Guide: Building with AI (as a Non-Developer)

By Coby Bergman

Like many of Kyle's readers, I'm a tad obsessed with AI. I run an AI-native GTM consultancy, and for the past 5 months I've been spending ~3-8 hours *daily* building AI applications - despite never having programmed before in my life.

It feels a lot like being a kid again playing with lego. Except now, instead of just building the sets, I get to design them, too.

At the same time, I've found there's often a substantial gap between an impressive looking AI demo and a solution someone might actually want to use.

This tactical guide contains my hard-won learnings covering the distance from AI prototype to production-grade AI application - the prompting techniques that actually worked, the context engineering principles that prevented failure, and the deployment gotchas that at times had me wanting to tear my hair out.

No theory. Nothing I don't personally use day in and day out myself.

*Examples in this guide are pulled from my recent experience building a production-grade, Star Wars-inspired generative text adventure game called "Echoes of Rebellion".

You can follow along with my up-to-date experiments and content here:

- Linkedin
- X (Twitter)
- Website - Leadership in Practice

Table of Contents

The Prompt *is* the Product

Unlike traditional software development, with AI-native applications the core application engine - its logic, personality, and constraints - is often dictated by the Master Prompt. The code is often just a scaffold to service the prompt.

To a significant extent, **the Master Prompt is the software**.

I've consistently found that my biggest challenges building AI-Native applications are not typical bugs, but figuring out how to craft the Master Prompt in a way that solves for three things:

1. **Quality:** Can the application deliver the expected quality of experience under ideal conditions?
2. **Consistency:** How consistently does it deliver that quality of experience under regular conditions?
3. **Anti-fragility:** How does it maintain that quality of experience in real-world, unexpected, or "red-teamed" conditions?

To put this into perspective, I probably spent around 36 hours building the Echoes of Rebellion game, and 24 hours of those hours were spent dialing in the Master Prompt with these three goals in mind.

Effective Prompting Techniques

My general approach was to treat the AI (primarily Gemini 2.5 Pro) as a strategic thought partner, not just an order-taker.

General-Use Prompting Techniques

- **Meta-Prompting (Prompting for Prompts):** To create the best possible Master Prompt, my first step was to have the AI help me design it.
    - *Prompt:* "Help me craft a prompt that I could feed to another LLM to generate the highest quality PRD and dev plan for a delightful and compelling application."
- **Meta-Meta Prompting (Guidance for Generating Meta-Prompts):** To help the LLM generate maximally effective prompts and meta-prompts, I fed it prompting guides.
    - *Prompt:* "I've attached three prompting guides. Please identify the relevant frameworks that might level up the quality of our prompt and overall project, and make sure to incorporate them into your responses as we go forward."

- **Handholding**: I frequently ask the LLM to guide me through the steps to complete the project. Find this is especially useful when doing something for the first time.
    - *Prompt:* “I have zero experience with doing work like this. Going forward, guide me through everything step-by-step, detail-by-detail, click by click. Leave no stone unturned, and assume zero upfront knowledge.”
- **Giving the LLM an Offramp**: LLMs often seek to please, and won’t always admit (reactively, much less proactively) when they’ve made a mistake. At the risk of anthropomorphizing, providing an upfront offramp makes it easier for LLM’s to admit or sidestep mistakes by helping them save face.
    - *Prompt:* “Please give me your confidence level in your analysis. Also, my priority here is accuracy and quality over completeness. If you’re not sure about anything, I’d rather you leave it blank than take a guess and risk making something up. If you leave anything blank, just let me know.”
- **The "Thought-Partner" Approach:** Rather than giving the LLM orders, I ended just about every prompt with an invitation for it to share it’s “candid thoughts”, “creative guidance”, or “candid feedback”.
    - Prompt: “I’d be grateful for your candid feedback and guidance here. I’m not married to the current approach.”
- **Forcing "Self-Critique":** A particularly high-value prompt was asking the LLM to role-play an expert and critique itself.
    - *Prompt:* “Play the role of an expert [PM/developer/designer] and conduct a comprehensive, candid, no-holds-barred analysis of what we've completed. What's good? What needs improvement? What risks should we consider?"
- **Using "What Good Looks Like" (WGLL) Examples:**
    - *My Experience:* With only 1-2 examples, the LLM tended to just recreate those same situations. With too many, the quality could become diluted.
    - *My Solution:* I used a small handful of high-quality examples (a minimum of 3, max of 7) and was explicit that they were for inspiration, encouraging the LLM to use its creativity to generate novel situations or reuse elements in new ways.

Consideration When Designing the Application Master Prompt

Balancing ‘Adaptive’ vs ‘Strict’ Master Prompt Mechanics: Found there was a delicate balance between using strict rules and adaptive principles when crafting the Master Prompt.

