# **Vibe Coding Playbook: AI Project with Roo Code, Coda Pack, DigitalOcean & TablePlus**

Building a software project with AI assistants can be fast and enjoyable – **if** you apply structure and best practices. This playbook walks through each phase of a project using **Roo Code** (VS Code AI extension), a **DigitalOcean** backend, **TablePlus** for the database, and a **Coda Pack** for front-end integration. We’ll highlight how to leverage Roo’s agents (Ask, Architect, Debug, Test, etc.), incorporate community tips, and avoid pitfalls using real examples from our project history.

**Note:** This guide follows the vibe coding philosophy[Google DriveGoogle Drive](https://drive.google.com/file/d/1AZWFTjk-7geKnGZ-_d0sXK2mFN8SqFG0) – *plan before you prompt*, work iteratively, use rules/guardrails, and roll back changes when needed. Treat your AI agents as a dev team: give them direction and review their work continuously.

## **Planning Phase: Define Vision & Architecture**

Start by clearly outlining what you’re building and how it should work. As the vibe coding mantra goes, *“if you skip the plan, your AI will skip the point.”* A thorough plan aligns your AI assistants with your intent[Google Drive](https://drive.google.com/file/d/1AZWFTjk-7geKnGZ-_d0sXK2mFN8SqFG0).

* **Use Roo’s Architect Agent for high-level design:** In Roo Code, **Architect Mode** is ideal for planning and technical strategy. Begin a new task in Architect Mode to draft a Product Requirements Document (PRD) or project plan. Describe features, user flows, and constraints in natural language. The Architect agent will often produce an implementation plan (e.g. creating a Plan.md or implementationPlan file) that future tasks can reference[spin.atomicobject.com](https://spin.atomicobject.com/roo-code-ai-assisted-development/#:~:text=Always%20Start%20in%20Architect%20Mode)[spin.atomicobject.com](https://spin.atomicobject.com/roo-code-ai-assisted-development/#:~:text=I%20want%20to%20architect%20a,app%2C%20along%20side%20domain%20types).
* **Break down the project:** Ask the Architect agent (or an Ask agent) to generate a structured to-do list from your PRD. For example, prompt: *“List all major tasks by feature, with dependencies and difficulty.”* This helps identify phases like *Backend API*, *Coda Pack Integration*, etc., and ensures the AI team knows the roadmap[Google DriveGoogle Drive](https://drive.google.com/file/d/1AZWFTjk-7geKnGZ-_d0sXK2mFN8SqFG0). Group tasks logically (database setup, auth implementation, etc.) so you can tackle them one by one.
* **Leverage Gemini for coordination (optional):** If you have access to **Gemini** (Google’s advanced AI), consider using it as an *AI project manager*. Gemini excels at big-picture reasoning. You can have it review your plan or verify assumptions made by other agents. For instance, our team consulted a Gemini assistant to double-check the deployment strategy; it provided a solid outline but misunderstood some specifics, which we then corrected[Google Drive](https://docs.google.com/document/d/1crnNwXIbJVDGfO8bjZRACqL9M87a56xcBOfynvm38ws).
* **Tip:** When using a powerful model like Gemini for oversight, feed it your Architect agent’s plan and any unique requirements, then ask it to confirm understanding. This can catch conceptual errors early, as we discovered when Gemini initially proposed an incorrect auth flow that we had to fix[Google Drive](https://docs.google.com/document/d/1crnNwXIbJVDGfO8bjZRACqL9M87a56xcBOfynvm38ws).
* **Define rules and constraints:** Before coding, set ground rules for your AI agents. Roo Code allows **Custom Instructions** or rules (similar to vibe coding “internal engineering culture”). Define things like “Use the existing coding style,” “Don’t expose secrets,” or “Prefer simple, modular solutions.” These act like guardrails so that generated code aligns with best practices[Google DriveGoogle Drive](https://drive.google.com/file/d/1AZWFTjk-7geKnGZ-_d0sXK2mFN8SqFG0). In Roo, you might add such rules in a .roo config or via the extension’s settings. For example:  
  + *“Always restart the dev server after code changes.”*
  + *“Don’t duplicate code – reuse functions if available.”*
  + *“Follow Node.js and React conventions used in this project.”* These guidelines help the AI make consistent decisions and avoid common pitfalls (e.g. rewriting large sections unnecessarily[Google Drive](https://drive.google.com/file/d/1AZWFTjk-7geKnGZ-_d0sXK2mFN8SqFG0)).
* **Plan iteratively:** Keep the planning scope focused. It’s tempting to have the AI outline everything at once, but too broad a plan can overwhelm the context. Our Architect agent’s first plan covered the entire data layer **and** UI wiring in one go – too much detail to execute in one step[spin.atomicobject.com](https://spin.atomicobject.com/roo-code-ai-assisted-development/#:~:text=Keep%20The%20Scope%20Of%20Each,Task%20Small). We refined the prompt to limit scope (just design the domain data types first), which produced a cleaner, more manageable plan[spin.atomicobject.com](https://spin.atomicobject.com/roo-code-ai-assisted-development/#:~:text=,Possible)[spin.atomicobject.com](https://spin.atomicobject.com/roo-code-ai-assisted-development/#:~:text=The%20plan%20that%20Roo%20created,created%20filter%20and%20sort%20types). *Be as specific as possible:* tell the agent exactly what to plan and what *not* to plan yet. If the plan still isn’t right, you can correct the agent or start a fresh planning task with clearer instructions[spin.atomicobject.com](https://spin.atomicobject.com/roo-code-ai-assisted-development/#:~:text=sort%20types)[spin.atomicobject.com](https://spin.atomicobject.com/roo-code-ai-assisted-development/#:~:text=Switch%20To%20Code%20Mode).

**Example:** In our project, the Architect agent produced a plan for a *Coda-to-App Data Sync* system, outlining phases from backend refactoring to Coda Pack setup[Google DriveGoogle Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU). This became our blueprint. We also wrote a short user story and listed key features (e.g. *“user can push a button in Coda to send data to backend”*, *“tasks sync from backend to Coda table”*). Having this clarity up front meant that when we prompted Roo to implement features, it “knew” the context and purpose. As Matthew Berman’s Vibe Coding Playbook says, *a clear plan is your anchor – when things get off track, refer back and course-correct*[Google DriveGoogle Drive](https://drive.google.com/file/d/1AZWFTjk-7geKnGZ-_d0sXK2mFN8SqFG0).

