How to develop your “pet project” using the power of LLMs. A practical guide for PMs and other non-coders

Author: Anton Kulikov (LinkedIn).

Example product: https://relistapp.app – PWA made with zero React knowledge.

Version: 1.01 – Dec 11, 2025.

Stage 0. Prerequisites

Lean and agile mindsets. Willingness to learn. A lot of spare time. Patience, grit, perseverance.

> Do your product discovery first.

Only after deciding to *actually* build something, invest in tools:

[December 2024]

- ChatGTP Plus: $20/month (you will need both `4o` and `o1` models)
- GitHub Copilot: $10/month (it will already have access to all ChatGTP models as well as Claude Sonnet 3.5 preview)
- VScode / Xcode, Android Studio: free

[/December 2024]

This guide is written from the perspective of building PWA in VScode using GitHub Copilot. Experience building a native app will be different in detail but should follow the same principles.

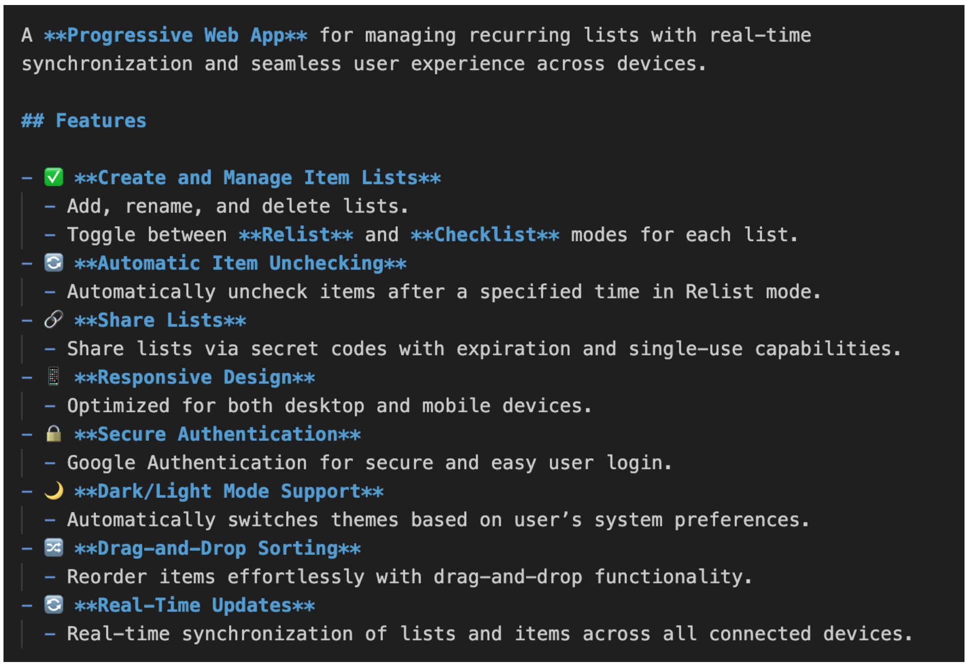
Stage 1. Preparation

Step 1. Describe the product

Define your product vision (one sentence) and feature set (the shortest list possible). Describe the intended value and anticipated user behavior; use any methodology you fancy, but be concise. You don't need to pitch the Chat the whole project or plan all upcoming features in detail right away.

Stay lean and prepare to act agile.

