# M365 CoPilot Researcher - Approaches-Reasoning-Journal

PROMPT:

|  |
| --- |
| // Context: I’m going to be working with VS Code and GitHub Copilot on creating a feature, from the idea and docs, specs, to the implementation in C#.  // Objective: I also want to record/save somewhere, in some kind of documentation, the whole process I’m doing, including what I think and I’m going to do and also every prompt I create and use with GHCP, so later I can showcase the process and reasoning I did. What would be the best way to save/write my whole process and reasoning including my own prompts?  // Requirements: Initially open to multiple approaches  // Format: Initially open to multiple approaches  // Constraints: I don't want to over-engineer, I need to be agile. |

**Documenting Your Development Process: Tools, Formats & Best Practices**

Developing a C# feature with GitHub Copilot is not only about writing code – it’s also an opportunity to capture your **ideas, reasoning, and AI interactions** in a structured way. By documenting the entire process from initial concept to final implementation (including every prompt you give to Copilot), you’ll end up with a valuable narrative akin to a team’s project journal. Such documentation can be public-facing, well-structured, and read as if a team collaborated on it (even if you’re solo). Below, we discuss what to record, how to structure it, and compare platforms (GitHub Projects/Markdown, Notion, Obsidian) for capturing and organizing your journey.

A screenshot of a computer

AI-generated content may be incorrect.

**What to Include in Your Development Log**

To **showcase your process and reasoning**, make sure your documentation captures the following:

* **Initial Ideas & Requirements:** Write down the feature idea, project goals, and any research or spec snippets you started with. This is your “project brief” or problem statement.
* **Design and Specifications:** Document the architecture or design decisions. For example, note why you chose certain approaches, perhaps in a mini design doc or Architecture Decision Record. If you considered multiple solutions, list pros/cons before deciding (**simulating a team discussion** where alternatives are evaluated).
* **Step-by-Step Implementation Progress:** Keep a running log of development sessions. For each coding session (or each day), note the goal (“Implement API endpoint X”, “Refactor module Y”), what you plan to do, and the outcome. This is essentially a diary entry of what happened.
* **GitHub Copilot Prompts & AI Interactions:** **Include every prompt you send to Copilot (and optionally a summary of its response).** You can format prompts as blockquotes or code blocks to distinguish them. For example, record a prompt like:

For example, record a prompt like:

\*\*Prompt:\*\* “Generate a C# method to validate email format”

\*\*Copilot Suggestion:\*\* \_[describe or paste the snippet]\_

\*\*Decision:\*\* accepted with minor edits (explain why or why not)

Logging prompts is crucial since it shows your problem-solving approach. It might feel tedious, but prompt documentation is considered a best practice in AI-assisted development. In fact, prompt engineers often keep a table or log of prompts, their intent, and outcomes for future reference.

* **Thought Process & Rationale:** Alongside each action or prompt, write **why** you did it. For instance, “I wasn’t sure how to parse JSON, so I asked Copilot for a quick example.” This reflective commentary simulates how a developer might explain decisions to teammates during a review.
* **Outcomes, Testing, and Fixes:** Document the results of running the code (e.g. “Tested function X with scenario Y, it failed, so I prompted Copilot to help debug…”). If Copilot suggestions led to bugs and you fixed them, note that too.
* **Final Implementation & Learnings:** Conclude with the completed feature and any key learnings or retrospective thoughts. This could be a summary that reads like a case study conclusion, useful for readers who want the gist after the detailed journey.

By capturing all these elements, your documentation will read like a story of the project, complete with requirements, design rationale, development twists and turns, and the interplay with AI assistance. The next step is choosing a format and platform that will present this information clearly.

**Structured Format – Simulating a Team’s Documentation**

**Structure your documentation as if it were a small project wiki or design journal maintained by a team.** This means using clear sections, consistent formatting, and perhaps a dash of “team voice” in the writing. Here are some tips for a structured format:

* **Chronological Entries vs. Thematic Sections:** You could organize the log chronologically (e.g. by date or development phase), or thematically (sections for “Design Decisions”, “Implementation Logs”, “Prompts & Solutions”, etc.). A chronological **daily log** is often most natural for capturing prompts and thought flow. For example, each day or major session gets a subsection with a date and subtitle (“Day 1 – Project Setup and Initial Ideas”). Under that, you narrate what happened in order.
* **Use Headings and Subheadings:** Leverage Markdown or your chosen tool’s heading styles to create a hierarchy. For example:  
  **H2: Design & Planning**  
  **H3: Use Case and Requirements**  
  **H3: Chosen Approach (and Alternatives)**  
  **H2: Development Diary**  
  **H3: Day 1 – Scaffold the Project**  
  **H4: Prompt: Setting up project structure** (then the prompt text, etc.)  
  **H3: Day 2 – Implement Feature X** … and so on.  
  This structured approach makes it easier for others to navigate and for you to maintain consistency. (GitHub-flavored Markdown, for instance, supports multiple heading levels and will automatically generate a table of contents in the repo UI for all these headings). [[How To Wri...ub Project]](https://dev.to/maddy/how-to-write-the-perfect-documentation-for-your-github-project-4k38), [[How To Wri...ub Project]](https://dev.to/maddy/how-to-write-the-perfect-documentation-for-your-github-project-4k38)
* **Write in an Explanatory Tone:** Since this is public-facing, prefer a clear, explanatory style. You can use first person (“I did X…”) or even collective voice (“We then decided to…”) to emulate team collaboration. Using **“we” can simulate a team environment**, but be careful not to invent fictional teammates – instead, frame it like you are explaining to future collaborators or stakeholders. For instance: “We’ll use approach A for this because it simplifies deployment (a decision I reached after considering B and C).”
* **Embed Code and Prompts Nicely:** Use formatting to separate your narration from technical content. For example, put code snippets or Copilot responses in fenced code blocks for readability. Prompts can be in **bold** or quotes to highlight them. Some people use a two-column table format for prompt vs response, but that can get unwieldy; simple inline formatting usually suffices. [[How To Wri...ub Project]](https://dev.to/maddy/how-to-write-the-perfect-documentation-for-your-github-project-4k38)
* **Use Lists and Tables for Clarity:** Wherever you find yourself discussing multiple options or pros/cons, use a bulleted or numbered list. This not only makes it easier to read, but mimics how a team might list considerations. For example, list the advantages of an algorithm or record several attempts at a prompt in a sub-list. If you want to go further, you could use a table to log Copilot prompts with columns like “Prompt”, “Outcome summary”, “Status (used or not)”. This tabular prompt log is exactly how some prompt engineers track their experiments. A spreadsheet or table ensures you **don’t lose track of which prompts worked or needed tweaking** – a practice recommended in AI development to handle the variability of responses. [[Documentin...s - Medium]](https://www.leeboonstra.dev/prompt-engineering/prompt_engineering_guide4/), [[Documentin...s - Medium]](https://www.leeboonstra.dev/prompt-engineering/prompt_engineering_guide4/) [[Documentin...s - Medium]](https://www.leeboonstra.dev/prompt-engineering/prompt_engineering_guide4/)
* **Templates and Consistency:** It helps to have a repeatable template for each log entry. For example, **every daily entry could be: *Goals*, *Prompts & Results*, *Decisions Made*, *Next Steps***. Or each prompt interaction could follow a template as illustrated earlier (Prompt -> Copilot output -> Your action). Consistency will make the document feel structured and “team-approved”. You mentioned simulating collaboration – a consistent format is something teams often enforce via templates, so it adds to that vibe. You can even prepare a small template in advance (perhaps a Notion template or a Markdown snippet) to fill in during each coding session. [[Documentin...ote-taking]](https://github.com/readme/guides/documentation-note-taking)

In summary, think of your documentation like a **running commentary** of the project with clear markers for each stage. Now, let's explore the best tools/platforms to use for writing and organizing this.

**Tool and Platform Options**

You have a few solid options for where to write and store this documentation. The ideal choice depends on your preferences for integration with code, richness of formatting, and how you plan to share it. We’ll look at:

* **A. GitHub (Repo Markdown, Issues, Projects)** – using GitHub’s native tools such as a Markdown file in the repository (or Wiki) and Project boards/Issues for tracking.
* **B. Notion** – an external all-in-one workspace known for flexibility and sharing.
* **C. Obsidian** – a personal knowledge base (Markdown editor) that excels at linking notes and offline editing.

Each has strengths and weaknesses for this use case. Many developers actually mix these tools (for example, using Notion during active development and then transferring notes to the repo for posterity). Let’s break down each option and then compare them: [[Using Gith...e projects]](https://dev.to/jonathanyeong/using-github-and-notion-to-organise-side-projects-424o), [[Using Gith...e projects]](https://dev.to/jonathanyeong/using-github-and-notion-to-organise-side-projects-424o)

**A. Using GitHub (Markdown in Repo, Issues & Projects)**

**What this looks like:** You document directly in your code repository and/or use GitHub’s project management features. For instance, you might create a DEV\_LOG.md file in your repo where you write the narrative. In parallel, you could track tasks or milestones via GitHub Issues and a Projects board to organize the work. All of this would be public (assuming your repo is public) and tightly coupled with your code.