## **Setup Phase: Environment, Repos, and Tools**

With a solid plan in hand, set up your project environment. This phase is about creating a clean foundation for development.

* **Repository and workspace:** Initialize a new Git repository for the project (if you haven’t already). Use meaningful structure: perhaps separate folders for backend, coda-pack, and any frontend if applicable. Roo’s Ask agent can assist if you need quick answers on git commands or project structure. For example, you can ask: *“How should I structure a monorepo with a Node backend and a Coda Pack?”* But keep it simple – clarity matters more than perfection at this stage.
* **Version control from day one:** As you set up, run git init and make the first commit (Gemini recommended an empty project scaffold and not even have a README). This gives you a safety net. Throughout development, commit often. If an AI agent introduces breaking changes, you can easily roll back via Git or Roo’s built-in checkpoints[Google Drive](https://drive.google.com/file/d/1AZWFTjk-7geKnGZ-_d0sXK2mFN8SqFG0)[spin.atomicobject.com](https://spin.atomicobject.com/roo-code-ai-assisted-development/#:~:text=Use%20Checkpoints%20to%20Course%20Correct). Roo Code automatically creates **checkpoints** at key moments (task start, file edits, command executions)[spin.atomicobject.com](https://spin.atomicobject.com/roo-code-ai-assisted-development/#:~:text=There%20have%20been%20times%20when,a%20checkpoint%20is%20made%20whenever). These let you review diffs and restore the project to a previous state with one click[spin.atomicobject.com](https://spin.atomicobject.com/roo-code-ai-assisted-development/#:~:text=Say%20for%20instance%2C%20Roo%20was,Task%E2%80%9D%20in%20this%20case). Use them!
* **Tip:** Don’t hesitate to revert when things go sideways – *“roll back without drama”* is a vibe coding mantra[Google Drive](https://drive.google.com/file/d/1AZWFTjk-7geKnGZ-_d0sXK2mFN8SqFG0). Better to undo and try a new approach than to fight tangled, AI-generated code.
* **Set up the backend skeleton:** You can have Roo Code scaffold a basic Express.js app. For example, in **Code Mode**, prompt: *“Create a Node.js Express server with an index.js (or server.js) listening on port 3000”*. Roo will generate the boilerplate. Approve the creation of files like package.json and server.js. Be sure to include any standard setup (CORS if needed, body parser middleware, etc.). This boilerplate will be expanded later. Save a checkpoint after the basic server runs.
* **Initialize the Coda Pack project:** Use the Coda CLI to create your pack structure. This might involve using the terminal. Roo’s **Ask agent** can guide you through the CLI steps, but you’ll execute them either manually or by instructing Roo’s terminal tool. The typical steps:  
  + **Authenticate the CLI:** Run npx coda register to log in and store your Coda API key (you’ll be prompted for an API token, which you can create in your Coda account settings).
  + **Create a new Pack:** Run npx coda create in the coda-pack directory. This registers a new Pack on Coda’s servers and returns a Pack ID.
  + **Link the project:** After creating, run npx coda link <PackID> to link your local folder to that Coda Pack ID. (If you ever reinstall dependencies or copy the project, you might need to re-link. In our case, an npm install reset the config, so we relinked to avoid confusion[Google Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU).)
  + **Install Pack SDK:** Ensure you have the latest @codahq/packs-sdk in your package.json. For example, we updated to a modern version so that features like context.document.id in formulas would be supported[Google Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU).
* **Project consolidation:** A common real-world issue is having multiple project copies or old code hanging around. We faced this early on – development had forked into a duplicate folder, causing sync issues and confusion[Google Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU).
* **Warning:** Always work in a single source of truth directory. If you have stray old project folders, consider archiving or deleting them to avoid accidentally editing the wrong files. We performed a “final consolidation” by merging the correct code into one folder and removing the duplicate, then updated VS Code and Git to point to the right path[Google Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU).
* **Example – Coda Pack configuration snafu:** Initially, our Pack wouldn’t upload or show any formulas in Coda. The culprit was misconfigured CLI setup. We resolved it by systematically doing all the CLI steps in order. Specifically, we:  
  + Registered the Coda API token with coda register.
  + Created the pack (getting a new Pack ID) with coda create.
  + Linked the local pack project to that ID with coda link (since a fresh npm install had cleared the link).
  + Bumped the packs-sdk version to the latest.  
     After this, the pack’s building blocks appeared correctly in the Coda doc[Google Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU).
  + **Tip:** If your pack formulas aren’t showing up in a doc, double-check these CLI steps and ensure your pack’s manifest.ts (or pack.ts) has a valid pack.addNetworkDomain and authentication set (more on that later).
* **Database setup with TablePlus:** Since our project uses a PostgreSQL database, we prepare that early. In TablePlus (or any DB GUI), connect to your DigitalOcean PostgreSQL instance. Create a new database (we named ours aiappcoda\_db) to serve as the project’s data store[Google Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU). Set up any necessary credentials or network permissions (on DigitalOcean, add your app’s IP or use “trusted sources” so the app can reach the DB). At this stage, you might not create tables yet – but it’s a good idea to have the empty DB ready and note the connection URL. We’ll create the schema during the Backend phase.
* **Gemini & AI credentials:** If you plan to use external AI models (like OpenAI, or Gemini via an API) within your project, secure those keys now. For example, if Gemini will be used in your app’s logic, get the API key and store it in your environment (e.g. a GEMINI\_API\_KEY env var).
* **Warning:** *Never* hardcode API keys in code or in prompts[Google Drive](https://drive.google.com/file/d/1AZWFTjk-7geKnGZ-_d0sXK2mFN8SqFG0). Always use environment variables or a secure secret manager. In Roo Code, you can add environment variables to the VS Code terminal environment so that any run commands have them available.