How I think about this:

- **Adaptive Principles**:
    - Gives the LLM more creative control on how to handle various situations
    - Relies on LLM judgement to deliver a great UX
- **Strict Rules**:
    - Kind of like hard-coding rules for how the AI will handle certain situations (but not actually, because the prompt is still a generative, statistical system; really we’re just dramatically influencing the probabilities)

    - Relies on human judgement to accurately predict the nature of the scenarios the LLM will experience
- **UX Tradeoff**:
    - Adaptive principles introduce more unknown risks to the system, will delivering a more natural UX (as long as the LLM's judgement is on point)
    - Strict rules introduce more known risks to the system, while reducing the range of unknown risks. The known risk is that the LLM will take a one-size-fits all (aka when you have a hammer, everything is a nail) approach to specific predetermined situations.
- *Example:* When I gave the LLM full adaptive control over the game's narrative pacing, it introduced an occasional but critical bug where the narrative would occasionally get stuck in a loop (among other occasional issues). To fix this, I set a strict rule that the LLM must find a way to progress the game's narrative in a substantive way within 2-3 'turns' max. This solved the narrative loop bug and reduced the range of unknown risks, but introduced a new known risk: the LLM would sometimes generate deux ex machina style events that felt forced, and detracted from the 'grounded' gaming experience I was going for.

Context Engineering

I find it helpful to think of LLM context like working memory. For example, info in my working memory is quicker to recall, and details are easier to recall. At the same time, try to hold too much in my working memory and the quality and reliability of recall degrades. LLMs work similarly.

This is why context engineering has been crucial for getting to quality outcomes, both during my AI-assisted build process and when designing the Master Prompt for production-grade applications.

The core challenge is always the same: providing enough context for consistent, high-quality, and anti-fragile outputs without bloating the prompt, which can lead to a loss of LLM focus, longer load times, and higher API costs.

The "Builder" Perspective: Managing My Workflow

- **The Inevitable Context Problems:** On any non-trivial project, I found I would inevitably run into one of three issues: the context window getting too long and confusing the AI (even with a 1M token window, I saw issues around the 150-200k token threshold), hitting token limits, or hitting rate limits on a frontier model.
- **My Suite of Solutions:** To manage these issues, I developed a few go-to tactics:
    - **Editing an earlier prompt to reduce token count.** I thought of this like loading a previous save state in a video game. Some AI applications are now introducing branching for this, where the old thread is saved while a new, pruned branch is created.

    - **Starting a new chat thread.** This is a "new game" approach—a blank slate where the LLM forgets everything. (Note: I found that even with features like ChatGPT's memory, it only retains snippets and doesn't solve the core problem of losing full thread context).
    - **Waiting for rate limits to refresh** or **falling back to a lower-capability model.** My takeaway here was that it was usually better to wait for the best frontier model than to switch to a lower-tier one and fight to get B-level outputs up to an A-level standard.
- **The Go-To Workflow: The "Context Transfer":** Since starting new threads became a regular part of my process, my core challenge was transferring all the relevant context without the cruft. My most effective technique for this was meta-prompting.
    - **My "Context Transfer" Meta-Prompt:** "Generate a prompt for a new LLM to be able to pick up from where we left off. Include all the relevant insight and context and design principles necessary for it to do an exceptional job without me having to provide it any additional information. Make sure to include an updated PRD and dev plan based on everything we've learned, as well as a detailed review of all the potential pitfalls and key success factors we've discovered, but don't limit your response to that. Be extremely thorough and detailed in thinking through the context the other LLM might need, and in crafting the prompt itself."
    - **Critical Documentation**: I also attach any other documentation to the new thread that I feel is important for the LLM to understand in full

The "Application" Perspective: Designing the AI Engine

- **Higher Stakes:** All the "Builder" considerations apply here, but the stakes are higher for the application's AI engine. Poor context engineering directly leads to hard costs (APIs) and a degraded user experience. This means building Context Engineering Principles directly into my PRD from day one.
- **A Case Study in Trade-offs (from Echoes of Rebellion):** I faced a series of context engineering trade-offs when designing the application.
    - **Application State Tradeoff:**
        - I used temporary browser storage for the application state. That said, any browser refresh (due to a bug, a lost connection, etc.) would wipe the user's progress.
        - I ended up engineering a "save" capability that could restore state from the minimum viable context.
    - **Game Length Tradeoff:**
        - My initial vision was an epic, endless generative gaming adventure, but this was bloating the context window and impacting narrative quality in later stages of the game.
        - I ended up capping game length to avoid context-related issues.
    - **Detail / Relevance Tradeoff:**

    - The more detail I gave the LLM about the game ‘universe’, the greater it’s ability to generate a richly detailed and emotionally resonant narrative experience
    - At the same time, too much context introduced focus/consistency risks, and lengthened load times (if I was using API calls that had fees, it might have contributed to operating costs as well).
    - Finding a healthy balance that met my bar for quality, consistency, and antifragility involved a lot of iterating and QA testing

The Deployment Gauntlet

The transition from my local machine to a live production environment was where I encountered the most difficult technical issues. While I solved countless bugs throughout the Echoes of Rebellion build process, 80% of my time spent resolving technical issues was dedicated to 6-8 deployment issues that the LLM I was working with struggled to resolve. In total, I spent ~6-8hrs on these items, or ~1hr/issue.