**How to do it:**

* **Markdown in Repo:** Create a Markdown file (or multiple files) for your documentation. For example, README.md could contain the high-level overview and design, and a separate Dev-Journal.md could hold the day-by-day log including prompts. GitHub will render these nicely in the browser. You can edit them directly in VS Code (which keeps you in flow) and commit changes as you go. Using Git means every update is versioned – so you’ll even have a history of your documentation edits, which is a bonus for showing progress over time.
* **GitHub Wiki:** If you prefer, enable the GitHub Wiki for the repo. The Wiki is essentially a collection of Markdown pages that you can organize with a sidebar. It’s great for multi-page documentation (one page for “Specs”, one for “Changelog/Journal”, etc.). The wiki can be edited via the web or pushed from a local repo. It’s less discoverable than a README in the main repo, but more structured for a mini “site” feel.
* **Issues & Project Board:** To simulate a collaborative team, consider using GitHub Issues to log significant tasks or questions. For example, open an issue for “Design Review” where you write down the considerations (as if it were a discussion – even if you’re the only commenter). You can reference these issues in your markdown documentation (and vice versa). A GitHub Project (especially the newer **Projects** which are like spreadsheets/boards) can show all issues as cards in columns (“To do / Doing / Done”), mimicking a Kanban board a team might use. This is **good for organizing the work and providing context**, but you would still write the detailed narrative in the issue bodies or the markdown files. The Project board itself doesn’t contain content beyond issue titles/notes; it’s more of an index. The strength of GitHub’s approach is that everything – code, tasks, and docs – lives in one place and links together smoothly (e.g., closing an issue via a commit, or referring to an issue number in your docs). [[Using Gith...e projects]](https://dev.to/jonathanyeong/using-github-and-notion-to-organise-side-projects-424o), [[Using Gith...e projects]](https://dev.to/jonathanyeong/using-github-and-notion-to-organise-side-projects-424o)

**Strengths:** GitHub is built for software collaboration, so it naturally supports this kind of documentation:

* *Integration with Code:* Your documentation sits alongside the code, which means anyone browsing the repo sees the docs. This ensures the documentation is **version-controlled and updated with the project** – a big advantage for accuracy and transparency. [[How To Wri...ub Project]](https://dev.to/maddy/how-to-write-the-perfect-documentation-for-your-github-project-4k38)
* *Single Source of Truth:* Using issues and markdown files in the repo can serve as the authoritative log of the project. You won’t have knowledge scattered across different apps.
* *Familiar Format:* Markdown is developer-friendly and supports headings, lists, tables, code blocks, images, and links. You can achieve pretty much any formatting you need for this journal (including tables for prompt logs or embedding screenshots of outputs if you want). [[How To Wri...ub Project]](https://dev.to/maddy/how-to-write-the-perfect-documentation-for-your-github-project-4k38), [[How To Wri...ub Project]](https://dev.to/maddy/how-to-write-the-perfect-documentation-for-your-github-project-4k38)
* *Public and Collaborative:* By default, a public repo’s docs are public. Others can contribute by opening pull requests to your docs or commenting on issues – which genuinely simulates collaboration. Even if no one else does, the potential is there. In a team setting, multiple people could commit to the documentation file or comment on the issues, and you’d have a true multi-user history.
* *Automation and References:* As noted, GitHub issues integrate with commits and PRs. If you make a commit that implements a feature and tag the issue number, the issue can auto-close and show that reference. This way, someone reading your issue about “Implement Feature X” can jump to the code that resolved it. This **traceability** is a unique strength of GitHub’s platform. [[Pros and c.... - Reddit]](https://www.reddit.com/r/Notion/comments/rzx5ym/pros_and_cons_of_github_projects_vs_notion_for/), [[Pros and c.... - Reddit]](https://www.reddit.com/r/Notion/comments/rzx5ym/pros_and_cons_of_github_projects_vs_notion_for/)
* *New Copilot Prompt File Feature:* GitHub has a new feature called **Copilot prompt files** (in public preview) which let you save prompts in your repo under .github/prompts/ as reusable snippets. This is more about reusing prompts than logging, but it could be repurposed for your needs. For example, you might keep particularly important or complex prompts as individual prompt files – effectively building a library of prompt templates that you (or others) can call up later. These .prompt.md files become shareable assets in the repo, and you can even invoke them easily in VS Code’s Copilot chat by name. If you foresee certain structured prompts being used repeatedly, this feature is worth exploring. (Even if not, the existence of prompt files underscores the benefit of storing prompts in your repo). [[Boost Your...ompt Files]](https://devblogs.microsoft.com/visualstudio/boost-your-copilot-collaboration-with-reusable-prompt-files/) [[Boost Your...ompt Files]](https://devblogs.microsoft.com/visualstudio/boost-your-copilot-collaboration-with-reusable-prompt-files/), [[Your first...itHub Docs]](https://docs.github.com/en/copilot/tutorials/customization-library/prompt-files/your-first-prompt-file)

