ChatGPT

Reimagining Generative UI Frameworks in Developer Tools

In the evolving landscape of developer experience (DevEx), a new wave of **generative UI frameworks** and AI-assisted tools is emerging. Unlike traditional IDEs that merely help write code faster, these tools aim to *orchestrate* development tasks at a higher level [1]. Cutting-edge examples like **Google Antigravity** and **Marimo** exemplify this shift – they integrate AI and novel UI paradigms to transform how we interact with code. This report dives deep into current frameworks, products, and design paradigms (with a focus on open-source solutions), and explores how to "steal, transform, and re-imagine" their ideas. We focus on three guiding principles:

1. **Terminal-First Interfaces:** Emphasizing text-based, command-line driven interactions for speed and composability.
2. **Functional & Compositional Paradigms:** Using functional programming concepts and highly composable components to build UIs and workflows.
3. **Open-Source Foundations:** Leveraging open frameworks (while learning from notable closed-source innovations) to ensure extensibility and community trust.

We'll survey key examples under each theme – from AI-powered CLIs to reactive notebooks – and present visual designs (screenshots, diagrams, ASCII charts) to illustrate how these ideas can invert and topple the status quo of UI/UX in developer tools.

Terminal-First Developer Interfaces

Modern developer tools are witnessing a renaissance of the **terminal**. A terminal-first approach means the command-line or text-based interface is the primary mode of interaction, even as richer UIs are layered on top. This philosophy harkens back to the Unix principle of composability (where simple text commands are combined to perform complex tasks), now supercharged with AI.

- **CLI Agents & TUI Mode:** Several AI coding assistants are designed to run natively in the terminal. For example, *Continue* – an open-source coding agent – provides a **"terminal-native AI coding assistance with TUI and headless modes."** Developers can chat with an AI assistant or run automated workflows entirely in the console [2] [3]. This means tasks like code generation or refactoring can be done through textual commands in a terminal UI (TUI) rather than clicking through GUI panels. The terminal interface keeps developers in their flow, enabling quick keyboard-driven actions.

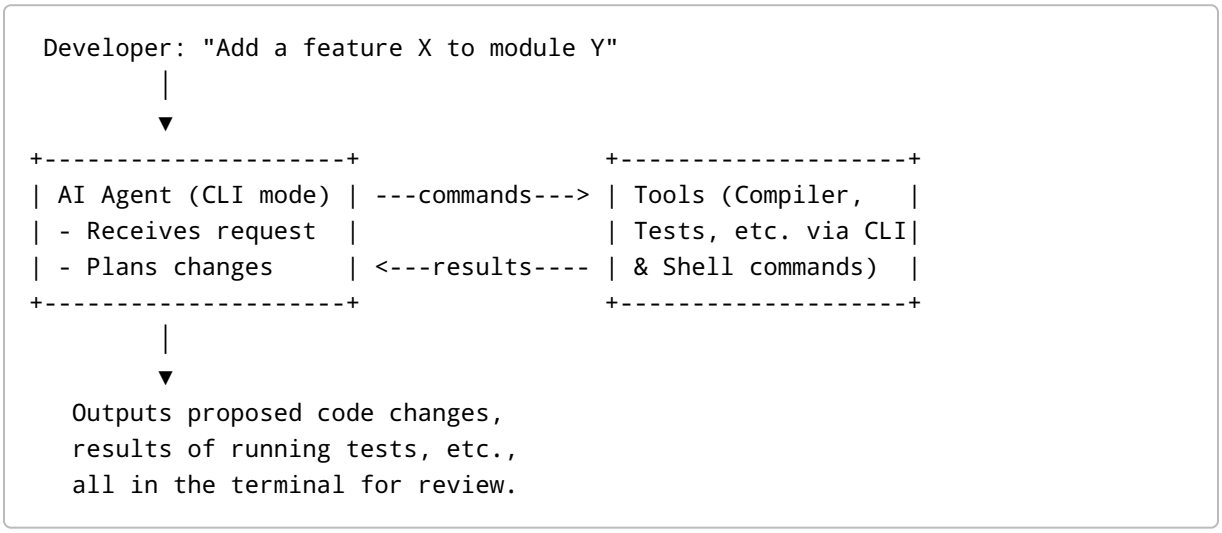
- **Composability via Commands:** A terminal-first tool often exposes functionality as commands that can be scripted and composed. This is evident even in proprietary offerings. **AWS's Amazon Q** (closed-source) integrates an AI assistant not only into IDEs but also as a CLI agent [4]. It supports specialized commands (like "/dev", "/doc", "/review") for different tasks (code implementation, documentation, code review), allowing developers to chain and automate these via scripts or

terminal sessions. The ability to pipe outputs between commands or include AI in shell workflows underscores the power of this approach.