## **Backend Implementation Phase: Building the API and Database**

With the groundwork laid, it’s time to implement the backend – the server logic and database integration. Roo Code’s **Code Mode** will be your main workhorse here, with **Debug** and **Ask** agents assisting as needed.

* **Design the API endpoints:** Review the plan for what backend endpoints are needed. In our project, we identified endpoints like GET /api/me (to fetch user info for authentication), GET /api/tasks and GET /api/responses (to read data, filtered by doc ID), and POST /api/tasks and POST /api/responses (to receive new tasks/responses from the front-end or Coda Pack)[Google DriveGoogle Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU). You might also have a special endpoint like POST /api/app\_content if the Coda Pack pushes a batch of data. Start implementing these one by one.

**Use Roo to scaffold routes:** In Code Mode, you can say: *“Implement a POST /tasks endpoint that inserts a new task into the database.”* Given the context (especially if you’ve shown Roo your DB schema or plan), it can draft an Express route. For example, we prompted Roo to add the routes for task and response submission. It generated something like:  
  
 javascript  
CopyEdit  
// POST /tasks - Receives a new task

app.post('/tasks', async (req, res) => {

const { docId, taskId, title, description, status } = req.body;

if (!docId || !taskId || !title) {

return res.status(400).json({ message: 'docId, taskId, and title are required.' });

}

try {

const query = `

INSERT INTO tasks ("docId", "taskId", "title", "description", "status", "createdAt")

VALUES ($1, $2, $3, $4, $5, NOW());

`;

await db.pool.query(query, [docId, taskId, title, description, status || 'New']);

res.status(201).json({ message: 'Task created successfully.' });

} catch (err) {

console.error('Error creating task:', err);

res.status(500).json({ message: 'Failed to create task.' });

}

});

* And similarly for POST /responses[Google DriveGoogle Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU). We reviewed and tweaked these as needed (for example, ensuring the DB pooling is set up, which we cover next).
* **Database integration:** Incorporate the actual database now. We used the pg library for PostgreSQL. Roo’s Ask agent can fetch documentation on using pg if you need (e.g., how to set up a Pool). We created a db.js module to initialize new Pool({...}) with our DATABASE\_URL. This module exports an object (e.g., pool) that we import in our routes to run queries[Google DriveGoogle Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU). Make sure to load the connection string from an environment variable for security.
* **Example:** In our first attempt, we forgot to export the pool from db.js, causing a frustrating 504 error at runtime. The server was calling db.pool.connect() on an undefined object and hanging[Google Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU). We quickly fixed the module export and re-deployed, eliminating the 504. The lesson: double-check that your DB helper is correctly wired up!

**Create the schema (with Roo’s help or manually):** Now that routes expect certain tables, create those tables in the database. You can do this via TablePlus UI or by running SQL commands through Roo. For instance, you might write a file schema.sql or just execute commands in a DB console:  
  
 sql  
CopyEdit  
CREATE TABLE users (

id SERIAL PRIMARY KEY,

email TEXT UNIQUE,

apiToken TEXT UNIQUE,

subscriptionStatus TEXT,

expirationDate TIMESTAMP

);

CREATE TABLE tasks (

docId TEXT,

taskId TEXT PRIMARY KEY,

title TEXT,

description TEXT,

status TEXT,

createdAt TIMESTAMP DEFAULT NOW()

);

CREATE TABLE responses (

docId TEXT,

responseId TEXT PRIMARY KEY,

content TEXT,

taskId TEXT,

submittedAt TIMESTAMP DEFAULT NOW()

);

* These columns align with the data our Pack will send or request. In our project, the **users** table holds each user’s email and API token (the token they use in Coda), and the **tasks**/**responses** tables store data partitioned by docId (so multiple Coda docs’ data can reside in one database)[Google Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU). We created these tables on the aiappcoda\_db database via TablePlus[Google Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU). Roo’s Ask mode could be used to double-check SQL syntax or best practices (e.g., *“What data type should I use for a UUID in PostgreSQL?”*). Once created, we also inserted a test user row (with a known apiToken) so that during testing we have a valid credential[Google DriveGoogle Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU).
* **Implement authentication & middleware:** Our new architecture relies on each API request from Coda carrying a bearer token header. On the server, we need to verify that token. Typically, you’d write middleware to check Authorization: Bearer <token> against the users table. Roo’s Code mode can help stub this out. For example: *“Create Express middleware that checks req.headers.authorization against the users table (using db.pool). If invalid, respond 401.”* This ensures all protected routes confirm the caller’s identity. In our case, we simplified by just validating the token on each relevant request by querying SELECT \* FROM users WHERE apiToken = $1. (Some might cache user info in req.user – you can decide based on complexity.)
* **Roo Agent usage tips (Backend):**
  + *Ask Agent:* Use it whenever you need quick knowledge – e.g., “How do I format a SQL INSERT with node-postgres?” or “What does a 504 Gateway Timeout generally indicate on DO App Platform?” It can pull answers with citations for you. Just remember to double-check critical details.
  + *Code Agent:* Great for writing boilerplate and even complex logic. We let Roo write most of the new endpoints and DB logic, but we watched each change. Keep tasks small: implement one route fully (with DB calls and error handling), test it, then move to the next. This one-feature-at-a-time approach keeps the AI focused[Google DriveGoogle Drive](https://drive.google.com/file/d/1AZWFTjk-7geKnGZ-_d0sXK2mFN8SqFG0).
  + *Debug Agent:* When something isn’t working, switch to Debug Mode. For instance, after deployment we saw the server wasn’t receiving requests properly. In Debug mode, you can paste an error log or describe a symptom (*“The /tasks endpoint returns 404 even though it’s defined”*) and Roo will systematically diagnose potential causes. In our scenario, Roo helped confirm that the DigitalOcean router was likely altering our paths (leading to 404s) – which was correct[Google Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU).
  + *Test Agent:* If your project has unit tests or you want Roo to generate some, a Test agent (or using Code mode to write tests) can be invaluable. You might prompt: *“Write a unit test for the /tasks POST route using Jest.”* Roo can draft a test file. We didn’t focus on unit tests in the initial sprint, but we did manual integration testing with the live system (more on that soon).
* **Community insight – keep scope small:** One piece of advice from Roo’s community is to **keep each task tightly scoped**. Our experience reinforced this. When we tried to have Roo implement too many things at once (e.g. “set up DB, add all routes, and also update the front-end calls”), it produced convoluted output or got confused. It’s often better to finish a task, close it, and start a new task for the next piece[spin.atomicobject.com](https://spin.atomicobject.com/roo-code-ai-assisted-development/#:~:text=Rinse%20and%20Repeat)[spin.atomicobject.com](https://spin.atomicobject.com/roo-code-ai-assisted-development/#:~:text=the%20defined%20domain%20types%E2%80%9D,new%20task%20with%20the%20prompt). For example, after completing the DB integration, we started a fresh Roo task for hooking up the Coda Pack, which cleared out any stale context and let us focus anew.