**Weaknesses / Considerations:**

* *Limited Rich Media:* Markdown in GitHub is powerful but not as flexible as Notion for embeds or visual layouts. You can attach images/gifs and they will display, but you can’t, say, collapse sections interactively (no toggles like Notion) or have advanced widgets. It’s mostly static content.
* *No Real-Time Editing:* If you were collaborating live with others, GitHub doesn’t allow simultaneous editing of a file with live updates. You’d merge changes via Git. For a solo effort this isn’t an issue, but it means the editing experience is not as slick as a Google Doc or Notion where you just type and it’s saved. That said, writing in VS Code with Markdown is pretty comfortable for developers, and you can preview it as needed.
* *Organization Overhead:* If you use many issues for narrative, jumping between issues and markdown files to follow the story can be a bit disjointed. A reader might have to click around – though you can mitigate this by including plenty of links in your docs (like “see Issue #5 for design discussion”). Alternatively, you might use issues minimally and keep most content in the single timeline document. In that case, you lose some of the structured feel of issues/boards, but the story is linear. Finding the right balance is key.
* *GitHub Projects UI:* The project board is nice for you as a developer to visualize progress, but it’s not where you’d write paragraphs of reasoning – it’s more of an index. So think of it as a supplement to your documentation rather than the main stage. In the context of showcasing the process, the board could be a screenshot or just something viewers optionally check. The core narrative would still be in the markdown or issues.

**Use Case Fit:** Overall, GitHub’s native tools are a great choice if you value **integration with the development workflow and a transparent, dev-centric presentation**. Open-source projects often use a combination of README (for overview), /docs markdown files or Wikis (for detailed guides/decisions), and issues for discussion. Your use case of logging Copilot prompts is a bit unique, but GitHub can handle it (just lots of text). The end result will feel like an **engineering journal published in a repository** – which is very credible as a professional artifact.

**B. Using Notion (a Collaborative Document Workspace)**

**What this looks like:** Notion is an online workspace where you can create pages, databases, and rich content. For your project, you might create a Notion **page (or a workspace)** that serves as the project hub. You could have your development log in a Notion page, and even separate sub-pages for design spec, code snippets, etc., all interlinked. Notion’s flexibility allows you to structure information in many ways: you can have a timeline, a table of prompts, toggle lists that hide Copilot outputs until clicked (keeping the page clean), and so on.

**How to do it:**

* Create a page for your project. On that page, you can write an introduction describing the project and maybe an index of sections (Notion can auto-generate a table of contents for a page if you add a Table of Contents block).
* As you work, keep adding to this page or creating new sub-pages for each day or phase. For example, a top-level “Feature XYZ Development Journal” page that contains links to “Day 1”, “Day 2”, or to sections like “Ideas & Planning”, “Implementation Diary”, etc. Many people simply keep one long page and use headings for each day – do what feels manageable.
* Make use of **Notion’s template and database features** if they help. Notion could allow you to create a **database of Copilot prompts** where each entry is a prompt record (with properties like date, purpose, outcome, etc.), and then you can roll-up or view them in a timeline or table. However, that level of structuring might be overkill; a simpler approach is a table in the page for prompts (Notion tables are easy to insert and edit like a spreadsheet). There are even community-made Notion templates specifically for coding journals and prompt libraries which you can use as a starting point. [[7+ Notion...- Gridfiti]](https://gridfiti.com/notion-coding-templates/)
* Use visual elements: Notion pages can include images, diagrams, even Figma embeds or code with syntax highlighting. If during development you sketch a quick design or get a notable code snippet from Copilot, you can drop it in. This can make your documentation more attractive and informative.
* When you’re ready to share with the world, **enable public access** to that Notion page (Notion allows pages to be shared via a public URL, either read-only or with commenting access). This way, anyone can read it without a login. You can also keep it private during work and only publish when done, or publish from the start so folks can follow along in real time.