- **Integration with Traditional Terminals:** Terminal-first doesn't mean purely text output; it can blend with richer media. Tools like *Antigravity* explicitly manage a **built-in terminal** as part of their agent's toolkit. In Antigravity, an AI agent can run shell commands and tests in an embedded terminal while concurrently editing code and testing in a browser [5]. All this is orchestrated from a unified interface, but crucially, the *textual* nature of the terminal operations (logs, shell outputs) is preserved for transparency and scriptability. Even traditional terminal emulators are becoming AI-enhanced (e.g. Warp terminal, though closed-source, introduced AI command search and explanations), pointing to a trend where the CLI itself gets smarter.

- **Example – Continue CLI:** *Continue* can serve as a concrete model for a terminal-first design. It offers both an interactive CLI and optional GUI/IDE plugins. In **TUI mode**, developers invoke an AI agent in the terminal to handle tasks like running builds, applying code transformations, or answering questions [3]. For instance, one could issue a high-level command like `continue refactor --goal "Optimize the sorting algorithm"` and the agent will plan and execute code changes across files, presenting the diff in the terminal for approval. Because it's text-first, these steps are easily reviewable and can even be saved or replayed (e.g., as part of CI pipelines in *headless mode* [3]). This shows how a text UI can actually improve **DevEx** by making interactions easily reproducible and automatable.

ASCII Diagram – A Terminal-Centric Workflow: Below is a simple ASCII illustration of how a terminal-first, agent-driven tool might operate through text commands:



In this flow, the developer's natural-language request is handled by an AI agent fully through CLI operations. The agent emits shell commands (e.g., to run tests, install packages) and modifies code, then returns textual results (diffs, test logs) in the terminal. The **composability** is evident – every action is a

discrete command that could be run manually or chained in scripts, embodying a *functional composition of commands*.

- **Efficiency and Ergonomics:** Terminal-first interfaces also encourage **keyboard-driven ergonomics**. Many modern GUI editors incorporate a *command palette* (e.g. VS Code, Sublime Text) which is essentially a popup CLI for actions. A truly terminal-first system takes this further: *everything* can be accomplished with commands. For example, Antigravity defines quick shortcuts to **"Switch to Agent Manager" (⌘E) or "Code with Agent" (⌘L)**, indicating that even opening the agent panel or invoking inline AI is done via a keystroke command【20†】. By prioritizing the terminal/command approach, we reduce context-switching – developers don't have to grab the mouse or navigate complex menus, they simply issue instructions.

In summary, a terminal-first philosophy yields a developer tool that is *fast, scriptable, and composable*. It aligns with developers' love for automation: if an AI-assisted UI can be driven by text commands, users can combine and repeat those efficiently. Next, we examine how **functional programming and compositional design** principles amplify these benefits.

Functional & Compositional UI Paradigms

Adopting a **functional interface paradigm** means treating UI and interactions as pure functions of state, emphasizing predictability and modular composition. This goes hand-in-hand with a "compositionality first" attitude – building complex behaviors by combining simpler parts. Many recent UI frameworks (especially in open source) draw on these ideas to improve reliability and reusability.

- **Functional Reactive Programming (FRP) in UI:** In academic circles, FRP has long been touted as a promising approach for GUI design. It provides *"high-level, declarative, compositional abstractions to describe user interactions and time-dependent computations."* [6] In simpler terms, instead of imperative step-by-step UI updates, you declare relationships between values and UI elements. For example, *Elm* – a functional language for web UIs – introduced the Elm Architecture where the UI is purely a function of an application state (the *Model*), and all updates are done via pure functions. The payoff is that **the outcome of the code becomes predictable, without surprises** [7]. When state changes, the UI *always* re-computes the same way. This eliminates whole classes of bugs related to hidden or stale state.

- **Composition of Components:** Modern UI frameworks like React (though not purely functional, it embraces a declarative component model) show the power of composition. A React app is built from small components (functions) that are composed in a tree. Similarly, Elm's view functions or SwiftUI's declarative views are composed hierarchically. The result is *UI reuse* and *clarity* – developers think in terms of assembling Lego blocks of UI. For a generative UI system, this is gold: an AI agent can generate or modify one self-contained component without breaking others, as long as the composition contracts are clear. Moreover, functional components with explicit inputs (props/state) make it easier for an AI to reason about where changes need to propagate.

- **Predictability & Determinism:** A functional paradigm also aids *DevEx* by reducing flakiness. For example, the **Marimo** notebook (explored more soon) is designed such that *"notebooks are executed in a deterministic order, with no hidden state"* [8]. Deleting a cell in Marimo will *"delete its variables while updating affected cells"*, meaning the system tracks dependencies and ensures no lingering

mutable state [8]. This determinism comes straight from functional thinking – treat the notebook as a **dataflow graph** of pure computations rather than an imperative log of execution. The result: if you change one thing, everything that depends on it updates automatically in a controlled way.

