ChatGPT

UX-Kit: A Spec-Driven UX Research Toolkit

UX-Kit is a command-line toolkit that brings a structured, spec-driven approach to User Experience (UX) research. It introduces a series of bash commands (for macOS/Linux) to guide an AI assistant through distinct phases of a UX research workflow – from formulating research questions to synthesizing insights. This approach treats research artifacts as *living documents* that evolve with the project, much like spec-driven development treats specifications as executables rather than static paperwork [1]. Think of it as **"version control for your thinking"** – capturing the *why* behind product decisions in a consistent, shareable format that grows with your understanding [1]. By breaking the research process into explicit steps with clear outputs, we reduce ambiguity and ensure the AI (and the team) always has unambiguous instructions at each stage [2]. The end result is a more reliable, repeatable research process, avoiding ad-hoc "vibe" research that can lead to lost context or contradictory findings.

Key Research Workflow Commands and Phases

UX-Kit defines five primary slash-commands (e.g. `/research:...`) corresponding to phases of the UX research cycle. Each command focuses on a specific goal and produces a Markdown artifact in a dedicated project directory (e.g. a new `.ux/` folder) to serve as the single source of truth for that phase. This ensures that at any point, the current state of research is documented and can be reviewed or refined before moving on (no more crucial insights scattered across chat logs or sticky notes!). The phases and their commands are summarized below:

Phase	**Purpose** (Focus)	**Command**
Questions	Define *what to learn and why* – capture user goals, assumptions, and key research questions.	`/research:questions`
Sources	Plan *how to gather data* – identify and log relevant source materials (documents, user studies, etc.) for investigation.	`/research:sources`
Summarize	*Distill findings* – for each source, extract key points and insights in a structured summary.	`/research:summarize`
Interview	*Structure qualitative data* – convert raw interview notes or transcripts into a consistent, analyzable format.	`/research:interview`
Synthesize	*Cluster insights* – analyze all summaries to identify themes, patterns, and prioritize insights or recommendations.	`/research:synthesize`

Each phase is *gated*, meaning you complete and validate the output of one step before moving to the next. This mirrors the philosophy of spec-driven development: **don't move on to implementation (in this case, final insights) until the foundation is solid** [3]. By dividing the workflow, we give the AI clear context at each step, which dramatically improves reliability [2]. As one GitHub AI blog noted, *"this structured workflow* *turns vague prompts into clear intent that [AI] agents can reliably execute."* [2] In practice, that means fewer misunderstandings and more trust in the research outcomes.

Below, we detail each command – what it does, how it's implemented, and how it contributes to a cohesive UX research process.

`/research:questions` – Define Research Goals & Questions

The `/research:questions` command is the starting point of the UX research journey. It prompts the AI to **draft a Research Questions document** (e.g. `questions.md`) that clearly lays out the problem space, user goals, and the key questions the research aims to answer. Just as a software spec defines *what* to build and *why*, this document defines *what we need to learn and why it matters*. The focus here is on high-level intent – **the user needs, pain points, and outcomes that would signal success**, not implementation details or solutions [4]. For example, it might include a brief project background, target user personas, and a list of primary and secondary research questions.

When you run `/research:questions` (typically as a chat command to the AI in your IDE or terminal), UX-Kit will scaffold a new research *workspace* for you. Under the hood, it creates a new subdirectory (for example, `.ux/studies/001-<topic>/`) to contain all files for this research project. Inside, a template file (say `questions-template.md`) is copied to `questions.md`. This template provides structure — e.g. sections for **Background**, **Objectives**, **Key Questions**, **Success Criteria**, etc. The AI assistant then **fills in this template** using your prompt and any existing context.

Implementation details: A bash helper script (e.g. `create-new-research.sh`) handles this setup. It will:

- Determine the next research ID (e.g. if this is the first study, ID = 001).
- Create the folder `.ux/studies/001-my-topic/` (using a slug from an optional title you provide).
- Copy the `questions-template.md` into that folder as `questions.md`.
- Optionally register this new study as the "current" context (so that subsequent commands know where to write outputs).