**Strengths:**

* *Flexibility & Rich Content:* Notion shines in letting you combine text, tables, boards, images, and more on one page. You can create a very polished document that feels like an interactive report or case study. For example, you can put your prompts in togglable sections – the user sees a prompt, and they can click a toggle to reveal the Copilot answer beneath it. This keeps the narrative tidy. You can also style things with callout blocks (to highlight “Key Insight” or “Note:” etc. in different colors) which add a nice visual cue for important points.
* *Database and Templates:* Notion’s database feature could allow you to track things in a structured way (like a table of prompts, or a Kanban board of tasks). One person described using Notion for a high-level Kanban of project ideas and statuses, complementing GitHub for low-level tracking. In your case, you might maintain a small task board in Notion or simply tag entries as “To Do / In Progress / Done”. There are existing templates (e.g., “100 Days of Code” journaling templates, Agile development templates) that you can adapt to give your documentation a systematic layout with minimal effort. [[Using Gith...e projects]](https://dev.to/jonathanyeong/using-github-and-notion-to-organise-side-projects-424o) [[7+ Notion...- Gridfiti]](https://gridfiti.com/notion-coding-templates/), [[7+ Notion...- Gridfiti]](https://gridfiti.com/notion-coding-templates/)
* *Collaboration & Comments:* Even though you are solo, Notion’s collaborative features can help simulate a team. You can leave comments on specific text blocks (like “TODO: Ask frontend team about this” – even if there is no frontend team, it shows awareness). If you did invite someone to view (or if you share it later), they can add comments or suggestions in-line, similar to Google Docs. Real-time editing is seamless for multiple users. Notion also keeps page history, so you can retrieve past versions if needed.
* *Publishing & Sharing:* A Notion page can be your *public document* without needing a separate site. The share link would let others read it in Notion’s web interface, which is pretty and navigable (including the sidebar for sub-pages and automatic table of contents). This is arguably simpler for readers than browsing a GitHub repository for docs, especially non-developers. It’s almost like publishing a blog post or case study article, but with the ability to include structured elements. Because you wanted a public-facing result, this ease of sharing is a big plus.

**Weaknesses:**

* *Not Integrated with Repo:* The biggest downside is that your documentation and your code are in totally separate systems. You’ll have to manually ensure that if code changes, the Notion doc is updated accordingly. There’s no automatic link like with GitHub issues or markdown files that live with the code. For example, if you refactor a class, you should remember to update any description of it in your Notion notes. In a fast-moving dev scenario, this might cause drift if not disciplined.
* *Requires Internet & Proprietary Format:* Notion is an online service; if you happen to be offline, you can’t edit (there is some offline support in the desktop app, but it’s not as frictionless as local Markdown). Also, your data is in Notion’s system – you can export to Markdown or PDF when needed, but it’s not as simple as having Markdown files already in Git. However, this is usually fine for documentation purposes, and Notion has a reliable platform. Just keep backups (Notion has export options).
* *Performance and Scale:* Very large pages in Notion can get a bit slow to load or edit. If you log everything for a long period, that page might become hefty. You can mitigate by breaking it into sub-pages per phase or per month, etc. Also, embedding a lot of code blocks can make the page long – consider linking out to GitHub gists or repo files for full code if needed, and keep excerpts in the doc.
* *Less Developer-centric:* This is subjective, but some developers prefer writing in their code editor (VS Code) rather than in Notion’s editor. Notion’s editor is WYSIWYG-ish and uses a “block” paradigm which is different from plain text Markdown. It’s generally easy to use, but certain things like editing tables or formatting code require keyboard/mouse interplay that some find slower than just typing in markdown. On the other hand, it frees you from worrying about some formatting syntax and provides slash commands for quickly inserting elements.

**Use Case Fit:** Notion is ideal if you want a **polished narrative with rich elements** and you don’t mind (or even prefer) separating documentation from the codebase. It encourages you to write detailed explanations and present them in an attractive way – perfect for showcasing your reasoning to a broad audience (including non-developers). If simulating a team environment, you can even create a fake “Documentation” database with entries that look like multiple people contributed (but that might be overkill). At minimum, Notion will give your documentation a professional, handbook-like feel. Many solo devs use Notion as a second brain for projects because it’s so adaptable to whatever workflow you want. Just be diligent about syncing knowledge between Notion and your code as things evolve. [[Using Gith...e projects]](https://dev.to/jonathanyeong/using-github-and-notion-to-organise-side-projects-424o), [[7+ Notion...- Gridfiti]](https://gridfiti.com/notion-coding-templates/)

**C. Using Obsidian (a Personal Markdown Knowledge Base)**

**What this looks like:** Obsidian is a lightweight, local-first app that works with Markdown files stored on your computer (your “vault”). It’s like having a supercharged text editor specialized for note-taking, with features such as **backlinking** (linking notes to each other and seeing a graph of connections), tagging, and a plugin ecosystem. Using Obsidian for your project would mean you keep your development notes in Markdown files on your disk (perhaps in a folder dedicated to the project). You’d write and organize them in Obsidian during development. When ready to share publicly, you could **publish them to the web** (Obsidian has a paid Publish feature) or more likely, just push the Markdown files to GitHub or convert them to a format for others to read.

