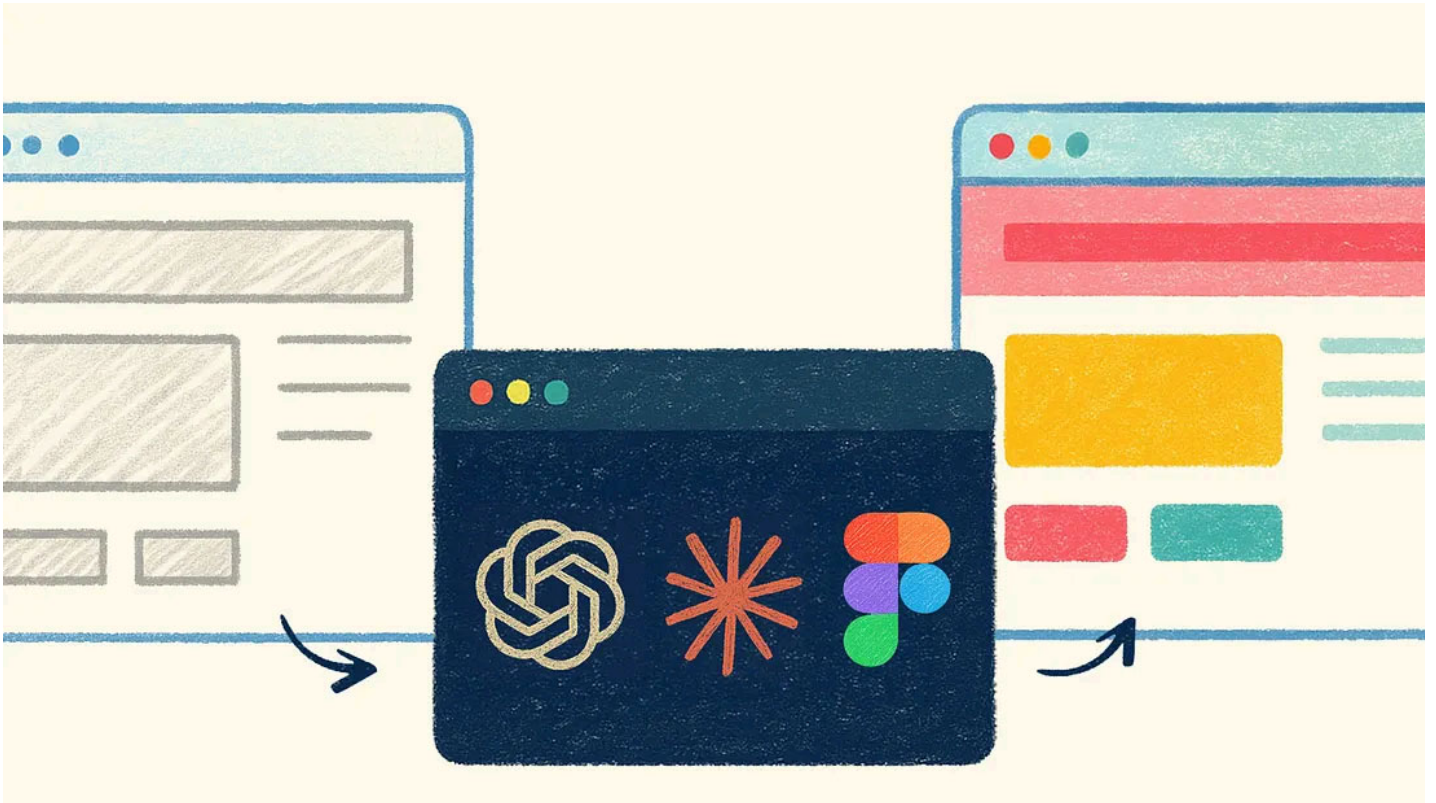


# Building AI-driven workflows powered by Claude Code and other tools

How agentic CLI tools extend Figma MCP and turn wireframes into production-ready prototypes



I set out to explore how designers can use agentic CLI tools like **Claude Code** and **Codex CLI** to build AI-driven workflows that turn rough wireframes into production-ready prototypes reflecting a real codebase, not generic mockups. After testing both tools, the results revealed that a product designer can truly prototype with real code and the help of **Figma MCP**.

With a solid design system in place, design and code share the same language, AI becomes a reliable partner, accelerating exploration and producing maintainable, real-world prototypes that teams can carry forward.

## The problem

If you are a designer already experimenting with AI tools, you know this feeling: gorgeous prototypes that, after looking at the code, are totally disconnected from the codebase your team is using and practically useless.

I've explored multiple AI tools myself to speed up my design workflow, but something was always missing. I tried **Replit**, **Lovable**, **vo**, and **Bolt** only to end up with prototypes built with generic code, based on **Shadcn** or **Tailwind CSS**. They didn't align with the existing code, and couldn't be reused in production.

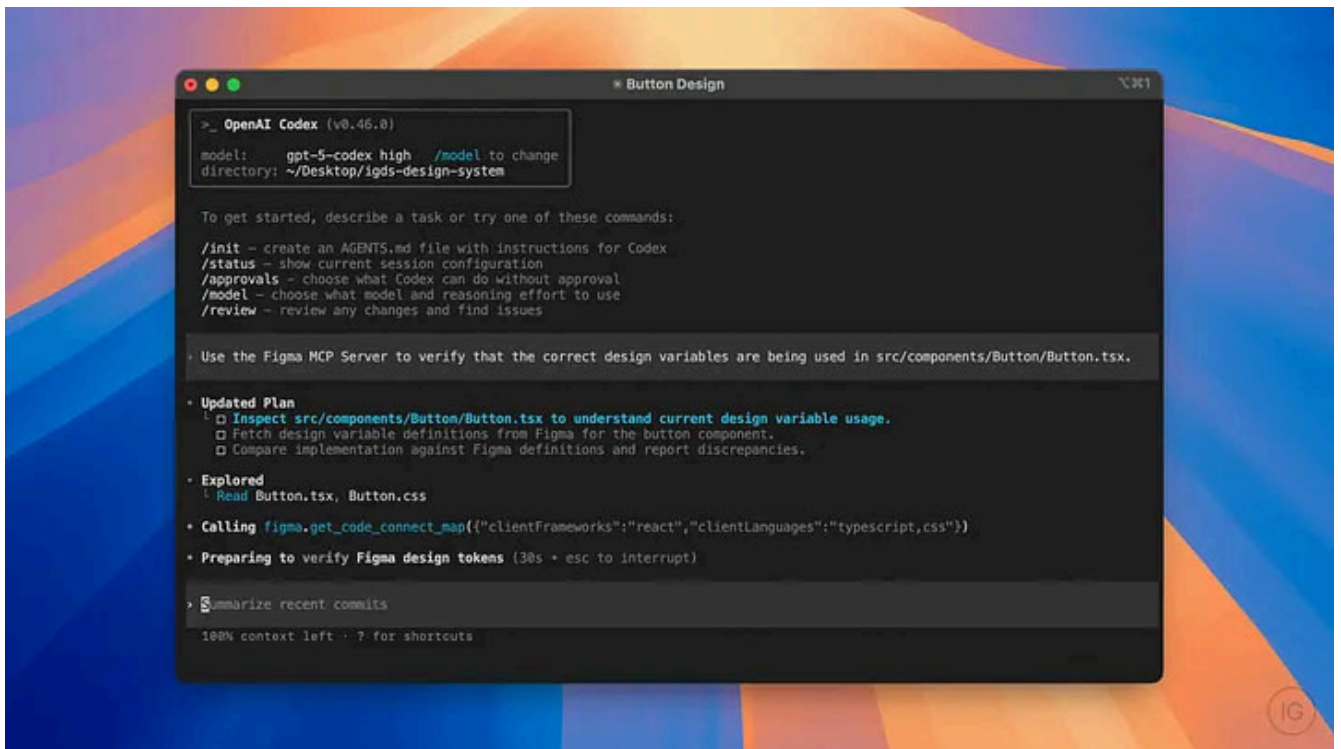
This changed significantly with **MCP servers**, and in my [previous article](#), I was optimistic enough to imagine a workflow where designers could instantly prompt rough ideas to prototypes, while also pulling from existing code and design tokens. This time, I brought this vision to life using **Code Connect UI** and its improved features like component mapping and MCP instructions.

In this article, I will share how I built an AI-ready workflow using agentic CLI tools like **Claude Code** and **Codex CLI**, combined with **Figma MCP** and **Code Connect UI**, to turn high-level prompts into production-ready code that directly reflects my design system and codebase.

## From Cursor to agentic CLI tools

Cursor was my first real look into what AI-assisted coding could look like. But soon after, agentic CLI tools like Claude Code came to play offering more control and capabilities, making AI integration into traditional workflows more efficient than any other IDE plugin ever did before.

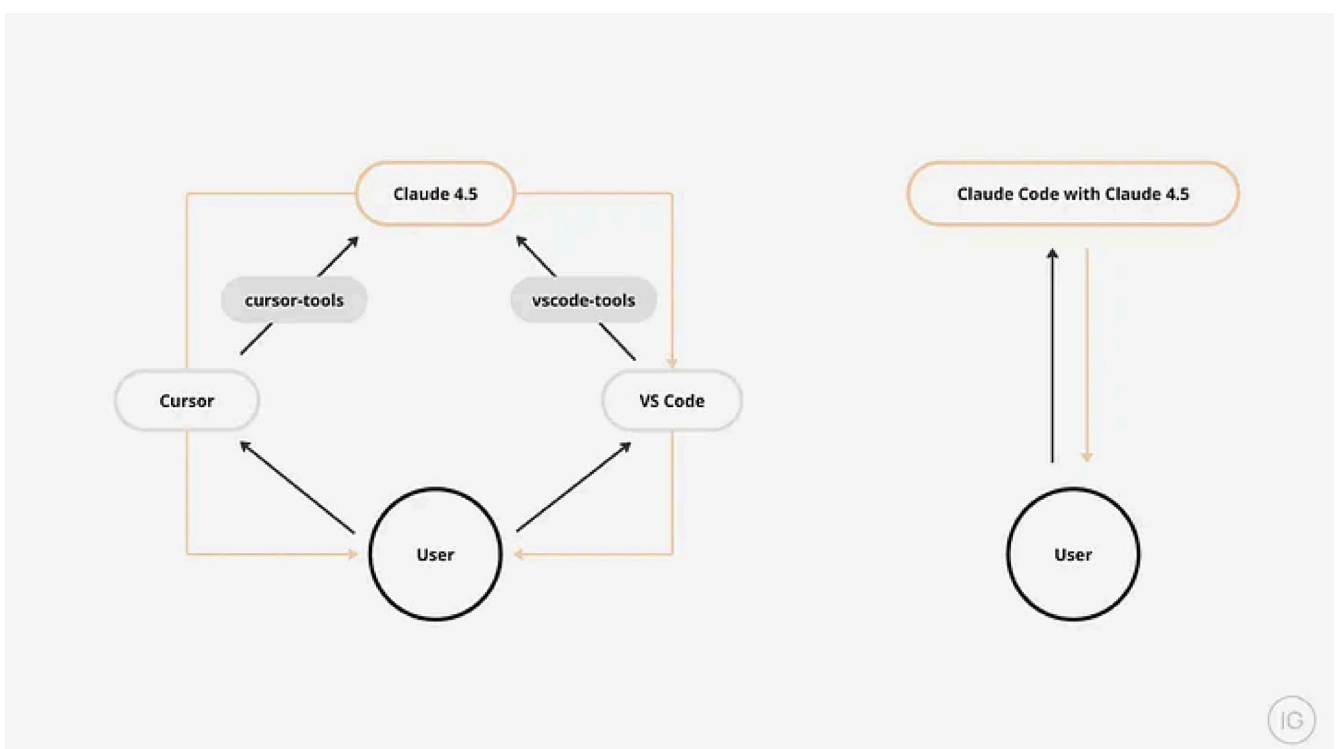
I began exploring Claude Code and Codex CLI, two command-line tools from **Anthropic** and **OpenAI**. Both can run locally inside a terminal or IDE, with immediate access to their respective models, and integrate with MCP Servers for external tool access.



The Codex CLI running in terminal and using Figma MCP tools.

