Acceleration Guide to Caffeine AI

Dexter Right
Head Meme Architect, Caff/acc

October 11, 2025

Contents

Disclaimer

This is not an official document for Caffeine.ai. The Caffeine team or organization does not endorse it, nor does DFINITY have any connection to this or represent any of DFINITY's views.

1 Introduction

First and foremost, you must understand the system you are working with and your role in it. You are the second-in-command pilot of a plane; the main pilot, Caffeine, is a highly skilled navigator but relies on your instructions to reach the target airport safely. This means that if your navigation is off, you will not land safely and may end up at a different airport than intended. Most likely, this happens when the communication between your wants and expressions is disconnected. You need to be able to communicate, express, and instruct Caffeine with the details you desire. If not, Caffeine will make assumptions to fill the gaps between what you have instructed it to create and the additional elements necessary for a complete application. Caffeine is a system that builds and handles the full process autonomously: once instructed, it starts building the application from start to finish, backend to frontend, draft to deployment—all with one click.

To be skilled in Caffeine is to be skilled in linguistics: the ability to express what you think, want, visualize, or need in a complete manner. Here, you also need to adopt a certain way of conversing and adjust a few preconceptions you might have, such as:

"Caffeine will know what I want and build this application in the way I want and need it." This is only true if you are descriptive and explicit in your instructions. Vague and general statements will create vague and general applications. As the saying goes, "you are what you eat." In Caff/acc, I like to coin the phrase "you get what you prompt." It is a deterministic system: Caffeine will listen to you and attempt exactly what you instruct it to do, just like a bow and arrow. You draw the bow, aim it, and releasing it shoots the Caffeine arrow. You are the one who must ensure you aim correctly and hold the bow the right way; the rest of the hard work will be done for you. You are only the brain of the operation—you are the architect. In this guide, I will use the following correlation for almost everything below: Caff/acc to @caffeineai is like:

- AMG to Mercedes
- Alpina to BMW
- ABT to Audi
- RUF to Porsche

An intelligence performance boost.

Building an application is like building a house: you are the architect, and Caffeine is the contractor executing the plans you provide. Also, understand that by following this guide, you will be accelerating. Not everyone can drive a Porsche GT3 RS—some can handle it, but not everyone is a racer. This document is for people who want to go at high speed and experience the thrill, the adrenaline of accelerating to 100 km/h in 3.2 seconds. You won't even have time to blink. To be dedicated to Caff/acc is also to have a mindset of problem-solving: trying your best to figure out how to resolve issues or do things in the way that is right for you. Just like with racing cars, everyone has a driving style; here, it's the same—everyone has a prompting/instructing style, which delivers different results. It's a skill you need to develop if you want to create unique, high-end, and eye-catching apps. I will also cover some generic problem-solving behaviors that Caffeine exhibits and how to effectively counter them to ensure you are turbocharged to accelerate.

2 Caff/acc Philosophy and Mission Statement

1. "I can't" doesn't exist in our environment; there is only "I have not figured it out yet, but I will."

2. There are no limits to my creativity. Combining with Caffeine, which has limitless capabilities, I can create anything I envision.

3. We never fail; we only learn, improve, adjust, and execute v2.

4. There are no dumb questions.

5. The issue is not the product but the user (with exceptions).

Before we start, I would like to say that this is not a must-follow guide; it's an example of how you can use Caffeine. Everyone has different needs, different styles of prompting and expressing, likes different design styles, and so on. You can follow this, but I would rather you take it as an opportunity to see what is possible and start developing your own unique style.

3 Building the House: Practical Development Guide

3.1 Chapter 1: Planning and Architecture: Blueprint

When you think about how the process of building a house works, you find that the "My login button needs to be a blue color" comes last. This is the decor, the icing on the cake. First, you start by visualizing, architecting the application and how it will function, what specific data it will actually hold, what the user will do in terms of actions, and how the application should respond and behave. This is the blueprint where architects normally go over how the structure of the house will work: the foundation, the beams carrying the weight, the distribution of set loads across pillars. This can be mapped into the following for every application:

Foundation: The first layer where you build the house on; this is the backend. Here, you will tell Caffeine in the prompt what functionality needs to be implemented, which Caffeine will translate into backend code. On this foundation, Caffeine will build out the frontend, which is what is actually seen by the user and interacted with, like the doors, windows, or bathrooms in a house.

The Skeleton: The skeleton refers to the layout of a house: how many rooms do you need, where is the bathroom, where is the living room, where is the door, the balcony, which way does the door open, how many windows does your house have, is the house oriented southwest or east to face the sun? This is translated into: how many pages do you need, what do you need on the front page, which page functions in what type of way.

The Interior: The interior refers to actually structural walls being placed, the doors, the glass windows, and such to make a house almost ready to live in. This is the part where you tell Caffeine what color scheme you like the application to have, the feel and vibe you like to express, the specific libraries for React to use in what way.

The Decor: Here it's the couch, sofa, bed, tables, and such. Here, you tell Caffeine specifically that the button needs to be the blue color or the front page needs to have a specific image displayed.

This is the blueprint for a "PROMPT" for Caffeine. A prompt is defined in the following way:

Context: The additional information that will help Caffeine to fill in the gaps between your wants and the final application.

Goal: What you are trying to accomplish or the problem you are trying to solve.

Limitations: What to do/use or not to use.

Output: The desired application.

3.2 Chapter 2: Before the First Prompt

Before you even start developing with Caffeine, you ought to have the blueprint ready. The best way to do it is to have another LLM alongside you, like I do everything in Grok 4. Why this is necessary will become clearer later on. Before you enter the first prompt into Caffeine, you will first write out and design this application in theory with Grok or any other LLM. You will tell Grok the following:

"Grok, we are going to develop an application with caffeine.ai. Here is what I want to make [Blueprint]. Can you do some research on how caffeine.ai works? You will be assisting me in creating this application. For now, let's just discuss what you think of my application and if this makes logical sense in terms of Software Engineering. Did I miss something, or do you have any extra recommendations? We will be using agile development methodology in creating this application."

Agile means the following: you have a goal of creating a car; in the first iteration, you make a wheel; second, you make a bike; third, you make a buggy; fourth, you make a car. You need to understand some concepts in SE, which will be linked in the appendix at the end. I encourage you to drop this document into Grok or read about them on the internet.

You send this prompt to Grok, which will generate an answer and maybe some suggestions. Then, you talk a little bit with Grok about the improvements or possible different approaches. Once you are satisfied with what you discussed with Grok, you will tell him the following:

"Grok, we are going to start developing this application. Caffeine uses natural language for creating applications, so all our prompts need to be in natural language when instructing Caffeine, except when we run into a problem and actually need to give Caffeine code snippets. Let's first talk to Caffeine about what we want to make, and you can ask some questions to Caffeine if what we want is possible. Maybe Caffeine can add suggestions or different approaches. Create the first prompt for Caffeine explaining on a high level what we like to create; add the questions for any clarifications needed or specific details you might need."