**How to do it:**

* Set up an Obsidian vault (which can just be a folder) for your project’s documentation. If your code is already in Git, you might even initialize this vault as the same folder (though mixing a whole codebase with Obsidian notes might be confusing). Alternatively, create a subfolder like dev-journal/ or use a separate folder and later copy things over.
* In Obsidian, create notes for different purposes. For example: Ideas and Requirements.md, Design Decisions.md, and a folder of daily notes (2025-09-12 Journal.md, etc.). You can use Obsidian’s **Daily Notes** plugin to quickly generate a new note each day with the date, which is great for chronological logging.
* Use Obsidian’s Markdown features to the fullest: you can embed one note in another (for instance, if you have a note that is just a log of prompts, you can embed that into your daily entry note for quick reference). You can tag items with keywords like #prompt, #decision, #todo, which later allows you to search or aggregate by tag.
* One of Obsidian’s killer features is the [[wikilink]] style reference. If on Day 3 you revisit a decision from Day 1, you can link to the “Decision X” section in the earlier note. This is akin to hyperlinking documents in Notion, but in Obsidian it also means you can see backlinks (Obsidian will show all notes that link to the current one – so you get a network of your knowledge). This mimics how a team might hyperlink design docs, tickets, etc., but it’s all in your personal vault.
* While writing, Obsidian stays out of your way – it’s essentially plain text editing with preview if needed. You can even open your code repo in VS Code and Obsidian side-by-side to copy snippets or prompts between them.
* When you’re done (or at checkpoints), you have a few sharing options:
  + **Publish via GitHub:** Since Obsidian notes are Markdown, you can push them to your GitHub repo (either the project repo or a separate one just for documentation). They can live as documentation files for readers to browse on GitHub. This is similar to the “Markdown in repo” approach, except you authored them in Obsidian which might have made it easier to manage many interlinked notes.
  + **Publish via Obsidian Publish:** This is a hosted solution (requires a subscription) where your vault can be published as a website (with all note links intact, and a nice explorer interface). This could be overkill unless you plan to maintain a public second-brain site.
  + **Export to other formats:** You can export notes to PDF or copy as HTML if you want to, say, create a blog post out of them. For instance, you might export the whole vault to a static site using a tool (people have created workflows to turn Obsidian vaults into Jekyll or Docusaurus sites, etc., if you ever want a more custom site).

**Strengths:**

* *Local and Privacy-first:* Everything is on your machine as plain text. You **own the data** fully. This means you can write freely without worrying about accidentally publishing something before it’s ready. It also means you’re not dependent on internet access or any platform’s availability. If something is sensitive (though in your case it’s all intended to be public eventually), it stays private until you push it out. Obsidian works offline by design, which some developers love for focus. [[Obsidian v...] - Zapier]](https://zapier.com/blog/obsidian-vs-notion/)
* *Markdown and Simplicity:* Obsidian is essentially a fancy Markdown editor, so you get all the goodness of Markdown (easy formatting, code fences, etc.) without a proprietary format. Your notes are just .md files that any other tool can read. This future-proofs your documentation – in 5 years those files will still be accessible, whereas a Notion page would need to be exported if you left the platform. [[Obsidian v...] - Zapier]](https://zapier.com/blog/obsidian-vs-notion/)
* *Linking and Navigating Notes:* It’s very easy to create and navigate links in Obsidian. This encourages a **wiki-like approach** where you don’t have to put everything in one long document; you can have multiple notes and hyperlink between them. For example, you might have a master timeline note that lists all your sessions with links, and each session is a separate note. Or have a dedicated note just to list all prompts and link each prompt to the day note where it was used. The ability to jump around and the presence of the graph view (which shows a web of connected notes) can give you insights into your own process. It’s like building a knowledge graph of your project as you go.
* *Extensibility:* Obsidian has community plugins for many things. Want to manage tasks in your notes? There’s a plugin that can aggregate all - [ ] checkboxes across notes into a view (like a little to-do list manager). Want to visualize your timeline? There are plugins for rendering mermaid diagrams or timelines if you give it data. There are even plugins to integrate with Git, so that every time you write notes, you can automatically commit them to a GitHub repo (this could be neat: your dev diary vault could be pushed daily to GitHub, making it public incrementally).
* *Personal Productivity:* Some developers use Obsidian not just for notes, but to **journal their coding activities and thoughts** as a daily habit. It can combine work and personal learning. In your context, as you log your Copilot prompts, you might also be building a personal prompt reference. Obsidian is well-suited for accumulating such knowledge that you can re-use in future projects. For example, six months later, you can quickly search your vault for “email validation” to find how you solved it (and what prompt you used) – much faster than digging through an old Notion or trying to recall which issue had it. The search in Obsidian is instantaneous across all notes. Features like tagging and linking further make retrieval easy (tag all Copilot prompts with #copilot, then one search shows all of them). [[Using Obsi...ng Journal]](https://a.bigmachine.io/posts/using-obsidian-as-a-programming-journal)
* *Publishing Flexibility:* Even if you don’t use Obsidian Publish, the notes being plain text means you can publish them anywhere. If you want a very clean public presentation, you could later copy the content into a blog format or the GitHub repo as discussed. So Obsidian can be your behind-the-scenes tool for content creation, and the final output can still end up on GitHub or a website.

