AI Projects through Prompting

Example Projects done through prompting only in Codex, Gemini CLI, v0 or OpenCode

Project Overview

?? Curriculum Roadmap

The bootcamp is structured into 10 weekly milestones, focusing on full-stack development and AI integration.

- **Week 01: Personal Dashboard**
  - **Project:** Personal Dashboard
  - **Goal:** Build a link organizer with a database for a custom browser "new tab" page.
- **Week 02: Build an App with Live Data**
  - **Project:** Events Dashboard
  - **Goal:** Live data visualization using APIs and charts.
- **Week 03: Build an App with Users**
  - **Project:** Shared Expense Tracker
  - **Goal:** Implementing User Authentication (Sign up/Log in) and data ownership.
- **Week 04: Build a Real-Time App**
  - **Project:** Live Chat Room
  - **Goal:** Create interfaces that update live without refreshing.
- **Week 05: Build an AI-Powered App**
  - **Project:** Ingredient Combiner
  - **Goal:** Integrate AI capabilities into your own application.
- **Week 06: Build a CLI Tool**
  - **Project:** Site Inspector CLI
  - **Goal:** Build a terminal tool that inspects any website.
- **Week 07: Build a Paid Product**
  - **Project:** Premium Version of a Previous App

- **Goal:** Turn your app into a business with real payments.

- **Week 08: Build an Online Store**
  - **Project:** Small-Batch Product Shop
  - **Goal:** Create a complete e-commerce experience with cart and checkout.
- **Week 09: Build an App with AI Agents**
  - **Project:** Link Sharing Community with AI Bots
  - **Goal:** Create AI bots that interact with your app autonomously.
- **Week 10: Build a Complete Product**
  - **Project:** Newsletter Tool
  - **Goal:** Combine everything into one polished, ship-ready application.

? Resources & Community

- **Platform:** Bootcamp Dashboard
- **Community:** Discord (Private Invite Required)
- **Support:** Access to a peer network for troubleshooting and building in public.

? Technical Constraints Note

> **Internal Note:** The student has identified a mismatch between the course content (Cloud/API heavy) and their professional environment (Air-gapped/Offline systems). This Wiki entry serves as a record of the curriculum prior to the refund request on **March 8, 2026**.

? Week 01: The Personal Dashboard

"Your Digital Command Center"

Welcome to the first real project. In the first week, we aren't just coding; we are reclaiming your browser. Instead of a cluttered "New Tab" page, you're building a lightning-fast, minimalist link organizer that lives on your machine.

The goal for this milestone is to master **CRUD** (Create, Read, Update, Delete) operations and understand how a frontend interface talks to a local database.

?? Phase 1: The Foundation

Before you paste your prompt into an AI, you need to decide on your **Tech Stack**. A good tech-stack is one that lets you ship the fastest, but here are some recommended paths:

Stack	Why choose it?
Next.js + Tailwind + SQLite	The modern industry standard. Fast, sleek, and everything stays in one folder.
Python (Flask) + Bootstrap + TinyDB	Great if you prefer a lighter, more logic-focused backend approach.
Deno + Typescript + HTMX + AlpineJS	Great if you prefer a lightweight stack with simple components that lets you create a single executable binary with the help of demo.
Astro + HTMX + AlpineJS	The AHA-Stack. Simple, minimal and effective.

>

> **Action Item:** Decide on your language. Do you want to go the JavaScript/TypeScript route or the Python route?

? The Master Prompt

Once you've picked your stack, use this comprehensive prompt to generate the "v1.0" of your dashboard:

```
Build me a personal link dashboard that I'll use as my browser's new
tab page.
[INSERT CHOSEN STACK HERE: e.g., Using Next.js and SQLite]

The app organizes links into categories. Each category has a name and
contains multiple links.
Each link has a name and a URL.

Features I need:
- Display links grouped by category in a clean grid/card layout.
- Add a new link (with name, URL, and category selection).
- Edit and Delete existing links.
- Create and delete entire categories.
- Store everything in a local database (setup instructions included).
- Run on localhost.

Design: Make it clean, dark-mode friendly, and minimal. It must load
instantly.
```

?? Phase 2: Iteration Prompts

Use the next prompts to enhacne your dashboard with more features. Often is less more and small iterations make it easier to get better results.

When you enable git you can also always go back and undo changes.

Also the Planing-Mode in many agents can help to layout a plan before doing any major tasks.