**Example:** After deploying our backend the first time, nothing worked – the Pack couldn’t authenticate with the server. We discovered multiple small issues in our backend config that we then fixed one by one:

* The DigitalOcean App was pointing to an outdated Git repo, not our new one – we updated the App spec to use the correct repository[Google Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU).
* Our server still contained code to serve static frontend files (from an old setup), which crashed the app (ENOENT file not found). Removing that unused static file serving code solved the crash[Google Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU).
* DigitalOcean’s health check was failing because we had no base URL route. We added a simple app.get('/') that returns “OK” to satisfy the health check[Google Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU).
* The **big one**: All our API routes were defined with /api prefix (e.g. /api/me), but DigitalOcean’s ingress was also configured to route requests under /api. This double prefix meant our endpoints didn’t match and returned 404. We realized we should remove the "/api" from our Express routes (so define app.get('/me') instead of /api/me), because DO was already adding /api externally[Google Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU). Once we did that, authentication succeeded and the Pack’s calls started hitting the right endpoints.

Each of these issues was identified by carefully reading logs and asking “why isn’t this working?” The Roo Debug agent and Ask agent were helpful in this troubleshooting, but so was simply methodically testing each piece.

## **Coda Pack Integration Phase: Bridging Coda and the Backend**

Now for the “front-end” of this system: the Coda Pack that connects the Coda doc to your backend. This phase involves writing Pack formulas (using the Coda Packs SDK) and setting up the Coda doc UI (tables and buttons). Roo Code can assist here too, especially in Code and Ask modes, but you’ll also rely on Coda’s documentation.

**Authentication with HeaderBearerToken:** We configure the Pack to use Coda’s built-in **HeaderBearerToken** auth type. This lets Coda handle user API tokens securely. In pack.ts (or manifest.ts), set up user auth like:  
  
 typescript  
CopyEdit  
pack.setUserAuthentication({

type: coda.AuthenticationType.HeaderBearerToken,

instructionsUrl: "https://<your-app-url>/settings/api",

getConnectionName: async function(context) {

const response = await context.fetcher.fetch({

method: "GET",

url: "https://<your-app-url>.ondigitalocean.app/me"

});

return response.body.email;

},

});

pack.addNetworkDomain("<your-app-url>.ondigitalocean.app");

* This snippet tells Coda: our pack uses a token (which the user will paste in once), and how to get the account’s name for display (by calling our /me endpoint)[Google DriveGoogle Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU). The instructionsUrl can point to a help page for obtaining the token – in our case we might set it to our app’s settings page where users can find their API key. After adding this, when a user installs the Pack, Coda will prompt them for the token (once), store it, and automatically attach Authorization: Bearer <token> to every fetcher request. This eliminated our old manual token exchange system completely – a huge simplification.

**Define schemas for sync tables:** Identify the data structures that will flow between Coda and your app. We created Coda object schemas for **Task** and **Response** to mirror our DB tables:  
  
 typescript  
CopyEdit  
// schemas.ts

export const TaskSchema = coda.makeObjectSchema({

properties: {

taskId: { type: coda.ValueType.String, required: true },

title: { type: coda.ValueType.String },

description: { type: coda.ValueType.String },

status: { type: coda.ValueType.String },

createdAt: { type: coda.ValueType.String, codaType: coda.ValueHintType.DateTime },

},

displayProperty: "title",

idProperty: "taskId",

featuredProperties: ["status", "createdAt"],

});

export const ResponseSchema = coda.makeObjectSchema({

properties: {

responseId: { type: coda.ValueType.String, required: true },

content: { type: coda.ValueType.String },

submittedAt: { type: coda.ValueType.String, codaType: coda.ValueHintType.DateTime },

taskId: { type: coda.ValueType.String },

},

displayProperty: "content",

idProperty: "responseId",

featuredProperties: ["submittedAt","taskId"],

});

* This defines how Coda should interpret a “Task” or “Response” record. We made sure these match the fields our API returns (JSON keys in the HTTP response)[Google DriveGoogle Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU). Roo Code’s Ask agent was handy to quickly confirm usage of coda.makeObjectSchema or the right ValueHintType for dates.

**Add sync table formulas:** With schemas ready, we add **Sync Tables** to the Pack. A sync table is essentially a table in Coda that pulls data from your backend on demand. For example:  
  
 typescript  
CopyEdit  
pack.addSyncTable({

name: "Tasks",

schema: TaskSchema,

identityName: "Task",

formula: {

name: "SyncTasks",

description: "Pulls tasks from the web app.",

parameters: [], // no parameters, we’ll use the context

execute: async function([], context) {

const docId = context.document.id;

const url = coda.withQueryParams("https://<your-app-url>.ondigitalocean.app/tasks", { docId });

const response = await context.fetcher.fetch({ method: "GET", url: url });

return { result: response.body.tasks };

},

},

});

* We do similarly for Responses (just hitting /responses)[Google DriveGoogle Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU). Notice we use context.document.id – Coda provides the current doc’s ID which our backend expects to filter data by doc. We also use coda.withQueryParams to append the docId as a query param. These sync formulas will be called by Coda behind the scenes when the user adds the table to their doc or clicks “Refresh”. Each returns an array of objects that Coda will map to rows using our schemas.
* **Tip:** Make sure to include pack.addNetworkDomain("<your-app-url>") for any domain your fetcher hits (we did earlier). Without it, Coda will block the requests for security.