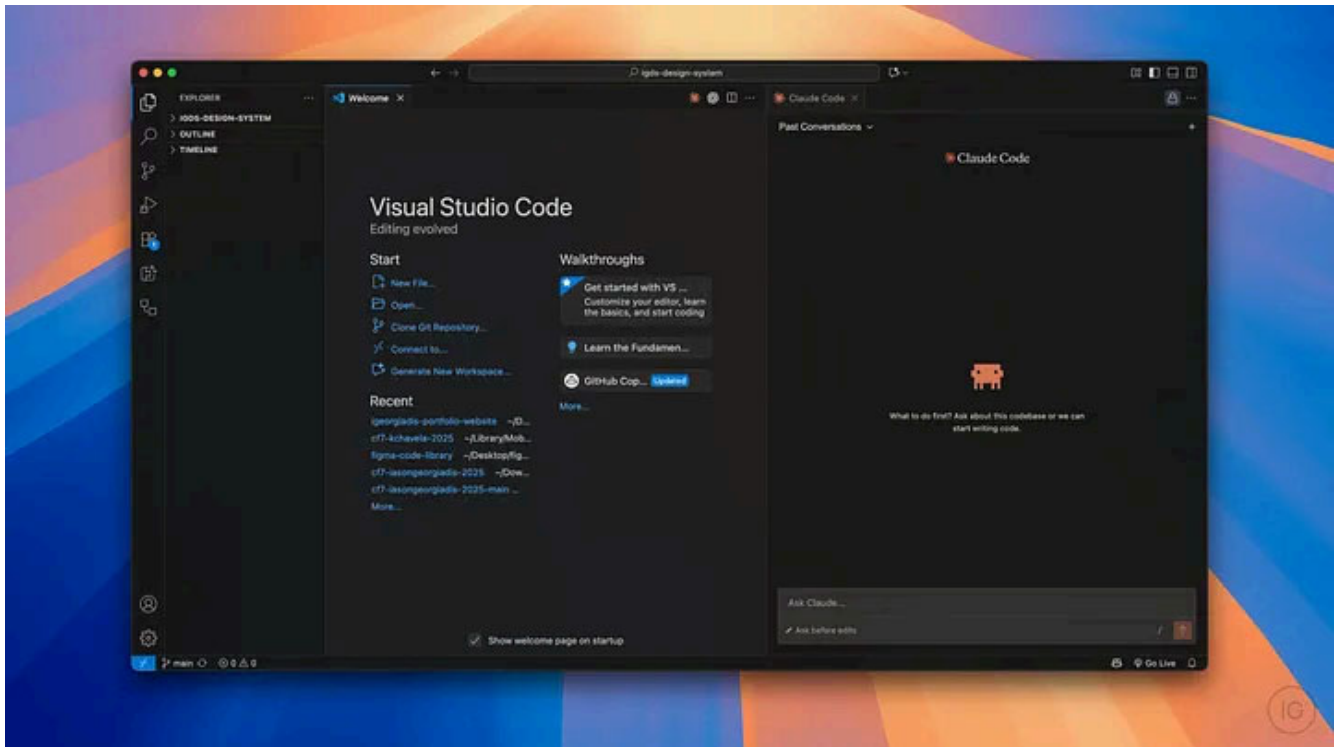
The key difference between using an AI-powered IDE and an agentic CLI lies in how they communicate with the models. While IDEs act as middlemen, translating user requests through APIs or extensions, agentic CLIs connect directly to the models themselves, making interactions faster, more contextual, and more reliable.

This direct connection enables a truly agentic experience: Claude Code and Codex CLI don't just respond to prompts; they reason, plan, and adapt within a project.



One of the key differences between using an AI-powered IDE and Claude Code lies in how they communicate with the same models.

Both Anthropic and OpenAI are actively supporting these tools with regular updates. One of them was a more user-friendly way of interacting with Claude Code on VS Code without the terminal.



Both tools come with a more user-friendly version within VS Code through their official extensions.

## Building an AI-ready design system

To ensure my workflow will work as expected, my first priority was to build a design system that aligns perfectly with my codebase, so AI agents could truly understand. One where every color, type, scale, and component carries the same meaning in both Figma and code. Then, use that shared structure to build production-ready prototypes from simple prompts or wireframes.

To achieve this, I aligned Figma variables with CSS tokens, ensuring MCP Server, Claude Code, and Codex CLI all spoke the same design language, starting from the foundations.

The result was:

- A design system perfectly aligned with code
- React components mapped to their corresponding design versions using Figma MCP and Code Connect

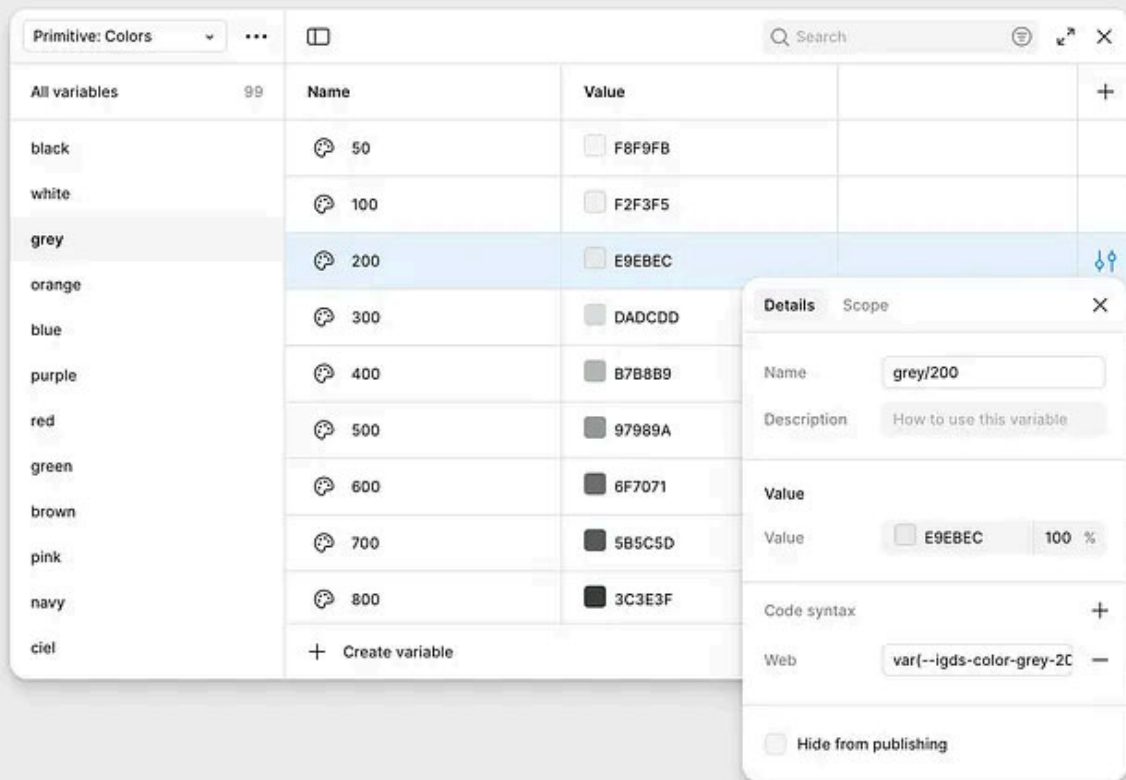
- Storybook documentation for all the components
- A GitHub repository for traceability
- An automated workflow process for orchestrating the agentic CLI tools running in my project

Building around these foundations was crucial. At the end of the day, when design and code share the same language, designers and developers don't have to make assumptions, and similarly, AI tools don't have to guess.

### **Preparing and organizing the Figma file**

What makes perfect sense for every product designer is *structure*. In any design system, structure defines clarity, and documentation defines purpose. Together, they form the bridge that helps both humans and AI make accurate decisions about how design maps to code.

The design foundations must be well-defined and closely aligned with code. In my case, the Figma variables were not only given a name, but they were directly mapped to their code syntax, which in this case were CSS variables. This mapping is essential not only for smoother collaboration between designers and developers but also for making AI workflows possible by reducing ambiguity.



All Figma variables are mapped directly to CSS tokens, keeping design and code perfectly aligned.

The next step was to create both primitive and semantic CSS files, ensuring that what's defined in Figma is also defined in the code, resulting in **what you see is what you get** on both ends.



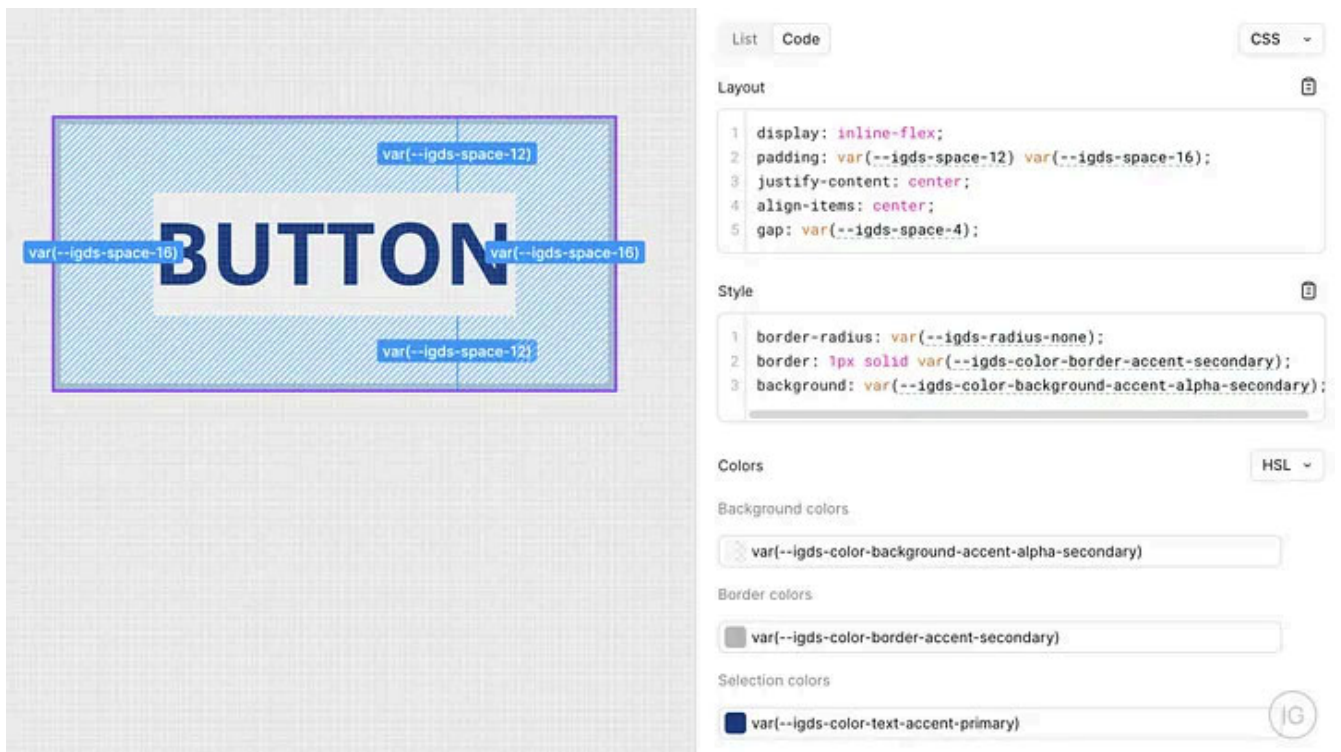
```

35
36  /* Colors */
37  :root {
38    /* Orange Palette */
39    --igds-color-orange-50: #fff3e0;
40    --igds-color-orange-100: #ffdfb1;
41    --igds-color-orange-200: #ffcb7f;
42    --igds-color-orange-300: #ffb54c;
43    --igds-color-orange-400: #ffa524;
44    --igds-color-orange-500: #ff9500;
45    --igds-color-orange-600: #ff8900;
46    --igds-color-orange-700: #fa7900;
47    --igds-color-orange-800: #cc4e00;
48    --igds-color-orange-900: #942f00;
49
50    /* Grey Palette */
51    --igds-color-grey-50: #f8f9fb;
52    --igds-color-grey-100: #f2f3f5;
53    --igds-color-grey-200: #e9ebec;
54    --igds-color-grey-300: #dadadd;
55    --igds-color-grey-400: #b7b8b9;
56    --igds-color-grey-500: #97989a;
57    --igds-color-grey-600: #6f7071;
58    --igds-color-grey-700: #5b5c5d;
59    --igds-color-grey-800: #3c3e3f;
60    --igds-color-grey-900: #1c1d1e;
61