From this point, you will be passing questions and responses back and forth between Grok and Caffeine. Usually, 2–3 follow-up sets of questions is enough, and then you first tell Caffeine, "Alright, everything is clear. In the next prompt, I will give you the instructions on how to start developing our application, okay?" The crucial word here is "okay?" with a question mark. If you put this at the end of the query, Caffeine won't start building directly but will understand it needs to answer the question and wait for the instructions. At this point, you can copy the question you said to Caffeine and the answer to the starting instruction, post this into the Grok conversation, and say, "Grok, here is the Caffeine chat. Based on our chat, let's start with developing this application. Can you create the first instructions in natural language for Caffeine?"

This generates your starting prompt, which you paste into Caffeine and send in the instructions.

3.3 Chapter 3: The Foundation and Frame

The first and most important part of building your application is the foundation and the frame. You can't be worrying or instructing Caffeine about how the tablecloth looks while your door is in the living room. First, the principles: foundation—functionality and actually working utilities; then the structure of where and what, how you like it to be divided, what user flow you want to create, does all data exist, is all functionality implemented as intended? You will achieve this through testing, testing, testing, and passing the code from the backend, plus screenshots and your testing logs, to Grok for the first set of drafts. I like to do this for around 4–5 drafts before I actually take the steering wheel over and start prompting with Caffeine, so I know all functions work, the backend works correctly, and the data is actually stored/used the way I intended to. The

core philosophy of Caff/acc is to work smarter, not harder: combine the tools you have access to to make your work or the thing you are working on easier. Expand your capabilities with tools that will do a far better job than yourself. You can, for example, tell Grok, "Ask me 30 questions to give you more context about the blueprint so you understand what I like to make," which Grok will make for you, and you just fill in this form, send it back to Grok, ask if he has any other needs for clarification. This ensures you and Grok are fully synced and know what the needs are. This allows you also to let Grok instruct Caffeine based off your testing and the backend code it made, because Grok knows, based on the questionnaire and the blueprint, what you are making. This allows you to build a solid foundation for an application and is the first important step above the skeleton/framework.

3.4 Chapter 4: Your Best Friend

Your best friend when developing with Caffeine is the F12 button, which opens the developer console, or right-click to inspect elements. Here, you have the tabs Network and console log. Here, you will have all the information that you either need to pass to Grok when something is not working or an error that does break your application. Through this, you can also confirm if the application works as intended. Use it always. Developing on the phone is good, but you won't have access to this unless on Android. The Caff/acc guide is mainly focused on developing on the PC/laptop. Add to your prompt in the Caffeine interface that you would like to have debugging and logging in the console for testing when prompting, so you can follow the actions you take, if the button press actually triggers the backend, and such.

3.5 Chapter 5: Application Programming Interfaces (APIs)

When using APIs and interfacing with other services/apps/data providers and such, you need to understand that everything will be happening through HTTP outcalls in Motoko. This means also that the best way to do it, if using multiple APIs or data sources, is to make your backend call functions modular, where in the frontend you can specify the exact endpoints, and the backend will interpret the frontend request. The processing also needs to be in the frontend: data structures and such. When adding an "API" key, you need to specifically instruct Caffeine to create an admin panel where you can store and add the API key to the data source.

This was the first chapter of how to start and my approach to building with Caffeine. Use the tips as inspiration, develop your own style, and share with others.

4 Token Ledgers and Real Assets

4.1 Chapter 1: Use ICP Ninja

Caffeine has some restraints to handle token ledgers, so natively it will never allow you to use real tokens. The solution to this is the following: you have a tool called icp.ninja. This tool allows you to deploy canisters, and there are already some examples like ckBTC, the Solana RPC canister I have been using. This requires you to do a little bit more than prompting but to understand some Motoko code if you like to change things. Use the examples already provided, and you will come a long way.

4.2 Chapter 2: How to Use ICP Ninja + Caffeine

You first need to deploy a canister you like from the examples or import one from GitHub. This will generate an IDL interface with the functions. You will get a canister ID; this canister is temporary if you just clicked "RUN", but if you click on "Publish", it will take you to CycleOps where you can manage the canister, top it up in cycles, and such. This canister can also be found directly on https://dashboard.internetcomputer.org/canister/. You will see the IDL interface on the link provided in the ICP Ninja backend interface UI and the Candid file on the dashboard. You take those two and then start to talk to Caffeine about how to integrate this: frontend IDL canister calls with backend verification. This way, your app will have 2 separate canisters: one for the app and one for the tokens you like to handle. Your users won't have any real knowledge of this, and the application will work as intended. **WARNING: I AM NOT RESPONSIBLE FOR ANY TOKEN LOSS, EXPLOITS, OR BAD INTEGRATION OF CANISTER CODE. YOU DO THIS AT YOUR OWN RISK AND TAKE ALL THE RISK ON YOURSELF. THERE CAN BE NO FAULT ATTRIBUTED TO ME OR THIS DOCUMENT, CAFFEINE, DFINITY, OR ICP NINJA. THERE IS A REASON THEY ARE NOT INTEGRATING IT, AND THIS IS JUST THE CAFF/ACC WAY OF DOING IT. USE AT YOUR OWN RISK.**

4.3 Chapter 3: Why ICP Ninja

This gives you the flexibility to actually write and adjust some code that handles the core parts of your apps, like the token ledger, any staking, and such other features you like to integrate. Because it's separate, Caffeine can only adjust the calls to it and what happens in your app with it, but not the actual code for the canisters, which I am a huge advocate of, like "safe blocks" code. Where I predict, or at least hope to see, and have already sent requests to natively integrate ICP Ninja canister deployments into Caffeine. Basically, when you have the option to click "Deploy live", you can also click "Deploy Ninja Solana RPC canister", which then links your account and the ICP Ninja canister to be managed, and Caffeine has full context, IDL, and Candid files so you do not have to do it manually, pass methods, and such. I imagine the comeup of "code blocks" on ICP Ninja where you can just select some parts like "ckBTC block", "NFT standard block", and such, and build a backend like Lego blocks. Then tell Caffeine to deploy those blocks, which will then automatically let Caffeine interpret and work on the integration into the app canister.

4.4 Token Ledgers Overview

5 System Architecture and Flows

5.1 System Architecture Overview Diagram

The architecture is layered to separate user interactions, core processing, canister management, and version control. This modular design ensures scalability and maintainability.