Ask the Chat to memorize this description. Save it in the text file (you will need it later). This also starts the habit of *self-produced documentation.*



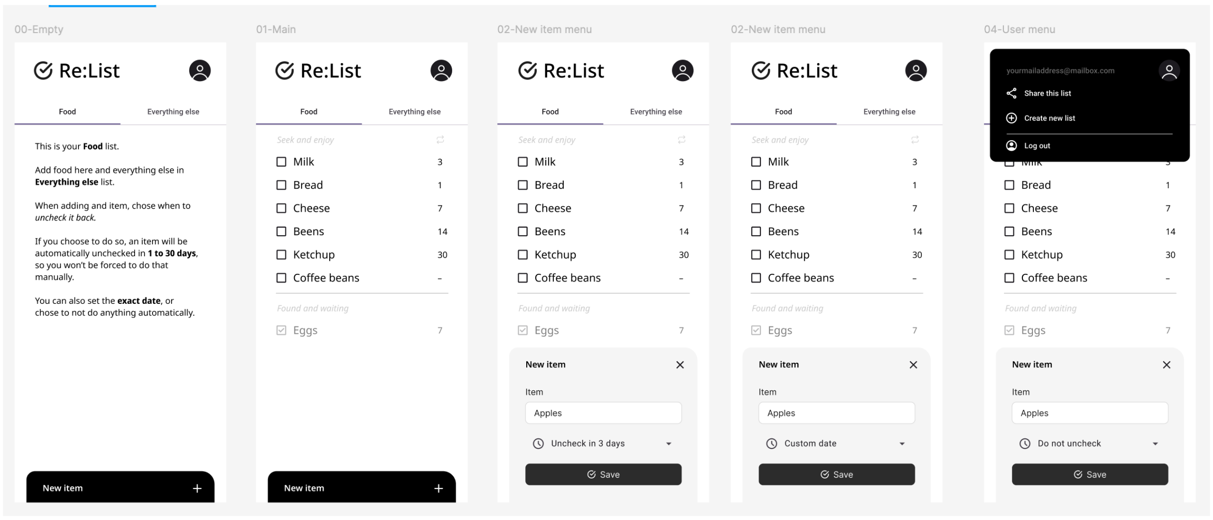
```
A **Progressive Web App** for managing recurring lists with real-time
synchronization and seamless user experience across devices.

## Features

- ✅ **Create and Manage Item Lists**
  - Add, rename, and delete lists.
  - Toggle between **Relist** and **Checklist** modes for each list.
- 🔄 **Automatic Item Unchecking**
  - Automatically uncheck items after a specified time in Relist mode.
- 🔗 **Share Lists**
  - Share lists via secret codes with expiration and single-use capabilities.
- 📱 **Responsive Design**
  - Optimized for both desktop and mobile devices.
- 🔒 **Secure Authentication**
  - Google Authentication for secure and easy user login.
- 🌙 **Dark/Light Mode Support**
  - Automatically switches themes based on user's system preferences.
- 🔀 **Drag-and-Drop Sorting**
  - Reorder items effortlessly with drag-and-drop functionality.
- 🔄 **Real-Time Updates**
  - Real-time synchronization of lists and items across all connected devices.
```

Part of my current README.md file

Prepare mockups of key screens and functions. Rough sketches will be enough. Don't waste your time polishing high-fidelity mockups – it's a rabbit hole down to a procrastination valley, don't follow that path.


This is your **Food** list.

Add food here and everything else in **Everything else** list.

When adding and item, chose when to *uncheck it back.*

If you choose to do so, an item will be automatically unchecked in **1 to 30 days**, so you won't be forced to do that manually.

You can also set the **exact date**, or chose to not do anything automatically.

The actual app looks... somewhat the same

Step 2. Define technologies and plan development

Ask the Chat to *memorize* that *you aren't a developer* and ask them to *explain any suggestions in detailed but simple language*.

Prompt the Chat to *assume the role of an experienced developer* and *make any upcoming suggestions considering the stability of technologies, existing alternatives, and ease of implementation*.

Feed the `4o` your product description and mockups and ask for suggestions for compatible technologies, outlining the pros and cons of each alternative.

Ask the `o1` to review suggested technologies, choose those that fit, taking into consideration product description, and suggest the most robust plan for implementing them.

Read both answers thoroughly (start developing the *habit of slow reading)*. Ask as many questions as you have until you think that you understand all the details.