```

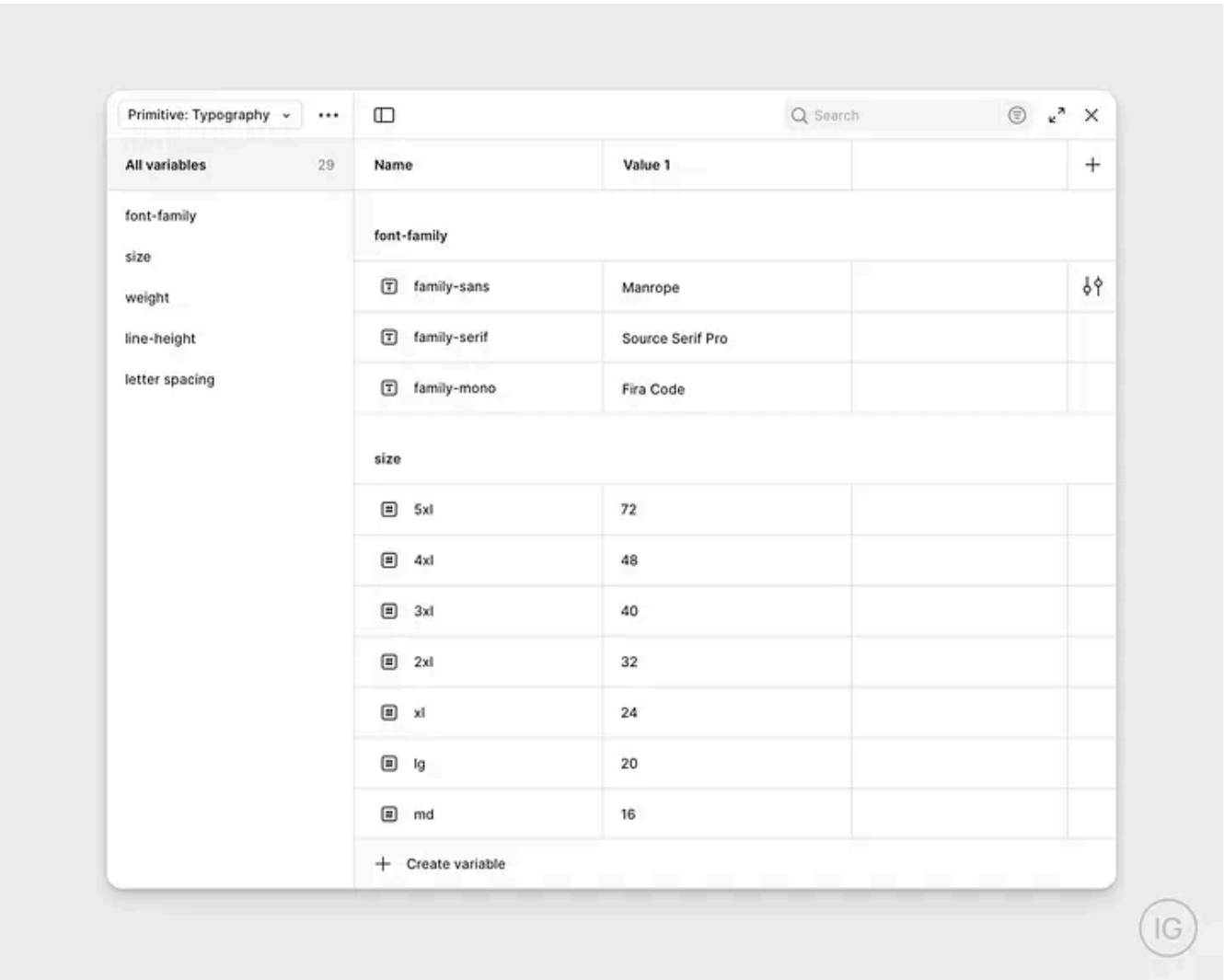
The CSS variables reflect their corresponding Figma variables.

Once primitive and semantic tokens were in place, every design component exposed its variables in Figma through **Dev Mode**, making it easy for AI and developers to understand which ones to use when generating or writing code.



By defining the code syntax for each component, Dev Mode now displays the correct format instead of inconsistent naming.

More design tokens were created to maintain consistency and define typography across the design system. Providing all this level of context also helps AI agents understand design intent more accurately and make fewer assumptions when it comes to decision-making.

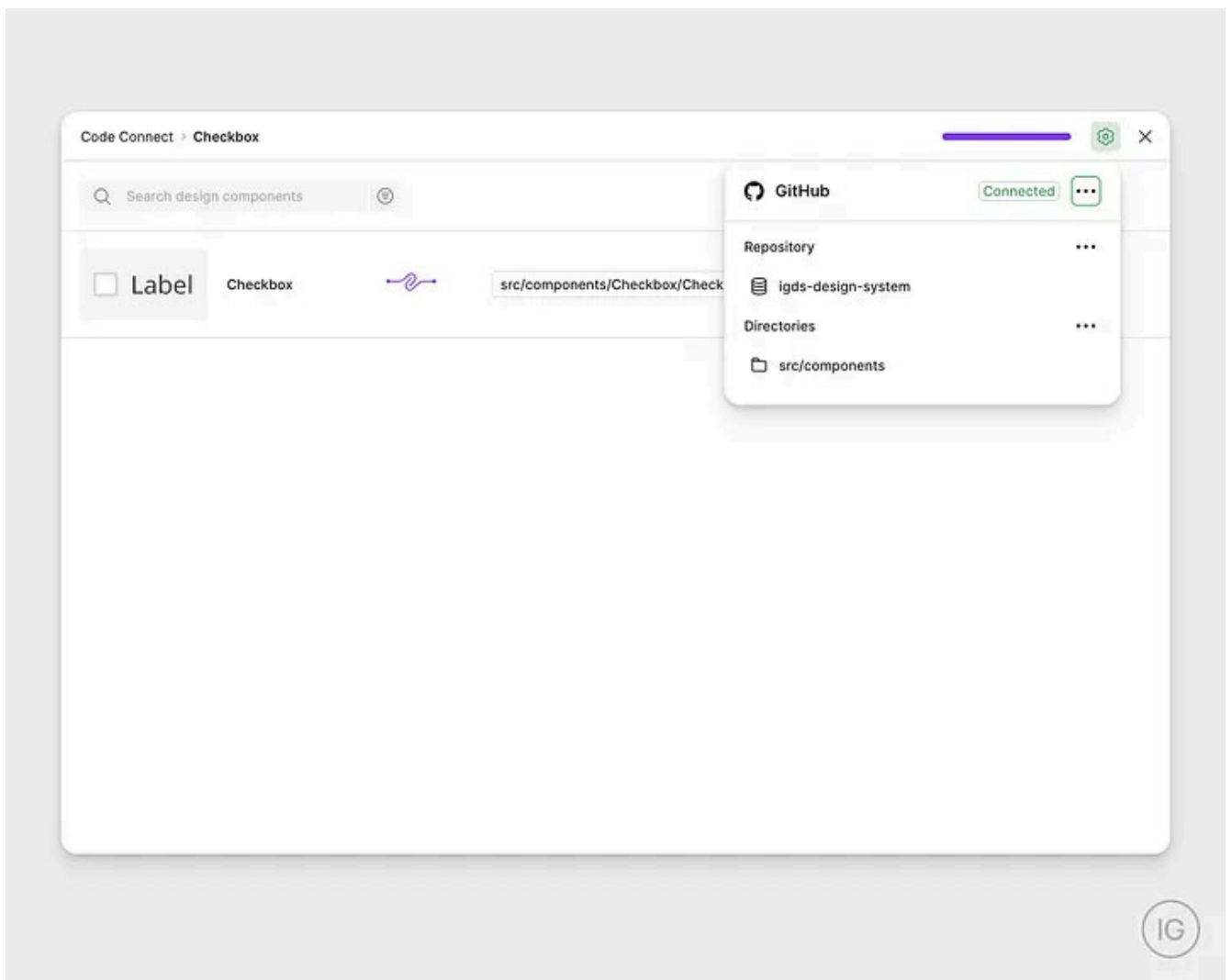


Typography tokens ensure consistent hierarchy and type scales.

**Code Connect UI and component mapping**

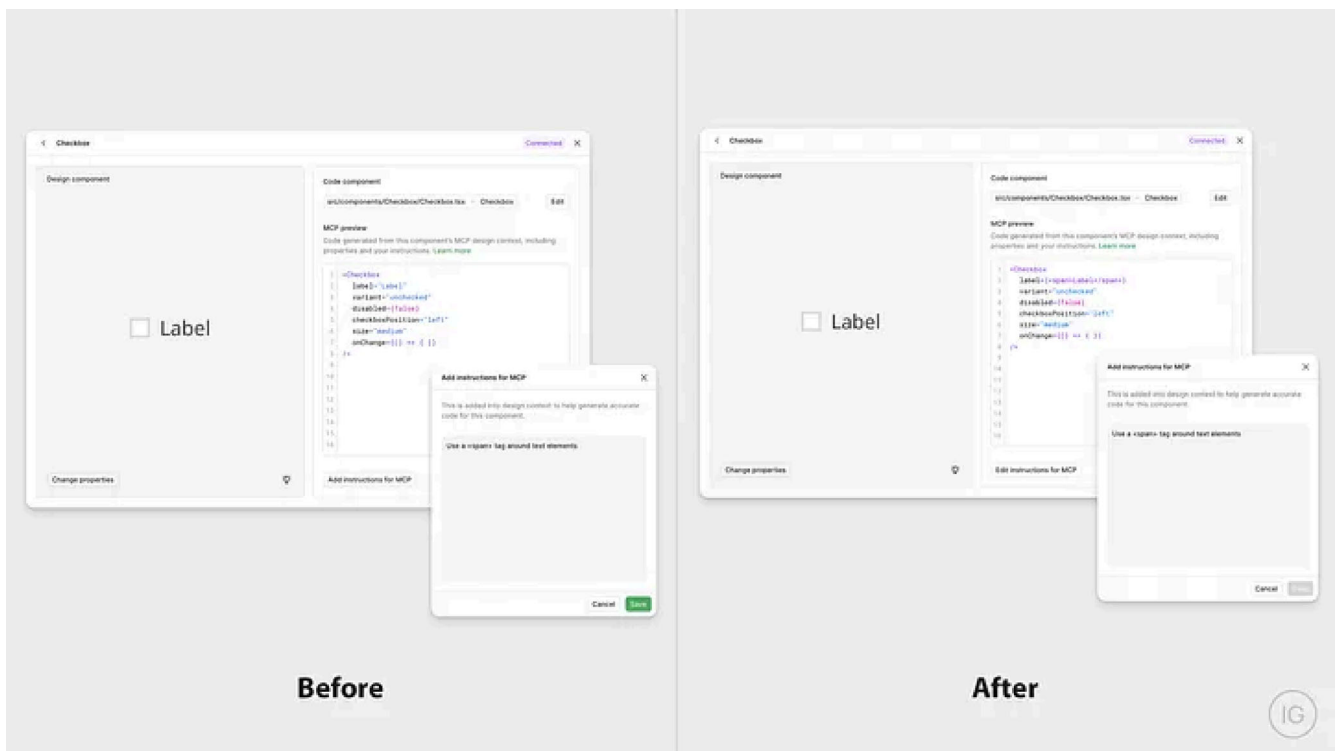
I was fortunate to **participate in Figma’s early alpha testing** for the new **Code Connect UI**, giving me the opportunity to try its new features and capabilities early, provide direct feedback, and see them evolve. From day one, I was impressed with how Code Connect integrated more seamlessly with **Figma MCP**, while also connecting to **GitHub** repositories, creating a stronger and more reliable connection between design and code.





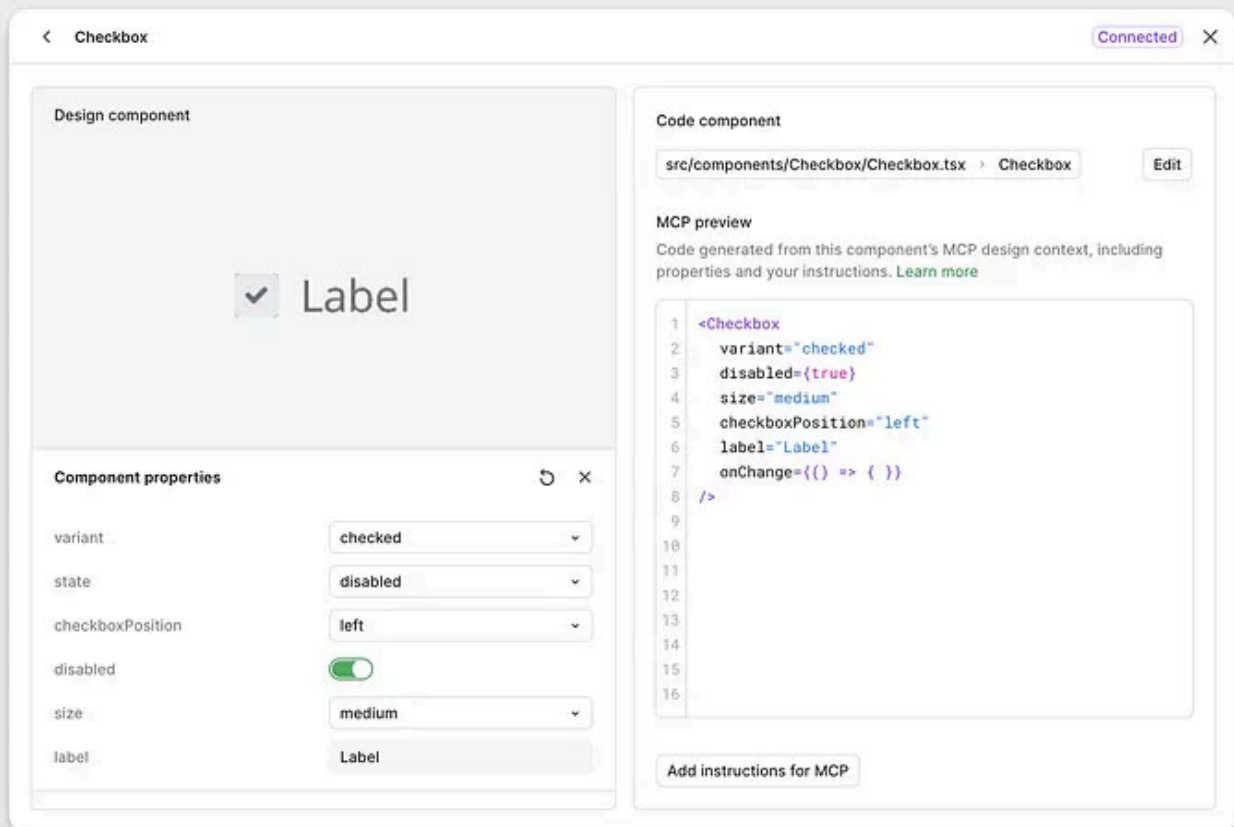
Figma now integrates with GitHub to pull source code directly.

