ChatGPT

AI Agent Workflow for a Scholarship Matching + Drafting App

Development Tools and Plugins for AI-Assisted Coding

- **VS Code with Copilot/Codex:** Use GitHub Copilot (powered by OpenAI Codex GPT-4) as your AI pair-programmer in VS Code. Copilot automatically looks at the code around your cursor to suggest completions or entire blocks [1]. It's fine-tuned for code, making it ideal for generating React components, API handlers, or utility functions from a brief description [2]. For example, a one-sentence comment describing a component's purpose can prompt Copilot to scaffold the JSX and logic. Copilot Chat (the VS Code chat extension) lets you ask GPT-4 coding questions and get fixes or explanations in-editor. This speeds up both frontend and backend development by handling boilerplate and syntax details.

- **Gemini Pro 1.5 (Google AI):** Set up access to Google's Gemini via Google AI Studio (you can obtain an API key in the Google AI Studio interface [3] and use it with the `google.generativeai` library as shown in Google's docs [4]). Gemini Pro 1.5 offers a *massive* 1,000,000-token context window [5] and multimodal understanding [6], which is invaluable for dev tasks like ingesting large documentation or even entire code files. **When to use Gemini:** for *structured prompt chains* and data-heavy tasks. For instance, you can feed a long chunk of TanStack Start docs or a large JSON dataset into Gemini and ask it to summarize or extract info in one go. It's also useful for web content tasks – e.g. scraping and structuring data. Gemini can help turn raw HTML from a scholarship website into structured JSON by reading the raw text and following your prompt instructions [7] [8]. While VS Code doesn't yet have a dedicated Gemini plugin, you can run Gemini queries via Python scripts or a Jupyter notebook during development to leverage its capabilities (e.g. use a small script to fetch and parse content with Gemini, then bring the results into your code).

- **Claude 4.5 for Prompt Tuning:** Even though Claude (Anthropic's LLM) will run in production, you should involve it during development for prompt design and testing. Claude 4.5 comes in two variants: *Haiku* (fast, lightweight) and *Sonnet* (deeper reasoning) [9] [10]. Use the Anthropic Console or API (Anthropic provides an official Node SDK: `npm install @anthropic-ai/sdk` [11]) to experiment with Claude's responses. **Dev-time usage:** draft your system prompts and few-shot examples and then call Claude (Haiku for quick tests, Sonnet for full outputs) to see if the responses align with expectations. This helps refine the *personality analyzer* or *essay generator* prompts before integrating into the app. For example, you might use Claude Haiku 4.5 via the API to quickly tag a sample student profile with relevant traits (to verify your tagging prompt), since Haiku is optimized for speed and cost-efficiency in classification tasks [12] [13]. Conversely, test your essay-generation prompt on Claude Sonnet 4.5 to ensure it produces well-structured, in-depth drafts (Sonnet excels at complex, nuanced outputs [14] [15]).

AI Agent-Assisted Development Workflow

Using AI agents strategically at each step can accelerate development. Below is a workflow outlining **when and how to call Codex (GPT-4) or Gemini during the project**:

1. Project Setup and Routing Structure

Use Codex to initialize the project structure once you've run the TanStack Start CLI to scaffold the app. For example, after generating a fresh TanStack Start app, **describe the high-level routes** (pages) you need (e.g. "Home page, Profile page, Scholarship Search page, etc.") in a prompt and let Codex suggest the file-based routing setup. TanStack Start v1 uses type-safe, file-based routes [16], so you can ask the AI to create stub route components or server functions for each page. *Tip:* Write comments in your route files like `// TODO: define scholarship search route params and loader` and let Copilot auto-complete or generate the needed code. Codex is fine-tuned to understand such context and produce the matching React Router/TanStack Router configuration.

2. React Component Scaffolding

Leverage GPT-4 (via Copilot or ChatGPT) to **scaffold React components** from plain English descriptions. This is one of Codex's strengths – a brief description can be turned into a working component within seconds [17]. For example, you might prompt: *"Create a* `<ScholarshipCard>` *component that displays a scholarship's title, amount, and a button to apply. Use props for the data and include basic styling."* The AI will output a functional component with props, JSX, and state if needed. It even handles event handlers or callback props (like an `onApply` function) as described. Use an iterative approach: get an initial component draft from the AI, then **refine it with follow-up prompts**. If the first pass is incomplete, you can say "Now add a modal using this component when the Apply button is clicked" – the model can integrate new requirements. Remember to provide context: if this component relies on certain existing types or context, have those files open or paste relevant code so the AI fits its output into your codebase [18] [19].