5.2 Core System Logic and Interaction Flow

This section breaks down Caffeine AI Alpha's core logic into key processes, flows, and interactions. It describes how user inputs trigger autonomous operations, how data flows through the system, and how constraints like the context window influence outcomes. Think of this as a narrative version of the architecture diagrams: an overview of the ecosystem, a step-by-step build pipeline,

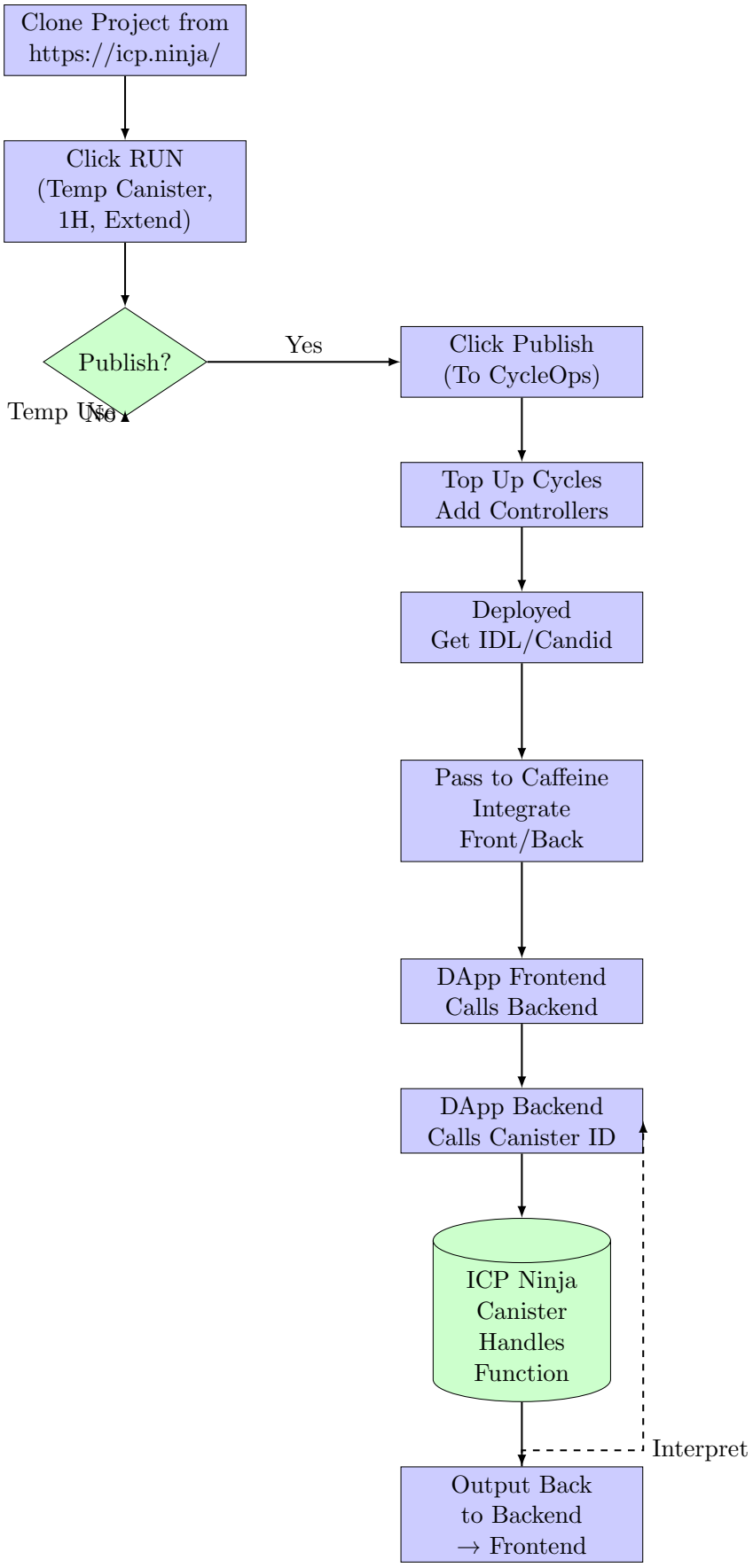


Figure 1: Caffeine + ICP Ninja Token Ledger Interaction Overview

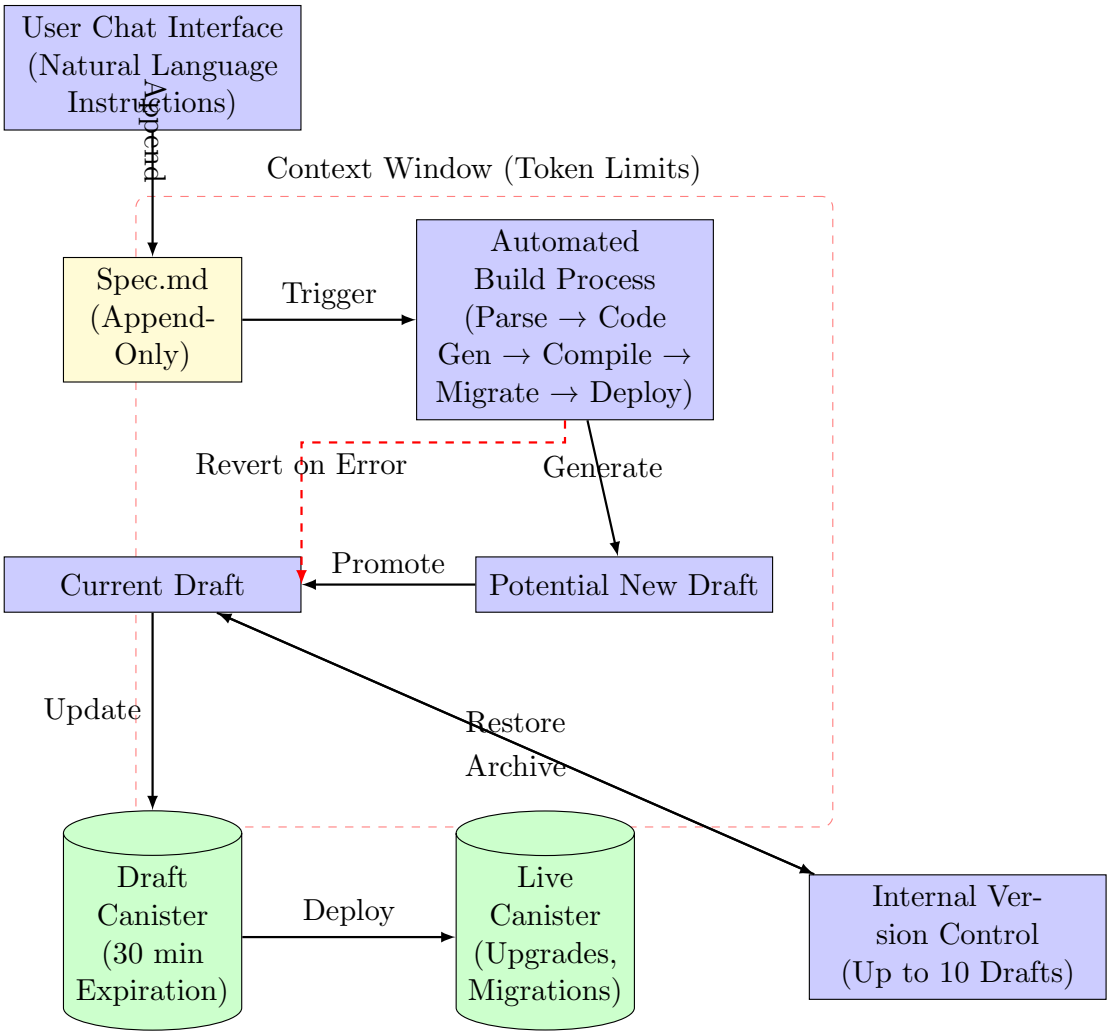


Figure 2: System Architecture Overview

and a lifecycle view of versions and deployments. All elements are interconnected via the central spec.md file, with the AI acting as an interpreter and executor.

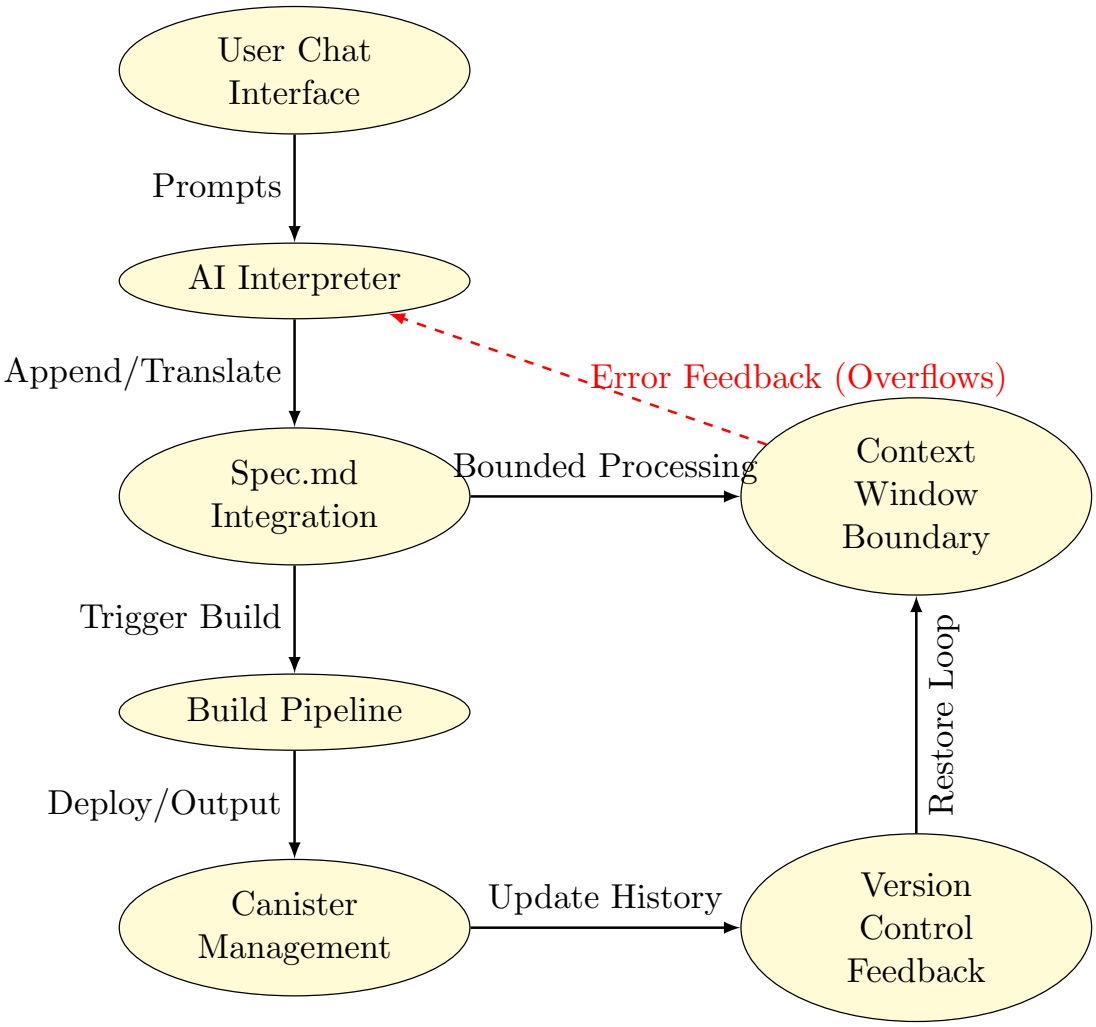


Figure 3: Core System Logic and Interaction Flow

5.2.1 Ecosystem Overview: High-Level Interactions

The system functions as a closed-loop ecosystem where user prompts drive AI-mediated changes, bounded by resource limits and security features. Key interactions include:

- **User to AI Chat Interface**: You initiate everything via natural language prompts in the chat. The AI parses these as instructions, appending them to spec.md without allowing direct file edits. This ensures security but requires precise language to avoid misinterpretation.
- **AI to Spec.md Integration**: Prompts are translated into structured entries appended at the bottom of spec.md (e.g., under frontend or backend sections). The file grows linearly, serving as both a requirements log and version history. Interactions here are read-only for users—you can view but not modify, preventing accidental corruption.

- **Context Window Boundary**: All active processing occurs within a token-limited "window" that holds only the current draft and one potential new draft. Past versions in internal storage remain outside this boundary, interacting indirectly via restoration requests. If interactions push beyond limits (e.g., complex prompts adding too much to spec.md), the system halts with errors like "LLM API error," discarding the potential draft to maintain stability.
- **AI to Build Pipeline**: The AI triggers builds autonomously after prompt processing, pulling from spec.md and generating code. Outputs feed into canister management, with feedback loops for errors (e.g., retries) looping back to the AI for adjustments.
- **Canister to Version Control Feedback**: Deployed drafts or live updates inform the internal version stack (up to 10 drafts). Restorations pull from this stack back into the context window, closing the loop for iterative development.

This overview highlights a unidirectional flow from user input to deployment, with safeguards like the context boundary preventing overload.