After setup, the script invokes the AI with a prompt to populate the `questions.md`. In practice, if using an AI coding agent interface, the script may load the template content into the AI's context and prepend your high-level description. For example, you might type:

```
/research:questions We need to improve onboarding for our app. The goal is to
identify why new users drop off within first week and how to increase
engagement...
```

The AI would receive the prompt along with the blank template and produce a rich first draft of the research questions document. *Critically, the researcher (you) should review and refine this output*. At this stage, you ensure the *'what and why'* of the research are correctly captured (much like reviewing a product spec). The document is a living artifact – you can always update it as new insights emerge, but it's important to get it roughly right before planning data collection. This up-front clarity prevents "misaligned assumptions" later on [5]. In other words, everyone (and the AI) should share the same understanding of the research goals before proceeding.

(Note: The idea of starting with a clear spec of questions follows the same rationale as spec-driven development's first phase. It forces explicit articulation of goals and prevents the AI from making unfounded guesses. As Den Delimarsky explained, many failures of AI assistance come from us giving vague directives and the AI having to fill in the blanks [6]. A well-defined questions doc ensures we're explicit about what we seek to learn, giving the AI a solid foundation.)

`/research:sources` – Plan & Log Source Materials

Once the research questions are established, the next step is to **gather context and data sources**. The `/research:sources` command helps discover, list, and organize the materials that will inform the research. This could include relevant user data, analytics, existing studies, academic papers, stakeholder interviews, competitor analysis, etc. The output is a `sources.md` log (in the same project folder) that serves as a catalog of all inputs to the study.

When triggered, this command first ensures that a questions document exists (so that we know *what* we're looking for). Then it either prompts the AI to suggest possible sources or helps the researcher record known sources. In practice, using an AI agent with internet access or a knowledge base is ideal here – the agent could perform preliminary discovery (e.g. search the web or internal knowledge repositories) for materials relevant to the questions [7]. For example, if the question is about user drop-off in onboarding, the AI might list: *"Product analytics reports on 7-day retention", "Customer support tickets related to onboarding", "Previous UX study on onboarding (2022)", "Industry benchmarks for onboarding engagement"*, etc. Each item can include a brief description and a link or pointer (URL or file path).

The `sources.md` file acts as both a plan and a record of what will be (or has been) reviewed. It may contain a table or bullet list of sources with fields like *Source Name*, *Description*, *Location/Link*, and *Relevance to our questions*. This establishes a **research plan** akin to a technical plan in software development – it defines the *inputs and tools* we will use to answer our questions [3]. Having this list upfront ensures that the subsequent analysis phase is comprehensive (covering all key angles) and traceable (anyone can see where insights came from).

Implementation details: The command can be implemented with a script `setup-sources.sh`. This script might:

- Read the `questions.md` to understand the context (the key topics or unknowns).
- If an AI with web access is configured, use the AI to query relevant sources (the script could feed the questions to the AI and ask it to suggest top N sources, with the understanding it might have tools to search).
- If web access is not available, the script instead might prompt the user (via the AI chat) to list any known sources or upload documents. The AI can still assist in formatting and organizing the list.

The script then creates or updates `.ux/studies/001-topic/sources.md` using a template. For instance, `sources-template.md` might contain section headers like **Existing Data**, **Primary Research (to be collected)**, **Secondary Research (literature)**, etc., which the AI fills out or the user edits. In terms of content, we leverage the AI to generate concise descriptions of each source (why it's relevant, how it might help answer the questions) – effectively justifying each item in the list. This is in line with the *"Research-Driven Context"* principle: gathering the necessary context so the AI can later make informed analyses [7]. By explicitly logging sources, we avoid the AI "hallucinating" facts later; every insight will trace back to a documented source.

Before moving on, it's wise to review `sources.md` and confirm two things: (1) **Coverage** – do the listed sources collectively address all the research questions? (If not, we may need to seek additional data or revise questions.) And (2) **Feasibility** – ensure we have access to these sources and the means to analyze them. This is analogous to a technical plan's review where you'd check if all requirements are covered and all tools are available [8]. Only after this vetting do we proceed, confident that we have a solid plan for data gathering.

