse Code
Meta developer documentation, accessed November 12, 2026
https://dev.meta.ai/docs/overview

[13] Transitioning Gemini CLI to Antigravity CLI
Google Developers Blog, May 19, 2026
https://developers.googleblog.com/an-important-update-transitioning-gemini-cli-to-antigravity-cli/

[14] Get your coding agent onto modern web primitives
Show HN discussion, accessed November 12, 2026
https://news.ycombinator.com/item?id=50443812

[15] Simon Willison's Tools
Practitioner project collection, accessed November 12, 2026
https://tools.simonwillison.net/

[16] Simon Willison's Weblog
Practitioner reporting, accessed November 12, 2026
https://simonwillison.net/

[17] Addy Osmani's writing
Practitioner articles, accessed November 12, 2026
https://addyosmani.com/

[18] Exploring Generative AI
MartinFowler.com, accessed November 12, 2026
https://martinfowler.com/articles/exploring-gen-ai.html

[19] Slashdot
Community technology coverage, accessed November 12, 2026
https://slashdot.org/

[20] Agentic HIL
Hackaday.io project, accessed November 12, 2026
https://hackaday.io/project/206788-agentic-hil

Builders, experiments and their discussions

[21] Huzzah
Daniel Vaughn, August 2026; checked November 12
https://www.danielvaughn.dev/posts/huzzah/

[22] Huzzah: the discussion
Hacker News; checked November 12, 2026
https://news.ycombinator.com/item?id=49378768

[23] Local-coder
Project documentation; checked November 12, 2026
https://github.com/gmarland/local-coder

[24] Local-coder: the discussion
Hacker News; checked November 12, 2026
https://news.ycombinator.com/item?id=49785853

[25] Stroq documentation
Project documentation; checked November 12, 2026
https://stroq.dev/docs/

[26] Stroq: the discussion
Hacker News; checked November 12, 2026
https://news.ycombinator.com/item?id=50380704

[27] Scope project documentation
Microsoft; checked November 12, 2026
https://microsoft.github.io/scope/

The trail behind the new workbench

[28] Scope: the discussion
Slashdot, October 25, 2026
https://news.slashdot.org/story/26/10/26/0420240/microsoft-announces-open-source-tool-evaluating-your-softwares-optimization-for-ai-coding-agents

[29] Introducing Scope
Microsoft Open Source Blog, October 19, 2026
https://opensource.microsoft.com/blog/2026/10/19/introducing-scope-open-source-evaluation-for-ai-coding-agents/

[30] OpenClaw documentation
Project documentation; checked November 12, 2026
https://docs.openclaw.ai/

[31] How AI Assistants Are Moving the Security Goalposts
Slashdot, March 9, 2026
https://it.slashdot.org/story/26/03/09/1912259/how-ai-assistants-are-moving-the-security-goalposts

[32] The original security report
Brian Krebs, March 2026
https://krebsonsecurity.com/2026/03/how-ai-assistants-are-moving-the-security-goalposts/

[33] Modern Web Guidance: get started
Chrome for Developers; checked November 12, 2026
https://developer.chrome.com/docs/modern-web-guidance/get-started

[34] Modern Web Guidance 1.0.0 package
npm registry; downloaded November 12, 2026
https://registry.npmjs.org/modern-web-guidance/1.0.0

[35] Prompting best practices
Anthropic documentation; checked November 13, 2026
https://platform.claude.com/docs/en/build-with-claude/prompt-engineering/claude-prompting-best-practices

[36] Prompt design strategies
Google documentation; checked November 13, 2026
https://aistudio.google.com/docs/prompting-strategies

The way into your first workbench.

[37] The City and the Stars / publisher-distributor synopsis
Official source checked November 13, 2026
https://www.ipgbook.com/the-city-and-the-stars-products-9780795300073.php

[38] OpenAI / desktop quickstart
Official source checked November 13, 2026
https://learn.chatgpt.com/docs/quickstart

[39] OpenAI / Codex CLI
Official source checked November 13, 2026
https://learn.chatgpt.com/docs/codex/cli

[40] Anthropic / Claude Code desktop quickstart
Official source checked November 13, 2026
https://code.claude.com/docs/en/desktop-quickstart

[41] Anthropic / Claude Code quickstart
Official source checked November 13, 2026
https://code.claude.com/docs/en/quickstart

[42] Meta / Muse Code overview and installation
Official source checked November 13, 2026
https://dev.meta.ai/docs/muse-code

[43] Meta / Muse Code authentication and billing
Official source checked November 13, 2026
https://dev.meta.ai/docs/muse-code/auth

Install, sign in and follow the workflow.

