Chapter 6. Documentation and Technical Writing

Documentation is vital for clarity, consistency, and knowledge transfer in software development. It ensures that team members understand the code when onboarding and reduces the learning curve during day-to-day work, leaving less room for lost context and the consequent errors and refactorings.

Documentation is also important for nontechnical stakeholders such as product managers, customer-support representatives, and those working in marketing, sales, and operations. Clear documentation fosters collaboration across teams and creates a single source of truth that prevents miscommunication. As software evolves, proper documentation simplifies codebase maintenance and onboarding for new developers, bolstering the longevity of the project.

Outside the company, documenting how to use a software product can help sales and marketing efforts, prevent difficulties during customer onboarding, and foster user engagement with the product. Writing features and workflows down for external stakeholders is also a great starting point for collecting their feedback on how to improve the product.

Despite its importance, documentation often doesn’t get written at all. Software engineers don’t usually enjoy writing for humans, so they often skip it if they can. But they are almost always under deadline pressure, and when they have to make compromises, documentation is often one of the things left behind. Even when it does get written, heavy workloads and time pressure often lead to rushed or incomplete content. Writing high-quality documentation takes time. Additional challenges include finding the right level of detail and keeping documentation up to date as systems evolve.

AI tools were helping generate written content for many years before the recent LLM-driven AI wave. Writing tools such as Grammarly, which helps find the correct words and fix mistakes, are especially helpful for those writing in a foreign language. In software development, tools such as Swagger

and Javadoc also use AI to automatically generate API documentation in tandem with code updates.

The tools I review in this chapter were launched more recently, mostly since the generative AI wave started in 2022, and all aim to extend the simplicity of generating documentation from code beyond simple modules (like APIs) and helpers (like Grammarly). Some aim to be competent enough to replace the need for human action in writing documentation.

Types of Documentation

There are four key types of documentation commonly found in software development:

API/SDK documentation

A critical resource for developers, documentation of APIs and software development kits (SDKs), provides clear, structured details about the functions, methods, and services available within a software system. These documentation interfaces serve as a bridge between different software components, ensuring that developers can integrate and use the system efficiently.

Internal documentation and feature specifications

When business stakeholders define a new product or feature to be developed in order to fulfill a business objective, they write feature specifications to let software engineers know what functionalities to implement. The engineers' role is to extend those specifications with technical system designs, architectural decisions, and workflows that document not just *what* was implemented, but also *how* it was implemented. This type of documentation is vital for maintaining and evolving software projects over time, especially when the original engineers are no longer around.

User guides and manuals

These documents help nontechnical users understand how to use the software. They include everything from installation instructions to troubleshooting tips. They're useful during the sales process as support material for sales and marketing colleagues, and as customers use the

product. The challenge here lies in writing documentation for users who don't have a technical background.

Release notes and changelogs

These documents are used to communicate changes to the software, such as bug fixes, new features, or performance improvements. More than just keeping everyone informed, effective release notes inform both internal and external stakeholders of the need to update integrations and workflows to accommodate the changes.

Evaluation Process

I evaluated more than 20 AI tools in the documentation and technical writing space in order to shortlist the four highlighted in this chapter. Every tool covered here meets the following criteria:

- It is a professional project with a competent team behind it.
- It generates high-quality results.
- It offers some level of functionality for free or on a trial basis.
- It has a high level of adoption at the time of writing (mid-2025).

For this test, I created a very simple authentication flow, with both frontend and backend. The full code, which is available in this book's GitHub repository, contains flows for signup, login, and logout. I've used the AI tools in this chapter to document my code. My main point of comparison is whether the documentation produced can be useful for any of the four documentation use cases explained previously.

Again, for this test I gave preference to tools that can be used with a simple signup and free trial, so I stayed away from enterprise tools.

The full documentation generated for each test can be found in the book's GitHub repository.