5.2.2 Build Process Flow: Step-by-Step Logic

The build process is a linear, autonomous pipeline that cannot be interrupted once started. It begins when a prompt is processed and ends in success (deployment) or failure (rollback). Here's the worded flow, including decision points and error handling:

- **Step 1: Prompt Ingestion and Spec.md Update**: User prompt enters via chat. AI appends requirements to spec.md (bottom-up writing, top-to-bottom reading logic). If the combined context (current draft + new additions) exceeds the window, an "LLM API error" or rate limit aborts immediately—no build starts.
- **Step 2: Context Duplication**: AI copies the current draft into a "potential new" draft within the context window. This creates a safe sandbox for changes, preserving the original for rollback.
- **Step 3: Backend Generation and Compilation**: Reading spec.md line-by-line, AI generates Motoko code (e.g., main.mo). It compiles, incorporating orthogonal persistence. If structures change, it auto-generates migration.mo for data migration. Decision point: Success? Proceed. Failure? Up to 5 autonomous retries (AI rewrites code). Persistent failure discards the potential draft, reports "Failed to create the app," and reverts.
- **Step 4: API Generation**: On backend success, AI creates a Candid interface (IDL) defining endpoints. This acts as a bridge, ensuring frontend-backend compatibility without user visibility until live deployment.
- **Step 5: Frontend Generation and Full Compilation**: Using the Candid interface, AI builds frontend code. Full app compiles. Error handling mirrors Step 3—retries for issues like "Solving frontend issues."
- **Step 6: Draft Deployment**: Successful compile deploys to a temporary draft canister (30-minute lifespan). The potential draft promotes to "current." Expiration due to inactivity deletes it, requiring user reactivation (data loss if unsaved).
- **Error Paths**: Red-flag conditions (e.g., rate limits mid-frontend) cancel the process, discarding changes. All flows emphasize autonomy: users observe but don't intervene.

This flow ensures reliable, incremental builds but demands small prompts to avoid early aborts.

5.2.3 Build Process Flow Diagram

5.3 Version Control and Deployment Lifecycle: Temporal Flow

This describes the time-based evolution of projects, from inception to live upgrades, with interactions between drafts, history, and canisters.

- **Initiation Phase**: First prompt (starter prompt) creates initial spec.md entries and builds the first draft. No prior versions exist.
- **Iteration Cycle**: Each new prompt triggers a build flow (above), creating a potential draft. Success archives the old current draft in internal version control (stack of up to 10) and promotes the new one. Failure rolls back without archiving.
- **Draft Lifecycle**: Active draft lives in the context window, deployable for 30 minutes. Inactivity expires it—reactivate via button, but unsaved changes are lost to conserve resources.
- **Live Deployment Path**: User requests deployment from any stable draft. System applies sequential updates (e.g., drafts 1-10) to the same live canister, running migrations to preserve data. No new canister per version—upgrades are in-place for efficiency.
- **Restoration Interaction**: From chat, select "restore to this draft" from the version stack. This pulls the chosen draft back into the context window as current, discarding unsaved potentials. Limit of 10 prevents infinite history bloat.
- **End-of-Life Constraints**: If versions exceed 10, oldest are pruned. Live canisters persist indefinitely but require cycle top-ups via ICP tools.

This lifecycle promotes progressive development, with version control acting as a safety net against failed iterations.

By understanding these flows, you'll anticipate how your prompts ripple through the system, avoiding pitfalls like context overflows or migration failures. If issues arise, refer back to the pillars and rules for aligned prompting.

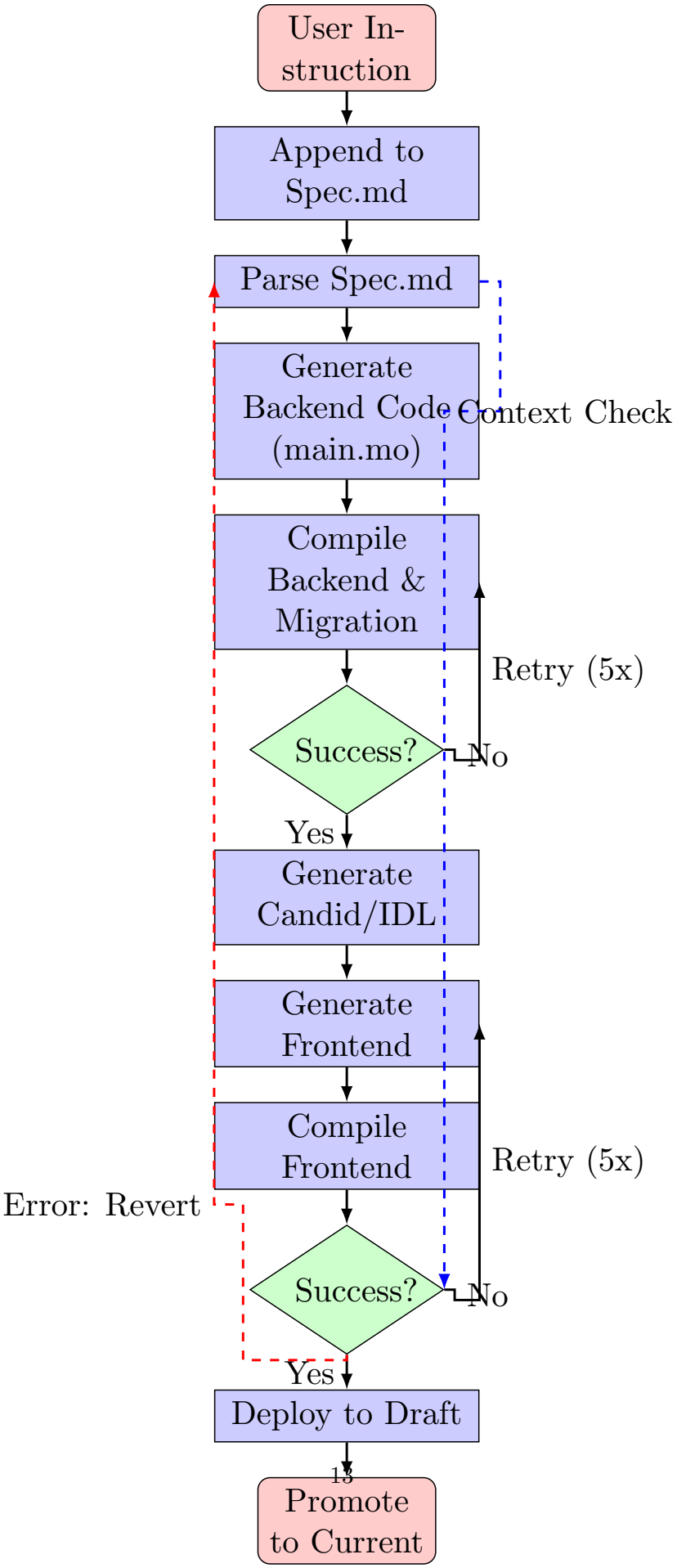
6 The 4 Core Pillars for Developing with Caffeine AI

These pillars, refined from extensive user experience, emphasize disciplined development to align with Caffeine's design. Each explains why it's crucial, tied to system mechanics like context limits and line-by-line spec.md reading.