`/research:summarize` – Distill Key Findings from Sources

With sources in hand, the next phase is **analysis** – extracting insights from each source. The `/research:summarize` command is used to generate summaries or analysis notes for one or multiple documents. The idea is to break down the research workload into *small, focused chunks*, much as one would break a software project into discrete tasks [9]. Instead of asking the AI a broad question spanning all data (which could lead to confusion or missed details), we tackle one source at a time, summarize it, then later combine the results. This method aligns with the pattern that *"the coding agent takes the spec and plan and breaks them down into actual work… each task is something you can implement and test in isolation"* [9] – here each "task" is summarizing a single source, yielding an isolated, reviewable piece of insight.

Using the command might look like:

```
/research:summarize sources/customer_support_tickets.pdf
```

Upon running this, UX-Kit will locate the specified source (it could be a PDF, text file, or even an external URL if the agent has access). It then guides the AI to produce a summary of that content. The output can be directed to a new Markdown file, e.g. `summary_<source-name>.md`, or appended under a section in a cumulative `summaries.md`. The format is typically a concise outline of key points, findings, quotes, or statistics from the source, with emphasis on information that relates to our research questions. For example, if summarizing a support ticket log, the AI might list the most common onboarding complaints or questions users had. If summarizing a usability test video transcript, it might highlight the steps where users struggled and any direct quotes illustrating frustration.

Implementation details: Under the hood, a script like `summarize-source.sh` handles this. It may use auxiliary tools to preprocess the source data as needed (for instance, extracting text from PDFs or spreadsheets). Once the raw text is ready, the script invokes the AI with a prompt template for summarization. This prompt would include instructions like *"Extract the key insights from [Source X] in bullet points. Focus on findings relevant to our research objectives listed in* `questions.md`*. Preserve important details or user quotes verbatim where useful."* The AI's response is captured and saved.

Notably, if multiple sources need summarizing, you can run `/research:summarize` repeatedly for each. UX-Kit can streamline this by reading the `sources.md` list and iterating through sources (possibly with confirmation before each). Summaries are stored under the `.ux/studies/...` directory, ensuring they become part of the persistent knowledge base for the project.

This phase is where the **AI truly acts as an analyst assistant**, combing through data systematically. The human researcher should review each summary for accuracy. Because the tasks are isolated, it's easier to verify accuracy (you can spot-check against the original source) – a crucial advantage over trying to verify a giant, mixed summary of everything. If something looks off, you can correct it now or even rerun the summary with adjusted instructions. By the end of this stage, we have a collection of *digested knowledge* – a much more manageable set of information that will feed the final synthesis.

(Note: This mirrors the benefit of test-driven development for AI: the model "validates its work" on each small piece before assembling them [9]. Likewise, summarizing source-by-source ensures the AI doesn't get "lost" trying to hold too much context at once. Large language models are exceptional at pattern recognition but poor at "mind reading" or dealing with unspecified complexity [6]. By explicitly giving them one document and a clear task ("summarize this with respect to our known questions"), we avoid burdening the AI with guesswork about relevance. The result is more trustworthy analysis.)

`/research:interview` – Structure Interview Notes/Transcripts

User interviews and other qualitative research produce raw, unstructured data (transcripts, notes, recordings) that can be hard for both humans and AI to parse directly. The `/research:interview` command addresses this by transforming such raw text into a **consistent, structured format**. Essentially, it's a specialized summarization tailored for qualitative data, ensuring no important anecdote or observation slips through the cracks. For example, if you have a transcript of a 30-minute user interview, this command can help produce an organized summary with sections like **Participant Background**, **Key Quotes**, **Pain Points**, **Positive Feedback**, **Observed Behaviors**, etc.

When invoked, the command will prompt the AI to ingest the interview text (from a file or copy-paste) and then output a formatted Markdown document (say `interview_<participant>.md`). A template (e.g. `interview-template.md`) guides this process, enforcing a uniform structure for all interviews. Consistency is key: if you interview 5 users, you want to easily compare their responses. By having the AI structure each one similarly, patterns will emerge more clearly in the synthesis phase. For instance, every interview summary might have a bullet list of "Top Frustrations" and if you see *"confusing onboarding flow"* appear in 4 out of 5, it's immediately evident.