- **Reactivity and the DAG of Dependencies:** Many compositional UIs implement reactivity via a dependency graph. In FRP terms, UI elements or outputs *react* to changes in input signals. Marimo's engine, for instance, builds a directed acyclic graph (DAG) of cell dependencies. Thus *"reactivity allows marimo to determine the correct running order of cells using a DAG"*, and *"marimo notebooks automatically update dependent cells, ensuring consistent results"* [9]. This can be visualized by an ASCII chart of a simple notebook:

```
[ Cell A ] <-- produces X
   ↓
[ Cell B ] <-- uses X from A
   ↓
[ Cell C ] <-- uses result of B
```

If **Cell A**'s code or output changes, the system knows to re-run Cell B and then Cell C, in that order. This automatic propagation is akin to a spreadsheet recomputing formulas – a concept even non-programmers find intuitive. For a developer tool, it means you don't waste time debugging why something didn't update; the framework guarantees consistency.

- **Immutable State and Debuggability:** A functional paradigm typically involves immutable state transitions, which hugely aids debugging (you can replay state changes without side effects). Elm famously aims to *"make impossible states impossible"* through its type system and update logic [10] [11]. While that level of type safety might be beyond a dynamic tool, the philosophy holds: by constraining how state changes (e.g., only via pure functions or approved agent actions), a generative UI system can avoid "weird" intermediate states. For example, an AI agent making UI changes could use transactions or diff previews to ensure the UI only goes from one valid state to another – a nod to Elm's guarantees but in an AI context.

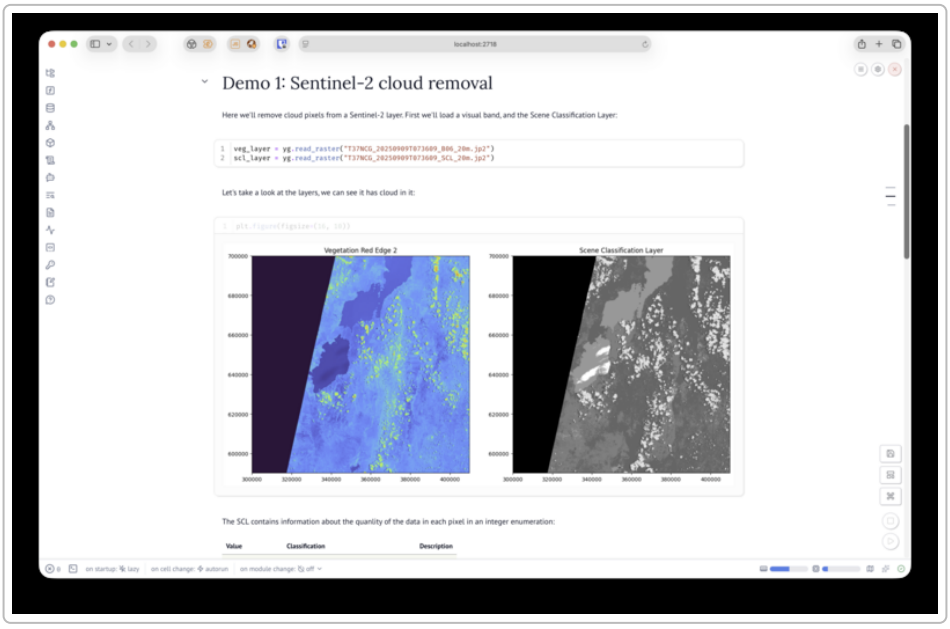
- **Functional Core, Imperative Shell:** One pattern to consider is designing the *core* interactions as pure functions (for instance, a function that, given a user request and current state, returns a set of changes or next state), while the *outer* layer performs the side effects (applying the changes to files, etc.). This way, an AI agent's logic could be unit-tested or at least simulated without actually changing things until approved. This concept isn't directly cited in sources, but it synthesizes the benefits we see: pure functions for reasoning and composition, imperative actions only when needed (like actually writing to disk or running a process).

In essence, **functional and compositional paradigms** lend *stability and clarity* to generative developer tools. By eliminating hidden state and favoring building-block design, they make the system's behavior more transparent – which is crucial when AI is involved (developers need confidence that the AI isn't doing something behind the scenes that they can't trace). Next, let's survey concrete frameworks and products exemplifying these principles, especially open-source ones we can learn from.