Precautions: technology stack

If you are building a native app, your technology choice is minimal. If you are building a PWA or a web-based SaaS, there are many alternatives, so you should really understand their limitations before kickstarting the project. **The poor choice at this stage will be tough to mitigate later.** You will most probably have to restart the whole project. It happened to me twice.

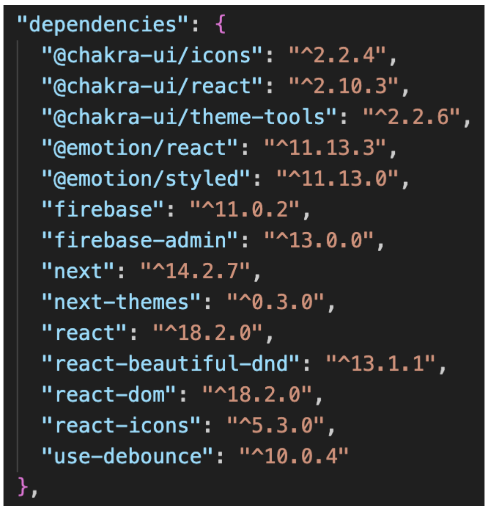
In the technologies list, **pay great attention to a visual components framework** you will be suggested to use. Go to that library website, look at your sketches, understand or define which components you will be using, and read about their features and behavior.

If you think a particular library lacks necessary components, doesn't support expected behavior, or doesn't look good enough – ask for more suggestions and examine them.

Never use the newest, the most recent version of any technology. Always stick to stable, preferably the LTS (long-term support) versions. Unfortunately, the state of the technologies is such that the newest versions of even established products has plenty of bugs and/or broken dependencies. As a non-programmer, you don't want to deal with that (programmers don't want to deal with that, either).

LLMs are trained on the most abundant code samples, hence, established technologies, and almost always generate code accordingly. You don't have time to retrain the model you are using to spell out the code for the latest version of your technology. If you struggle with the newest releases of your components library or framework, it will almost guarantee a "restart" of the whole project. It happened to me twice.

After making final choices, write them down in the same document with your product description. It will later become `README.md` file in your Git repository. (They will be also reflected as dependencies in your `package.json` file – you can always use it to refresh your memory)



```
"dependencies": {
  "@chakra-ui/icons": "^2.2.4",
  "@chakra-ui/react": "^2.10.3",
  "@chakra-ui/theme-tools": "^2.2.6",
  "@emotion/react": "^11.13.3",
  "@emotion/styled": "^11.13.0",
  "firebase": "^11.0.2",
  "firebase-admin": "^13.0.0",
  "next": "^14.2.7",
  "next-themes": "^0.3.0",
  "react": "^18.2.0",
  "react-beautiful-dnd": "^13.1.1",
  "react-dom": "^18.2.0",
  "react-icons": "^5.3.0",
  "use-debounce": "^10.0.4"
},
```

Step 3. Kickstart the project and build a “pipeline” to production

> This stage aims to get a functional page in the production environment.

Set up project

Ask `o1` to walk you through setting up the project.

Read instructions slowly and follow them to the t.

When you see the first terminal command – that is your clue to open the VScode.

Get through onboarding and install plugins for the technologies you will be using. By the way, ask `4o` for suggestions on plugins.

Open the project folder and terminal window inside VScode.

Continue following LLM’s instructions using the terminal to the t.

If you are building a PWA, at the end of this step, you should get the first functional page (based on your framework template) running in your local development environment (aka `localhost`).

Hold your horses, don’t start developing actual pages and features.

Set up the git repository and push all current changes to it first.