For *TanStack Start routing* specifics (which involves server-side logic via "server functions" and client components), you can ask Codex to **generate an isomorphic server function**. For instance: *"Write a TanStack Start server function to handle form submissions for scholarship applications, saving data to Postgres."* Given TanStack's design, the AI might not know exact function signatures without docs, so consider feeding it a small example from TanStack Start docs as a few-shot example (showing how a server RPC is defined). Codex with context can then mimic the pattern for your specific use case.

3. UI Design with Tailwind CSS and Shadcn/UI

Tailwind CSS: When building the UI, describe the layout and styling you want, and let the AI suggest Tailwind classes and markup. Codex is adept at producing Tailwind-heavy JSX if your prompt specifies the design. For example: *"Create a responsive form layout with two columns using Tailwind (one for inputs, one for a preview panel). Include proper spacing and use Tailwind utility classes for a modern look."* The AI will output JSX with the appropriate Tailwind classes (`flex`, `p-4`, `grid cols-2`, etc.). Including styling requirements in the prompt ensures the code comes out with the desired classes [20]. If Copilot's inline suggestions don't give the exact look, open the Copilot chat and explicitly request the Tailwind CSS for a certain style; it can even fix your class names if you mention an issue.

Shadcn/UI components: Instead of hand-writing complex interactive components (like modals, date pickers, etc.), use the Shadcn UI component library. Shadcn provides pre-built, accessible components styled with Tailwind [21] [22]. The fastest way is to run the Shadcn CLI to add a component, e.g. `npx shadcn@latest add dialog` for a modal dialog [23]. Once added, **use AI to integrate it**. For example, after adding a `Dialog` component, ask Codex: *"Implement a login form using the Shadcn Dialog for the modal. Include email/password fields, and Tailwind classes for spacing."* Given an understanding of common usage, the AI can produce the code to render the `<Dialog>` with a form inside, and even manage open/close state. If the AI isn't familiar with Shadcn specifics, provide a snippet from Shadcn's docs (for instance, the code snippet for a basic Dialog usage) as a pattern – the AI will follow that style in your implementation [24]. This saves time combing through docs and ensures consistency. **Pro tip:** also use VS Code extensions like *Tailwind CSS IntelliSense* (for class name auto-suggest) alongside AI – it's not an AI tool, but it helps validate the classes the AI writes.

4. Database Schema and Embeddings Setup

Designing the Postgres schema can be accelerated by AI. Start by describing your data model to GPT-4 or Gemini: *"We need a Postgres schema. Tables: Users (id, name, profile info, vector), Scholarships (id, title, criteria, vector), Applications (id, user_id, scholarship_id). Use pgvector for the vector fields (1024-dim embeddings)."* The agent can produce SQL DDL statements or a Prisma schema with these tables. It will likely include the vector column (e.g. `VECTOR(1024)` type if using raw SQL) and foreign keys. Review and adjust types as needed, but the heavy lifting of writing out column definitions and constraints is handled. Since *Voyage 3.5* is our chosen embedding model, note its vector size (1024) in the prompt so the AI uses the correct dimensionality. Voyage 3.5 is a high-quality embedding model with 1024-dim vectors that outperforms many alternatives [25] – you'll use it to embed user essays and scholarship descriptions for matching.

Once the schema is ready, use AI to **generate boilerplate DB code**: if using an ORM like Prisma, have the agent write the model definitions and initial migration. If using a query builder or raw SQL, have it draft a few queries, e.g., *"Write a SQL query to find the top 5 scholarships most similar to a given 1024-dim embedding vector (using cosine similarity in pgvector)."* An AI like GPT-4 can produce the SQL using `cube` or pgvector functions. It might output a snippet with an `ORDER BY embedding <-> $1 LIMIT 5` (pgvector's cosine distance operator) – this is a good starting point that you can fine-tune.

For **TanStack Server Functions** on the backend, you can similarly prompt Codex to create server endpoints that perform DB operations. For example: *"Implement a TanStack server RPC called* `searchScholarships` *that takes a userId, fetches the user's embedding from Postgres, finds similar scholarships via a vector search, and returns the results."* Providing the earlier SQL or the Prisma client calls as context will help the AI craft a coherent function. The key is to specify the data flow (input/output) so the agent knows what to do.

