AY AUTOMATE — PLAYBOOK

Vibe Coding SaaS MVPs: The Ultimate Guide

intro who i am and where work is going

hi, i'm walid. i co-founded ay automate and i'm also the co-founder of humanoidz, our venture studio. after scaling the business, i now manage the team and keep the work flowing smoothly day to day. i spend my time talking to people, listening for the real problems under the surface, and shaping simple, agent-powered systems that save time, cut costs, and unlock growth. i lead discovery calls, scope clearly, write clean proposals, and make sure handoffs to the build team are tight and aligned.

i mostly work with fast-moving teams who are overwhelmed by repetitive workflows and curious about ai, but don't have the hours to figure it all out. think founders, coos, and operators in ecommerce, saas, services, and ops-heavy spaces. my style is direct, collaborative, and calm. no pressure. just clarity, options, and next steps.

on linkedin, i share behind-the-scenes builds, real use cases, and honest notes about agents and automation. i try to keep it practical and close to the work.

the future of work, simply

- work is shifting from clicking to orchestrating. we'll spend less time doing and more time deciding what should be done.
- agents won't replace judgment. they'll make room for it. the value moves to problem framing, taste, and systems thinking.
- small teams will feel bigger. an operator with good prompts and clean processes becomes a force multiplier.
- the winners will be boring on purpose. simple, reliable systems beat flashy tools. consistency compounds.

if you're overloaded and want space to think again, we can start small. map the flow, remove the drags, and let agents handle the busywork. keep what matters human.

Prompt Pack 2.0 polished and ready to paste

Kickoff: clarify the idea

```
I'm planning to build [describe your app briefly]. Ask clarifying questions to i
```

Create a PRD from the discussion

```
Based on our conversation, draft a concise PRD that includes:
1) Goals and non-goals
2) Target users and primary JTBD
3) MVP feature list and user flows
4) Key screens and states
5) Data model (entities, fields, relationships)
6) Tech notes and constraints
7) Risks and assumptions
Keep it under 2 pages. Use bullet points. End with a checklist I can review.
```

Tech stack evaluation

```
Recommend a stack for rapid MVP delivery with good AI tooling. Compare 2 options
- Speed to first prototype
- Developer ergonomics
- Hosting and cost
- Scalability to v1
Propose: FE, BE, DB, Auth, file storage, and infra. Call out trade-offs and what
```

Project setup in Cursor

```
Create a Next.js + TypeScript project with Tailwind and ESLint. Plan first, then
- App directory structure
- Shared UI components folder
- API routes scaffold
- Env var placeholders (.env.example)
- Minimal layout and theme
After changes, print the exact commands to run dev and a sanity checklist.
```

Generate the Home screen component

```
Build a responsive Home screen with:
- Header with product name and tagline
- Primary CTA button “Find Recommendations”
- Recent activity section with empty state
```

```
- Bottom nav (Home, Favorites, Profile)
Use Tailwind classes, semantic HTML, and accessible labels. Return a single sel
```

Core recommendation flow

```
Implement a flow where the user selects mood, cuisines (multi-select), and price
Requirements:
- Local state management
- Validation and disabled state while loading
- Call a stub recommend() service and render results list with empty and error s
Return component code and a stub service with types.
```

Backend API endpoint (filter + simple ranking)

```
Create /api/recommend that accepts mood, cuisines[], priceRange.
- Load a JSON dataset of ~20 restaurants
- Filter by cuisines and price
- Rank by mood keyword matches in description
- Return top 10 items with score
Include TypeScript types and basic input validation.
```

Replace mock data with Google Places later

```
Plan the steps to swap mock data with Google Places:
1) Add env vars for API key
2) Server-side fetch by user location
3) Map Places fields to our Restaurant type
4) Rate limit and caching strategy
5) Error handling and fallbacks to mock data
Output code snippets for each step without executing them yet.
```

QA bug report and fix loop

```
I’m seeing a bug: multiple cuisine selections are not respected in filtering. Re
```