**Implement data push (Coda to backend):** Sync tables cover pulling data into Coda. For pushing data (e.g. user inputs from Coda to your app), we set up a Pack **Action**. In our case, users can press a button in Coda to send all content rows to the app. We added a formula like:  
  
 typescript  
CopyEdit  
pack.addFormula({

name: "SendAppContent",

description: "Pushes all rows from the Content table to the web app.",

parameters: [

coda.makeParameter({ type: coda.ValueType.String, name: "contentJson", description: "Content JSON" })

],

resultType: coda.ValueType.String,

isAction: true,

execute: async function([contentJson], context) {

let payload;

try {

payload = JSON.parse(contentJson);

} catch {

throw new coda.UserVisibleError("Invalid JSON payload.");

}

const response = await context.fetcher.fetch({

method: "POST",

url: "https://<your-app-url>.ondigitalocean.app/app\_content",

headers: { "Content-Type": "application/json" },

body: JSON.stringify(payload),

});

return `Pushed content. Server responded ${response.status}.`;

},

});

* This formula expects a pre-stringified JSON of the content (since Coda actions can’t natively handle arbitrary object parameters easily). It posts that JSON to our /app\_content endpoint on the backend[Google DriveGoogle Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU). We marked it isAction: true so it can be used in a button.  
    
   The trick was **producing the JSON string of all content rows inside Coda**. Initially, we attempted a complex formula with Format() and FormulaMap to build a JSON string manually (concatenating strings for each row). This led to errors when content contained quotes or special characters. We even tried a non-existent JsonEncode() formula (doesn’t exist in Coda) and hit a wall[Google Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU). The breakthrough was discovering a community-made Pack that provides a toJSON() formula. We installed that Pack in our Coda doc, which allowed us to simplify the button formula to just RunActions(SendAppContent(toJSON([ContentTable]))). **Example:** Our earlier approach looked like a long Format('{{ "docId": "{1}", "contentRows": [...] }}', ...) formula with manual escaping[Google Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU), which was brittle. By using a proper JSON serializer, we avoided all those edge-case bugs[Google Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU). This is a great example of AI + community knowledge: our AI agents didn’t know about the toJSON Pack, but by researching (Ask agent searching Coda forums) and a bit of user insight, we found the solution. *Lesson:* When Roo or even ChatGPT hits an unknown, leverage community plugins or ask the forums – someone might have built the exact helper you need.
* **Integrate with the Coda doc UI:** After coding the Pack, update the Coda doc (template) to use it:  
  + Add the **Tasks** and **Responses** sync tables to the doc (these will appear once the Pack is linked to the doc and published or in dev mode).
  + Create a button for sending content. For example, a button labeled “**Send Content**” that runs the SendAppContent() action. The button formula should assemble the payload. Using the toJSON approach, it might be RunActions(SendAppContent(toJSON(thisTable))) if thisTable is the table of content you want to sync. In our final setup, we placed this button prominently and removed older buttons from the deprecated flow[Google Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU).
  + Remove any legacy tables or formulas from the old system (for us, that meant deleting the no-longer-used doc\_registry or regKey-related elements).
* **Roo Agent usage tips (Pack/Coda):**
  + *Ask Agent:* Great for querying Coda Pack documentation. You can copy-paste snippets from Coda’s docs (the Ask agent will preserve attributions) or ask targeted questions: *“How to add a sync table in a Coda Pack?”* or *“What does context.sync.continuation do?”* (if you had paginated sync). The agent can retrieve answers from Coda’s official guides[Google Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU) or community posts.
  + *Code Agent:* You can use it to write the pack code, but ensure you test each part. For example, we let Roo draft the schema and sync table code from our descriptions and the API spec. It did well, but we had to correct minor things (like adding cacheTtlSecs: 0 if we wanted no caching on sync fetches). Always review the generated code against the Coda SDK types – TypeScript will help catch mistakes too.
  + *Debug Agent:* If your pack isn’t working, use Debug mode with any error messages. For instance, if coda upload fails or a formula throws an exception, share that with Debug agent. In one case, our pack wasn’t uploading because of a mis-set Pack ID and authentication error. The Debug agent suggested re-linking the pack (which was exactly the fix) and checking our manifest for any missing fields.

**Example:** One real-world issue during Pack development was the Pack not displaying the connection properly. We noticed that after a user entered their token, the Pack should show “Connected as [email]”. It wasn’t at first. Using the Debug agent and logs, we found the mistake: our /me endpoint wasn’t returning exactly { email: userEmail }. We adjusted the backend to match the Pack’s expectation. The next test showed the user’s email in Coda, confirming the getConnectionName was working. Small details like response shape can cause integration hiccups – having Roo double-check the contract (Pack expects vs. server returns) is useful.

Also, after implementing the SendAppContent action, our first test gave a DEADLINE\_EXCEEDED error in Coda (meaning the action took >30 seconds and timed out)[Google Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU). Nothing was obviously wrong – the payload was small. We suspected maybe a cold server or network issue. The Debug agent helped us realize the action wasn’t timing out due to size, but because the server was never responding at all. This led to the discovery that our database calls were hanging (the DB URL was wrong). In summary, when a Coda action times out with no response, it often points to a server-side hang. We ultimately fixed the env var and the action completed in seconds[Google DriveGoogle Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU).

## **Testing and Troubleshooting Phase**

With both backend and pack in place, thorough testing is crucial. This phase is about verifying each component works end-to-end and ironing out any bugs. It’s also where Roo’s **Debug** and **Test** agents, along with good old manual testing, come into play heavily.