[44] Meta / Muse Code subscription account requirements
Official source checked November 13, 2026
https://dev.meta.ai/help/subscriptions/subscribe-to-muse-code

[45] Google / Gemini CLI getting started
Official source checked November 13, 2026
https://geminicli.com/docs/get-started/

[46] Google / Gemini CLI installation and requirements
Official source checked November 13, 2026
https://geminicli.com/docs/get-started/installation/

[47] Google / Gemini CLI authentication
Official source checked November 13, 2026
https://geminicli.com/docs/get-started/authentication/

[48] Microsoft / set up GitHub Copilot in VS Code
Official source checked November 13, 2026
https://code.visualstudio.com/docs/setup/copilot

[49] Microsoft / first task with an agent
Official source checked November 13, 2026
https://code.visualstudio.com/docs/agents/quickstart?agent-surface=chat-view

[50] Node.js / official download and installers
Official source checked November 14, 2026
https://nodejs.org/en/download

A DATED STARTING POINT

Product walkthroughs were checked November 13 and clarified November 14, 2026; Node’s installer page was checked November 14. Use official guides for current platform, plan and interface details. Record your own exercise results.

Your next idea belongs here.

A magazine about making things grows through the people who try them. Send us an idea for a prompt-in, a response to an article, a question for the editor or a short account of something you built.

Show us what happened

Tell us what you wanted to make, which tools you used and what the result did. A short prompt, a screenshot or one useful observation can make a good starting point for a future reader feature.

Ask a question worth exploring

What should we build next? Which agent workflow deserves a fair test? What hardware would you like to put to work? What would make these projects easier to learn from?

Send your idea or comment / support@codefarmer.com

> **A SIMPLE SUBMISSION**
>
> Include your preferred byline, the idea or result, and whether we may quote your message or reproduce your attached image. Selected contributions may be edited for length and clarity.

This first issue opens the conversation. Future reader pages can include your experiments, corrections and useful disagreements.

Editorial mail: support@codefarmer.com

ONE MORE THING

Make a change. See what unfolds.

A computer becomes more interesting when an idea becomes yours.

Move the landing pad.

Plant a pattern at the edge.

Leave a beat empty.

Then ask why the result feels different.



THE NEXT WORKBENCH CHALLENGE

Make a tiny tool for something you know well. Give it one input, one useful result and one example that proves it works. Keep the first version small enough to explain to a friend.

Creative Agentic Computing
Issue 001 / CodeFarmer / First edition

My toaster has a roadmap.

By Arthur / Back-page editor

Imagine asking a coding agent for a five-minute timer. One button. A bell. Something to remind you that tea exists.

The first proposal includes a dashboard. By the second, the timer has teams. Apparently the kettle is now a stakeholder.

This is how our imaginary kitchen acquires an enterprise architecture. The toaster needs an account. The bread has a lifecycle. Someone is asking whether breakfast should be a platform.

Computers make another thing possible. A coding agent can turn that possibility into a button before anybody has finished the sentence. A small project suddenly has room for a very large future.

The useful question arrives from the person holding the cold cup: does the bell ring?

That question is a gift. It gives the project somewhere to land. We can make a timer that rings, then spend our imagination on the detail that makes it ours: a tiny sunrise, a favorite chord, a perfectly unnecessary animation of a kettle taking a bow.

I would happily keep the bow. Delight earns its place when it helps us enjoy the thing we came to do. The strategic breakfast dashboard can wait until the toast has requested it.

Finish one small thing, then give it one small pleasure. Test the bell. Keep the bow. Ask the kettle before adding it to a team.

If your next project develops a business model, the toaster would like to compare roadmaps.