Implementation details: The `format-interview.sh` script will handle this. Steps include:

- Taking the raw text (transcript or notes). The script might strip timestamps or irrelevant metadata from transcripts for clarity.
- Feeding it to the AI with the interview template. The prompt might say: *"Format the following interview transcript into the given structure. Identify and list: (a) User's background, (b) Goals, (c) Main frustrations or pain points mentioned, (d) Features or aspects they liked, (e) Direct quotes that are insightful (with timestamps or markers if available), (f) Any suggestions the user made."*
- The AI's output is saved to `interview-<id>.md` under the study folder.

By using an AI for this, we speed up what is typically a very time-consuming task for researchers (transcribing and summarizing interviews). However, **human oversight remains critical** – you should read through the structured notes and adjust if something is misinterpreted or if tone/context is lost. The advantage is that the bulk of the sorting and grouping is done, so you can focus on refining meaning.

This step yields a set of uniform interview summaries, each capturing the essence of a conversation in a way that's easy to scan and compare. It effectively turns qualitative data into semi-quantitative insights (e.g. counting how many participants mentioned X). Moreover, having these in Markdown means you can easily share them or even push them to version control for collaboration. It exemplifies how **templates and helper scripts enforce consistency** in AI-driven workflows [10] – every interview follows the same pattern, guided by the template, which reduces randomness in how the AI might format things. That consistency will prove invaluable when synthesizing insights across all these interviews and other sources.

`/research:synthesize` – Derive Insights & Recommendations

The final and most crucial phase is turning all the collected knowledge into actionable insights. The `/research:synthesize` command directs the AI to **cluster findings, identify patterns, and prioritize insights** across all sources and analyses conducted so far. This results in an `insights.md` (or similarly named) document, which is essentially the *research report*. It answers the original research questions and often includes recommendations or next steps for design/product decisions.

When you run `/research:synthesize`, the AI is prompted to pull together information from the various artifacts: the initial questions (to remember the goals), the source summaries, the interview highlights, etc. Because we've been diligent in the earlier steps, the AI now has a rich, structured pool of context to draw from – each piece explicitly documented. The prompt for synthesis might be something like: *"Given the research questions in* `questions.md` *and the findings in* `summaries/` *and* `interviews/` *documents, produce a summary of insights. Identify recurring themes or points of friction that answer our questions. Organize the insights by theme or research question, and prioritize them (e.g. which issues seem most urgent or impactful). Provide evidence for each insight (e.g. reference which source or how many users mentioned it). Finally, suggest any recommendations or potential solutions based on these insights."*

The AI will then generate a draft insights report. A typical structure could be:

- **Introduction** (restate the objectives briefly)
- **Key Insights/Themes** – e.g. *"Onboarding confusion is the top issue: 4 of 5 users struggled to find feature X* [11] *; analytics show 60% drop-off at step 3. This suggests... (Insight #1)"*
- Each insight can be a subsection, with bullet points of evidence. We ensure the AI cites back to the source (by name or ID) so we maintain traceability (this can be in plain text since all sources are internal at this point).
- **Recommendations** – e.g. *"Simplify onboarding flow (address Insight #1 by doing A/B/C)..."* or *"Further research needed on Y..."*
- **Conclusion** – optional, summarizing how the findings answer the questions and what actions will be taken.

The output might be several paragraphs long, combining narrative explanation with lists or tables if needed (for example, a table of user pain points by frequency).

Implementation details: This command leverages a script `synthesize-insights.sh`. Key tasks for the script:

- Gather all relevant files (it can programmatically read the `summaries` and `interview` files from the `.ux/studies/001/` folder, as well as `questions.md` and `sources.md`). Using a bash script, it might concatenate key parts into a single prompt context or feed them in pieces if the AI tool supports it. Some advanced AI agent CLIs allow referring to files directly in a prompt (for instance, Cursor's commands can load file content into the prompt). UX-Kit will take advantage of that.
- The script then uses a synthesis prompt (possibly defined in a `templates/synthesis-template.md` or directly in the command definition) to instruct the AI on how to combine everything.
- Once the AI produces the insights draft, the script saves it as `insights.md`. It might also append a timestamp or version info.