* **Local testing of the backend:** Before integrating with Coda, test your backend with tools like Postman or curl. For instance, run the server locally (or on a staging URL) and try hitting GET /tasks?docId=<some-id> with an Authorization: Bearer <token> header. Does it return the expected JSON? You can use Roo’s terminal tool to run curl commands and even automate some checks. We wrote a quick script to test all endpoints with a sample token and docId. This helped ensure the backend logic and SQL were correct without waiting for Coda.
* **Coda Pack test deployment:** Use coda upload to push your Pack to Coda (in development mode) and coda release when you’re ready for others to use it[Google Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU). In the Coda doc, install the pack (if not already) and add the sync tables. Enter a known API token via the pack settings and verify data pulls in. During dev, you can also use coda execute to run formulas locally, but we found using the actual doc more straightforward for integration testing.
* **Full integration test:** Try the core user actions in sequence:  
  + **Authentication flow:** Does connecting the Pack prompt for a token and then display “Connected as [the user’s email]”? Test with a test user’s token. If the email shows up, your /me endpoint and getConnectionName are working[Google DriveGoogle Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU).
  + **Data pull sync:** Populate your database with some test tasks/responses for the test user & docId. Then click “Refresh” on the Tasks sync table in Coda. The rows should appear, matching what’s in the DB for that docId. If not, check the Pack’s log or the network request (Coda has a developer panel for Packs where you can see fetcher errors). We had to verify that context.document.id was indeed sending a valid ID and that our backend filtered correctly. It did, and tasks came through.
  + **Data push action:** In Coda, modify some data (or create a new content row if you have a content table). Click the “Send Content” button. Then check the backend – did it receive it and insert into the DB? We watched the server logs (via DigitalOcean’s logs) to see the incoming request and any errors. Our first try timed out (50s) with no log output, hinting the server hung. This is where we dug into the **DigitalOcean config** – eventually finding the mis-set DATABASE\_URL and fixing it[Google Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU). On re-deploy, the next button press succeeded (server log showed an insertion query, and Coda got a success response).
* **Troubleshooting with Roo’s Debug agent:** When tests reveal issues, describe them to the Debug agent. Some examples:  
  + *“My Coda action is timing out with DEADLINE\_EXCEEDED.”* Roo suggested checking server performance and then connection settings, which led us to the environment variable insight[Google Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU).
  + *“The /tasks sync table isn’t returning any data.”* Roo might remind you to verify that the tasks endpoint is included in pack.addNetworkDomain and that response.body.tasks actually contains an array. (We once forgot to wrap the result as { result: ... } in the sync formula, which caused an empty table – an easy miss that the Debug agent caught.)
  + *“Getting 401 Unauthorized on all requests.”* Roo will prompt you to double-check that the token in Coda matches one in your DB and that the auth middleware is correctly implemented. Indeed, we had to ensure the token format matched (no extra spaces, “Bearer ” prefix handled, etc.).
* **Task continuation vs new tasks:** During testing and debugging, decide whether to continue an existing Roo task or start fresh for a new angle. There’s no strict rule, but experience helps[spin.atomicobject.com](https://spin.atomicobject.com/roo-code-ai-assisted-development/#:~:text=I%20could%20do%20what%20I,to%20do%20so%20with%20experience)[spin.atomicobject.com](https://spin.atomicobject.com/roo-code-ai-assisted-development/#:~:text=for%20now). A guideline:  
  + If the fix is small and directly related to the last AI output, you can continue in the same task (e.g. “Oops, update that query to use parameterized inputs”).
  + If you feel the context has gotten muddled or you’re shifting to a different sub-problem, it’s often better to start a **New Task**. This clears the conversational history (except what’s in files and memory bank) and gives the agent a clean slate focused only on the current issue. We did this when moving from backend coding to pack coding – starting a new task helped Roo focus on the Pack without the distraction of backend context. Also, after fixing one bug, we’d sometimes start a new task to test the next feature, to avoid any lingering confusion.
  + The Roo community suggests keeping tasks atomic and ending them once the objective is met[spin.atomicobject.com](https://spin.atomicobject.com/roo-code-ai-assisted-development/#:~:text=Rinse%20and%20Repeat)[spin.atomicobject.com](https://spin.atomicobject.com/roo-code-ai-assisted-development/#:~:text=the%20following%20text%3A%20%E2%80%9CThe%20next,new%20task%20with%20the%20prompt). You can always open a new task saying “Continue with the next step: X” and Roo will pick up based on the updated codebase (thanks to Roo’s memory of files). In fact, after completing our data push feature, the Architect plan recommended the next step (e.g. *“Next, create the frontend UI or next integration”*). We took that text and literally fed it as the prompt for a new task: *“Implement the next step: create the database schema based on the types”*[spin.atomicobject.com](https://spin.atomicobject.com/roo-code-ai-assisted-development/#:~:text=Once%20you%E2%80%99ve%20completed%20a%20task,works%20to%20say%20something%20like)[spin.atomicobject.com](https://spin.atomicobject.com/roo-code-ai-assisted-development/#:~:text=the%20defined%20domain%20types%E2%80%9D,new%20task%20with%20the%20prompt). This way, each task stays focused and the AI agents remain efficient.
* **Performance checks:** It’s easier to address performance now than post-deployment. We did a quick check on how our sync was performing with larger data. Using Ask agent, we inquired about any Coda limits – e.g., Pack actions have a 30 second limit and sync tables a 30 sec limit per sync (with 5MB response cap). Our data was small, so no issues. But if you anticipate scale, consider adding pagination to sync formulas or limiting how much the Pack sends at once. Roo can help implement a continuation token if needed.
* **Security audit (AI-assisted):** Before deploying widely, run a security and maintainability pass. You can actually use Roo (or another AI) to do this. Prompt the Ask agent or Debug agent with something like: *“Audit the code for any security issues or secrets.”* Or *“Check that all inputs from Coda are validated server-side.”* In vibe coding practice, AI can help spot things like missing input sanitization or uses of eval. We made sure:  
  + All our SQL queries use parameterized queries (to prevent SQL injection).
  + The Pack doesn’t expose sensitive info (it doesn’t; the token is stored by Coda, not visible in docs).
  + The backend only allows authorized tokens and only accesses data for that user’s docId.
  + No secrets (like our own API keys) are in the repo – they’re in environment variables and .gitignore.  
     These are basic, but easy to overlook when moving fast. The Vibe Coding Playbook emphasizes: **never hardcode API keys**, always use .env for secrets, add rate limiting, and validate all inputs[Google DriveGoogle Drive](https://drive.google.com/file/d/1AZWFTjk-7geKnGZ-_d0sXK2mFN8SqFG0). We followed those principles. For example, we set up Express rate limiting on the /app\_content endpoint (to avoid abuse, since it writes to DB) and added basic input checks in the Pack (so the action won’t send malformed JSON).