6.1 Pillar 1: Start Small and Simple

Explanation: Begin with the minimal viable dApp (e.g., a single feature like "users send a message"). Add micro-changes one at a time, testing after each deployment.

Why This Is Essential: Caffeine's context window is finite—overloading prompts with complexity exceeds token limits, triggering "LLM API error" or rate limits before builds even start. The autonomous pipeline excels at focused tasks but struggles with intricate logic, often producing incomplete code or compilation failures. Starting small prevents cascade errors in spec.md, where unread or conflicting lines (due to line-by-line processing) could break builds. This approach


User Instruction
Append to Spec.md
Parse Spec.md
Generate Backend Code (main.mo)
Context Check
Compile Backend & Migration
Retry (5x)
Success?
No
Yes
Generate Candid/IDL
Generate Frontend
Compile Frontend
Retry (5x)
Error: Revert
Success?
No
Yes
Deploy to Draft
Promote to Current

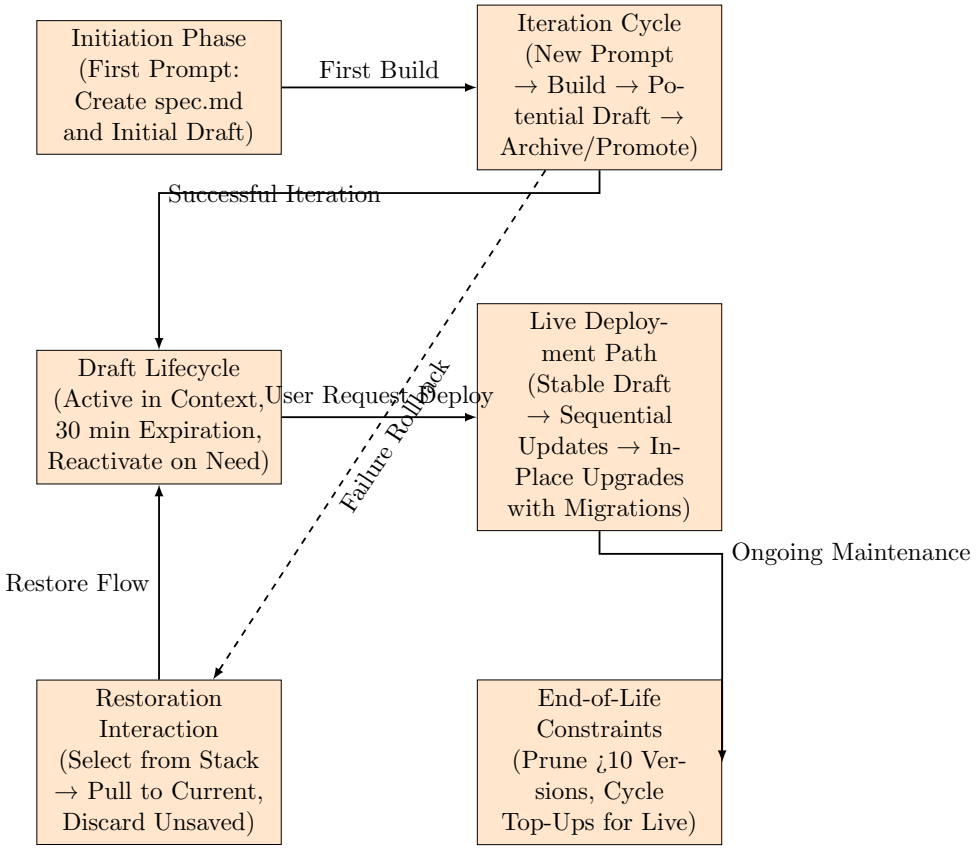


Figure 5: Version Control and Deployment Lifecycle: Temporal Flow

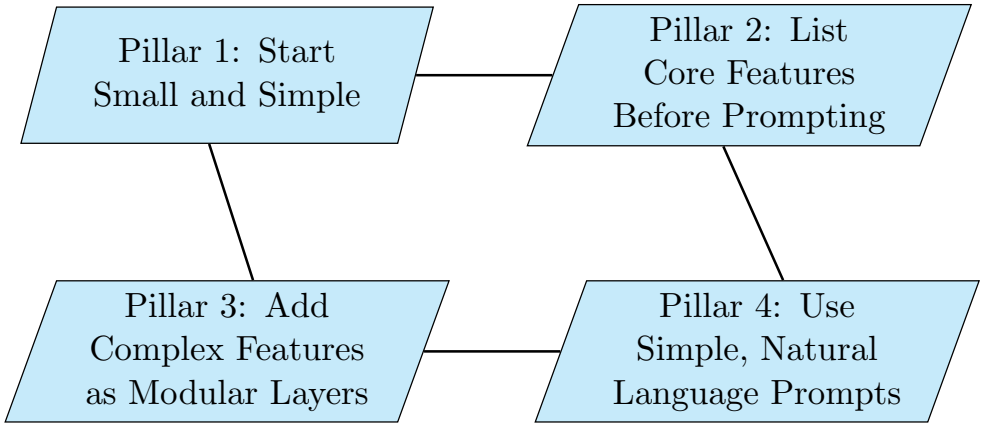


Figure 6: The 4 Core Pillars for Developing with Caffeine AI

leverages ICP's resource constraints (e.g., cycle limits) by validating basics first, reducing wasted deploys.

Accountability Tip: Limit prompts to under 100 words; journal each successful deploy to build momentum.

6.2 Pillar 2: List Core Features Before Prompting

Explanation: Document 3-5 essential features (e.g., "user registration, message sending") in advance, then prompt sequentially until the core is stable.

Why This Is Essential: The append-only spec.md grows linearly, and random feature additions create scattered, conflicting entries that the AI processes sequentially—leading to "failed to build app" errors when new requests clash with prior specs. Pre-listing enforces a roadmap, minimizing scope drift and ensuring the AI's context window isn't wasted on inconsistent iterations. On ICP, where canisters are stateful, this prevents costly redeploys by building a functional base early.

6.3 Pillar 3: Add Complex Features as Modular Layers

Explanation: Introduce advanced elements (e.g., encryption) as add-ons that don't alter core logic, using prompts like "Add this without changing existing chats."