1. Real-Time Fuzzy Search & Filtering

```
Implement a global search bar at the top of the dashboard.
As the user types, it should filter the displayed
categories and links in real-time. Use fuzzy-matching
```

```
logic so searching for 'git' matches 'GitHub' or
'GitLab'. If a category has no matching links, hide the
entire category heading from the view to keep the UI clean.
```

2. Dynamic Favicon & Metadata Fetching

```
Enhance the link display by adding a 16x16px favicon next
to each link name. Generate the icon URL dynamically
using: https://www.google.com/s2/favicons?domain=[URL]&sz=32
Add a fallback 'Earth' icon using your icon library if the
favicon fails to load, ensuring the layout remains
consistent and aligned.
```

3. Persistent Dark Mode & System Preference

```
Add a theme toggle component (Sun/Moon icon). The system
should check for the user's OS preference using
'prefers-color-scheme' on first visit but allow manual
override. Store the chosen theme in localStorage. Apply
a 'dark' class to the root HTML element and ensure all
CSS transitions for colors are smooth (300ms duration).
```

4. Drag-and-Drop Reordering (Persistent)

```
Integrate a drag-and-drop library (like dnd-kit) to allow
reordering of links within a category. When a link is
dropped, send a PATCH request to the backend to update a
'sort_order' integer field in the database. Ensure the UI
updates optimistically so there is no visual lag while
the database saves the new order.
```

5. Data Portability: JSON Backup & Restore

```
Create a 'System' modal that allows data management.
Include an 'Export' button that generates and downloads
a 'dashboard_backup.json' file containing all data.
Also, include a file upload input for 'Import' that
parses the JSON file, validates the schema, and performs
a bulk-insert into the database to restore the setup.
```

6. Smart URL Validation & Auto-Naming

```
Improve the 'Add New Link' form. When a user pastes a URL,
use a regex to validate it. If valid, use a client-side
fetch or server-side route to attempt to scrape the
<title> tag of that website. Automatically populate the
'Link Name' field with this title, allowing the user to
```

```
edit it before saving.
```

7. Keyboard Navigation & "Quick Actions"

```
Implement global 'Hotkeys' for power users. Pressing '/'
should instantly focus the search bar; pressing 'n'
should open the 'Add New Link' modal; and pressing 'Esc'
should close any open modals. Add a small footer or
tooltip that visually reminds the user of these
shortcuts to improve discoverability.
```

? Implementation Tip

When using these, I recommend pasting the **relevant file code** (e.g., your `page.tsx` or `api/links.js`) along with the prompt. This prevents the AI from hallucinating variable names that don't exist in your project.

? Week 02: The Centralized Event Hub

Welcome to **Week 02**! You've already knocked out your personal dashboard; now we're stepping into the world of **Dynamic Orchestration**. This week, you aren't just building a display—you are building the "Nervous System" for every application you will ever write.

? The Mission: Observation & Architecture

The goal is to build a **Centralized Event Dashboard**. This application acts as a private "Log Sink" or "Command Center." It provides a secure API that listens for "Events" from your other apps—whether it's a successful user signup from a web app, a cron job failure in a Python script, or a simple `cURL` message from your terminal.

Build a two-part system:

1. **The Receiver (Remote API):** A cloud-hosted endpoint that is "always on," waiting to catch events from your other apps, scripts, or servers.
2. **The Viewer (Local Dashboard):** A high-performance real-time feed where you can search, filter, and visualize the data flowing through your API.

?? The Architecture

- **Project-Based Isolation:** Manage multiple apps from one hub.
- **API Key Authentication:** Secure your endpoints so only your authorized apps can post data.
- **Persistent Cloud Storage:** Use a cloud database so your data is safe and accessible even when your local machine is off.

? Step 0: Choose Your Stack

Before you run your first prompt, decide on your **Tech Stack**. You will need a database (like Supabase or PostgreSQL) to store your projects and historical event data.

Component	Option A: Modern Serverless (Recommended)	Option B: Robust Python
Backend API	Next.js API Routes (Vercel) or Hono (Cloudflare)	FastAPI (Render or Railway)
Database	**Supabase** (PostgreSQL + Realtime)	**MongoDB Atlas** or Supabase
Frontend	Next.js + Tailwind + Shadcn UI	React (Vite) + Tailwind
Live Updates	Native Supabase Realtime	Pusher or Socket.io

> **Student Note:** Are you a **Next.js + Tailwind** fan? Or do you prefer **Python (FastAPI) + React**? Specify this in your initial prompt so the AI builds the API routes correctly.