Performance pass on long lists

```
The results list lags with 500+ items. Suggest and implement optimizations:
- Virtualized list
- Memoization and stable keys
- Suspense-friendly data fetching
```

```
- Lightweight item components
Provide before/after profiling notes.
```

Deployment prep and checklist

```
Prepare for production:
- Build-time and runtime env vars
- Logging and minimal analytics
- Error boundary and fallback UI
- Vercel config for Next.js
- Post-deploy smoke tests
Return a copy-paste deployment checklist and the exact commands.
```

part one define your idea in simple words

just start small. write what you want to build in one or two lines. who is it for, what problem it fixes, and what “done” looks like. keep it chill and honest.

- what is the app for
- who is going to use it
- what is the one thing it must do well for v1
- how will someone use it from start to finish

here’s a tiny prompt you can paste to your ai buddy to talk it through

```
hey, i want to build [your app idea].
ask me simple questions to nail the users, the core jobs-to-be-done, and a tiny
please summarize what you heard in plain bullets and list any open questions.
```

when the chat feels clear, ask for a mini prd (nothing fancy)

```
please write a short prd (under 2 pages) with
- goals and non-goals
- target users and main job-to-be-done
- mvp features and the basic user flow
- key screens and empty/error states
- simple data model (entities and fields)
- risks and assumptions
finish with a checklist i can review.
```

if you want a quick visual, this is the flow

```
flowchart LR
  A[idea] --> B[clarify with ai]
  B --> C[draft mini prd]
  C --> D[agree mvp scope]
```

pro tips (soft and simple)

- save your mini prd. you'll reuse it a lot
- keep scope tiny now. add nice-to-haves later
- if you feel stuck, write the flow as a story "a user opens the app, does x, sees y"

---

part two choose a simple stack that won't fight you

keep it easy. pick tools that are common and have lots of examples. you can always swap later.

- one-shot route: lovable, bolt, or replit builds most of it for you fast
- guided route: cursor or windsurf helps you build piece by piece

how to decide fast

- if the app is simple and you just want something working today → one-shot
- if you want control and to learn the code → guided

recommended defaults

- web: next.js + typescript + tailwind
- data: supabase or firebase
- auth: the one built into your data choice
- deploy: vercel

paste this to your ai if you want a quick pick

```
recommend a simple stack for this idea. favor common tools with great docs.
propose: frontend, backend, database, auth, and hosting. explain in plain words
if something is overkill, say so. keep it minimal.
```

quick visual

```
flowchart LR
  A[requirements] --> B[compare options]
  B --> C{pick approach}
  C -->|one-shot| D[lovable/bolt/replit]
  C -->|guided| E[cursor/windsurf]
  D --> F[accounts + config]
```

```
  E --> F
  F --> G[frontend + backend + db]
```

pro tips

- choose boring, popular tech. it's easier to debug
- don't chase perfect. ship the mvp, then improve
- write down any trade-offs so future you remembers

---

part three scaffold the app fast with ai

goal get the skeleton running so you can click around. no polish. just a working base.

if you use onedshot

- create a project in lovable, bolt, or replit
- paste your mini prd and any mustdhave tech choices
- wait a few minutes, then run it and click everything

if you use guided (my usual)

- open cursor
- ask it to set up next.js + typescript + tailwind + api routes
- let it create folders, configs, and scripts
- run the dev server and make sure it boots

copydpaste prompt

```
set up a next.js + typescript project with tailwind and a simple api folder.
add a components folder, a utils folder, and .env.example with placeholders.
when done, show me the exact commands to run and how to verify it works.
```

what to check

- can you start it locally without errors
- does the home page render
- do you see the folders and basic files you expect

visual

```
flowchart LR
  A[start] --> B{generation mode}
  B -->|one-shot| C[platform builds]
  C --> D[run locally]
```

```
  B -->|guided| E[cursor scaffolds]
  E --> D
```

pro tips

- if something fails, paste the exact error into the chat
- keep commits small "setup ok", "home page ok"
- move on once it runs dondt chase details yet