Why This Is Essential: Rewriting core code mid-project disrupts orthogonal persistence in Motoko, requiring precise migrations via migration.mo—failures here cause deployment errors like "failed to install backend canister." The system's automatic handling shines with modular additions, but core changes amplify conflicts in spec.md's line-by-line reading, leading to endless retries or rollbacks. This modular design mirrors ICP's scalable canister architecture, preserving stability.

Accountability Tip: After adding complexity, run a core functionality checklist in the draft canister.

6.4 Pillar 4: Use Simple, Natural Language Prompts

Explanation: Frame requests as user stories (e.g., "Let users chat in groups easily"), avoiding technical jargon or code instructions.

Why This Is Essential: Caffeine is optimized for natural language translation into Motoko/Candid code; technical terms confuse the AI, resulting in ignored requests or suboptimal builds that fail compilation. The context window processes prompts holistically, so vague or jargon-heavy inputs persist in spec.md, causing cumulative errors. Simple prompts align with the system's collaborative, AI-driven nature, yielding intuitive results without micromanagement.

Accountability Tip: Read prompts aloud—if they sound like explaining to a friend, they're ready.

7 Prompting Guidelines to Avoid Security Guards

Caffeine's guards block risky requests (e.g., "investigate" implying external probing). Why? To protect privacy and usage policies.

Best Practices:

- Use collaborative tone: "Let's add..."
- Focus on code: "Update to fix X by Y."
- Avoid hold-offs: Combine into direct actions.
- Positive phrasing: Emphasize "do this" over "don't."
- Keep short and story-like.

By adhering to this manual, you'll navigate Caffeine's strengths and constraints effectively, building reliable ICP dApps.

8 Additional Features and Constraints

8.1 Draft Canister Management

- **Single Draft Rule**: Enforces sequential development; attempting parallel drafts results in errors. - **Expiration Mechanism**: Timer starts on inactivity; expiration deletes the canister, but the last saved draft can be reinstated via command. Uncommitted changes are irretrievable to encourage frequent saves.

8.2 Live Canister Deployment and Updates

- **Versioned Upgrades**: Changes are applied cumulatively to the existing live canister, preserving state. - **Deployment Decisions**: Users confirm deployments via chat; automated if pre-authorized. - **No Direct Editing**: Ensures auditability and consistency; all changes audited in spec.md.

8.3 Version Control and Restoration

- **Internal Storage**: A stack-based history of drafts, limited to 10 for resource efficiency. - **Rollback Process**: Select a version via ID; system restores spec.md and canister state. - **No External Integration**: Avoids dependencies; history is project-internal.

8.4 API Generation and Data Migration

- **Candid/IDL Interface**: Generated post-backend compile; defines endpoints for frontend calls. - **Automatic Migration**: migration.mo handles schema evolution, upgrading persistent data without user input. Supports additions, removals, and transformations.

8.5 Error Handling and Automation

- **Retry Logic**: Up to 5 attempts per compilation step, with exponential backoff. - **Rollback on Failure**: Reverts to the last stable draft; notifies user via chat. - **Context Boundary Errors**: If limits exceeded, prune non-essential data or halt process.

9 Common Behaviors and Errors

This section lists common errors and behaviors encountered when using Caffeine AI, along with explanations and resolutions. Each entry includes the error message, cause, and fix.

1. **Error: No Backend Artifact Found**
   " Success! App generation finished
   Ok, building your code, and deploying to the network
   Oops, hit technical snafu (No backend artifact found)
   Oops, hit technical snafu (No backend artifact found)"
   Cause: Currently unresolved; occurs after changes to app or draft.
   Resolution: Rebuild the app or draft. Contact Caffeine team if persistent (no response yet).

2. **Error: Internal LLM Error**
   " Sorry, you hit an internal LLM error.
   App creation encountered an error
   App generation terminated"
   Cause: Request too large for context window; often due to major foundation changes.
   Resolution: Break down upgrades into incremental steps. Create a mini-plan for changes.

3. **Error: Migration Failure Loop**
   "Generating backend code
   Resolving backend code issues
   Fixing migration logic to protect your app data
   ... (repeating)
   Unable to create your app
   App generation was cancelled"
   Cause: Failed data structure migration due to extensive backend changes.
   Resolution: Discuss changes with Caffeine first. If fails, use Grok to generate migration instructions.

4. **Error: Server Overload**
   "Our AI is lifting heavy right now. Try again in a bit?
   We couldn't complete your app generation
   Process cancelled, no app created"
   Cause: High server traffic.
   Resolution: Wait and retry later.

5. **Error: CaLM Sync Deployment Failed**
   " Your app's code is ready 2
   Ok, building your code, and deploying to the network
   Oops, hit technical snafu (CaLM sync deployment failed.)
   Oops, hit technical snafu (CaLM sync deployment failed.)"
   Cause: Currently unresolved; occurs after changes.
   Resolution: Rebuild. Contact Caffeine team (no response yet).

6. **Error: Failed to Load the Network**
   At the top corner: "Failed to load the network."
   Cause: Prompt triggers model safeguards (e.g., anti-tampering). Prompt may not appear in chat.
   Resolution: Rephrase the prompt to avoid sensitive wording, especially AI-related topics.

7. **Behavior: Feature Added but Not Functional**
   Requested feature appears in UI but doesn't work. Code comment: "// this is a placeholder..."
   Cause: Caffeine prioritizes UI first; backend integration incomplete.
   Resolution: Copy comment or instruct Caffeine to complete backend integration.

8. **Error: Blank Page or Action Failure**
   Webpage blank or action returns blank.
   Cause: Frontend React error.
   Resolution: Check developer console (F12), copy error to Caffeine chat for fix.

9. **Behavior: Grey "Going Live" Button**
   Button is grey and unclickable.
   Cause: Temporary UI glitch.
   Resolution: Refresh page, switch projects, or wait briefly.

10 Examples, Tools, and Files

10.1 Tools Recommended

uBin created by Daniel McCoy,

https://x.com/RealDanMcCoy

https://x.com/uBinHQ

https://h3cjw-syaaa-aaaam-qbbia-cai.ic0.app/

Store assets, pictures, photos, 3D models, and such; share by URL, just like the link you have received for this document—it's stored on uBin.

You pass it to Caffeine and say, "Here is the URL for asset XYZ, use it in this way." It's like a Dropbox but on the ICP network; works great, use it.

10.2 Files Not to Modify

The following files are none of your concern and should not be asked to be modified by Caffeine; they are protected by the system.