Swimm

Swimm is an AI-powered documentation tool designed specifically for software engineers. It automates the creation and maintenance of code documentation. To ensure that it stays current with every code change, Swimm integrates directly into the code repository. Engineers can create

documentation for a certain code file or snippet, or create/update documentation with each new pull request (PR). The latter option is a great fit for most software development teams' processes, since a PR represents the most granular level of iteration to the codebase. Each such iteration needs to be documented, and each has the potential to render the existing documentation outdated.

I think this flow is comparable to the automated code reviews in Chapter 3. I can see how embedding these tools into a repo can provide a seamless integration into existing software development processes.

While Swimm can be blended into the repo and create or update documentation upon each PR, for the sake of this comparison test, I haven't used that exact flow. I've simply used Swimm's browser-based UI, which allows me to connect the repo, select specific files to be documented, and prompt for what to include in the documentation, as shown in Figure 6-1.

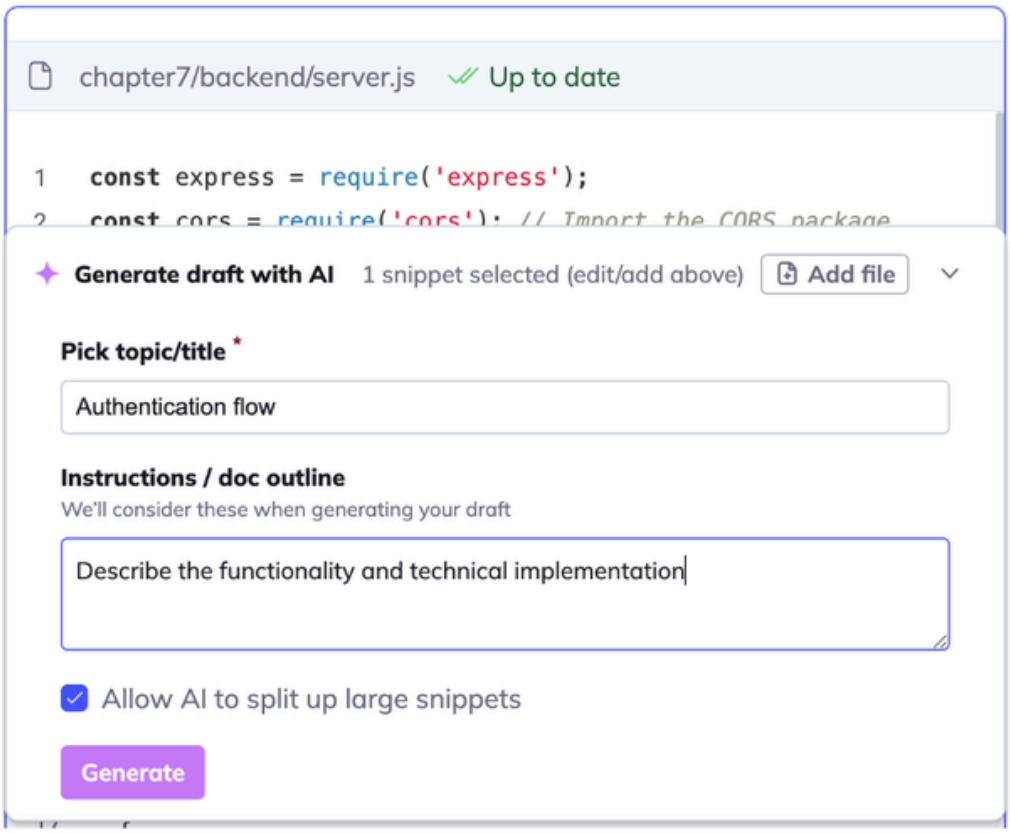

Figure 6-1. Swimm's widget to create a piece of documentation

In this flow, I've asked Swimm to document the backend part of my authentication flow with a simple prompt:

```
Describe the functionality and technical implementation
```

The desired output is a document that can be used for internal visibility on ongoing initiatives and for onboarding future team members. You can see a sample of the result in Figure 6-2.