? The Starter Prompt

Copy and paste this into your AI chat to generate the foundation.

```
Build me an events dashboard. Other applications send events to it
through an API, and I see them in a real-time feed.

The app has two parts:
1. A REST API that accepts events via POST request (with API key
authentication). This needs to run on a remote server so it's always
available, even when my computer is off.
```

```
2. A dashboard that displays events in a feed, with search, filtering,
and charts. This can run locally.

Each event has: a channel (category like "orders", "signups",
"deploys"), a title, an optional description, an optional emoji icon,
and optional tags.

Features I need:
- POST /api/events endpoint that accepts JSON and stores events in the
database
- API key authentication (generate a key when creating a project)
- A feed page showing events in reverse chronological order
- Filter events by channel
- Search events by title, description, or tags
- At least one chart showing event activity over time
- The dashboard should update in real-time when new events arrive
- Use a cloud database that's always available (Supabase, Convex, or
similar)

Make it clean and functional. I want to actually use this to monitor my
own projects.

Before making any decisions on the stack. Make a stack proposal and ask
me which I want to use.
```

Once you have the initial dashboard frontend, api and database running to your liking, you can continue with the next prompts, to make it better or add additional features to it.

? Evolution: 7 Prompts to Pro Power

Once your API can catch a message, use these iterative prompts to turn a "basic table" into a professional monitoring tool.

The Roadmap:

- **The Real-Time Subscription:** Implement a real-time listener (e.g., Supabase Realtime) to push new events to the feed instantly with a highlight animation.
- **Smart Emoji & Auto-Parsing:** Add logic to automatically assign emojis based on channel names (e.g., ? for orders, ? for deploys) if one isn't provided.
- **Multi-Project Management:** Build a settings page to manage multiple projects, each with its own name and unique API key validation.

- **Advanced Time-Series Analytics:** Integrate Recharts to visualize events per hour and top channels using bar and pie charts across various time ranges.
- **Desktop & Critical Alerts:** Add native browser notifications triggered by specific tags like #error or #urgent, even when the dashboard is in the background.
- **The "Deep Dive" Inspector:** Create a clickable side-drawer for every event to display the raw JSON payload and include a "Copy as cURL" button for debugging.
- **Key Rotation & Security:** Implement a security feature to regenerate API keys, instantly voiding old credentials to protect against leaks.

?? The Detailed Prompt List

1. The Real-Time Subscription

> "Since our database is in the cloud, implement a **Real-time Listener** (e.g., Supabase Realtime). Ensure that when the remote API inserts a new event, the local dashboard pushes it to the top of the feed automatically with a subtle 'new item' highlight animation."

2. Smart Emoji & Channel Parsing

> "Enhance the API logic: if an incoming event doesn't specify an emoji icon, automatically assign one based on the `channel` name (e.g., 'orders' gets ?, 'deploys' gets ?, 'errors' gets ?). Display these icons prominently next to the event title in the feed."

3. Multi-Project API Key Management

> "Build a 'Project Settings' page in the dashboard. Allow me to create multiple projects, each with its own name and unique generated API key. The API should now validate the key against the database and tag the incoming event to the

> correct project automatically."

4. Advanced Time-Series Analytics

> "Add a 'Metrics' tab. Use **Recharts** to create a bar chart showing 'Events per Hour' and a pie chart showing 'Top Channels by Volume.' Allow me to toggle the time range between the last 24 hours, 7 days, or 30 days."

5. Desktop & Push Notifications

> "Add a toggle in the dashboard for 'Critical Alerts.' If an event is received with a specific tag (like #error or #urgent) or a 'High' priority status, trigger a browser-native desktop notification so I see the alert even if the dashboard tab is hidden."

6. The "Deep Dive" JSON Inspector

> "Make each event card clickable. When clicked, open a side-drawer (Slide-over) that shows the full raw JSON payload received by the API formatted for readability. Include a 'Copy as cURL' button so I can easily replicate the exact request for debugging."

7. API Key Security & Rotation

> "Implement 'Key Rotation' logic. In the Project Settings, add a button to 'Regenerate API Key.' This should instantly void the old key in the database and provide a new 32-character secret to the user, ensuring security if a key is ever accidentally leaked."