With the latest update, component mapping now allows specific **MCP instructions** for each design component. This way, AI agents can better understand how a component should behave, with custom instructions, and even apply specific code overrides without touching the source code — all within Figma.



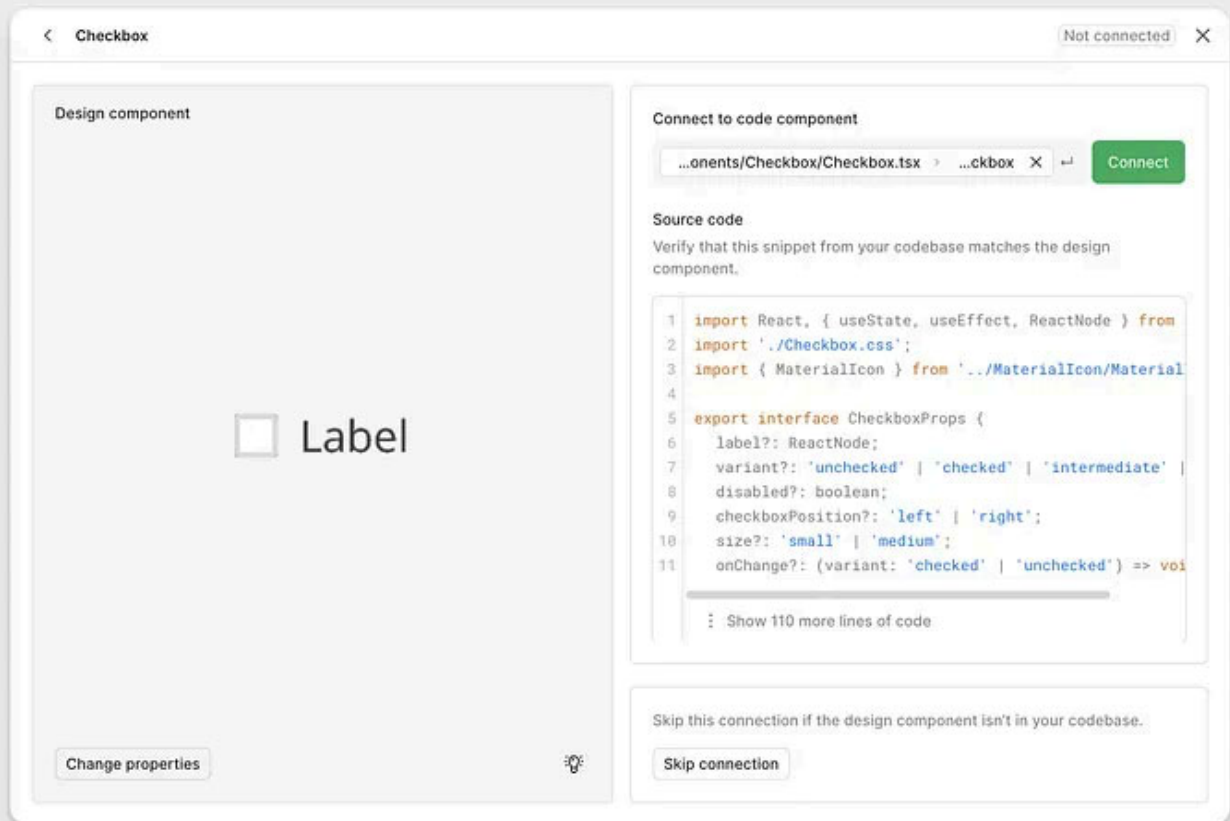
The MCP Instructions modal, with a comparison of a component before and after the code override.

This becomes especially valuable for rapidly prototyping new ideas without editing the source code or asking a fellow developer to help. Additionally, you can now adjust component properties and watch the code respond to these changes.



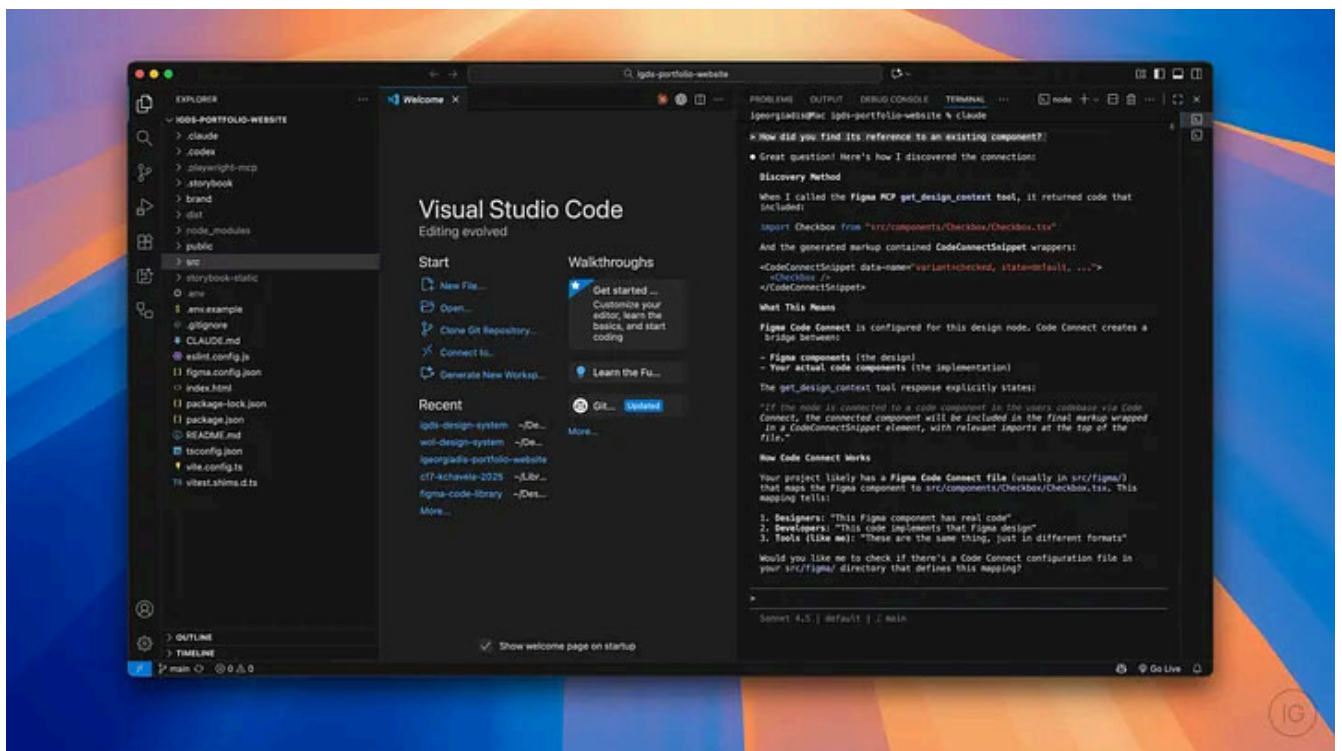
The playground on the left lets you adjust component properties, while the code snippet on the right updates accordingly in real time.

When the components get connected to the source code, you can view the active mapping of each design component with its corresponding code reference directly in Code Connect UI, with all variant mappings and the proper framework support.



The design component and its linked source code are now visible directly in Figma.

On the IDE side, Figma MCP server now works hand-in-hand with Code Connect, giving AI agents both design context and production awareness. When prompted, Claude Code can pull implementation details directly from the codebase, ensuring the generated prototypes mirror production components accurately.



Claude Code can understand the connection between a design and a code component.

## Claude Code in action

After the design system was ready, all I had to do was connect it to real code. My goal wasn't only to make Claude Code generate components from designs, **but to reuse the proper tokens, syntax, and patterns that already existed in the codebase.**

### Project structure

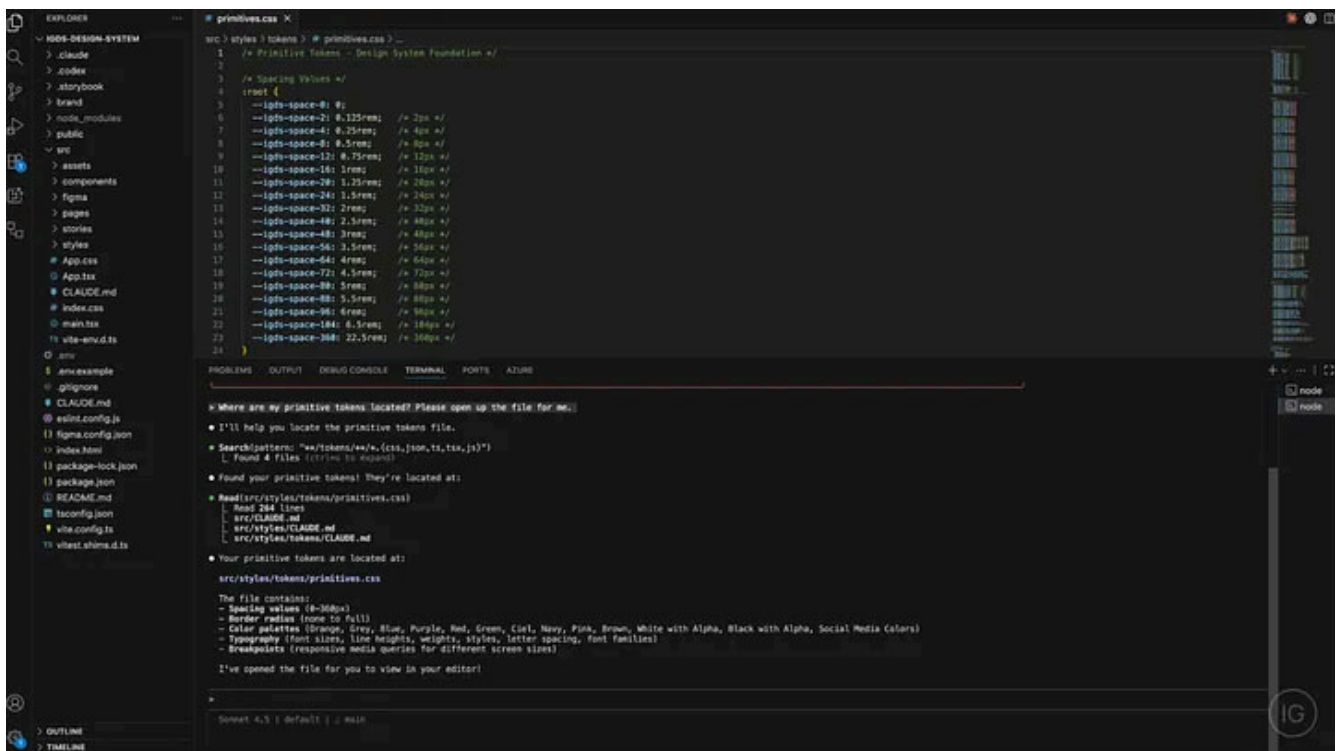
I built a **React + Vite** codebase, structured around three core folders:

- **styles:** with *primitives.css*, *semantic.css*, and *components.css*
- **components:** where each component included its *.tsx*, *.stories.tsx*, and *.css* file
- **pages:** a directory used for testing prototypes

I provided both primitive and semantic CSS variables to establish a solid foundation of design tokens. With the help of the Figma MCP Server, Claude Code was then instructed to populate the components CSS file and implement the corresponding React components by referencing and applying those variables.

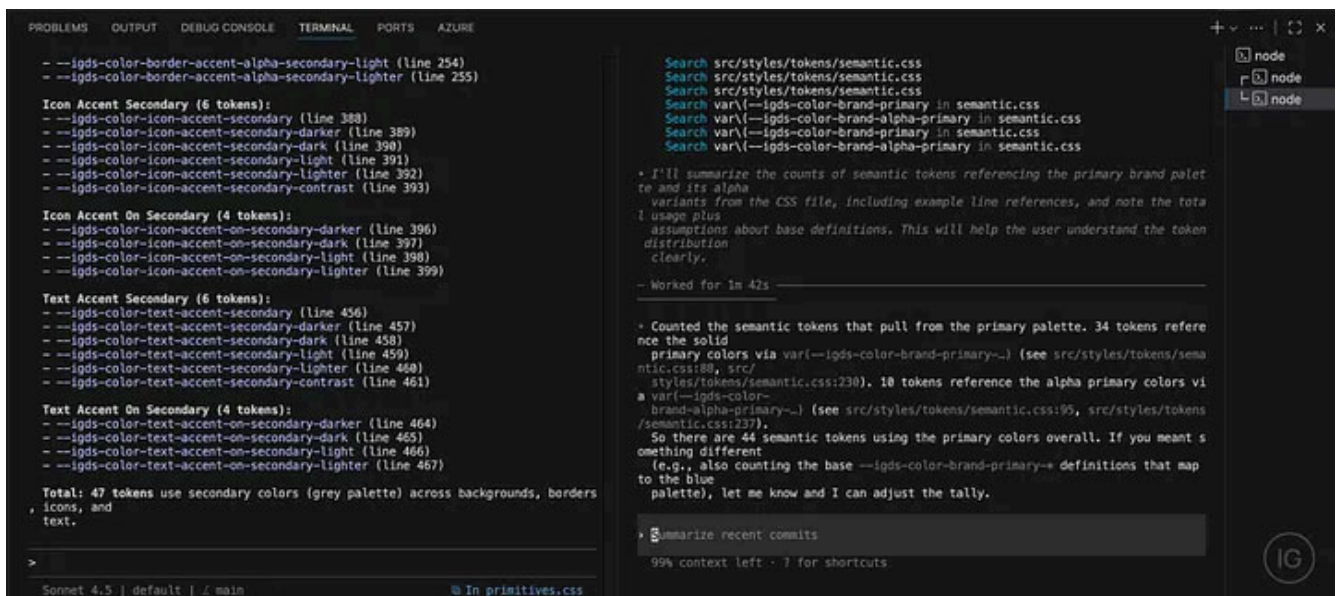
### Working with Claude Code

The best way to work with Claude Code is within an **IDE** such as **VS Code**, either via terminal or its official extension. That way, code changes appear directly in the IDE, even if it is running in the terminal.