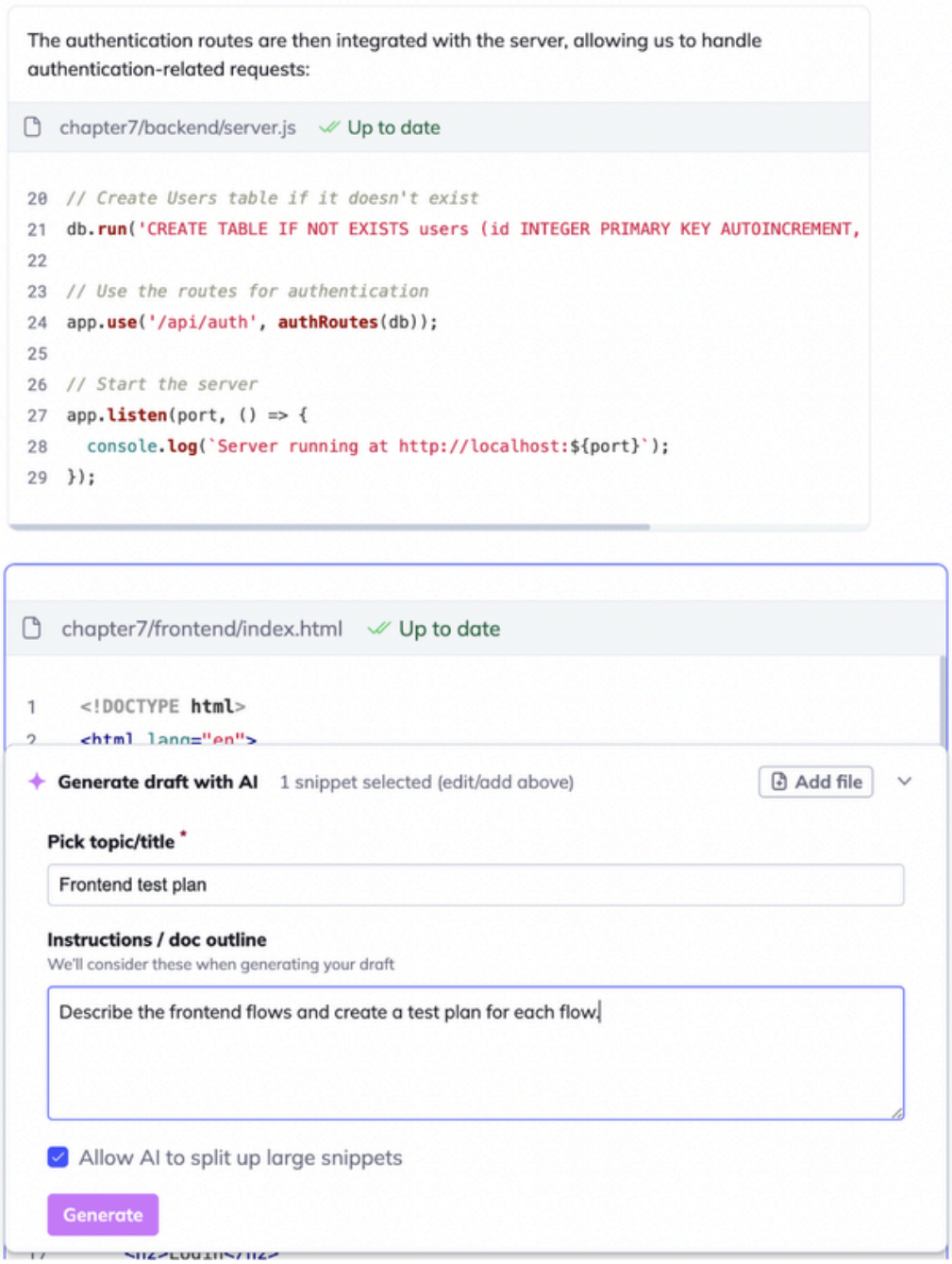

Figure 6-2. Sample of Swimm's output for the backend documentation

This output is quite good. I like the structure of the document as well as its content. However, my authentication flow is probably too simple to showcase Swimm's full potential. So I tested a second example for a more complex document:

```
Describe the frontend code and create a test plan for e
```

The result was again very good. It generated a full document (whose table of contents can be found in Figure 6-3), including a high-level introduction and then a deep dive into specific code components that impact the flow and thus should be documented.