After generation, **human review is again important**. This is where you validate that the insights truly follow from the data. Thanks to the structured approach, this review is more straightforward: you can cross-check each insight against the summary docs. If an insight lacks support, you catch it now and can prompt the AI to revise or omit it. The final `insights.md` can be considered the deliverable of the UX research agent – something you might share with stakeholders or the product team. It closes the loop by directly answering the questions posed in the beginning, with evidence.

By enforcing a disciplined, stepwise approach ending in this synthesis, we dramatically improve the quality of outcomes. We avoid the AI's tendency to either hallucinate conclusions or overlook contradictory data, because at each phase we constrained it to work with verified information. It's analogous to how in spec-driven dev, by the time you let the AI code, it "knows what to build, how to build it, and in what sequence" because the spec, plan, and tasks spelled it out [12]. Here, by the time we ask the AI to draw conclusions, it *knows what to look for, where to find it, and why it matters*, thanks to the prior phases.

Implementation and Tooling Details

To make all the above possible, UX-Kit follows a very similar architectural pattern to other spec-driven toolkits, but focused on Mac/Linux bash and a research context. Upon initialization, it will set up the necessary project structure and integration points with your AI assistant/IDE. Key components of the implementation include:

- **Project Directory Structure:** All files are organized under a **.ux/** directory at the root of your project (to avoid clutter and keep things version-controllable). For example:

```
.ux/
 ├── memory/                 # (Optional) Persistent context like research
guidelines or "constitution"
 │    └── principles.md      # e.g. ethical rules, company guidelines for UX (if
any)
 ├── templates/              # Prompt and document templates for each command
 │    ├── questions-template.md
 │     ├── sources-template.md
 │     ├── summarize-template.md
 │     ├── interview-template.md
 │     └── synthesis-template.md
 ├── scripts/                # Helper bash scripts that implement command logic
 │     ├── create-new-research.sh
 │     ├── setup-sources.sh
 │     ├── summarize-source.sh
 │     ├── format-interview.sh
 │     └── synthesize-insights.sh
 └── studies/                # Subdirectories for each research project or
feature study
      └── 001-onboarding-experience/
           ├── questions.md
           ├── sources.md
           ├── summaries/
           │     ├── summary_analytics.md
           │     └── summary_support_tickets.md
           ├── interviews/
           │     ├── interview_user1.md
           │     └── interview_user2.md
           └── insights.md
```

Each research study is numbered (001, 002, …) and has its own folder. This mirrors how a development spec kit might number feature specs, helping to manage multiple efforts in parallel. The numbering and naming are handled by the `create-new-research.sh` script. It's possible to have just one ongoing study in a project, but the structure allows for growth (for example, a product team might run multiple research efforts for different features and keep them all tracked in one repo).

- **Command Integration (Cursor, etc.):** To provide a seamless experience within AI coding environments, UX-Kit registers custom slash commands for the supported IDE/agents. Initially, we target the **Cursor** editor/CLI (which natively supports user-defined slash commands). During init, if `--ai cursor` is selected, the toolkit will create markdown files for each command under `.cursor/commands/`. For instance, `.cursor/commands/research_questions.md`, `research_sources.md`, etc. These files describe the command behavior and link to our scripts and templates. This mechanism is exactly how Cursor is extended by spec-driven development toolkits: by dropping in command files that Cursor reads, users can simply type `/specify` or `/plan` in chat [13]. We leverage the same capability for `/research:...` commands. Each command definition will typically do the following when invoked: (1) call the appropriate shell script from `.ux/scripts/`, and (2) include the content of relevant templates or context files for the AI to use.

For example, the `research_questions.md` command file might contain instructions for Cursor like: *"Run `.ux/scripts/create-new-research.sh` with the user's prompt as input, then open the generated `questions.md` file."* Similarly, `research_synthesize.md` might instruct the agent to run the `synthesize-insights.sh` and then open `insights.md` for the AI to start writing into. This approach ensures the heavy lifting (filesystem operations, prepping context) is done by our scripts, while the AI focuses on content generation. *In short, we treat our AI assistant as a "literal-minded pair researcher" that follows our structured prompts and uses our tools, rather than leaving everything to the AI's imagination* [12].