---

part four shape the ui in small, clear pieces

think lego. one block at a time. get the screen to show up, then add little details.

start with the home screen

- title and a short line that says what this does
- one big button that starts the main action
- an empty state area that says “nothing here yet”
- simple nav at the bottom or top

copy-paste prompt

```
make a simple, responsive home screen:
- header with app name and one-line purpose
- big primary button “find recommendations”
- empty state section for recent items
- basic nav with home, favorites, profile
use tailwind. keep the markup clean and accessible.
return one self-contained component.
```

add more screens the same way

- list page
- details page
- a small form for inputs

visual

```
flowchart LR
  A[ui plan] --> B[home]
  B --> C[list]
  C --> D[details]
  B --> E[form]
  D & E --> F[polish later]
```

pro tips

- make it work on mobile first. simple layouts scale up easily
- name components clearly so you can find them later
- after each screen renders, commit and move on

---

part five make the app actually do the thing

now we wire buttons to real work. keep it tiny and test as you go.

what this means

- when a user clicks a button, something happens
- we read inputs, run a function, and show a result

start with the smallest path

- take the form values
- call a simple function recommend()
- show a loading state, then results, or a friendly error

copy-paste prompt

```
add a simple recommendation flow:
- gather mood, cuisines[], and price
- call recommend() in services/recommend.ts
- while loading, disable the button and show a spinner
- render a list of results with empty and error states
return the component update and the recommend() stub with types.
```

visual

```
flowchart LR
  A[click] --> B[read inputs]
  B --> C["recommend()"]
  C --> D[show results]
  C --> E[handle error]
```

pro tips

- test with fake data first
- always show empty and error states it feels polished
- commit after it works once end-to-end

---

part six connect data without stress

we're adding real data, but keep it calm. start fake, then go real.

start with mock data

- make a data/restaurants.json with ~20 items
- build a tiny api route that reads from it and filters

then add a real source later (like google places)

- add env vars
- fetch on the server
- map their fields to your fields
- keep a fallback to the mock json if the api fails

copy-paste prompt

```
create /api/recommend that accepts mood, cuisines[], priceRange.
- load a local json dataset first
- filter by cuisines and price
- rank by simple mood keyword matches
- return top 10 with a score
include types and basic input validation.
```

when you're ready to switch to google places

```
plan the swap from mock to google places:
1) add env var GOOGLE_PLACES_API_KEY and read it safely
2) server-side fetch by user location
3) map places fields to our Restaurant type
4) add simple rate limiting and caching
5) fallback to mock json on error
return code snippets for each step without executing anything.
```

visual

```
flowchart LR
  A[mock json] --> B["/api/recommend"]
  B --> C[frontend]
  A -.later .-> D[google places]
  D --> B
  B --> E[fallback on error]
```

pro tips

- keep secrets in .env, never in code

- start simple, then harden with auth and logs later
- test with empty, small, and large results

---

part seven test it, fix it, ship it

keep this super light. test like a user, then fix the rough edges.

basic loop

- try each flow end-to-end
- when you see something weird, write one short note
- fix one thing at a time, then retest

quick prompts you can paste

```
make a tiny test plan for this app:
- list the main flows in bullets
- add 2 edge cases per flow
- include a 5-minute smoke test before deploy
keep it short and plain.
```

```
here’s a bug i see: [describe briefly].
reproduce it, explain why it happens in simple words, propose a fix, and give me
add a quick test so it doesn’t come back.
```

visual

```
flowchart LR
  A[test flow] --> B[find issue]
  B --> C[small fix]
  C --> D[retest]
  D -->|ok| E[deploy]
  D -->|not ok| B
```

ship with confidence

- run your smoke test
- deploy to vercel
- click through the main flow in prod
- jot down any follow-ups for later

pro tips

- tiny fixes ship faster than big refactors

- write down the exact steps to reproduce a bug
- if performance feels slow, try loading fewer items on screen first

---

that's it

start small, keep it simple, ship early. then improve.