**Weaknesses:**

* *No Native Collaboration:* Unlike Notion or even Google Docs, Obsidian doesn’t let multiple people edit the same note simultaneously. It’s single-player (though you can share vaults via Git for multi-user, it’s asynchronous). For simulating a team, this means you won’t have comments or live feedback in the tool itself. It’s truly your personal space. You can mimic collaboration by writing imagined Q&A or including feedback you got from others in the text, but Obsidian won’t track who said what.
* *Learning Curve and Setup:* Obsidian is pretty intuitive for those used to Markdown, but it might take some time to set up your preferred structure. You have to decide how to split notes, how to name and link them, etc. There’s also a bit of setup in installing plugins if you want extra functionality. Compared to just writing in a single Notion page or a single README, Obsidian’s limitless vault can be a blessing or a curse if you over-organize or spend too much time tweaking it. The Zapier comparison article noted Obsidian’s interface and approach can be intimidating at first. [[Obsidian v...] - Zapier]](https://zapier.com/blog/obsidian-vs-notion/), [[Obsidian v...] - Zapier]](https://zapier.com/blog/obsidian-vs-notion/)
* *Not Immediately Public:* If your goal is to have a live public log that others can view while you work, Obsidian doesn’t do that out-of-the-box (unless you manually push to a public repo frequently). Notion or a GitHub markdown file would be available online as soon as you write them. With Obsidian, you’d typically write a lot locally and then publish after cleaning up. That’s fine (sometimes better to wait until it’s polished), just a different mode.
* *Lacks Some Rich Media Tools:* Obsidian handles images and has some diagram plugins, but it won’t embed a YouTube video preview or a fancy widget like Notion can. If you needed such things (maybe not likely for a dev log), you’d have to link out instead. It’s largely text-focused. However, you can always embed images of charts or architecture diagrams that you saved, so it’s usually sufficient for technical documentation.

**Use Case Fit:** Obsidian is great if you prefer a **markdown-driven, personal documentation process** and want full control of your notes. It’s like keeping a developer diary that you refine over time. For a public showcase, you’d do an extra step to publish it, but the quality of content you’ll produce can be very high since Obsidian helps you think clearly (distraction-free writing, quick cross-reference, etc.). It may appeal to you if you already enjoy tools like VS Code or Vim for writing text, because it doesn’t abstract things away like Notion does. Also, if you foresee this project’s log becoming part of a larger personal knowledge base (e.g., adding to a portfolio of “build journals”), Obsidian is excellent for maintaining a lifetime collection of such notes. You could later merge this vault with others and have a single repository of all your project journals, easily searchable. This “second brain” approach is a reason many developers choose Obsidian for note-taking in general. [[Using Obsi...ng Journal]](https://a.bigmachine.io/posts/using-obsidian-as-a-programming-journal)

Having looked at each option individually, here’s a **comparison table** summarizing how GitHub (native), Notion, and Obsidian stack up for the purpose of documenting a full development process including Copilot prompts:

A black background with a black square

AI-generated content may be incorrect.

**In summary:** GitHub’s native tools keep you in the development context and are great for authenticity and integration; Notion offers a polished, shareable notebook feel; Obsidian gives you personal control and robust note-taking power. All three can achieve the goal of recording your entire Copilot-assisted development process – the decision comes down to how you want to work and present the information.