5. LLM API Integration and Prompt Design

In this app, Claude 4.5 is the runtime LLM, so you'll integrate it via API calls in your backend. Use GPT-4/Codex to **scaffold the API wrapper**: for instance, ask it to *"Write a Node.js function that calls the Anthropic Claude API (Claude 4.5 Sonnet) with a given prompt and returns the completion. Include the correct API endpoint, headers, and error handling."* If you supply your API key (or a fake key) and model name (e.g. `claude-4.5-sonnet`), the AI will output a function using `fetch` or the Anthropic SDK to call the model. You can speed this up by citing the official Anthropic SDK usage – e.g., mention `@anthropic-ai/sdk` in the prompt so the AI uses the SDK's client. The Claude TypeScript SDK makes calling the model straightforward [11].

Prompt crafting: With your Claude API wrapper in place, involve the AI agents to help draft and refine the prompts. For the *personality analyzer* feature, for example, you might prompt GPT-4: *"Draft a system prompt for Claude that instructs it to analyze a student's profile and return 3 key personality traits or strengths relevant to scholarship applications. The tone should be supportive and professional."* The result can be iterated – maybe GPT-4 writes a decent prompt, then you test it by feeding a sample profile to Claude (via your wrapper or using the Claude console) to see if the output suits your needs. Refine wording with the help of the AI (it can suggest modifications if you say "make the analysis shorter" or "ensure it's in JSON format").

Similarly, for the *essay generator*, you can co-design a prompt: *"System prompt: You are an AI writing coach. Given a student's profile and a scholarship description, write a 500-word draft essay highlighting the student's fit. Ensure the tone is enthusiastic and the content stays on topic."* – have GPT-4 suggest the full prompt structure and maybe few-shot examples. Testing this with Claude Sonnet is crucial; Sonnet's stronger reasoning will yield a better essay, and you can verify the length and coherence. On the other hand, plan to use Claude Haiku for tagging tasks in production (like labeling scholarships with categories or quickly scoring a draft) since it's **much faster and cheaper** for real-time tasks [26] [13] . During dev, you can simulate this by calling Haiku with some sample inputs to ensure it produces the tags or classifications you expect (e.g. does Claude Haiku properly output a list of keywords for a given text?).

Don't forget you can use AI to generate **test cases and dummy data** as well. For instance, after implementing the essay generator, ask Codex: *"Generate a sample student profile and a scholarship description for testing the essay generator."* It will produce fake but plausible data you can use to manually test your end-to-end flow. Moreover, consider employing the AI in a TDD style: have it write a few **unit tests** (using Jest/RTL) for critical components or functions. For example, *"Write unit tests for the* `searchScholarships` *server function, mocking the DB and ensuring it returns results sorted by similarity."* LLMs are surprisingly good at generating tests and will cover edge cases you might miss [27] . This not only improves reliability but also assures you that the agent-produced code does what it should.

Stretch Feature Integration Timeline

If time permits, a few advanced features can be integrated. Here's a plan on **when to tackle these stretch goals** during or after the 6-day hackathon:

- **Qdrant Cloud (Vector DB):** For the MVP (by Day 6), you can rely on Postgres + pgvector for similarity search since the scale is likely small. Integrating Qdrant – a specialized vector database – is a stretch goal. Qdrant excels in performance at scale and offers hybrid search capabilities (combining vector similarity with filters) and might be worthwhile if you expect millions of embeddings or complex queries. However, adding it too early can consume time with deployment and data syncing. **Recommendation:** Defer Qdrant integration until **after the core matching is working**. If you're ahead of schedule (say around Day 5) and the core features are done, you could spend half a day deploying Qdrant Cloud and adjusting the search calls. Otherwise, safely push this to post-MVP. (For context: benchmarks show pgvector starts to lag behind specialized vector DBs by an order of magnitude at millions of vectors [28] , but for a prototype scale it's fine. You can note in documentation that Qdrant would be a next step for production.)

- **GBDT Scorer (XGBoost with LLM pseudo-labels):** This is an advanced, experimental idea – using Claude to generate "match scores" or labels for training a gradient-boosted tree model (e.g. using XGBoost) that can then predict scholarship fit on its own. Implementing this would involve

generating a dataset of profiles + scholarships with Claude-given match ratings, then training XGBoost and integrating it into the pipeline. It's a cool approach to make a fast, interpretable model influenced by the LLM's reasoning. **However, this should be last priority**. Only attempt this after the MVP is fully functional (likely after Day 6 or in a post-hackathon phase). It requires time to set up data and training loops, and it's not strictly necessary for a demo since Claude can handle the matching and scoring live. If you do tackle it, you might schedule it as a post-MVP optimization (e.g., Day 7+), once you have some usage data. Keep it in mind for future work rather than a core hackathon deliverable.