**Example – The case of the mysterious timeout:** During end-to-end testing, our “Send Content” Coda button kept timing out at 50 seconds. No error logs, just hanging. Using TablePlus and logs, we confirmed the data wasn’t reaching the DB – it was never inserted. This pointed to the server stalling on the DB connection. We checked DigitalOcean’s “trusted sources” firewall – all good. Finally, we discovered the true issue: the DATABASE\_URL variable in the App Platform was still pointing to an old database (which we had deleted)[Google Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU)! Essentially, our server was trying to connect to nowhere and hung. We updated the env var to the new DB and re-deployed. Immediately, the next test succeeded in ~2 seconds[Google Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU). This saga taught us to double-check environment config whenever there’s a baffling timeout. It’s an example of a bug that’s not in code, but in deployment settings – something an AI won’t catch from reading code. **Tip:** When AI-driven tests don’t reveal an obvious code issue, consider external factors (env vars, network, platform-specific quirks). Sometimes the “human intuition” layer is needed to ask the right question. In our case: “Could the DB URL be wrong?” was the key – and that was answered by old-fashioned inspection.

## **Deployment Phase: Launch and Monitor**

Deployment is the final step – getting your backend running in the cloud and publishing your Pack for users. A smooth deployment requires coordination between AI helpers and manual oversight.

* **Deploying the backend to DigitalOcean:** We were using DigitalOcean’s App Platform. We connected our GitHub repo to DO, set the build commands (e.g., npm install and npm start for the server), and configured environment variables like DATABASE\_URL and any API keys (ensuring they are set to the correct values). **Example:** We encountered a deployment issue where DO kept redeploying our app due to failing health checks. By adding a simple health-check route and removing the faulty static file serving, we satisfied the platform’s requirements[Google Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU). **Warning:** Cloud platforms often have default behaviors (like prefixing routes, or expecting a response on /). Read the platform docs or use Ask agent to learn these nuances. Roo’s Ask mode helped us find that *DigitalOcean’s router was stripping the /api path*, which guided our fix of removing /api in code[Google Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU).
* **Final Coda Pack release:** Once everything works in dev, publish the Pack so others (or your production doc) can use it. Use coda release <pack-id> <version> to push a new version. Update any doc that uses it to that version. Verify one more time in the live doc that authentication and sync/push all work with the production endpoints.
* **Effective use of Roo and Gemini in deployment:** Deploying is often more procedural, but you can still utilize AI:  
  + Let Roo’s Ask agent create a **deployment checklist**. We asked something like, *“What are all the steps to deploy a Node/React app on DO App Platform?”* and cross-checked we did each (set correct repo, config vars, build commands, domain mapping, etc.). This was similar to what our Gemini assistant initially did – it outlined a generic deployment plan[Google Drive](https://docs.google.com/document/d/1crnNwXIbJVDGfO8bjZRACqL9M87a56xcBOfynvm38ws). We then modified it to fit our project specifics (like ensuring the Pack’s domain is whitelisted, etc.).
  + Use the Debug agent for any issues during deploy. If DO throws an error log (e.g., port not exposed, or build failed), feed that to Roo Debug. It can pinpoint common causes (e.g., “Ensure your app listens on PORT env var” – DO requires that).
  + Use Gemini or another agent to simulate a *user acceptance test*. We wrote a short scenario: “User installs the pack, enters token, adds a task in the web app, sees it in Coda, adds a task in Coda, sees it in the web app.” Then essentially role-played it to make sure all parts were covered. AI can help enumerate these user steps so you don’t forget a corner case.
* **Post-deploy monitoring and optimization:** Once live, keep an eye on logs and performance:  
  + **Logging:** Make sure your server logs errors meaningfully. We added console.error in catch blocks (which DO captures). In production, consider integrating with a logging service or at least logging to a file for later analysis.
  + **Performance tweaks:** Our initial deployment was on a basic instance. We watched CPU/memory usage during heavy syncs – it was fine. If your AI-generated code isn’t the most efficient, you might need to optimize. For example, if sync is slow, maybe add DB indexes on docId, or if memory usage spikes, stream large responses instead of loading in memory. Roo’s Ask agent can give quick suggestions on performance tuning if needed (*“How to optimize an Express app for many simultaneous requests?”*).
  + **Security audit (final):** Do a final check that no secrets are in client-side code (for example, if you had a React frontend, ensure any API keys are secure). Our stack kept secrets on the server and Coda’s side, so we were okay. We did ensure the Coda Pack’s source (which could be visible to others if they inspect the Pack) doesn’t contain anything sensitive – it doesn’t, only the network domain and the logic, which is fine.
* **User onboarding and documentation:** Now that the system is live, consider writing a brief guide (could even be AI-assisted). We documented how a new user should get their API token from our app, install the Pack, and press “Send Content” to sync. Clear instructions reduce support burden. This aligns with vibe coding’s emphasis on maintainability – good docs and clear workflows are part of a solid project.