Open-Source Generative UI Frameworks and Tools

Many open-source projects are pushing the envelope in AI-assisted development, UI generation, and new dev workflows. Below is a rundown of notable projects (all open-source) that embody the **terminal-first** or **functional-compositional** ideas – or both – often with innovative UI/UX designs. Each offers valuable concepts to draw from:

- **Marimo – Reactive AI Notebook:** *Marimo* is an open-source reactive Python notebook that reimagines Jupyter notebooks. Marimo notebooks are **stored as plain .py files** (so they're Git-friendly) and execute as a proper Python program rather than a linear scratchpad [12] [13]. It guarantees that *"code, outputs, and program state are consistent"* – run a cell and it auto-runs any other cells that depend on it [13]. There's **no hidden state**: deleting a cell removes its variables from memory, preventing the usual Jupyter gotchas [13]. Marimo also packs a rich set of **built-in UI components** (sliders, dropdowns, tables, etc.) that you can embed in your notebook. These *interactive elements are automatically synchronized with Python:* if you move a slider or select an option, any cell using that value re-runs immediately [14]. This reactive UI makes it *"substantially more useful than Jupyter widgets"* because it feels instantaneous and is easier to use [14]. Notably, Marimo positions itself as an **"AI-native editor"** – it integrates with AI coding assistants (Copilot, etc.), supports inline AI edits and chat, and even lets you generate entire notebooks from natural language [15] [16]. It's essentially a marriage of a functional reactive notebook (inspired by ideas from Pluto.jl and Streamlit [16]) with modern AI features. Marimo's design shows how a *compositional UI framework* (the notebook with its UI widgets and DAG execution) can work seamlessly with AI: for instance, you can prompt it to create a data visualization, and it will insert the necessary code cells and UI elements. (Marimo's open-source nature also means one can extend it or even fork its UI component system for other projects.)



Screenshot: A Marimo notebook UI. In this example, the notebook combines text, code, and outputs (plots) in a clean layout. Marimo's interface is less cluttered than classic Jupyter, with a sidebar of icons for adding different cell types, etc. Under the hood, each cell is annotated (see the `@app.cell` decorators in code) to

define its behavior, reflecting Marimo's functional, compositional approach to notebooks [17] [18]. The two side-by-side plots shown were produced reactively from the preceding code cells. If the data or code changes, Marimo would automatically recompute and update these visuals, demonstrating real-time generative UI capabilities in a notebook setting.

- **Continue – AI Agent in IDE & CLI:** *Continue* (by continuedev) is a popular open-source AI coding assistant that exemplifies a **hybrid terminal + IDE** approach. It provides a **CLI/TUI** for headless or terminal use, as well as extensions for VS Code and JetBrains IDEs [2] [19]. In VS Code, Continue appears as a sidebar chat/agent that can modify code, but notably, the core is editor-agnostic – you can run it purely via `continue` CLI in a project. Continue's agent uses a *"Plan-and-Act"* loop: it will draft a plan of steps to implement a request, then execute them one by one (editing multiple files, running tests, etc.). This is similar to how Antigravity's agents plan tasks [20]. The open-source nature of Continue is key: you can self-host it, use your own model API (OpenAI, local LLMs, etc.), and all operations happen locally on your codebase [21] [22]. It being terminal-first is evident from its documentation stressing *"terminal-first development"* and automation in CI/CD [3]. For instance, you could run `continue ci --fix-failing-tests` on your CI server; the agent would analyze failing tests and attempt fixes, outputting the diff for review. **Lesson:** Continue shows the viability of **keeping the developer in the loop** even with an autonomous agent – it prints out each action and waits for confirmation if configured, which in a terminal is just a [Y/n] prompt. This kind of design (transparent step-by-step AI) could be emulated in new tools to build trust.

- **Aider – CLI Code Assistant with Multi-File Edits:** *Aider* is another open-source tool, focused purely on the terminal. Described as *"a popular open-source CLI tool for AI-assisted coding"*, it pairs with GPT models (user provides an API key) and has direct **write access to your repository** [23]. The developer explicitly tells Aider which files to work on (e.g., `aider main.py utils.py`) and can then issue natural language commands; Aider will edit those files accordingly. What sets it apart is powerful multi-file refactoring: you can say *"Update these two files to use dependency injection"* and Aider will intelligently modify both in tandem [23] [24]. This ability to coordinate changes across a codebase via one command was noted by Thoughtworks as a major innovation – many early coding assistants couldn't handle simultaneous multi-file context [24]. Aider's **local-first approach** (it runs on your machine, uses your editor for reviewing diffs, and nothing leaves your environment except API calls to the model) makes it attractive for sensitive projects [25] [26]. It's essentially giving the terminal superpowers: you still use your text editor or `git diff` to review changes, but the heavy lifting of writing those changes is done by the AI. For our purposes, Aider is a lesson in **granular command design** – it's simple (one command-line invocation to start, then a chat loop in terminal) yet effective. One could imagine extending such a tool with an even more compositional UI (perhaps a TUI that shows a tree of file changes or a small preview window for images if the code produces any). In sum, Aider reinforces how a **terminal-first UI** can tackle complex coding tasks while remaining understandable and controllable by the developer [27] [28].