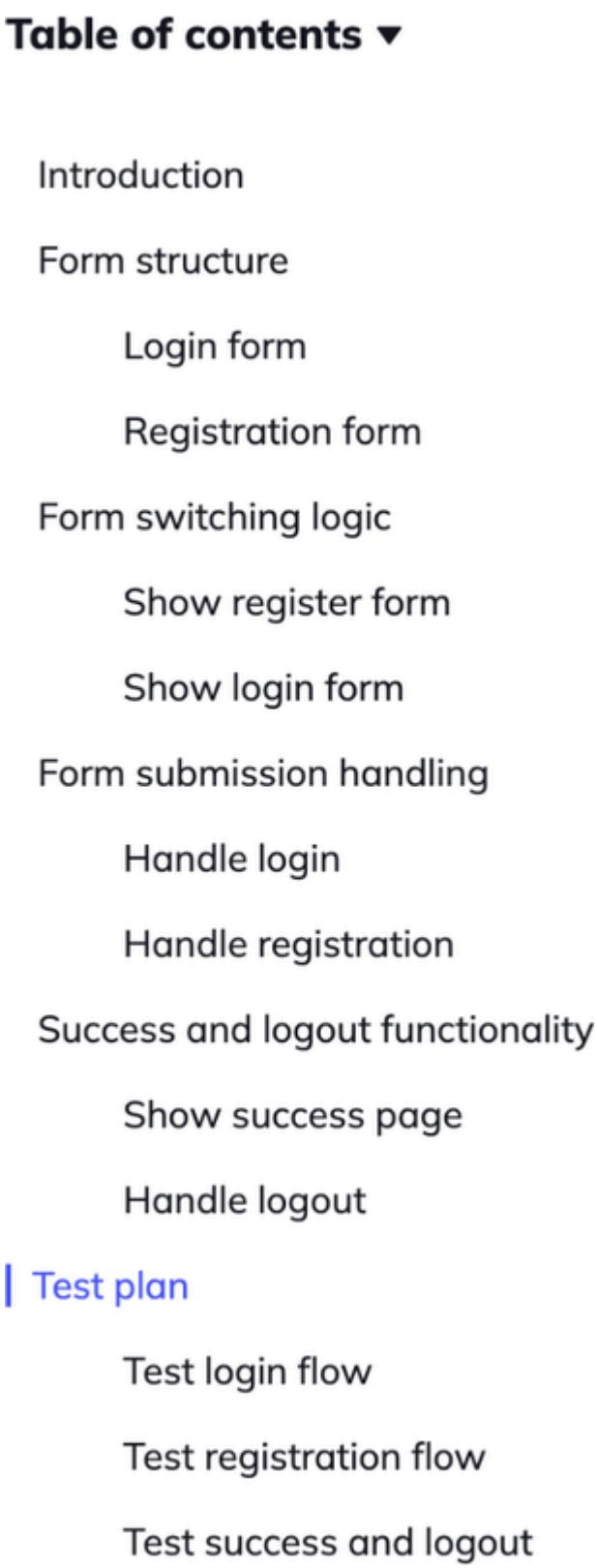

Figure 6-3. Table of contents of the document generated by Swimm for the frontend code

The last section of the document, as I asked, identifies the main flows of my code and provides test plans for each. The actual test plans are quite simplistic, but that's probably a byproduct of the simplicity of the underlying flow, as shown here:

Test plan

Test login flow

1. Verify the login form is visible by default.
2. Enter valid credentials and submit; expect a success message.
3. Enter invalid credentials and submit; expect an error alert.
4. Click “Register here” and ensure the registration form appears.

Test registration flow

1. Click “Register here” to switch to the registration form.
2. Enter valid details and submit; expect a success message.
3. Enter invalid details and submit; expect an error alert.
4. Click “Login here” and ensure the login form reappears.

Test success and logout

1. After successful login or registration, verify the success message is displayed.
2. Click the logout button and ensure the login form is shown again.