**Recommendations and Best Practices**

Given the options, you don’t strictly have to choose just one. You could **combine tools to leverage their strengths**, as some developers do. For example, you might draft daily notes in Obsidian (for speed and convenience) and then paste them into a polished Notion page later, or maintain a Notion journal but periodically export important parts to your GitHub repo (so the repo has an up-to-date README or Wiki for the project). However, maintaining one source of truth is usually simpler. Here are some final recommendations to consider:

* **Use GitHub for permanence, plus another tool for comfort:** If you enjoy Notion or Obsidian’s environment for writing, use it during development, but ultimately **store a copy of your documentation in the repo**. This way, all critical information lives with the project code. For instance, you can write in Notion, then export or copy to a Markdown file in the repo each time you hit a milestone. This gives you the best of both – a nice editing experience and a version-controlled record in GitHub.
* **Establish a routine for logging prompts:** Since keeping track of every prompt is a bit unusual, build it into your workflow. For example, after each Copilot suggestion is applied (or rejected), take a minute to jot it in your log with a note. It’s much easier to do it as you go than to reconstruct later. Because this can disrupt coding flow, some folks use techniques like splitting their editor view (code on left, log on right) to write notes continuously. Figure out a habit that works: maybe typing a quick bullet in Obsidian during a compile, or using Notion on a second monitor. The key is consistency.
* **Leverage templates for Copilot prompts:** You can **standardize how you record prompts** similar to how an AI researcher would. For example, maintain a table (in Notion or a section in your Markdown) where each row has: Prompt, Why I asked it, Result summary, Outcome (used/fixed code, etc.). This adds structure and makes it easy to scan later. If using Notion, a database with these fields could be handy; if using Markdown, just a simple table or list is fine. This is not required, but if you anticipate analyzing your prompt usage, it’s a nice structured addition. [[Documentin...s - Medium]](https://www.leeboonstra.dev/prompt-engineering/prompt_engineering_guide4/), [[Documentin...s - Medium]](https://www.leeboonstra.dev/prompt-engineering/prompt_engineering_guide4/)
* **Simulate peer review and team input:** To really give a collaborative vibe, document some discussion or review notes. For instance, after finishing the feature, you might add a section “Code Review & Future Improvements” where you (pretending to be a reviewer or a teammate) critique the solution or suggest enhancements. This shows reflective thinking. You can frame it like: “During a hypothetical team review, it was noted that X could be refactored for clarity… I agree and would address that next.” This kind of commentary makes the documentation richer and feels like a multi-perspective effort.
* **Keep it transparent and honest:** Since this is public-facing, don’t shy away from including mistakes or dead-ends (with reasoning). Showing how you identified a bug from a faulty Copilot suggestion and then fixed it, for example, is extremely valuable. It demonstrates problem-solving. Often, people showcase only the end result, but including the journey – “Copilot suggested approach A, which I tried but it didn’t work because of Y, so I reverted and did Z” – provides learning moments for readers. It also mirrors how teams document decisions (“we tried X, but it had these issues, so we went with Y instead”).
* **Review and refine the document:** After the development is done, **do a pass over your documentation to organize and clarify it**. This is like editing a draft. You might find you want to pull out an executive summary at the top, or ensure the chronology is clear. Maybe add links between sections, or a short table of contents for long pages. If on GitHub, ensure all references (issue links, file links) are correct. If on Notion, perhaps add an introduction that frames the whole story for new readers. This editing phase will turn a raw journal into a polished case study of your feature development.

Finally, remember that the “best way” is one that you will actually follow. The fanciest documentation setup means nothing if it’s too cumbersome and you abandon it. So choose the tool that you **feel most comfortable with for daily use**. If you live in VS Code and GitHub, writing a Markdown journal in the repo might keep you most in the zone. If you love structured thinking and visual organization, Notion might motivate you to write more. If you like taking lots of notes and connecting ideas, Obsidian could be the perfect nerdy solution (with a publishing step at the end).

To wrap up, here’s an example of a possible workflow that ties many of these pieces together:

A screenshot of a computer program

AI-generated content may be incorrect.

By following a process like the above, you’ll end up with a thorough documentation of your journey. The result will be a structured narrative (complete with your Copilot interactions) that you can proudly showcase. It demonstrates not just the end code, but **how** you arrived there – which is often what teammates, recruiters, or future-you will find most interesting and educational.

Good luck with your project, and happy documenting! By choosing a suitable platform and disciplined approach, you’ll create a resource that reflects a collaborative, thoughtful development process from start to finish. Each tool has its merits, but the true star will be the story you tell with it. 🚀