- **Cline – Plan/Act Autonomous VS Code Agent:** *Cline* (by Roo) is an open-source project that integrates with VS Code as an extension, offering an autonomous agent similar to Cursor or Antigravity's agent mode. Cline's hallmark is its dual-phase approach: *"Plan" and "Act" modes.* In Plan mode, it analyzes the request and formulates a step-by-step plan (e.g., *"1. Modify function X, 2. Update unit tests, 3. Run tests"*), which the developer can review. Then in Act mode, it executes those steps, modifying code and possibly running commands [29] [22]. It can read and write across the entire project, thanks to VS Code's API, and supports either OpenAI's GPT-4 or local models via API [21] [30].

Cline being open-source and **extensible (written in TypeScript)** means one could integrate additional tooling – perhaps hooking it into a custom test runner or CI trigger [30]. The key takeaway from Cline is the **explicit planning UI**: by surfacing the plan, it adds a functional flavor (the plan can be seen as a *pure description* of what will happen, separate from the side-effect of applying it). This again echoes the benefits of a functional paradigm, now in an AI agent's UX – the developer sees a declarative "what" before approving the imperative "do it". A future generative UI might present such plans in a nice UI panel or even as an editable checklist (imagine converting the plan to a TODO list that the dev can tweak before execution).

- **Jupyter AI – Generative Extension for Notebooks:** *Jupyter AI* is an official open-source extension that adds AI assistance to JupyterLab [31]. It's worth noting as an example of adding generative UI features to an existing interface. Jupyter AI provides a chat UI inside JupyterLab where you can ask questions about your code or have it generate code in new cells [32]. It supports multiple models and even allows working with local models [33]. The interesting UI aspect is the *conversational assistant native to the notebook UI* [34] – meaning users don't have to switch to a separate tool, the generative suggestions appear in context (for example, you highlight some code and ask the assistant to explain or refactor it, and it can insert the result into the notebook). While not a standalone framework, Jupyter AI demonstrates that *even established UIs can be enhanced with generative, contextual widgets*. For a new system, however, one could design this in from the start – e.g., a terminal-first notebook could have a special "AI cell" or command palette for queries, fully integrated rather than bolted-on.

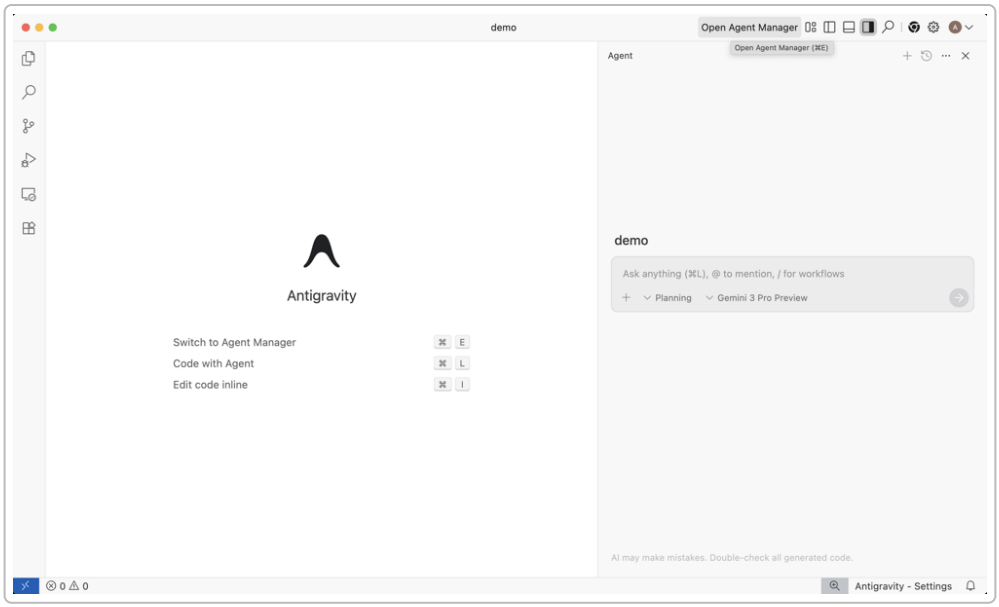
These open-source tools illustrate a spectrum of approaches to AI + UI for developers. We see truly **text-centric interfaces** (Continue, Aider) as well as **mixed GUI** ones (Marimo's notebook, Cline's VS Code integration), all embracing *openness* and *compositional design*. By studying these, we can identify what works well: fast iterative loops, clear visualization of AI actions (plans, diffs), preserving developer control, and seamless blending of code and UI elements.