Swimm did well in this test. It was easy to get started with this tool, and it generated relevant documentation for my requests in correct Markdown format, which is the standard in technical documentation. However, I found it quite limiting that Swimm can only document one file of code at a time. This produces very fragmented pieces of documentation that are closer to a *read.me* file than a higher-level codebase and flow documentation.

I see working with a larger scope of source material as a natural evolution for Swimm, which could leverage its superior integration flow to create documentation for the whole codebase, or at least groups of files. It could work horizontally, documenting the structure of frontend code by using all frontend files as the object of a document, or vertically, documenting a feature flow by using all files related to that feature.

As such, I’m rating Swimm a 6/10. While the UX is good, the output is still far from the quality of documentation that I would accept from my teams.

ChatGPT

ChatGPT is most software engineers' go-to LLM tool for creating documentation, so I'm including it in this chapter, specifically the GPT-4o model, the most advanced available at the time of writing (mid-2025).

I started by prompting ChatGPT to generate the documentation for my code. I included in the prompt all six code files, a screenshot of the repository structure (so it understands the relationships between the code files), and instructions for what the documentation should include, as seen in Figure 6-4.

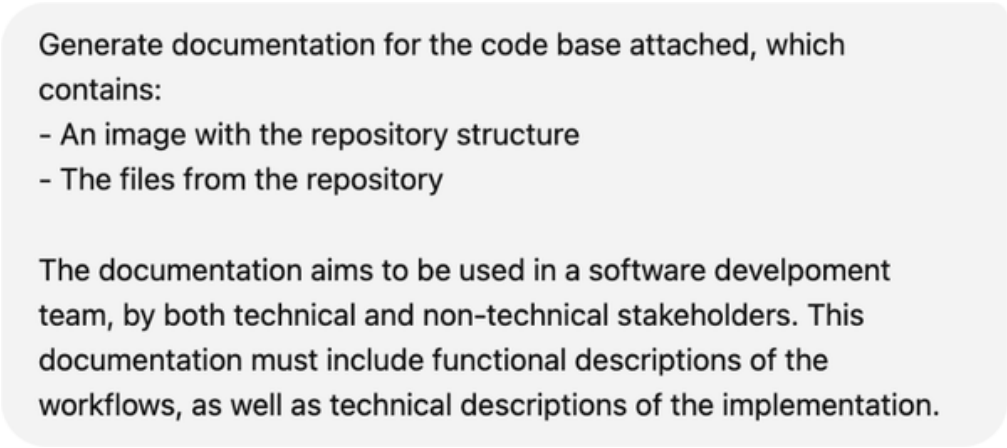

Figure 6-4. Instructions to ChatGPT to document my code

ChatGPT generated very comprehensive documentation, as seen in the table of contents in Figure 6-5.