- **Observability and Monitoring:** Incorporating logging, analytics, or error tracking is important for a polished product, but in a hackathon you should keep it lightweight. Basic logging (to console or a file) can be done from the start for debugging. For real monitoring: **Sentry** (for error tracking) or similar services can be integrated in just an hour or two, so it's relatively *safe to do early* (e.g., Day 2 or 3, once you have some backend functions to monitor). Sentry is actually highlighted as a TanStack partner [29], and adding it simply involves an npm package and an init call with your DSN. Doing this early means that as you test, any crashes are caught and visible. On the other hand, **usage analytics** or metrics (like tracking how many times users generate essays) can be deferred until the app is stable. It's nice-to-have for judging usage patterns, but not critical for MVP functionality. If you find yourself with a bit of extra time towards the end (Day 6), you can sprinkle in a simple analytics script or at least log key events (e.g., log each Claude API call with user ID and prompt type – this data can help with debugging and demonstrating what's happening under the hood). In summary: minimal observability (error logs, maybe one monitoring tool) can be set up alongside development, but deeper analytics or dashboards are stretch goals best added once core features are in place.

AI Agent Workflow Cheat Sheet

- **Key Dev Tools & Plugins:**
- *GitHub Copilot (VS Code)* – AI autocompletion and chat, powered by Codex. **Use for:** on-the-fly code suggestions in React & Node. *Tip:* Write a doc-comment or function signature, then pause – Copilot will suggest the implementation.
- *OpenAI GPT-4 (ChatGPT or VS Code chat)* – higher-level help and code generation via natural language. **Use for:** asking "how do I…?" or generating larger code snippets (components, functions) from descriptions. Also great for debugging errors (paste stacktrace + ask for cause).
- *Google Gemini Pro 1.5 (API via Python or Node)* – long-context AI with multimodal input. **Use for:** absorbing large docs (like feeding in part of TanStack's docs or a whole Markdown file) and asking summary or integration questions. Also for data munging tasks (e.g., use it to parse HTML or JSON content that's too large for other models). *Setup:* Get API key from Google AI Studio [3], use the `google.generativeai` library as shown in their tutorial [4].
- *Anthropic Claude (Console or SDK)* – test your production prompts. **Use for:** interactive prompt tuning. Try both Claude Haiku 4.5 (fast, ~200k context) and Claude Sonnet 4.5 (more powerful reasoning, 200k context) to ensure your instructions yield the desired output. *Setup:* Use the Anthropic console for quick tests, and the Node SDK (`@anthropic-ai/sdk`) in code for integration [11].

- **When to Call Which Agent:**

- *React UI or Tailwind code:* **Call Codex (Copilot/GPT-4).** It's trained on tons of frontend code and will produce accurate JSX and Tailwind classes given a proper description [17] [30]. e.g. *"Create a NavBar with a logo on left and profile menu on right using Tailwind."*
- *Complex questions or planning:* **Call GPT-4 or Gemini.** If you need a structured plan (like an outline of steps to implement a feature) or to digest a large reference, use these. Gemini can handle giant context (e.g. entire library docs); GPT-4 is good for logical breakdowns.
- *Repetitive code (models, CRUD, tests):* **Call Codex.** It will churn out boilerplate quickly and even write unit tests for you [27]. For example, ask *"Generate a Jest test for the signup form validation."*
- *Scraping or external data needs:* **Call Gemini via a script.** Its ability to read long text means you can fetch a web page and give it to Gemini to extract exactly what you need (as shown in the Gemini scraping demo [7] [8]). This can populate a prototype dataset of scholarships if needed.

- *Prompt or content generation:* **Call Claude (dev-time).** When crafting prompts for Claude, there's no substitute for testing with Claude itself. Use Haiku for quick iterations (cheap and fast) and Sonnet for final verification of quality. For example, after Codex helps you draft a prompt, run it by Claude Sonnet with a sample input to make sure the output is on point.

- 📝 **Best Practices for Prompting Agents:**

- *Provide context.* Include relevant code or examples in your prompt so the AI can tailor answers [18]. If asking to use a certain library or API, show an example call.
- *Be specific with requirements.* State the language, frameworks, or constraints (e.g. "use TypeScript and TanStack Router v1 conventions") [31]. The more you pin down, the less generic the output.
- *Few-shot for style.* Show the AI a sample of the style or format you want [24] – for instance, give a small React component with your preferred patterns, then ask it to create another in the same style. It will mimic things like hooks usage, naming, comment format, etc.
- *Iterate in steps.* Don't expect a perfect solution in one go. Prompt for a basic version, test it, then ask the AI to improve or extend it [32] [33]. Treat the AI like a junior dev: gradually increase the task complexity.