- **Helper Scripts (Bash):** The shell scripts in `.ux/scripts/` are crucial for coordinating between the file system and the AI's output. They are written in bash (with `sh` compatibility) targeting Unix-like environments (macOS, Linux, WSL). We explicitly avoid Windows batch/PowerShell in this toolkit to keep things simple (spec kits often provide PowerShell equivalents, but here we focus on Bash). Each script is lightweight and focused on one task. For instance:
- `create-new-research.sh`: Creates the new study directory, copies templates, and perhaps stores the new study ID in a `.ux/current` file for reference.
- `setup-sources.sh`: Might trigger a web search or simply set up the sources.md from a template, depending on configuration. (We might integrate with CLI tools or APIs for web search if available, but by default it will rely on the AI's suggestions or user input.)
- `summarize-source.sh`: Opens a source file (convert to text if needed), maybe chunking it if it's very large, then calls the AI with the summarization prompt.
- `format-interview.sh`: As described, cleans up raw text and applies the interview template via AI.
- `synthesize-insights.sh`: Gathers content from multiple files to feed into the AI's synthesis prompt.

Common functions (like reading the "current study" or ensuring prerequisites) can be factored into a `common.sh` that all scripts source. We also ensure paths are handled in a POSIX-compliant way (no hard-coded Windows paths, etc.), and use standard utilities (`mkdir`, `cp`, `sed`, etc.) that are available on macOS by default. By using bash, we enable transparency – users can open and inspect these scripts to understand or modify the workflow as needed, which is important for trust.

- **Templates and Memory:** Following the spec-driven philosophy, templates encode the structure we expect at each step, and a "constitution" or principles file can encode overarching rules. For example, we might include a `principles.md` in `.ux/memory/` that states guidelines like *"Always maintain a neutral, objective tone. Base insights on evidence from sources (avoid speculation). If conflicting data exists, note it rather than ignoring. Adhere to company privacy rules (don't include user PII in summaries),"* etc. These principles can be injected into the AI's context during planning and synthesis to guardrail the output [11]. This is analogous to Spec Kit's *constitution* concept – a set of non-negotiable rules for the project [8]. For our UX use-case, the principles might cover research ethics and quality standards. They help ensure consistency and can be updated as the team refines their research approach.

Each command has a corresponding Markdown template in `.ux/templates/`. These templates are essentially prompt blueprints that the AI will follow. For example, `plan-template.md` in a coding spec kit contains sections for architecture decisions; in UX-Kit, `synthesis-template.md` might outline how to structure the insights report. By providing these, we reduce the cognitive load on the AI: it doesn't have to figure out *how* to organize the output – we've already decided the format [10]. This leads to more predictable results, which are easier for a human to read and compare. Templates also make it simple to tweak the output style without changing AI logic (e.g. if the team wants insights presented as an ordered list vs narrative, you adjust the template).

- **Deployability and Extension:** UX-Kit is designed to be easy to set up and extend. To deploy it, one could package it similarly to the spec toolkit. For instance, it can be distributed as a Python package or shell script bundle that installs the `.ux` folder structure. We can leverage the `uv` tool (from the Astral project) or even a simple Homebrew formula for macOS for one-line installation. For example, using `uvx` (which Spec Kit uses for quick Git-based installs), one could initialize a research project with:

  ```
  uvx --from git+https://github.com/ourorg/ux-kit.git uxkit init --ai cursor
  ```

  This would fetch the toolkit and run the `init` command (very much like `specify init`). The `uxkit init` process sets up the `.ux/` directory, copies all templates and scripts, and, if `--ai cursor` is specified, places the `.cursor/commands/` files for our five commands [13]. (If a different `--ai` option is given, it can adjust accordingly – e.g. for a hypothetical VS Code integration, it might print instructions for the user to trigger commands manually, since VS Code's Copilot might not allow custom slash commands yet.)