Best Practices for Working with AI Agents: A Verification-Driven Approach

Working effectively with AI agents requires a fundamental shift in how we approach development. While AI can generate vast amounts of code instantly, the primary challenge is no longer authorship, but **verification**. Modern software engineering with AI is less about crafting the "perfect prompt" and more about maintaining a **disciplined, step-by-step process**.

Here is a comprehensive guide on how to optimally interact with AI agents, supported by real-world examples.

1. The Mindset Shift: Engineering Over Prompting

In the AI era, your core value shifts from typing speed to three essential competencies: **Problem Definition, Decomposition, and Verification**.

- **You are the Architect, AI is the Typist:** You are entirely responsible for the logic, security, and data flow.
- **Avoid the "One-Shot Trap":** A common beginner mistake is the "5-second high"—asking the AI to generate a complete application from a single sentence. This creates a massive **technical debt of understanding**. If you cannot verify the output, you do not own the code, making it a liability rather than an asset.

2. Precise vs. Imprecise: The Power of Constraints

Your prompts must be **highly precise when it comes to rules, constraints, and edge cases**. Ambiguity is the enemy of secure AI-generated code.

- **Example: The Server-Side Cart Calculator** If you simply ask an AI to "build a shopping cart," you risk getting vulnerable client-side logic where a user could manipulate prices. Instead, you must define a strict trust boundary where the server is the single source of truth. A precise prompt establishes rigid constraints:

- **Ignore Client Prices:** Explicitly state to never accept a price sent from the browser.
- **Validation Constraints:** Define mathematical rules, such as $Quantity \ge 1$ and $Tax/Discount \ge 0$.
- **Order of Operations:** Mandate that discounts must be applied *before* calculating tax.
- **Precision:** Require the system to round money to 2 decimal places (or use integers/cents to avoid floating-point errors).

3. Short vs. Long Prompts: The Iterative Workflow

Instead of writing one massive prompt, the most effective strategy is **iterative prompting**. Start with a structured, medium-length prompt to define the goal and constraints, then transition to short, highly focused commands to build and refine the output incrementally.

- **Example: Rapid UI Iteration via Short Prompts** During the development of a real-time events dashboard, a developer used extremely short prompts to polish the UI once the foundational context was established by the AI.
  - The developer prompted: *"make the bar chart smaller and horizontal. different colors for channels. randomly assigned."*.
  - Because the AI already understood the established architecture, this short prompt was enough for the agent to formulate a highly detailed implementation plan—creating a compact horizontal layout and using a deterministic hash-to-color function so that channels kept a stable pseudo-random color across page reloads.
  - The developer then followed up with rapid micro-prompts like *"increase font contrast of labels of bar chart"* and *"the colors of the background and the overlays do not match the dark"*. The AI executed these perfectly by tuning CSS overlay tokens and applying theme-aware colors.

4. The 7-Step Verification Loop: Trust but Verify

Never assume the agent's first output is flawless. You must **treat AI-generated code like code from a stranger—useful, but untrusted until proven by tests**. Fundamentals matter more than ever: security, data flow, and edge-case thinking are your primary tools.

To ensure quality and maintain control, adopt this repeatable **7-Step Iterative Loop**:

1. **Define the Goal:** State the objective in one clear sentence.
2. **Establish Rules:** List the non-negotiable technical constraints (what *must* be true).
3. **Provide Examples:** Define the exact expected Input $\rightarrow$ Output mappings.
4. **Identify Edge Cases:** List "weird" or bad situations the system needs to handle.
5. **Request a "Small Piece":** Ask for a specific function or logic gate, not the whole system.
6. **Demand Tests:** Require the AI to provide runnable assertions to prove its logic.
7. **Iterate:** Treat failing tests as data. Use them as a "flashlight" to refine your next prompt and fix ambiguities in your rules.

- **Example: Actively Verifying System Logic** Verification isn't just about automated tests; it's also about actively questioning the AI's architectural decisions. When the AI added data filters to a dashboard's charts, the developer didn't just accept the code. They verified the logic by asking: *"how do the filters work? do they filter visible data or data on the server?"*. Only after the AI confirmed that the filters were applied securely on the server side via SQL query parameters, did the developer instruct the AI to solidify this architecture: *"document the changes do you?"*, ensuring the verified logic was permanently recorded in the project's README.