- *Validate and refine.* Always run and verify the AI-generated code. Write or generate tests to ensure it works (agents can help write these tests, too). If something fails, tell the AI the error – it can often debug its own output.

- 🕒 **Hackathon Time Management:**

- Focus on **core features first** – e.g. profile input, matching algorithm, essay generation. Use the agents to get those working end-to-end by mid-hackathon. Avoid getting sidetracked on perfecting the vector search or complex ML tuning initially.
- Integrate **stretch features only after MVP** is functional. For example, only hook up Qdrant if you've got spare time and a working core app; the Postgres+pgvector will serve well enough for demo purposes. Clearly mark stretch tasks (like the XGBoost ranker) as *post-MVP enhancements* so you don't feel pressure to complete them under time. Judges appreciate a working core app over half-done extras.
- Keep an eye on **budget (API usage)**: Using GPT-4 and Claude can incur costs. During dev, use lower-cost options when you can (Claude Haiku instead of Sonnet for quick tests, GPT-3.5 or Gemini for less critical queries). This way, you won't burn through tokens on something that Copilot could handle

locally. Plan a budget for LLM calls and stick to it – e.g., allocate most of it to final testing of Claude prompts and use the free/dev tools for coding help.

- **Leverage agent speed:** What might take you hours (scouring docs, writing boilerplate, debugging trivial errors) the AI can do in minutes. Let it handle the grunt work so you can spend time on integration and fine-tuning, which are where a developer's eye is most needed. Essentially, use the AI as a force-multiplier to **build a lot in a short time**, but always keep yourself in the loop to steer the project towards the hackathon goals.

By following this AI-assisted workflow and toolkit, a solo developer can move rapidly from zero to a functional **scholarship matching + AI essay drafting app**. The key is balancing the strengths of each tool – using Codex and Gemini during development for maximum velocity, while preparing Claude's role for the production intelligence. With this guide and cheat sheet at hand, you can confidently navigate the build, knowing when to summon your AI assistants and when to code things directly, all within the tight timeline of a hackathon. Good luck, and happy building! [2] [6]

---

[1] GitHub Copilot · Your AI pair programmer
https://github.com/features/copilot

[2] What is the difference between OpenAI Codex and GPT models?
https://milvus.io/ai-quick-reference/what-is-the-difference-between-openai-codex-and-gpt-models

[3] [4] [6] [7] [8] Web Scraping with Google Gemini. Introduction to Google Gemini | by Madhu Shree Aravindan | Nerd For Tech | Medium
https://medium.com/nerd-for-tech/web-scraping-with-google-gemini-0b4a45765794

[5] The killer app of Gemini Pro 1.5 is video - Simon Willison's Newsletter
https://simonw.substack.com/p/the-killer-app-of-gemini-pro-15-is

[9] [10] [12] [13] [14] [15] [26] Claude Haiku 4.5 vs Sonnet 4.5: Detailed Comparison 2025
https://www.creolestudios.com/claude-haiku-4-5-vs-sonnet-4-5-comparison/

[11] Client SDKs - Claude Docs
https://docs.claude.com/en/api/client-sdks

[16] [29] TanStack Start v1 Release Candidate | TanStack Blog
https://tanstack.com/blog/announcing-tanstack-start-v1?utm_source=weeklyfoo&utm_medium=web&utm_campaign=weeklyfoo-104&ref=weeklyfoo

[17] [18] [19] [20] [24] [27] [30] [31] [32] [33] Stop Prompting Like It's 2023: 20 Codex Moves That Print Velocity | by Dálio Lage | Sep, 2025 | Medium
https://medium.com/@dalio8/stop-prompting-like-its-2023-20-codex-moves-that-print-velocity-17c5019a06bd

[21] [22] [23] Building a UI in 60 seconds with Shadcn and Framer AI - LogRocket Blog
https://blog.logrocket.com/building-ui-60-seconds-shadcn-framer-ai/

[25] voyage-3.5 and voyage-3.5-lite: improved quality for a new retrieval frontier – Voyage AI
https://blog.voyageai.com/2025/05/20/voyage-3-5/

[28] pgvector vs Qdrant- Results from the 1M OpenAI Benchmark
https://nirantk.com/writing/pgvector-vs-qdrant/