Innovations from Proprietary Products (Closed-Source Insights)

While our focus is open-source, it's valuable to glean ideas from major proprietary offerings in this space. Tech giants and startups have released AI-driven dev tools that push boundaries. We can "draw upon learnings" from these to inform our own designs:

- **Google Antigravity – Agentic Development Platform:** Unveiled in late 2025, **Antigravity** represents Google's vision for an AI-first IDE [1]. Its core innovation is an **"agent-first interface"** where AI agents are first-class citizens, not just chatbots in a corner [35]. Antigravity provides two main UI surfaces: (1) the **Editor View**, a familiar IDE with AI completions and inline commands, and (2) the **Agent Manager Surface**, a dedicated panel to *"spawn, orchestrate, and observe multiple agents working asynchronously across different workspaces."* [36]. In practice, a developer might have one agent coding a new feature, another running the app in a sandbox, and another scanning the docs – all visible in the manager UI. This *multi-agent orchestration* is a leap in UX: instead of one monolithic assistant, you have a team of specialized helpers. Antigravity also emphasizes **task-oriented delegation**. You can ask the agent to "implement feature X", and it will *"plan and execute the task across the editor, terminal, and browser"*, for example writing code, launching the dev server, then opening a browser to verify the UI [20]. The developer doesn't watch a stream of tokens to see what the AI did; instead Antigravity produces **Artifacts** – tangible results of the agent's work. Artifacts

include things like *"screenshots and browser recordings"* of the running app, test results, or an *"implementation plan"* document [37]. The idea is to *"verify with Artifacts, not logs"* [38]. This dramatically improves UX by showing the outcome (a picture of the UI change) rather than just the textual log of commands. If something is wrong, the developer can **leave feedback directly on an Artifact** (like commenting on a screenshot or plan) and the agent will incorporate that feedback mid-execution [37]. This loop is reminiscent of a Google Doc review, applied to code generation.



Screenshot: Google Antigravity interface. Left: a minimalist code editor view (with a big Antigravity logo when no file is open). *Right:* the Agent panel (titled "demo" here) where the user can "Ask anything" and see agent tasks. Antigravity's design uses keyboard shortcuts (shown as ⌘L, ⌘E, etc.) to toggle modes, reinforcing a *keyboard-first* interaction even in a GUI. The clean separation of the editor and agent manager illustrates how a UI can accommodate asynchronous AI work without overwhelming the main coding area [36]. The agent manager likely lists multiple agents, each with status (Planning, Executing, etc.) and allows the user to oversee or intervene in each.

- **Cursor – AI-augmented Code Editor:** *Cursor* is a closed-source (subscription) product that gained popularity by forking VS Code's open source and baking in AI deeply. Cursor introduced an **"Agent Mode"** similar in spirit to Antigravity's: you give a high-level goal (e.g., "Add dark mode to this React app") and Cursor's agent will *generate and modify files across the project, run the app, iterate, etc.* towards that goal [39]. It essentially automates the edit-run loop within the editor. The UI in Cursor, being VS Code-based, wasn't radically different at surface – but users note that it feels like the AI is a co-developer in the same workspace, thanks to thoughtful touches like a TODO list that gets auto-generated from your request, which the agent then ticks off as it completes steps. **Learning:** Even without a fancy new interface, integrating AI tightly with an existing editor can transform UX. It suggests that *workflow integration* (autonomous actions tied to familiar UI elements like file explorers, problem lists, etc.) is crucial. A brand-new framework we design could go further, of course, but we should ensure AI tasks appear as structured data (like tasks/artifacts) not just as raw text.

- **Replit Ghostwriter & Agent:** *Replit*, an online IDE, launched an AI **"Agent"** that can *"generate entire projects from descriptions"* and an **Assistant** that can answer questions and do small edits [40]. What stands out is the *seamless cloud integration* – you can go from an idea in chat to a deployed application on Replit without leaving the browser [40]. The UI here is chat-centric; you simply converse, and behind the scenes the agent creates files, configures environment, etc. For example, you might say "Build a simple to-do list app with a Flask backend and publish it," and Replit's agent will scaffold the backend, the frontend, and deploy on a Replit URL. The **zero-setup deployment** is a big UX win (no "it runs on my machine but now I need to deploy" friction). This teaches us that integrating **DevOps actions** (like environment setup, containerization, deployment) into the generative flow can set a tool apart. Open-source alternatives could leverage technologies like Docker or Terraform, but an AI that handles those via CLI (with user approval steps) could similarly streamline the "go live" process as part of UI/UX.

- **Domain-Specific Generators (Vercel v0, Bolt.new):** Two notable closed solutions focus on UI generation. **Vercel v0** is *"an AI-powered UI generator for React components and Tailwind CSS"* [41]. You describe a desired component in natural language, and v0 produces the code for a styled, polished component using a library of pre-designed UI elements (shadcn/ui). Its tight integration with Vercel means it can even instantly deploy the component to a live site for preview [42]. **Bolt.new** (from StackBlitz) took a similar prompt-to-app approach: you could type "create a to-do app" and it *"scaffolded it live"* in the browser [43]. People marveled at the demo where the app literally built itself step by step. The lesson from these: visual feedback during generation is compelling. Bolt.new showed a sort of real-time **flow visualization** of the code being created, which gave users a mental model of progress. For our designs, providing a **progress UI** (like watching files appear or UI components render one by one) might greatly enhance the generative experience. Also, v0's success with *UI design generation* suggests an opportunity: generative tools can tackle not just code logic but also **visual design** (layouts, styles). An open-source generative UI framework might incorporate an AI that suggests UI mockups or auto-adjusts design based on guidelines, potentially with a live preview.