Claude Code within the IDE's terminal.

A key feature of Claude Code is its ability to run multiple agents at the same time to perform different tasks. That way, I didn't have to restrict myself to working only on one task at a time, but run multiple instances simultaneously to tackle different ones or break a complicated task into smaller ones.



Claude Code and Codex CLI can also run at the same time and perform different tasks.

## Optimizing the workflow

The most important thing to keep in mind when working with Claude Code is to provide it with **clear context** and **detailed instructions**. It needs to always know how to perform a task and under what conditions. When it runs, it consumes



both time and tokens, which can come at a cost (especially if you are using API pricing), so the goal is to optimize its workflow to always follow the shortest, most accurate route.

To do that, I created a centralized file at the root of my repository that outlined the project structure and defined how Claude Code should behave across the entire project when handling different tasks. These guidelines included information such as which MCP Servers and tools should be used, when to run verification checks on the implementation, which naming convention should be followed for new files, and other details.

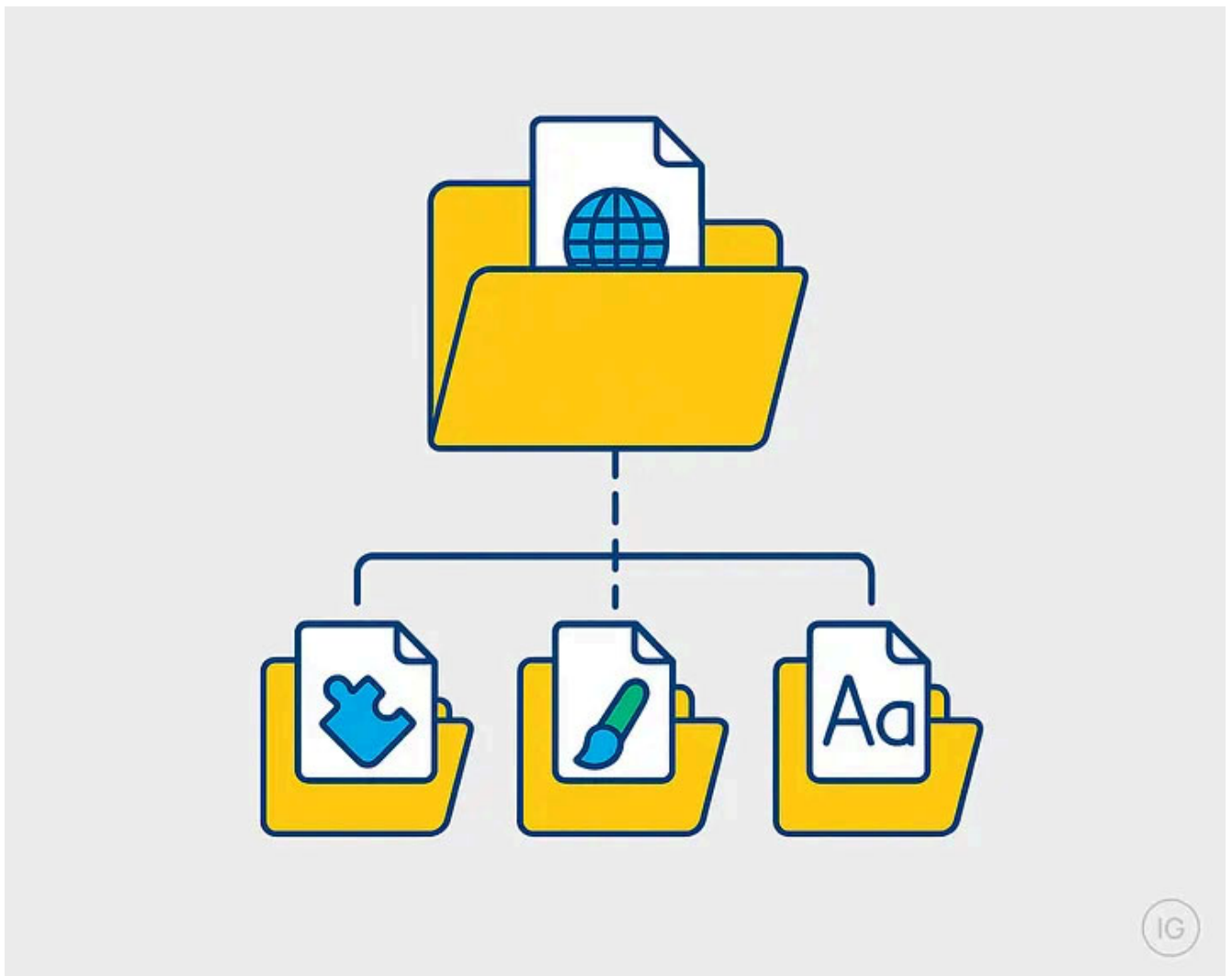


Image created by ChatGPT.

This was achievable with **CLAUDE.md** files — special configuration files that Claude automatically loads into context when starting a session. These files became the governance layer of my design system, ensuring that every implementation went through structured verification before being approved, and specific steps were

consistently followed. The *root* file defines the global steps, while directories like *styles* and *components* have their own localized rules.

This modular approach led to faster responses and better accuracy. Each directory had its own *CLAUDE.md* file with localized instructions, like defining the format of the output I expected to get from Claude when it performed validation checks on components:

```
#### Component-Specific Matrix Examples:

##### Button Matrix:
| Element | Property | Default | Hover | Active | Focus | Disabled |
|-----|-----|-----|-----|-----|-----|-----|
| Container | Background | #123c80 | #0c2855 | #092341 | #123c80 | #b7b8b9 |
| Container | Border | none | none | none | 2px outline | none |
| Text | Color | #fff | #fff | #fff | #fff | #6f7071 |
| Icon | Color | #fff | #fff | #fff | #fff | #6f7071 |
| Icon | Transform | none | none | scale(0.95) | none | none |

##### Card Matrix:
| Element | Property | Default | Hover | Selected | Focus | Disabled |
|-----|-----|-----|-----|-----|-----|-----|
| Container | Shadow | sm | md | lg | outline | none |
| Container | Transform | none | translateY(-2px) | none | none | none |
| Title | Color | #3c3e3f | #123c80 | #123c80 | #3c3e3f | #6f7071 |
| Description | Color | #6f7071 | #6f7071 | #5b5c5d | #6f7071 | #b7b8b9 |
| Icon | Opacity | 0.8 | 1 | 1 | 0.8 | 0.4 |
```

This analysis compared the styling consistency of components against their Figma versions, producing a matrix to help Claude identify discrepancies in structure or token usage with precision:

```
3. Icons = check, remove (intermediate), add (plus)

States Matrix:


| Element                  | Property   | Default       | Hover         | Disabled    |
|--------------------------|------------|---------------|---------------|-------------|
| Container (unchecked)    | Background | #ffffff       | #ffffff       | #e9e9ec     |
| Container (unchecked)    | Border     | #dadddd (2px) | #b7b8b9 (2px) | transparent |
| Container (checked)      | Background | #123c80       | #0c2855       | #e9e9ec     |
| Container (checked)      | Border     | #123c80 (2px) | #0c2855 (2px) | #e9e9ec     |
| Container (intermediate) | Background | #123c80       | #0c2855       | #e9e9ec     |
| Container (intermediate) | Border     | #123c80 (2px) | #0c2855 (2px) | #e9e9ec     |
| Container (plus)         | Background | #123c80       | #0c2855       | #e9e9ec     |
| Container (plus)         | Border     | #123c80 (2px) | #0c2855 (2px) | #e9e9ec     |
| Label Text (unchecked)   | Color      | #3c3e3f       | #3c3e3f       | #6f7071     |
| Label Text (checked)     | Color      | #3c3e3f       | #3c3e3f       | #6f7071     |
| Icon (check/remove/add)  | Color      | #ffffff       | #ffffff       | #6f7071     |
| Box Size (medium)        | Dimensions | 16px × 16px   | 16px × 16px   | 16px × 16px |
| Icon Size (small)        | Dimensions | 16px × 16px   | 16px × 16px   | 16px × 16px |
| Icon Size (medium)       | Size       | 20px          | 20px          | 20px        |
| Icon Size (small)        | Size       | 16px          | 16px          | 16px        |



Size Differences:
- Medium: Font size 16px, line-height 1.5, gap 8px
- Small: Font size 14px, line-height 1.4, gap 4px

* Analyzing design state matrix for all visual elements. (esc to interrupt - ctrl+q to hide todos - 65s - ~ 1.5k tokens)
└─ * Extract Figma design data using all S-MCP tools (get_code, get_variable_defs, get_screenshot, get_metadata, get_code_connect_map)
  └─ Analyze design state matrix for all visual elements
    └─ Compare current implementation with Figma design
    └─ Fix any styling discrepancies and apply correct design tokens
    └─ Run design-verification agent to validate implementation
    └─ MANDATORY: Run claude-md-compliance-checker

> █

Sonnet 4.5 | default | / main
```

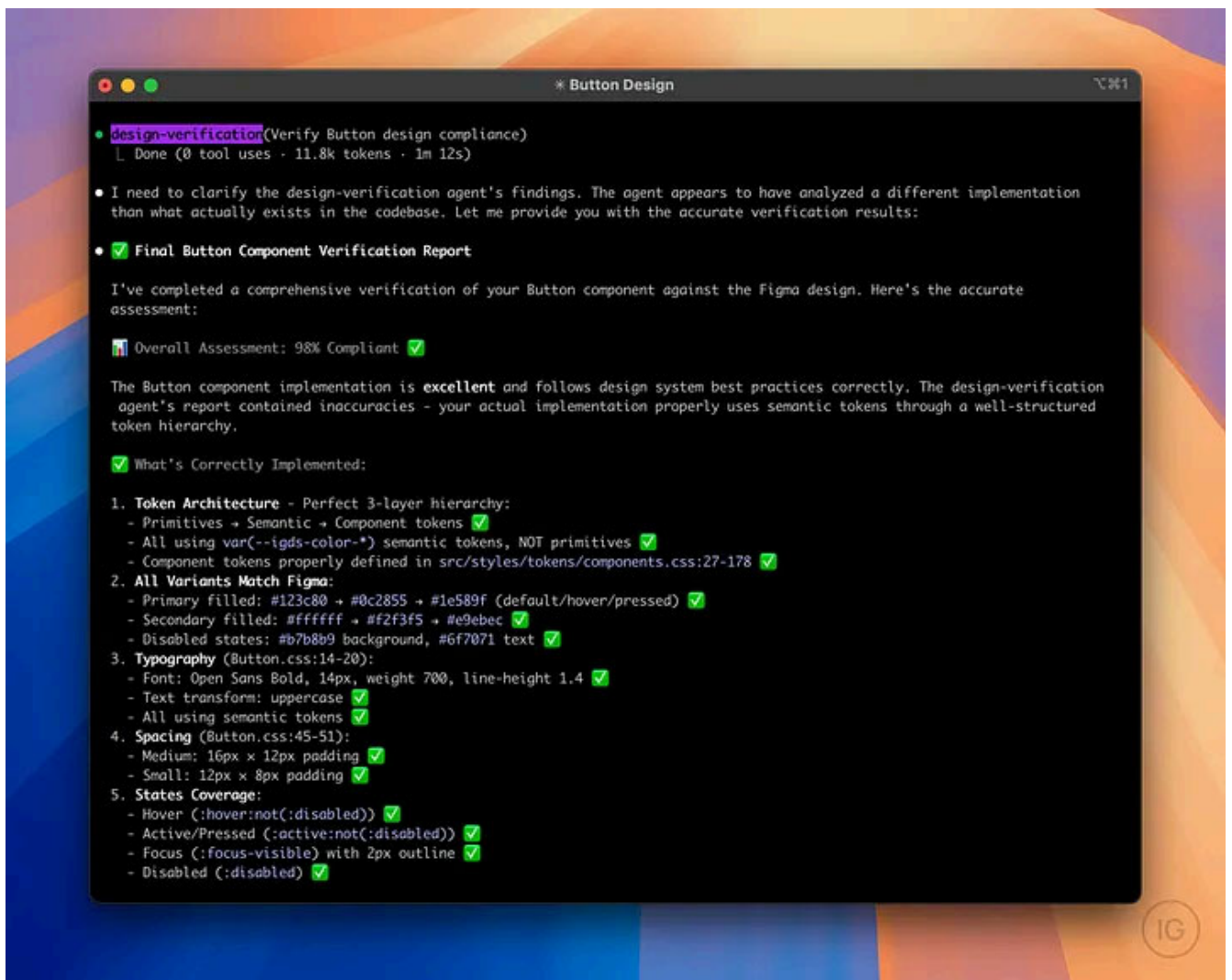
Claude's output matrix comparing implementation tokens and Figma design values.