Supercharging Your Prompts

The quality of your prompts directly impacts the quality of AI-generated code. Use these techniques to get better results:

1. **Be specific and detailed** – Instead of "create a login form," specify "create a login form with email and password fields, validation, error handling, and a 'forgot password' link"
2. **Provide examples** – When available, show the AI examples of similar features or styling you like
3. **Establish context** – Remind the AI of previous decisions or the broader architecture
4. **Request explanations** – Ask the AI to explain its approach before implementing
5. **Break complex requests into steps** – For intricate features, outline the steps and have the AI tackle them sequentially

Handling AI Limitations

Even the best AI assistants have limitations. Here's how to navigate them:

1. **Chunk large codebases** – Most AI tools have context limitations. Focus on specific files or components rather than the entire application at once.
2. **Verify third-party interactions** – Double-check code that integrates with external APIs or services, as AI may generate outdated or incorrect integration code.
3. **Beware of hallucinations** – AI might reference nonexistent functions or libraries. Always verify dependencies and imports.
4. **Plan for maintenance** – Document AI-generated code thoroughly to make future maintenance easier.
5. **Establish guardrails** – Use linters, type checking, and automated tests to catch issues in AI-generated code.

Managing Technical Debt

Rapid development can lead to technical debt. Here's how to minimize it:

1. **Schedule refactoring sessions** – After implementing features, dedicate time to clean up and optimize code.

2. **Use AI for code review** – Ask your AI assistant to analyze your codebase for duplications, inefficiencies, or potential bugs.
3. **Document architectural decisions** – Record why certain approaches were chosen to inform future development.
4. **Implement automated testing** – Even simple tests can catch regressions when making changes.
5. **Monitor performance metrics** – Track key indicators like load time and memory usage to identify optimizations.

Building a Restaurant Recommendation App with AI

Let's walk through how this process worked for building the restaurant recommendation app shown in my video:

Initial Concept and Requirements

I started with a basic idea: an app that recommends restaurants based on a user's mood and preferences. Using Gemini 2.5 Pro, I fleshed out this concept into a detailed PRD that included:

- Core features: mood-based filtering, restaurant browsing, favorites
- User flows: search, view details, save favorites
- Data requirements: restaurant information, user preferences
- MVP scope: focus on just restaurants first, with basic mood matching

Development Approach and Implementation

I demonstrated both approaches:

With Lovable (One-Shot Generation):

- Pasted the PRD into Lovable
- Generated a complete app in minutes
- Explored the generated code and UI
- Found it had created:
- A clean, functional UI
- Mock restaurant data
- Basic filtering functionality
- Simple "vibe matching" based on keyword matching

With Cursor (Guided Development):

- Set up a React Native project using Expo
- Created individual components for screens and functionality
- Built a backend with Express.js
- Implemented “vibe matching” using OpenAI
- Connected everything with proper API calls
- Fixed issues as they arose through debugging

Challenges and Solutions

Both approaches encountered issues:

- Endpoint mismatches between frontend and backend (fixed by aligning route paths)
- API key configuration (resolved by setting proper environment variables)
- Data sourcing (initially used mock data, with plans to integrate Google Maps API)

The Result

Within our session, we successfully built:

- A functional restaurant recommendation app
- The ability to filter restaurants by mood, cuisine, and price
- A simple but effective “vibe matching” algorithm
- A clean, intuitive user interface

The entire process took less than an hour of active development time, demonstrating the power of AI-assisted coding for rapid application development.

Best Practices and Lessons Learned

After dozens of projects built with AI assistance, here are the key lessons and best practices I’ve discovered:

Planning and Architecture

1. **Invest in clear requirements** – The time spent defining what you want to build pays dividends in AI output quality.
2. **Start simple, add complexity gradually** – Begin with a minimal working version before adding advanced features.
3. **Choose proven technologies** – Stick to widely-used frameworks and libraries for better AI support.
4. **Break down large features** – Decompose complex functionality into smaller, manageable pieces.