- **Multi-Modal and Agentive UIs (Devin, OpenAI Code Interpreter):** *Devin* (by Cognition AI, closed-source) markets itself as a *"complete AI software engineer"* working in a controlled sandbox [44]. It can use a browser, terminal, etc., similar to Antigravity's agent, and even do web searches for you [44]. OpenAI's Code Interpreter (now called Advanced Data Analysis in ChatGPT) allowed uploading files, running code and returning results/visualizations in a chat. These systems indicate that a truly **agentive dev tool** might need to be *multi-modal* – not just editing code, but also handling data, images, web content as part of its UI. For example, if the task is "generate a dashboard for this dataset," the agent might produce the code *and* show the rendered chart in an output panel. Antigravity's artifacts (screenshots, recordings) are along these lines [45] [37]. We should consider UI affordances for this: maybe an *"Artifact viewer"* panel in our tool where images, PDFs, or diagrams from the agent are shown (rather than dumping base64 or links into a text log).

In summary, **closed-source tools confirm and extend** the trends we see in open ones: agent orchestration, integrated planning, visual feedback, and end-to-end workflows (from coding to running to deploying) under AI guidance. They also highlight new UX paradigms like *Artifact-based validation* (which could inspire, say, an open-source tool to implement an "artifact diff" view instead of a plain console log). By studying these, we can avoid reinventing the wheel and instead focus on innovating beyond – for instance, delivering similar multi-agent capabilities but in a more user-customizable, open way.

Synthesis and Future Outlook

Bringing it all together, how can we **“invert and topple”** the status quo of developer interaction? By synthesizing the above ideas, a clear vision emerges:

- **The Terminal as the Foundation, GUI as Progressive Enhancement:** Start with a powerful CLI core – every action (from creating a component to deploying an app) should be invocable via a command or script. Then, add a slick UI on top for convenience, *without ever requiring the UI*. This is akin to how modern tools like Git have a CLI but also GUIs. In our context, this could mean a developer can chat with the AI agent in a terminal or alternatively use a web UI that internally just sends the same commands. A *terminal-first but not terminal-only* approach captures both power and approachability.

- **Functional Reactive Backbone:** The internal architecture should treat the dev environment as a live dataflow. Think of *files, tests, running app, outputs* all as streams of data that react to changes. For example, if the developer asks the agent to create a new UI component, the framework could create a *state node* for that component. Any changes (by the agent or human) to that component’s file automatically propagate to reload the app preview (like React Fast Refresh meets FRP). This way, the developer sees effects immediately and consistently. Using a DAG of dependencies (like Marimo does for cells [46]) ensures we know what to recompute or restart on each change – eliminating inconsistent states.

- **Agent as a Co-worker, Not a Magician:** A reimagined UI would present AI actions in familiar forms: tasks, diffs, plans, artifacts. We can invert the black-box nature of some AI assistants by making everything the agent does *inspectable and editable*. For instance, instead of executing code changes immediately, the agent could open a “proposed change” buffer (similar to a Git staging area) that the developer can review. This is already seen in Aider (showing diffs to accept) and Antigravity (plans and artifact comments) [27] [37]. Our unique design could be an **“AI Task Board”** in the UI: imagine a panel that lists ongoing tasks (with states like *Planning, Waiting for Review, Executing*), each expandable to show details (commands to run, code to be written, results obtained). The developer can prioritize or even rearrange these tasks – effectively *pair-programming with multiple agents* in a structured way.

- **Composable Extensibility:** Open-source means people will want to extend the system. By prioritizing compositional design, we ensure each part of the tool can be modified or replaced. For example, if the UI widgets (say a slider, or a log viewer) are themselves modular components, users could add new ones (maybe someone wants a **UML diagram view** that an agent can output to when asked to “visualize architecture”). Our framework should allow plugging in new AI models, new artifact types, new CLI commands easily. This way, it can evolve with the community – a key advantage over closed systems.

- **Human-in-the-Loop by Design:** Ultimately, the goal is to **amplify developer productivity and creativity** – not to remove the developer from the equation. Every feature should be designed with a fallback or oversight mechanism. Terminal-first helps because the user naturally types commands (an intentional act) and sees textual feedback. Functional paradigms help because the system’s reactions are deterministic and explainable. To truly topple current UX, we should aim for an experience where using the tool feels like *collaborating with a clever colleague*. You ask for something, it does the grunt work, but keeps you posted and seeks approval for big changes. When errors occur,

it should present them clearly (perhaps even suggesting *"I can try to fix this test failing, should I proceed?"*).