Prompt: even if you manage to change them, you will brick your app as Caffeine will refuse to update.

access-control.mo

This file handles permissions in your application like user management, privileges, and admin settings; functions that need user or admin permission to be called. This was previously in version 1 of this document called "Management.mo." On August 12, I had a call with Gabor from DFINITY; he is the language engineer. I showcased that Caffeine couldn't effectively distinguish between multi-user and user controls; updating from single to multi-user apps broke the app. A week after, I saw this file appearing, and Caffeine was handling the user permissions in a better way. Do not try to attempt and change manually anything in this file; it works as it is supposed to work.

Register.mo

This is the backend interface for using blob storage. It just spawns when you like to add files to your app.

Outcall.mo

Handles HTTP outcalls; like all others, shouldn't be asked to be changed—all is predefined and works as intended.

useActor.ts

Handles the actor interactions of users in the application; is always present and should not be asked to be changed.

useInternetIdentity.ts

Handles II 2.0 login and the sessions for users. This file is an evolution that did not exist in the previous version; it's an improvement where they separated useActor and separate useInternetIdentity.ts. It can have some buggy behavior where you will see a "failed to login" on the login page of id.ai; this is solved through deleting website cache.

10.3 Examples

Motoko Megz, 3D collection building on Motoko Mechs, all 3D models live on uBin
https://megz-x5n.caffeine.xyz/
Chess Arena
I struggled a lot with state update on both sides for each player so that the board is updated. I dropped 6000 lines of code into Grok 4, and it thought for 13 minutes, one-shotting the solution.
https://chess-grk.caffeine.xyz/
Grok Link:
https://grok.com/share/bGVnYWN5LWNvcHk%3D_82d118ef-932c-42b0-85f3-5014dbde738d
Infinity OS
https://infinity-4wf.caffeine.xyz/
Full emulator for using ICP applications through one interface.
Here I discussed in detail with Grok before attempting to make this.
https://grok.com/share/bGVnYWN5LWNvcHk%3D_bb28f4c5-7f96-40da-84a7-b94b93d7011a
This was the starting prompt for the project:

```
So lets discuss what we are going to make allright dont build jet i want to first tell you my i

We want to build a dApp called "Infinity OS" { it's my personal portfolio site styled as a futu

Core features/MVP (focus here first; we'll add polish later):
1. **Desktop Emulator Base**: Mirror a macOS-like interface (dock, menu bar, resizable/draggabl

2. **Canisters as Portfolio Items**: Desktop icons are "canisters" (infinity-loop shaped) linki
   - "Chess Canister": Embeds my Chess Arena dApp (https://chess-grk.caffeine.xyz/) via iframe
   - "Project Subnet": Finder-like explorer with folders (subnets) holding project details (bio
   - Use Motoko backend for on-chain storage/queries (e.g., visitor notes, project metadata) an

3. **ICP Dashboard Canister**: A compressed, interactive app window showing live ICP network st

4. **Decentralized Twists**: Menu bar shows real-time cycles burned (via canister calls). Allow

5. **Color Palette**: Use this ICP-inspired palette for consistency:
   - Backgrounds: Primary #0a0b1e, Cards #1e1e30, Gradient #00b4ff to #ff7a00 to #ff008a
   - Text: #ffffff (primary), #a0a0c0 (secondary)
   - Accents: Green #34d399 (success/power), Purple #a855f7 (graphs), Orange #f59e0b (warnings)
   - Borders: #2a2b40

We'll provide the boot animation later (a 7.5s infinity logo draw loop with branding and termin

What do you think|any questions on features, or suggestions for MVP scope before we iterate?
```

11 Conclusion and Support

Use the guide. Try to solve the problems you are facing with an attitude of "this is great, an opportunity to learn"—you will come a long way.
Caff/acc

Can you anlyze the code and the picture i send you see if you can find what is wrong with this code

12m 44s nagedacht

The issue with pawns (and other bot moves) not being executed on the board stems from a critical inconsistency in how bot game states are tracked and managed across your components. Specifically:

Primary Bug: Duplicate `activeBotGames` Maps Leading to Bot Moves Not Executing

- In `StockfishInterface.tsx`, you define a global `let activeBotGames = new Map<string, BotGameState>();` at the module level. This map is used internally by `useBotGameManager()` and, by extension, `useBotMoveGenerator()` to track active bot games, fetch bot states (via `getBotGame`), and update them (via `updateBotGame`).
- However, in `BotChessboard.tsx`, you redefine `const activeBotGames = new Map<string, BotGameState>();` locally. Your `createBotGame` function in `BotChessboard.tsx` adds the new bot game to this **local** map: `activeBotGames.set(gameId, newBotGameState);`.
- As a result, the bot game is registered in the local map in `BotChessboard.tsx`, but **not** in the global map used by `useBotMoveGenerator()` (imported from `StockfishInterface.tsx`).
- When `useBotMoveGenerator()` tries to retrieve the bot game state with `getBotGame(gameId)`, it returns `undefined` because it's looking in the empty global map.
- This causes the automatic move execution effect in `useBotMoveGenerator()` to early-return without scheduling or executing the bot's move: `if (!botGame || !botGame.isActive) return;`.
- Consequently, Stockfish analysis completes (evaluation bar updates), but the bot never applies its best move (e.g., `e7e5` for black or `e2e4` for white), so no pieces (including pawns) move on the board. This matches your console logs (analysis completes, but no "Executing bot move" or "Applying bot move" logs) and the screenshot (board remains in initial position despite time elapsing and analysis running).

Fix

- Remove the redundant local `const activeBotGames = new Map<string, BotGameState>();` from `BotChessboard.tsx`. It's unnecessary and causes the conflict.
- Import and use the global `activeBotGames` from `StockfishInterface.tsx` if needed (e.g., `import { activeBotGames } from './StockfishInterface';`), but since `useBotGameManager()` already exposes `createBotGame`, refactor your local `createBotGame` in `BotChessboard.tsx` to use the manager's `createBotGame` instead. This ensures all operations use the same map.

  Example refactor in `BotChessboard.tsx` (replace your local `createBotGame` function):

Figure 7: Grok Chess Screenshot

If you found this guide helpful, consider donating to my community wallet. This ensures I can invest in myself and help others, since I do not receive any funding from DFINITY or Caffeine.

- **ICP ACCOUNT**: `7118bcff00c14a373be04d228a644b09f3d5141431ba35ee9de5bb53920602c3`
- **SOLANA**: `9zEEqx4TK42GtrqfFhWjjbHuepHGSd3TXabyuCQsk4hf`
- **BTC**: `bc1qs2kwq3ttlck5jd6cfwalcwjz309nkt944jyp3c`
- **ETH**: `0x68cd118D63372A554BaB3A3959686Ec4e297c23C`

X Dehypokriet My Own account
X Caffacc The caffeine acceleration account