Understanding the Break Points

- **Anticipating the Production Break:** I found that code working perfectly locally didn’t always work ‘as is’ in production. The hosting environment is different.
- **Common Points of Failure I Encountered:**
    - **Environment Configuration:** Setting up Docker files, YAML files, and getting GitHub CI/CD working.
    - **API Keys & Dependencies:** Improper handling of keys and dependency version mismatches were frequent culprits for failed deployments.
    - **LLM Hallucinations:** I experienced Google AI Studio Build lying about its capabilities (e.g., "I've generated the Docker file," when it couldn’t) while Gemini 2.5 Pro made a variety of unsanctioned and unacknowledged changes to my planning documents.

Solutioning

Proactive Solutions

- **Building with Foresight**: To proactively mitigate production issues, I now include context regarding the future production environment in my PRD + dev plan. If I don’t know the exact stack I’ll be using upfront, I share examples of the types of tools and hosting environments that I’m considering (e.g. Google Cloud Run, Google Analytics, etc.)
- **Meta meta prompting:** I also now include a list of common production issues in my ‘Pre-flight Checklist’ meta meta prompt when crafting my initial PRD and dev plan.

Reactive Solutions

- **Troubleshooting**: When troubleshooting production issues, my process now involves asking the LLM to consider the specific tools I'm using (and their unique configuration requirements) in my production environment as a potential cause.
- **Rules for Stuck LLMs:** I found my issue-resolution time was highly correlated with how quickly I was able to spot an LLM spinning its wheels.
    - I learned to immediately switch to a different (frontier) model any time I spotted the following situations:
    - **The '8 Ball':** If an LLM tried the same failed solution more than once I immediately switched to a different model.
    - **The "3-Strikes" Rule:** When the LLM failed to solve a bug within three attempts, I would pause and immediately switch to a different LLM.
    - Typically the new LLM, with its 'fresh set of eyes', was able to resolve the issue in minutes.

LLM Tool Stack: Evaluation

Vibe-coding tool eval coming soon (e.g. Bolt, Lovable, Cursor, Windsurf, Gemini CLI, Claude Code). Follow along on Twitter for details.

ChatGPT o3

Plus Subscription

STRENGTHS

- Good code
- History is easily searchable
- Sizeable context window
- Generous rate limits

WEAKNESSES

- Not as opinionated
- Inconsistent artifact use
- Inconsistent at identifying up-to-date versions of project dependencies + tooling
- Can be a bit lazy without deep research for anything requiring search
- Particularly susceptible to 'capability-related hallucinations' - saying it can do things it can't, and then faking the output

USE CASES

- Bug fixes
- Researching AI tooling options and tool strategy considerations (find it's hit and miss, but with a higher 'hit' rate than the other models)

Claude Opus

Pro Subscription

STRENGTHS

- Solid thought partner
- Great code
- Acts as a decent tutor
- Can navigate the transition to a production environment
- Decent web search
- Workhorse

WEAKNESSES

- Small context window
- Brutal rate limits
- Hasn't been great at thinking through AI tool strategy

USE CASES

- Polishing PRDs & development plans
- Coding
- Refactoring codebases
- Best at creative writing - when fine-tuning an existing draft

Gemini 2.5 Pro

Google AI Studio

STRENGTHS

- Solid thought partner
- Decent 'contemporary' web search
- Workhorse
- Great code
- Acts as a fantastic tutor
- Great at Google Cloud Run configuration
- Virtually unlimited context window
- Virtually unlimited queries
- Cost efficient APIs with generous free tiers
- Great at creative writing when crafting the first draft

WEAKNESSES

- A bit sycophantic
- Struggles with the transition to a production environment (esp. re API key management, and versioning/dependencies)
- May lose focus around 150-200k tokens
- Frequently makes unsanctioned edits when I feed it existing creative writing drafts, and often doesn't own up to it

Prototyping Tools

Claude Artifacts & Google AI Studio Build

STRENGTHS

- Easily accessible prototyping tools

WEAKNESSES

- (Over)-optimized for rapid prototyping:
- Minimal if any focus on building a modular code base
- Minimal if any in-line code explanation
- **Production-relevant files don't always get built (e.g. .gitignore, Docker, YAML, etc.)**
- Frequent issues with API key handling when transitioning to a production environment

USE CASES

- I've stopped using these for projects with any degree of complexity
- Examples of where Claude Artifacts performed well: A space-themed tic tac toe game my daughter vibecoded
- Turning information-dense documents into de shareable 'webpages' (like this LLM evaluatio

Leadership in Practice

Click here to visit the Claude Artifact.