The architecture is **agent-agnostic**: adding support for a new AI assistant is usually a matter of providing the right command interface. For example, support for OpenAI's Codex CLI (a terminal-based coding agent) can be added by creating analogous command triggers or using Codex's CLI API to run our scripts. In fact, the spec-driven development toolkit we reference has seen community contributions to support new agents like OpenAI Codex, Qwen, Windsurf, etc., via small adaptations – e.g. adding a Codex mode that writes extension files for Codex CLI [14]. We anticipate doing the same for UX-Kit. The core logic (templates, scripts) remains the same; we just wrap them appropriately for each environment. This means as new AI research assistants or IDE plugins emerge, UX-Kit can integrate with them by following the same pattern of *steering commands and structured prompts*. Our initial focus is on Cursor because of its open extensibility, but the toolkit is built to expand. For instance, a future `uxkit init --ai codex` could drop files in the format that Codex CLI expects (perhaps a specific directory or configuration file).

- **No Windows?** At this time, we've chosen to target Unix-like systems. The scripts are written in bash and use Unix paths (`/` separators, etc.). We intentionally ignore Windows PowerShell support in this initial version to reduce complexity. Users on Windows can still use UX-Kit via Windows Subsystem for Linux (WSL) or Git Bash. Cross-platform support could be added later (potentially by porting scripts to a cross-platform runtime like Node.js, as some have proposed [15], or by adding PowerShell scripts). For now, by focusing on one environment, we minimize the maintenance overhead and can move faster in refining the toolkit's research capabilities.

Trust but Verify: Throughout the workflow, UX-Kit encourages an iterative **verify-and-refine** mindset. Each generated artifact (questions, sources, summaries, etc.) should be reviewed by the researcher and can be improved by either editing directly or re-prompting the AI with clarifications. This echoes the idea that *each phase should be fully validated before proceeding* [16] [12]. The toolkit provides the structure and initial content, but the human in the loop ensures accuracy and relevance. By the time you reach the final insights, you have high confidence in them because you've been curating the content all along.

In summary, UX-Kit adapts a spec-driven, **multi-step prompting strategy** to the domain of UX research. It leverages the same kind of scaffolding – custom AI commands, templates, and scripts – that have proven effective in managing complex coding tasks with AI [17]. The result is a deployable package that turns an AI coding assistant into a **UX research assistant**. Rather than ad-hoc queries, you now have **purpose-built commands** like `/research:questions` and `/research:synthesize` orchestrating the AI's behavior in a reproducible way. This not only yields better results in the moment (more focused outputs) but also creates an auditable trail of how conclusions were reached. All research knowledge is systematically captured in the `.ux` directory, which can be checked into version control alongside code – ensuring that design decisions and user insights are not lost. By closely mirroring the successful patterns of spec-driven development (without ever mentioning it by name in our process), we give UX practitioners a powerful, AI-augmented workflow that is logical, transparent, and extensible.

Sources:

- GitHub Blog – *Spec-driven development with AI (open source toolkit)* [4] [9]
- Visual Studio Magazine – *GitHub Spec Kit Experiment & Principles* [1] [7]
- GitHub Blog – *Why structured AI prompts work (clarity over vagueness)* [2] [6]
- Robot Paper (Roy Osherove) – *Spec-Driven Dev Implementation Notes* [13] [11]

---

[1] [3] [5] [7] [8] [10] GitHub Spec Kit Experiment: 'A Lot of Questions' -- Visual Studio Magazine
https://visualstudiomagazine.com/articles/2025/09/16/github-spec-kit-experiment-a-lot-of-questions.aspx

[2] [4] [6] [9] [12] [16] Spec-driven development with AI: Get started with a new open source toolkit - The GitHub Blog
https://github.blog/ai-and-ml/generative-ai/spec-driven-development-with-ai-get-started-with-a-new-open-source-toolkit/

[11] [13] [17] GitHub Spec-Kit Spec Driven Dev is in the Right Direction: Disciplined, Incerementall Vibe-Coding
https://robotpaper.ai/i-tried-out-github-spec-kit-and-all-i-got-was-this-not-terrible-website/

[14] Pull requests · github/spec-kit · GitHub
https://github.com/github/spec-kit/pulls

[15] Consolidate Spec Kit scripts into a single Bun/Node.js CLI to reduce ...
https://github.com/github/spec-kit/issues/280