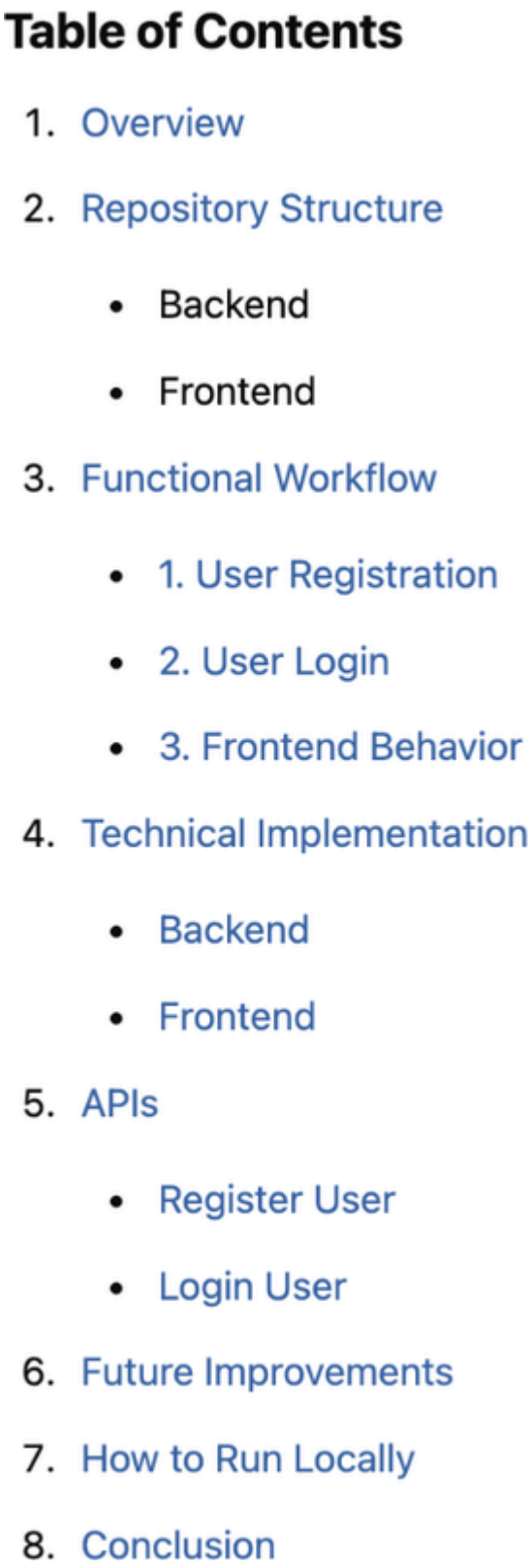

Figure 6-5. Table of contents of the documentation generated by ChatGPT

This is a really good output; it's very complete documentation with sections for all of the expected main components, from high-level context (such as repository structure) to a detailed deep dive in each specific component, such as the API, visible in Figure 6-6.

Figure 6-6. ChatGPT's documentation of the API module

You can ask ChatGPT to output the documentation directly into a Markdown file. I committed the final documentation generated by ChatGPT (as well as the other tools in this chapter) to the book's GitHub repository.

As expected, ChatGPT performs very well in this limited-scope test. It will work with up to 20 files at a time, and the file size limit varies by file type. While that's totally OK for small projects like my authentication application, it is insufficient for most production-level applications. On top of those limits, ChatGPT also offers an inconvenient UI, compared to tools that connect to the repository. The need to upload files manually and give ChatGPT contextual information about their structure and relationships makes it more challenging to use, especially for large projects.

As such, I'm rating ChatGPT a 7/10 for this use case. The quality of the documentation is very good, with the caveat of the limits and inconvenient UI. It would take a software engineer some creativity to document clusters of an application (by functionality or part of the stack, or module) within the limit of 20 files per piece of documentation.

Cursor

Cursor is a relatively new player in the AI coding tool space. It was launched in 2023 and has captured massive market share in the specific category of IDEs with AI code-assistant capabilities, which has been led by GitHub Copilot. As of August 2024, Cursor had 40,000 customers.[1]

Cursor's product is an AI-native IDE that started as a fork from the popular Visual Studio Code. It allows software engineers to select which LLM should power the tool; I've used Anthropic's Claude 3.5 Sonnet. As an actual IDE, Cursor has visibility into all code files in my repository, regardless of their number or size. You enter prompts through a chat feature, as shown in Figure 6-7.

Figure 6-7. Prompt to Cursor to generate documentation

The document Cursor generated was good, with sections for the expected main components, as seen in the table of contents in Figure 6-8.

Figure 6-8. Table of contents of the documentation generated by Cursor

Despite the very comprehensive outline and the relevancy of its content, Cursor has a significant pitfall when it comes to generating Markdown documents. For some reason (perhaps a bug), the generated content is only partially formatted as a Markdown file. It outputs some sections as raw text, such as the snippet in Figure 6-9. This makes it much harder to read.

Figure 6-9. Formatting issue in Cursor's generated Markdown document

Despite these formatting issues, the documentation generated is extensive, covers the right topics, and has the correct level of technical depth. It's definitely in line with what I would consider acceptable documentation from my teams. As such, I rate Cursor 8/10.