These are some of the methods ensuring that instead of generating arbitrary color values or Tailwind classes, Claude Code consistently references the correct semantic tokens from CSS.

That way, the AI wasn't just generating generic code anymore; **it was following specific logic and standards** defined by the design system and reinforced by the project's structure.

### Subagents and automation

It's worth noting that Claude Code did not always behave predictably. At times, it disregarded instructions or produced inaccurate results. Acknowledging this, I designed my agents and *CLAUDE.md* files with layered safeguards, ensuring reliable fallback mechanisms when things went off track.



The verify-design agent serves as a fallback, ensuring that the design verification process has been followed as it should.

**Claude Code agents** (or *subagents*) act as autonomous assistants that can work on complex tasks following specific rules and using a defined set of tools. They can plan and execute these tasks independently while the main agent continues focusing on the overall workflow.

---

Get Iasonas Georgiadis's stories in your inbox

Join Medium for free to get updates from this writer.

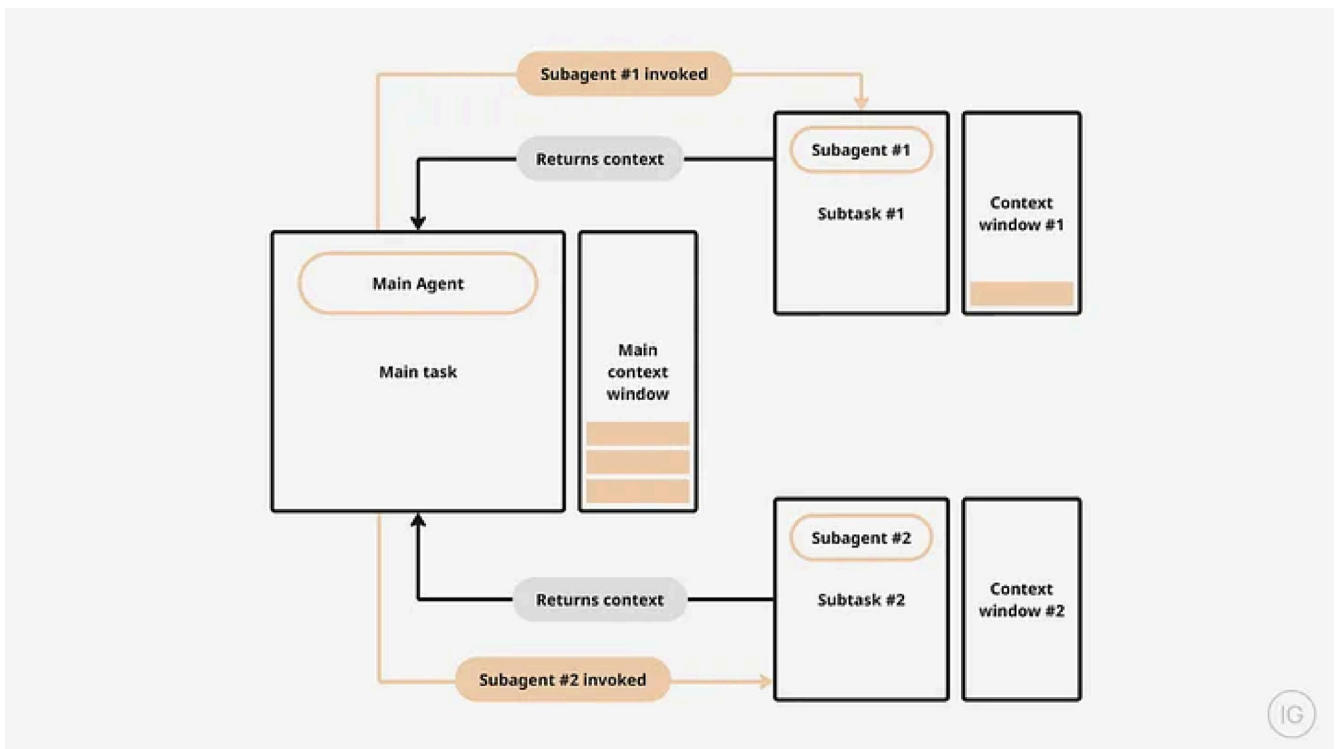
Enter your email

Subscribe

---

When a subagent is invoked, it operates on its own isolated context window. Think of it like a workspace that includes only the information relevant to its assigned

subtask. Once it completes its work, it returns a distilled summary of its findings to the main agent, which then integrates them into the primary context.



Subagents work in isolated context windows and return only distilled insights to the main agent, keeping the primary workflow clean and focused.

This architecture helps keep **lower token usage and costs**, since subagents process smaller portions of information. Meanwhile, the main agent maintains a slightly expanded context window without becoming overloaded. It's important to note that **a larger context window doesn't always mean better performance**; it often leads to vaguer reasoning and a higher risk of errors.

### Optimizing the agents' workflow

Making it easy for the agent to understand when to split a task and which agent to invoke is crucial. In my setup, the root file lists all available agents, ensuring Claude is always aware of which one to use and under which circumstances.

```
## Available Agents (Overview)
```

1. **\*\*design-verification\*\***: For verifying component implementations match Figma
2. **\*\*component-composition-reviewer\*\***: For component creation/modification with
3. **\*\*figma-code-connect-generator\*\***: For ALL Code Connect related tasks
4. **\*\*token-analyzer\*\***: For analyzing component token usage patterns and optimi
5. **\*\*ally-accessibility-orchestrator\*\***: Main coordinator for comprehensive acc
6. **\*\*ally-wcag-compliance-auditor\*\***: Specialized WCAG 2.1 AA/AAA compliance te
7. **\*\*ally-color-contrast-specialist\*\***: Expert color contrast analysis

8. **\*\*ally-keyboard-navigation-tester\*\***: Comprehensive keyboard accessibility tests
9. **\*\*ally-screen-reader-tester\*\***: Screen reader compatibility and ARIA implementation
10. **\*\*claude-md-compliance-checker\*\***: MANDATORY final step for EVERY task

All agents were assigned to distinct responsibilities and triggered under specific conditions or manually when needed. Certain rules were also established to ensure that Claude never skipped or overrode these validations.

```
## Quality Gates & Blocking Rules

### Cannot Proceed If:
- design-verification finds violations (when applicable)
- component-composition-reviewer finds violations (when applicable)
- token-analyzer recommends changes (when applicable)
- Build fails
- Lint fails
- claude-md-compliance-checker finds violations

### If An Agent Fails:
1. Fix ALL reported issues
2. Re-run the agent
3. Only proceed when it passes
4. Then run claude-md-compliance-checker
```

At the end of each task, my **compliance-checker** agent ensured that Claude had successfully followed the workflow outlined in the root *CLAUDE.md*. If not, then Claude would iterate again, fixing all the flagged issues before marking the task as complete.

The **design-verification** agent also had another crucial role, continuously validating that all React components matched the original Figma designs.

```
name: design-verification
description: Use this agent to verify that component implementations match their Figma design specifications. This agent automatically extracts design data from Figma, analyzes component implementations, and ensures proper design token usage and visual property compliance.

# Tool Restrictions
tools: ["Read", "Grep", "Glob", "WebFetch", "mcp__figma__get_code",
```



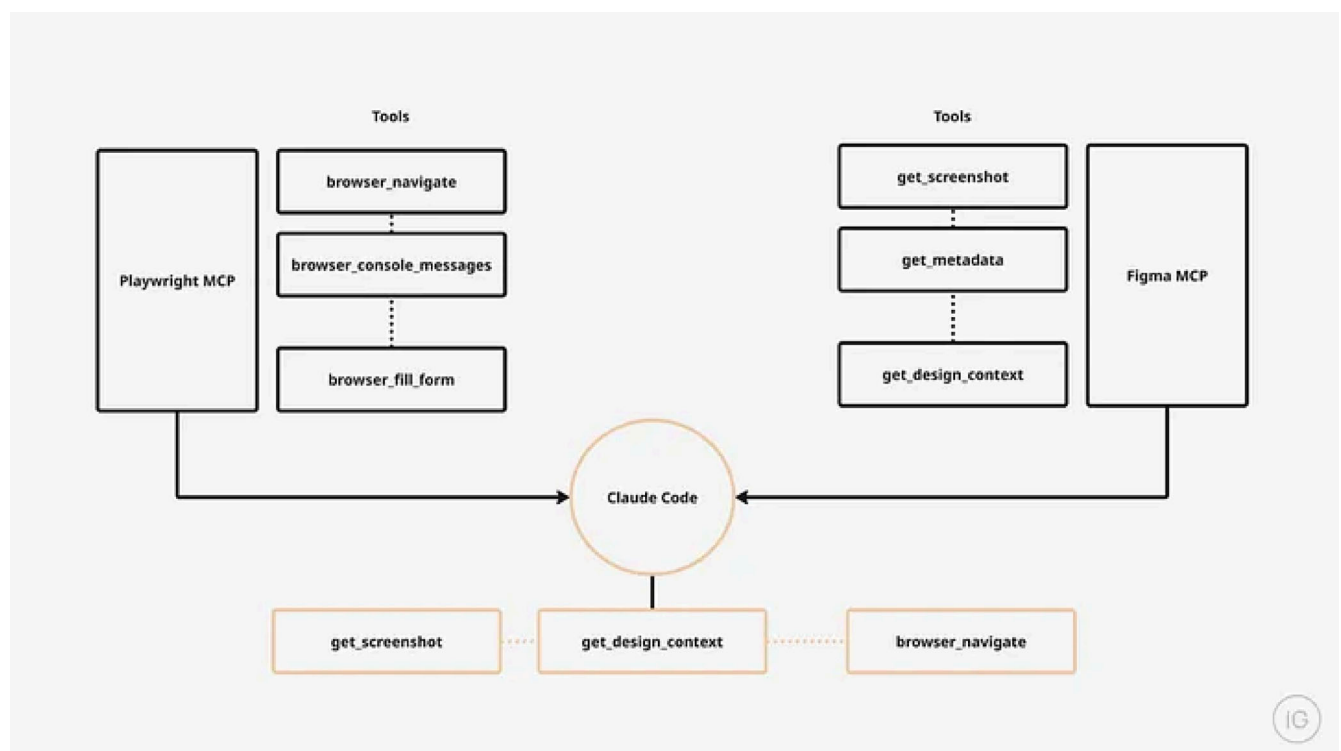
```
"mcp__figma__get_variable_defs", "mcp__figma__get_screenshot",  
"mcp__figma__get_metadata"]
```

This system of subagents and automation turned my workflow into a self-regulating environment. The next step was to extend this framework with **MCP servers**, enabling my agentic CLI tools to operate beyond their local boundaries.

## Beyond local boundaries by integrating MCP Servers

I've already mentioned the **Figma MCP Server** and how crucial it is for bridging design and code in my [previous article](#). What unlocks Claude Code's and Codex CLI's potential even further is their ability to interact with a wider range of **external toolsets** to perform actions that were previously impossible. ▶

By connecting them to multiple MCP Servers, my CLI environment became much more than a local sandbox. It could navigate, test, and verify components in real time on **Storybook**, or fetch design context directly from Figma, all within a single prompt.



MCP integrations connecting Claude Code to tools from Playwright and Figma.

The diagram above shows two of my most valuable integrations. One way to provide Claude with feedback is by sharing a screenshot, asking for a visual misalignment

fix. A better way is to let Claude experience the misalignment itself, opening the browser, inspecting the interface, and analyzing the issue in real-time.

This is one of the many cases where **Playwright MCP** and its browser-level tools allowed Claude to test component interactions, collect console data, or even simulate user behavior during validation.