Working With AI

1. **Test after every significant change** – Don't wait until you've implemented multiple features to test.
2. **Don't blindly accept AI suggestions** – Always review and understand what the AI is proposing.
3. **Be specific in your requests** – Vague prompts lead to vague results.
4. **Keep track of the bigger picture** – It's easy to get lost in details; periodically step back and ensure alignment with your overall vision.
5. **Use version control religiously** – Commit frequently and create checkpoints before major changes.

Code Quality and Maintenance

1. **Document as you go** – Add comments and documentation during development, not as an afterthought.
2. **Implement basic testing** – Even simple tests help catch regressions.
3. **Refactor regularly** – Schedule time to clean up and optimize AI-generated code.
4. **Maintain consistent patterns** – Establish coding conventions and ensure AI follows them.
5. **Prioritize security** – Verify authentication, data validation, and other security practices in AI-generated code.

Conclusion: The Future of Development

We're experiencing a profound transformation in how software is created. AI code-generation and tools built on them are changing who can build applications and how quickly ideas can be turned into working software.

This doesn't mean traditional development skills are becoming obsolete. Rather, the focus is shifting from syntax mastery to system design, user experience, creative problem-solving, and effective AI collaboration. The most successful developers in this new landscape will be those who can clearly articulate their intent and effectively guide AI tools while maintaining a strong foundation in software engineering principles.

As you embark on your own vibe coding journey, remember that AI is a powerful collaborator but not a replacement for human judgment. Your creativity, critical thinking, and domain expertise remain essential. The tools will continue to evolve rapidly, but the process outlined in this guide (defining clear requirements, building incrementally, testing rigorously, and refining continually) will serve you well regardless of which specific AI assistants you use.

Now it's your turn to build something amazing. Start small, embrace the iterative process, and watch your ideas come to life faster than you ever thought possible.

Visual overview with Mermaid

```
flowchart TD
    A[Step 1: Define Concept + PRD] --> B[Step 2: Choose Tech Stack]
    B --> C{Approach}
    C -->|One-shot| D["Step 3a: Generate app (Lovable/Bolt/Replit)"]
    C -->|Guided| E["Step 3b: Init project (Cursor/Windsurf/Claude Code)"]
    D --> F[Step 4: Build/Adjust UI]
    E --> F[Step 4: Build UI]
    F --> G[Step 5: Core Functionality]
    G --> H[Step 6: Backend + Data]
    H --> I[Step 7: Test, Debug, Deploy]
    I --> J[Iterate + Refine]
```

```
sequenceDiagram
    actor U as You
    participant AI as AI Assistant
    participant IDE as IDE/Platform
    participant DB as DB/API

    U->>AI: Share concept, ask questions
    AI-->>U: Clarify and draft PRD
    U->>IDE: One-shot generate OR init project
    IDE-->>U: Scaffolds codebase
    U->>AI: Request UI components
    AI-->>U: Components and styling
    U->>DB: Connect to database/API
    DB-->>U: Data flows
    U->>AI: Debug help, fixes
    AI-->>U: Patches, guidance
    U->>IDE: Deploy
    IDE-->>U: Live app
```

Copy-paste prompts

```
Kickoff clarification prompt:
I'm planning to build [your app idea]. Help me flesh this out by asking questior
```

```
PRD request prompt:
Based on our discussion, please create a comprehensive PRD with:
```

```
1. Core features and user flows
2. Key screens/components
3. Data requirements
4. Technology considerations
5. MVP scope vs future enhancements
```

```
Tech stack eval prompt:
For the app we've described, recommend a tech stack that:
1) Enables rapid development
2) Has strong AI tool support
3) Scales reasonably
4) Stays simple for an MVP
Explain trade-offs and limitations.
```

```
Cursor project setup prompt:
Create a new Next.js TypeScript project, set up pages/components/API routes, cor
```

```
Core feature prompt:
Implement the recommendation feature with mood, cuisine multi-select, price slic
```

```
Backend prompt:
Define a Restaurant model, an API endpoint that filters by preferences, and a ba
```