Scribe

Scribe is a very different tool from the others reviewed in this chapter. This tool is best suited for creating user guides, standard operating procedures (SOPs), or bug reports in a visual way. While my use of Swimm, ChatGPT, and Cursor focused on creating written documentation about the technical implementation of a certain product or functionality, I used Scribe to produce a guide about the product's functionality.

While Scribe was created in 2019 as a basic screen capture tool, the functionality I used for this test, called Scribe AI, was only launched in 2023.

It leverages the original functionality that allows a user to record a browser session, but instead of simply storing the video of the recording, it also creates an entire workflow with annotations, based on the screen transitions in the recording. That's why it caters to UI-related use cases, like bug reports and product guides.

To start the test, I installed Scribe's Chrome extension and used it to record a simple session of registering a new account and logging in to that account. My goal was for Scribe to generate a user guide that I could share with external nontechnical stakeholders, like users of the product.

The experience of recording my first session was quite seamless; I got the recording I needed easily on my first try. It's called a Scribe, the name for both the video recording and the annotated workflow that's generated, and it's available in this public link. I'd say this output is good, since it identifies the screen transitions in my workflow and captures the screenshots of each screen, highlighting the action that the user did on the screen to cause the transition. The result is in line with user shadowing tools like Hotjar or Fullstory, which are commonly used for user research and bug tracking.

Scribe offers a feature that converts the raw HTML document in the preceding public link into an AI-generated document. I used the authentication flow to test the feature, which allows the user to write a prompt specifying the documentation piece to be generated from the screen recording. My instructions were simple, as shown in Figure 6-10.

Figure 6-10. Instructions to Scribe to generate a document from raw tracking of website actions

The resulting document is publicly available here. I found this output underwhelming. It's generic; it feels like it could have been written about any application, not specifically about mine. It generated a document and embedded Scribes (specific flows) into it, as opposed to generating a document based on the flow I recorded, which was my intention. This makes me think that the tool might be a better fit to generate larger pieces of documentation that involve several different Scribes merged together in a large document (e.g., a product guide). The content of the generated document is not very relevant to the use case. As such, I'm rating Scribe a 5/10.

Tool Comparison

Table 6-1 compares the ratings for each of the tools discussed in the chapter.

Table 6-1. AI documentation tools overview

Tool	UX	Test performance
Swimm	Repository extension	6/10
ChatGPT	Website	7/10
Cursor	IDE	8/10
Scribe	Chrome extension	5/10

Conclusion

As a CTO for over a decade, I've found that documentation is one of those things that's always lacking, but never to the point where it's worth pausing ongoing work to fix it. In fact, bad documentation is a form of technical debt, but one that doesn't break systems or degrade performance. It *does* degrade the *team's* performance, however, which is a less visible and perhaps more damaging form of debt in a software development team.

I've always found it hard to push software engineers in my teams to write documentation in the first place, and even harder to keep that documentation organized, accessible, and updated. I think that AI tools like the ones I reviewed in this chapter can play a fundamental role in making that process easier. With a simple prompt, they can generate documentation within seconds. It would take a human at least an hour or two to generate a similar document. And that time commitment compounds with complexity: the larger a system is, the more challenging and time-consuming it is to document it properly and keep that documentation up to date. In a team of a few dozen people, that work could easily add up to thousands of collective hours of work a year.

While AI tools can create documentation instantly, they can also create bad documentation (just like humans can). I recommend that teams take the same approach to documentation as to setting coding guidelines: create a template for prompts or even for documents, with predefined sections and subsections. This serves as a backstop to avoid unnecessarily long documents, and facilitates readability by making content easier to find.

With all that said, documentation created by AI tools must *always* be thoroughly reviewed and edited by team members. While it takes seconds to produce 90% of the deliverable, the final revisions and quality control must be performed by human beings, since the output does not always fulfill the objective. See the case with Scribe, where the document generated is generic; a human reviewer would have caught that flaw and improved the documentation manually.

1 Anysphere Team. August 22, 2024. "Series A and Magic". *Cursor* (blog).