**Tip:** Continue to use vibe coding practices even post-deployment:

* If you need to add a feature or fix a bug, start by updating your plan or to-do list. Maybe Phase 8 is *“Frontend App Integration”* (in our case, building a React UI that consumes the same API – indeed in our ChatGPT thread we see references to an interactive slides front-end with Gemini integration[Google Drive](https://docs.google.com/document/d/1crnNwXIbJVDGfO8bjZRACqL9M87a56xcBOfynvm38ws)). Plan it out with Architect mode, then implement with Code mode.
* Keep using Git for version control. Tag releases, use branches for experimental changes and merge when tested.
* If something goes wrong in production, don’t panic-debug in prod – recreate it locally and let the Debug agent help. We actually simulated a scenario locally where a user’s token expired, to see how the system behaves (it returned 401 and Coda would prompt to re-auth – which is acceptable). Having AI to analyze hypothetical failures (*“What happens if the DB is down? How to handle?”*) can lead you to add fallback logic or better error messages.
* **Gemini and Roo agents in harmony:** A final note on multi-agent workflow. We found that using **Gemini** (as a separate assistant) in conjunction with Roo’s specialized agents gave us the best of both worlds. Roo was deeply integrated with our codebase, great for specific coding tasks, whereas Gemini (or ChatGPT-4, etc.) was useful for high-level Q&A and brainstorming. For example, after completing the backend and pack, we asked a high-level AI: *“Are there any edge cases or failure modes we should test for this Coda-pack-to-app system?”* It suggested things like token expiration, multiple docs with same user, handling of null data – some of which we might have missed. We then used Roo to implement handling for those cases (like making sure deleting a doc’s data is possible, which we added as a cleanup endpoint). Using a coordinator AI to overview the project and specialist AIs (Roo agents) to do the heavy lifting proved to be a productive strategy. It mirrors the SPARC framework idea of orchestrator vs. specialist agents[github.com](https://github.com/Mnehmos/Advanced-Multi-Agent-AI-Framework#:~:text=Boomerang%20Task%20Delegation).

**Example – Gemini’s role in final prep:** We had Gemini generate a “pre-launch checklist” and it reminded us to double-check OAuth vs token (to ensure we didn’t accidentally leave an OAuth flow piece in), to ensure the Pack’s network domain is correct (especially if our app URL changed for production), and to confirm our Coda pack permissions are minimal (just the one domain, no unnecessary scopes). This was a nice sanity check. In one instance, Gemini incorrectly mentioned a non-existent “Static\_Token” concept (a misunderstanding from a previous conversation)[Google Drive](https://docs.google.com/document/d/1crnNwXIbJVDGfO8bjZRACqL9M87a56xcBOfynvm38ws), which we recognized and ignored. This highlights that while these AIs are powerful, you must filter their output with your own knowledge – don’t accept everything blindly. Use them as advisors, not sources of absolute truth.

## **Conclusion**

By following this playbook, we navigated the project from idea to deployment in structured phases. Along the way we applied vibe coding principles:

* **Plan before you code:** we invested in detailed planning and it paid off with smoother implementation[Google Drive](https://drive.google.com/file/d/1AZWFTjk-7geKnGZ-_d0sXK2mFN8SqFG0).
* **Work in small tasks:** focusing the AI on one feature at a time kept quality high and debugging easier[Google DriveGoogle Drive](https://drive.google.com/file/d/1AZWFTjk-7geKnGZ-_d0sXK2mFN8SqFG0).
* **Use the right agent for the job:** Architect for planning, Code for coding, Ask for research, Debug for troubleshooting, Test for validation. This “AI team” approach (in Roo Code, your whole dev team is literally in the editor) makes development feel collaborative and efficient.
* **Keep the human in the loop:** We constantly reviewed AI output, set rules, and intervened when necessary (e.g. halting Roo when it got off-track, or doing a quick manual fix that was faster than prompting again)[spin.atomicobject.com](https://spin.atomicobject.com/roo-code-ai-assisted-development/#:~:text=Manually%20Intervene%20When%20Necessary)[spin.atomicobject.com](https://spin.atomicobject.com/roo-code-ai-assisted-development/#:~:text=Remember%20how%20I%20mentioned%20earlier,makes%20them%20and%20course%20correct). AI is a partner, but you’re the lead developer.
* **Embrace version control and rollback:** This saved us more than once – with both Roo’s checkpoints and Git commits, we had the freedom to experiment without fear. As the vibe playbook says, *sometimes the fastest way to fix a bug is to rewind*[Google Drive](https://drive.google.com/file/d/1AZWFTjk-7geKnGZ-_d0sXK2mFN8SqFG0).
* **Prioritize security and clarity:** We treated things like API tokens, error handling, and documentation as first-class tasks, not afterthoughts. The result is a system that’s not just working, but maintainable and secure for real users[Google DriveGoogle Drive](https://drive.google.com/file/d/1AZWFTjk-7geKnGZ-_d0sXK2mFN8SqFG0).

This playbook can serve as a template for your own projects. Feel free to adapt it – add “Warnings” for gotchas you encounter, “Tips” from your experience, and new phases as your project demands. By combining Roo Code’s powerful AI agents with a solid vibe-driven process, you (and anyone you share this with) can build reliable, tasteful, and bug-free software faster than ever. Good luck, and happy vibe coding!

**Sources:**

1. EmployeeImpactAI Project Plan – Coda-to-App Data Synchronization[Google DriveGoogle Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU)
2. EmployeeImpactAI Project Progress – Technical Challenges & Resolutions[Google DriveGoogle Drive](https://docs.google.com/document/d/1tQP3iwvz4xbJX4hhq62jQkw92DKg6UHQ6gMpfxsI4zU)
3. Matthew Berman, *The Vibe Coding Playbook*, 2025 – Best Practices for AI-Assisted Development[Google DriveGoogle Drive](https://drive.google.com/file/d/1AZWFTjk-7geKnGZ-_d0sXK2mFN8SqFG0)
4. Roo Code Documentation – AI Agents and Workflow (Roo Code 3.21)[github.com](https://github.com/RooCodeInc/Roo-Code#:~:text=Multiple%20Modes)[spin.atomicobject.com](https://spin.atomicobject.com/roo-code-ai-assisted-development/#:~:text=There%20have%20been%20times%20when,a%20checkpoint%20is%20made%20whenever)
5. “How I Effectively Use Roo Code” – Developer Blog (Atomic Object, 2025)[spin.atomicobject.com](https://spin.atomicobject.com/roo-code-ai-assisted-development/#:~:text=Always%20Start%20in%20Architect%20Mode)[spin.atomicobject.com](https://spin.atomicobject.com/roo-code-ai-assisted-development/#:~:text=There%20have%20been%20times%20when,a%20checkpoint%20is%20made%20whenever)
6. Project Chat Transcript – Verification of Architecture (Gemini AI and ChatGPT)[Google DriveGoogle Drive](https://docs.google.com/document/d/1crnNwXIbJVDGfO8bjZRACqL9M87a56xcBOfynvm38ws)