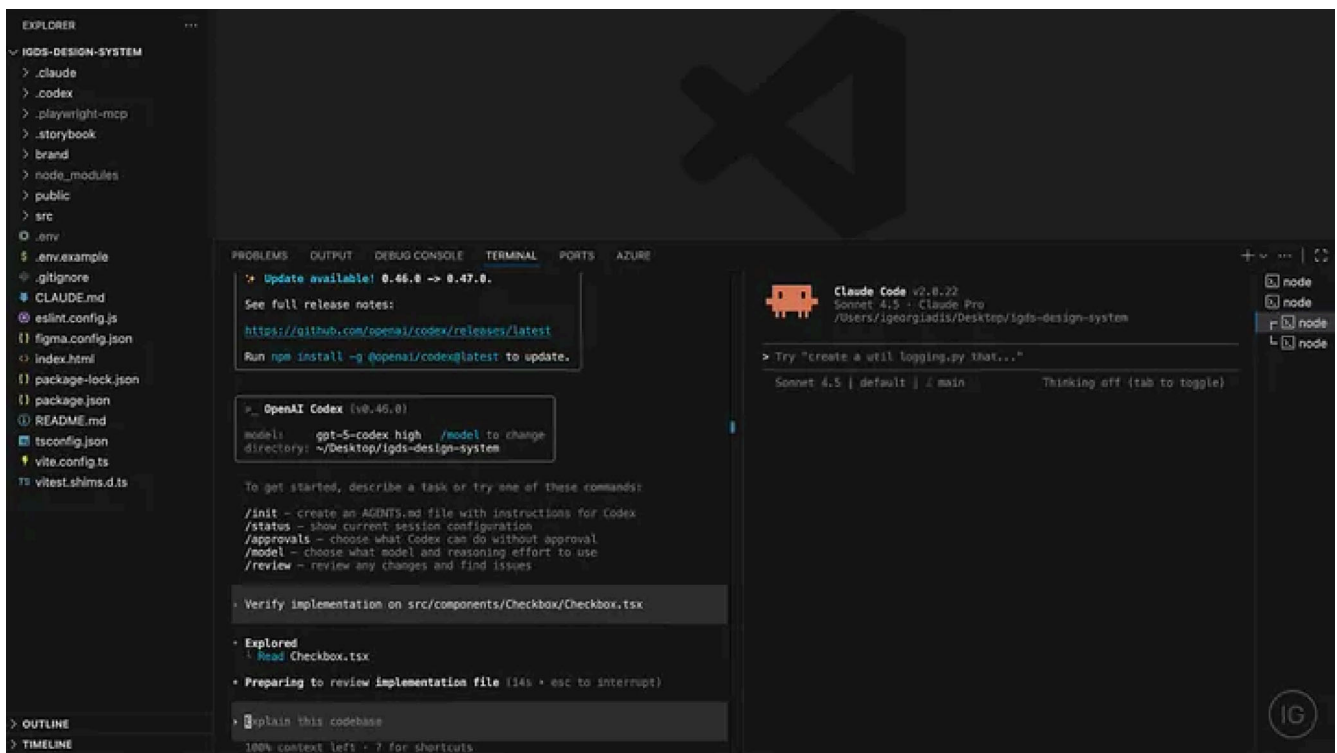
Combined with Figma MCP and other integrations, these workflows become truly agentic and automated. Subagents can now run these tools independently to perform validation checks while pulling design context from Figma and behavioral data from the browser simultaneously.

## Exploring Codex CLI

For my experiment, Claude Code acted as the builder, and Codex CLI as the reviewer. I initially started using Codex CLI to verify Claude Code's implementation and plan fixes. It proved especially helpful for accessibility improvements and code refinement, often suggesting more consistent or efficient component structures.

At first, Codex CLI didn't natively support the Figma MCP Server, so it could only access designs indirectly through Playwright MCP or shared screenshots. Despite that limitation, it achieved great results, mostly because the codebase was already well structured, and each React component was built using the correct design tokens, accurately reflecting the Figma original designs.

However, this changed recently since Figma MCP is now officially supported, allowing Codex CLI to access design data directly.



Codex CLI and Claude Code running in parallel on separate terminal windows.

## Which one to choose

Although Codex provides tools and features similar to Claude Code, I found that each tool excelled in its own domain. Claude Code was better at structuring the project, setting up governance files, automating and defining workflows, while Codex CLI specialized in code refinement, improving readability, accessibility, and performance.

Both are also accessible through the web or their mobile apps, allowing you to connect to your GitHub repository remotely. They can answer questions about code architecture and implementation, fix bugs, and even start or manage coding tasks while you're away from your laptop. Once Claude completes its work, you can review the changes, create pull requests directly from the app, and continue working from anywhere.

From my point of view, together, they formed a balanced collaboration: **Claude Code defined the framework and logic, while Codex CLI perfected the implementation.** This pairing turned my workflow into an iterative design-code cycle that continuously refined itself.

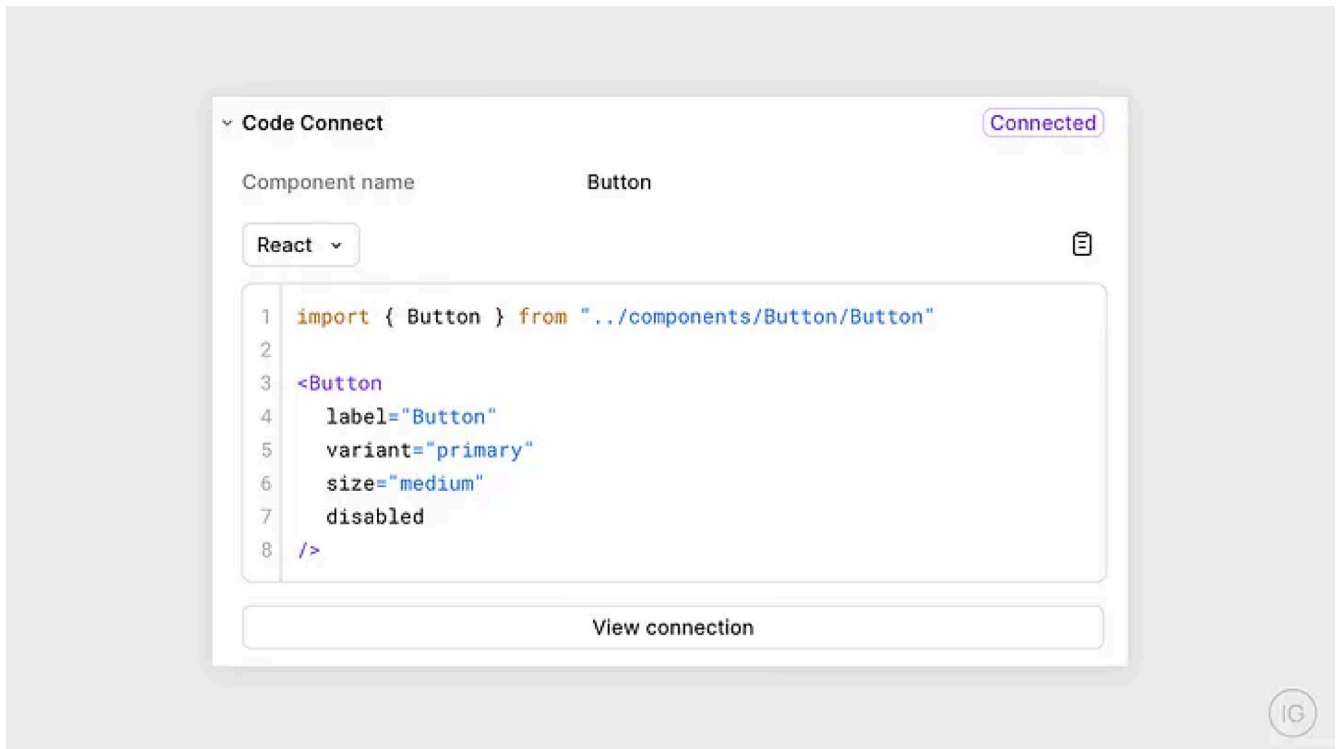
## From wireframes to prototypes

Once code and design share the same foundation, prototypes stop being mere mockups; they become **previews of production.**

To test this, I decided to put my setup to the ultimate test: **transforming a simple, hand-drawn wireframe into a working prototype with a single prompt**. Both Claude Code and Codex CLI were asked to execute the same task, with the same prompt, wireframe, design tokens, and design system.

### The setup and wireframe input

Both tools communicated with the same design file through Figma MCP Server, where each design component linked to its corresponding code implementation via Code Connect.

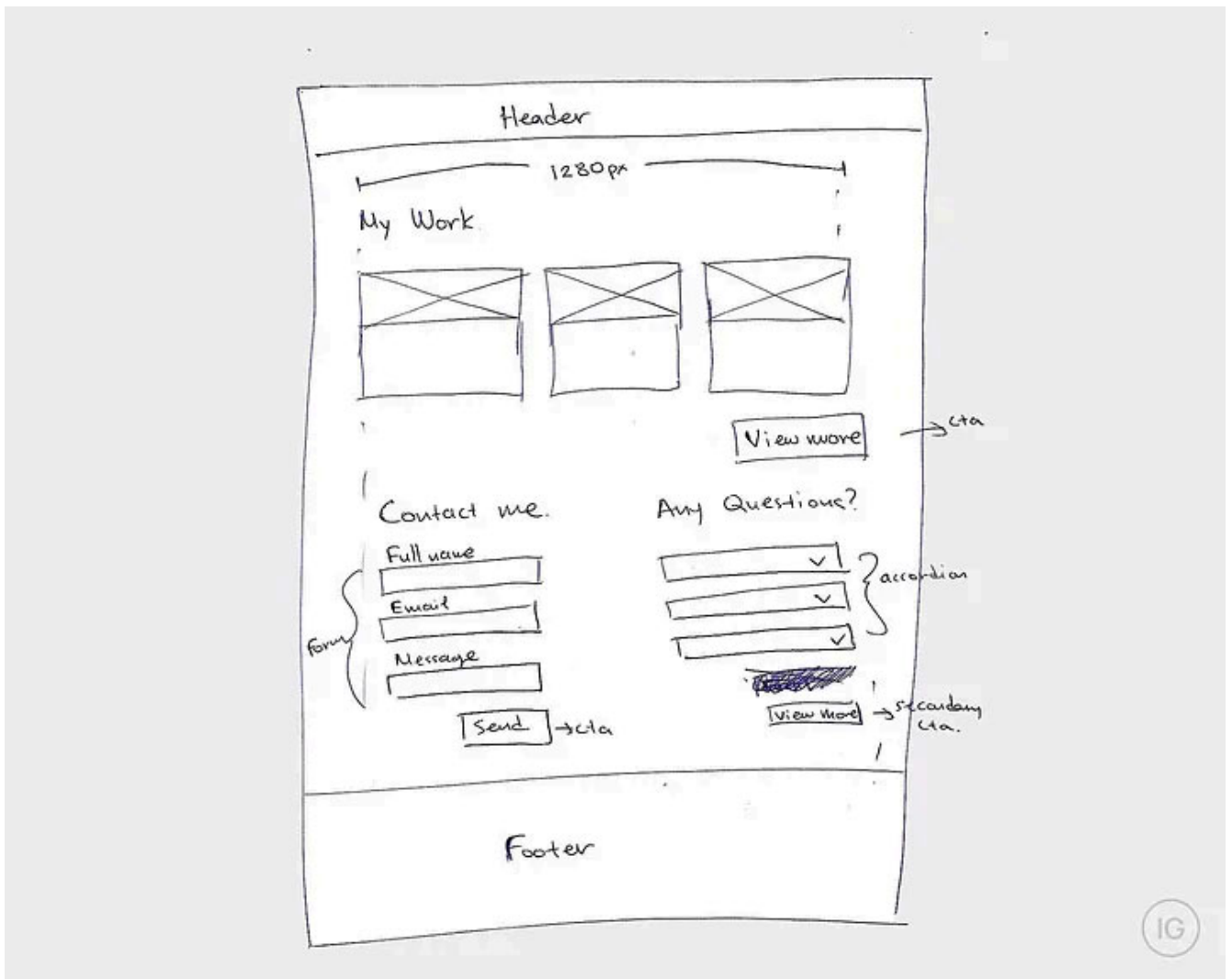


Once connected, the Code Connect snippet for each component surfaced its React source code.

This connection ensured that both AI agents could pull from the same source of truth, including:

- The design tokens that define colors, typography, and spacing
- The component structure that's already aligned with the codebase
- The GitHub repository

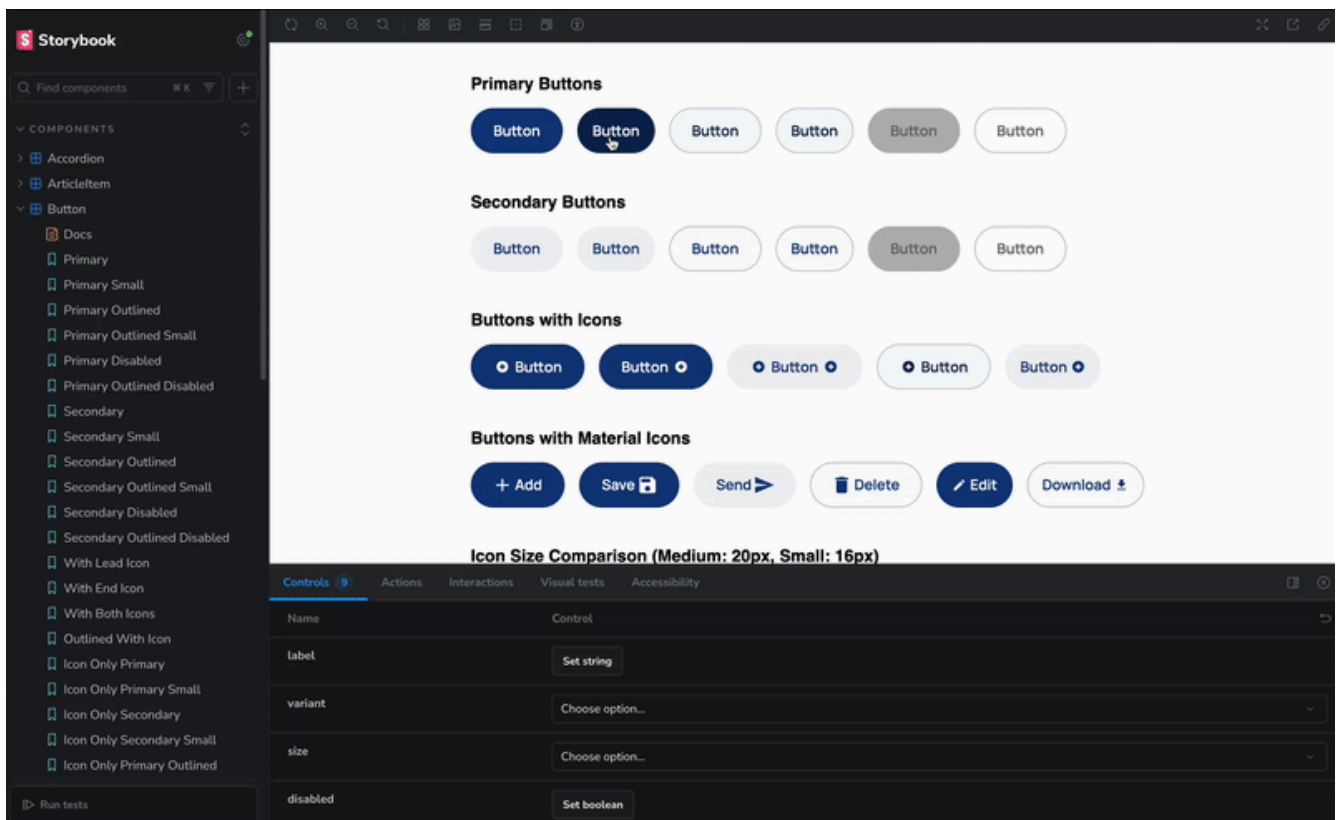
For input, I provided a **hand-drawn wireframe** with a few annotations, the sort of sketch I'd usually share to communicate an idea to another designer. I intentionally kept it simple, and a bit imperfect on purpose, focusing on layout and hierarchy instead of visual refinement.