Envisioned Example: Imagine a developer wants to add a new feature. In our dream tool, they might open the integrated terminal (or chat panel) and say: *"Add a dark mode toggle to the settings page."* The system (leveraging an agent) will maybe create a new branch (as a command), then **plan**: *"1. Add a button in UI, 2. Save preference in user settings, 3. Update stylesheet for dark theme, 4. Include user preference in page load."* This plan appears in an ASCII-formatted checklist in the terminal for quick reading, and also as a graphical checklist in the GUI. The developer ticks "Approve" on each, or edits a step (maybe they want a different toggle style). The agent then executes step 1: it generates the UI code. Because of our reactive architecture, the running app preview on the side *live updates* to show the new toggle (perhaps styled obviously to indicate it's not functional yet). Step 2 and 3 are done similarly, each time updating outputs (maybe step 3's artifact is a screenshot showing the page in dark mode). Finally, for step 4, the agent might run the app to ensure it defaults to the saved preference – opening a browser window or an embedded webview showing the dark mode in action. Throughout, the developer could intervene: all changes are in a version control diff, any long-running action (like running the app) is streamed in the terminal. In the end, the developer sees the feature working and can either **approve merge** (one command) or iterate further. All of this happened within one cohesive tool, terminal and GUI dancing together, with functional reactivity ensuring nothing fell out of sync.

Such a system, built on the principles of **terminal-first interaction, functional compositional design, and open extensibility**, truly has the potential to *invert* the current developer workflow paradigm. Instead of context-switching between CLI, editor, browser, and multiple tools, the experience becomes **unified**. Instead of the user adapting to the tool's limitations, the tool (via AI) adapts to the user's intent. And because it's open-source, the community can continuously inject new ideas – preventing stagnation and ensuring the *best* ideas from everywhere (even closed tools) can be incorporated.

In conclusion, by studying and combining the approaches from Antigravity, Marimo, and others, we can envision a generative UI framework that is **greater than the sum of its parts**. It would empower developers to work at a higher level of abstraction (tasks and goals), while maintaining the low-level control and transparency that experts need. The path is clear: **embrace the terminal, embrace functional composition, stay open, and relentlessly focus on a sublime developer experience.** With these guiding stars, we're poised to build the next revolution in UI/UX for software creation – one that just might defy gravity.

Sources: The information and examples above draw from a broad survey of current tools and research. Key references include Google's announcement of *Antigravity* [1] [36] [38], details from Marimo's documentation and creator posts [13] [14], Continue's and Aider's project docs [3] [23], and a 2025 overview of AI coding assistants comparing these platforms [29] [41]. Academic insight on functional reactive UIs is drawn from the Elm language design [6]. These sources collectively paint the picture of the state of the art in generative UI frameworks and inspire the vision described.

---

[1] [5] [20] [35] [36] [37] [38] [45] Build with Google Antigravity, our new agentic development platform - Google Developers Blog
https://developers.googleblog.com/build-with-google-antigravity-our-new-agentic-development-platform/

2 3 19 Welcome to Continue - Continue
https://docs.continue.dev/

4 21 22 23 24 25 26 27 28 29 30 39 40 41 42 43 44 Best AI Coding Assistants as of December 2025 | Shakudo
https://www.shakudo.io/blog/best-ai-coding-assistants

6 people.seas.harvard.edu
https://people.seas.harvard.edu/~chong/pubs/pldi13-elm.pdf

7 10 11 Elm at Rakuten | Rakuten Engineering Blog
https://engineering.rakuten.today/post/elm-at-rakuten/

8 15 marimo | a next-generation Python notebook
https://marimo.io/

9 46 marimo: A Reactive, Reproducible Notebook – Real Python
https://realpython.com/marimo-notebook/

12 13 14 16 [P] I built marimo — an open-source reactive Python notebook that’s stored as a .py file, executable as a script, and deployable as an app. : r/MachineLearning
https://www.reddit.com/r/MachineLearning/comments/191rdwq/p_i_built_marimo_an_opensource_reactive_python/

17 18 Marimo notebooks for demos
https://digitalflapjack.com/blog/marimo/

31 Generative AI in Jupyter
https://blog.jupyter.org/generative-ai-in-jupyter-3f7174824862

32 Jupyter AI Assistant - TAMU HPRC
https://hprc.tamu.edu/kb/Software/Jupyter-AI-Assistant/

33 Jupyter AI — Jupyter AI documentation
https://jupyter-ai.readthedocs.io/

34 jupyterlab/jupyter-ai: A generative AI extension for JupyterLab - GitHub
https://github.com/jupyterlab/jupyter-ai