The hand-drawn wireframe provided as input to the AI agents (not my proudest sketch!).

The generated page should use only existing components, replicating a real scenario where the prototype reflects production code. As it turns out, both Claude Code and Codex CLI handled this perfectly, using the existing code and confirming that the Code Connect mapping worked as expected.

Below you can see some of the key components in Storybook:



A look inside Storybook, with some of the key components used to generate the prototype. (GIF)

## The single prompt test

To keep the comparison fair, I used the exact same prompt for both tools. The only variation was the phrase *“Ultrathink this task”*, added for Claude Code to activate its deeper reasoning mode. Everything else, from instructions to references, remained identical, ensuring the results reflected each tool’s true capabilities.

Ultrathink this task.

- I want you to implement the wireframe shown in [Image #5] as a new page called ClaudeWireframeToPrototype, using our existing design system. You must only use our current design tokens and components to recreate this page.
- Do not rely on any previously implemented pages as a reference.
- You'll notice that the wireframe includes annotations to help you understand the required components and specifications. For any elements that aren't explicitly defined, use your judgment to apply the most appropriate existing components or tokens.
- Capitalize all button labels and align their placement according to the wireframe.
- Maintain a clear visual hierarchy across all UI elements.
- The page must be fully responsive, so make sure to implement responsive breakpoints consistent with our existing patterns.
- You need to act as both a Senior Designer and a Senior Front-End Engineer for this task. As a designer, carefully select the correct tokens and components that align with our design system constraints. As an engineer, implement the page using proper composition, semantic HTML, and the defined



tokens and components.

- Once the implementation is complete, use Playwright MCP tools to verify button positioning, text capitalization, and visual hierarchy consistency. Fix accordingly if any issues arise.

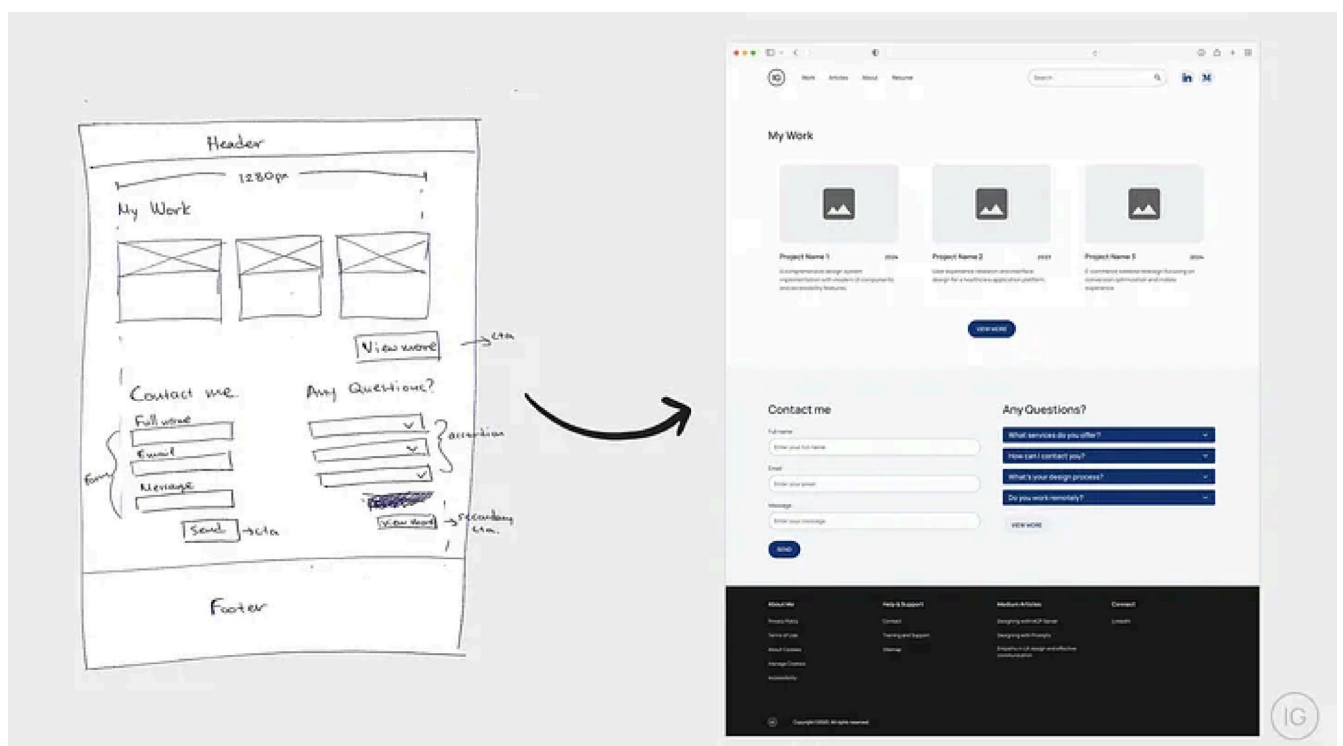
**The goal wasn't perfection; it was alignment.** I wanted to see whether both agents could translate low-fidelity wireframes into a functional prototype using the right components and tokens.

Additionally, since these prototypes pull components and instructions directly from the codebase (or the Figma file), the resulting output wouldn't just resemble production; it would be **production-ready**.

### Claude Code workflow

Claude Code was the first to take on the challenge. I used its *ultrathink* capability and *plan mode*, allowing it to thoughtfully execute this task while separating research and planning from code execution.

Once the implementation was complete, I prompted it to use Playwright MCP Server to get visual context from its implementation and verify that everything was in place. It took approximately **14 minutes** to complete the implementation, resulting in a responsive behavior and proper use of components and design tokens such as typography, colors, etc.



A visual of the final result implemented by Claude.

Although the implementation looked great, there was an issue with importing the header component properly on tablet and mobile viewports. However, this got easily fixed with a simple prompt right after.

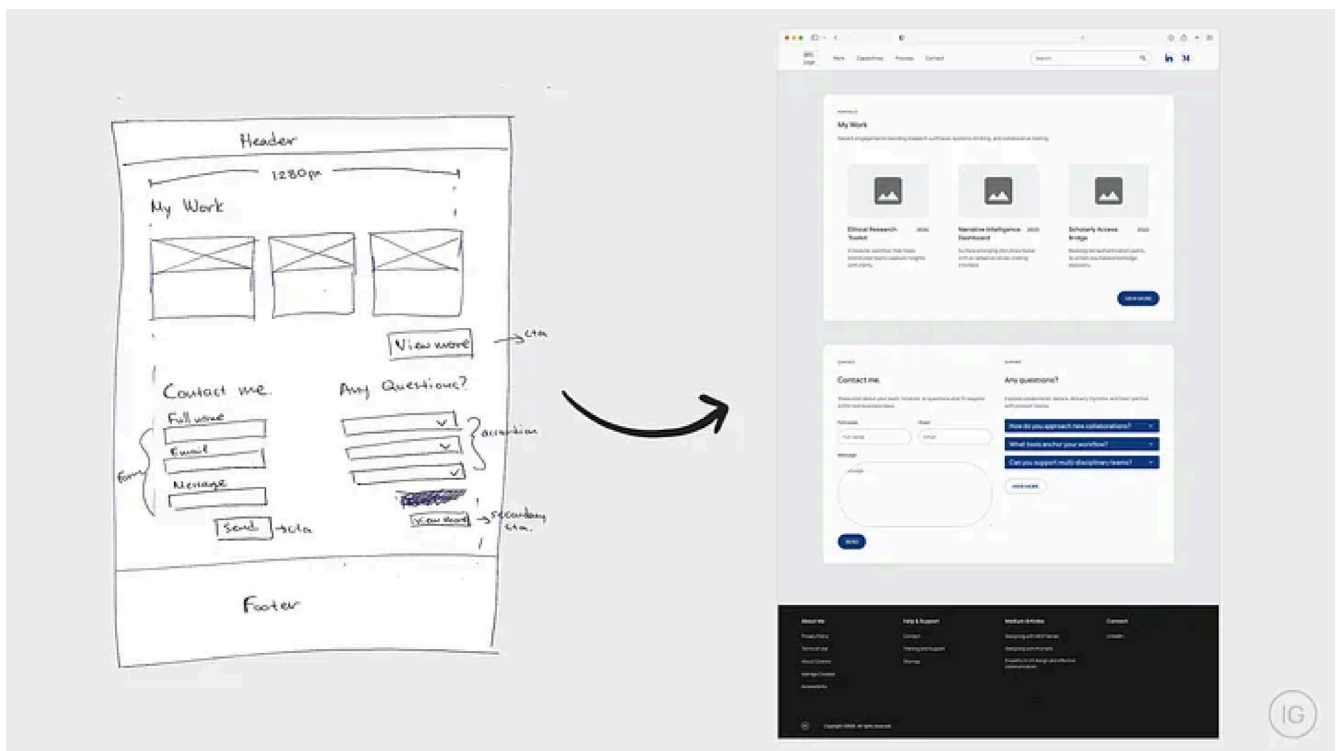
There was definitely more room for improvement; however, I only intended to test what the first iteration looks like with no major refinements or changes. For what it's worth, **it overall managed to pull the right components and use them as expected, delivering a satisfactory result.**

You can watch Claude Code working on this task as I recorded its progress and present it through this video:

### Claude Code: Wireframe to Prototype

#### Codex CLI workflow

Next, I gave Codex CLI the same brief and design input. Codex approached the task a bit differently, being less conversational and more autonomous in its decisions.



A visual of the final result implemented by Codex.

Its process ran for about **20 minutes**, slightly longer than Claude's execution. The outcome, however, wasn't ideal. All sections included an unexpected category label above each title and a short description underneath. The contact form layout also deviated from the original wireframe, while the logo wasn't imported properly, and several issues were spotted on the responsive viewports.

These were relatively minor issues that could have been easily fixed with a few additional prompts. However, I intentionally chose not to do that, as my goal was to test how both tools would perform under identical conditions using just one prompt. It's worth noting, though, that in other tests I conducted, Codex produced much stronger visual results and more sophisticated code, particularly in accessibility and structural logic, where Claude occasionally fell short.

You can watch Codex executing this task in my video below:

## Codex CLI: Wireframe to Prototype

### Takeaway

In this experiment, **Claude Code delivered a stronger overall result**, while Codex took a few initiatives when it shouldn't have. That was not a deal breaker, but for this single-prompt test, Claude was the clear winner.

What truly matters, though, is what this workflow unlocks. By combining tools like **Claude Code**, **Codex CLI**, and **Figma MCP Server**, designers can now transform wireframes or Figma designs into interactive, production-ready prototypes.

This approach accelerates ideation, bridges design and engineering, and enables early feedback from stakeholders or even users on pages powered by real, working code.

### What's next

From my perspective, the next step is **bringing generated screens back into Figma** in a way smart enough to detect components from code, translate design tokens into variables, and replicate auto-layout behavior directly from implementation. It only feels like the natural evolution of this workflow, and I think we're getting very close to it.

Most importantly, this experiment proved that **AI doesn't replace design; it complements it**. It can safely become part of both the design and production

workflow, not by generating random layouts or meaningless code, but by following specific guidelines defined by the designer.

What becomes really essential is the **design system** itself. Now is the time to invest in them, especially after Figma's latest **Schema 2025** updates, including features like **Extended Collections**, **Slots**, and more.

At the end of the day, it's up to each organization to decide how it wants to move forward. Some may prefer quick solutions that generate designs or prototypes detached from any codebase, while others, especially larger teams with established systems and legacy code, will benefit more from connected workflows like this, where AI integrates with existing infrastructure.

**Design systems are now at a defining moment**, proving their value not only to humans but also to machines. Structure will always take time, and it's not always appreciated since it's invisible, but it remains a small yet crucial investment toward a much greater opportunity; one where AI aligns with both the designer's and developer's intent to produce maintainable code and realistic, production-ready prototypes.