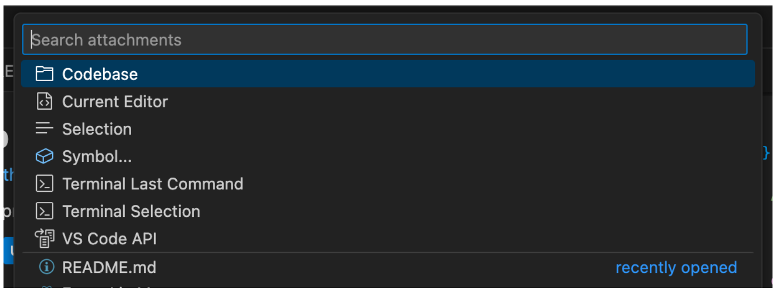
> The second goal of this stage is to set up a pipeline for pushing changes and seeing them in the production environment.

Interlude: Copilot – the best $10 you’ll spend this month

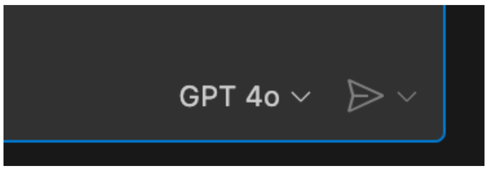
From now on, when you work with actual code, it’s more efficient to use Copilot.

Install two connected VScode plugins (GitHub Copilot, GitHub Copilot Chat), start your trial, and invest time in watching and understanding onboarding videos the team made. They are short and very useful.

Open the Copilot Chat panel and start using it. Understand how context works and how to add several files or the whole codebase to it.



Pay attention to which model you choose for each request. Since `o1` requests are highly limited, use them wisely. Never use them to implement actual changes.



Edit a README.md file in your project, adding your product description, features, behavior, and technologies. Use it as attachment (*add to context*) when asking `o1` to plan the next iteration/feature.

Push to production

Follow `o1`'s instructions on deploying your first (template) page into the production environment. If you didn't get such instructions, ask for them.

If you encounter any problems at this stage – ask `4o` for help. You don't need to spend limits on `o1` queries on simple stuff. Use `o1` only when you need to review, plan, and refactor. Or when encountering complex bugs and implementing complex behavior. If you are doing everything according to the principles of this guide, such occasions should be pretty rare.

At the time of writing this guide `4o` response time and code quality is noticeably depended on the current OpenAI load. Living in Europe I enjoy fast answers and good code quality when the North Americas are sound asleep and struggle with a noticeable degradation when heavy users wake up.

Make the slightest change possible on the page you already have (edit any sentence).

Ask `o1` how to ensure *end users get an updated version of the product*, when changes you are making will be pushed into a production environment. This is how you will learn about caching and version management.

Implement necessary techniques and bump the version before pushing the commit. Deploy to production.

If you are useing GitHub, ask `4o` for instructions on setting up GitHub Actions (GitLab should have something similar). **Your goal is to deploy to production on each successful commit to the repository**. Don't set up staging environments, feature flags, beta programs, or other state-of-art techniques yet. You will get to that stuff later, if at all.

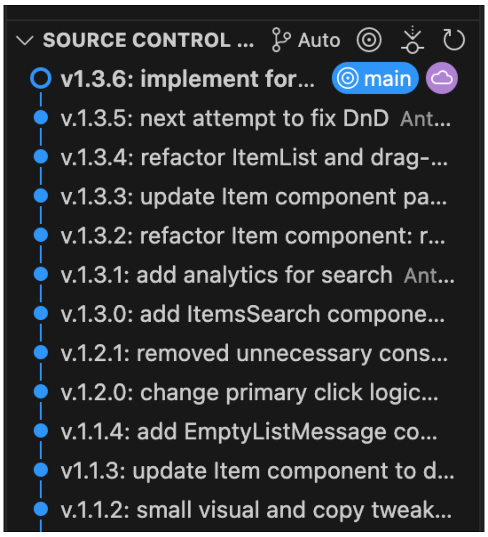
Interlude: Proper versioning and git practices

What you can and should do at this stage is to develop three connected habits:

1. work on one thing at a time;
2. make incremental commits;
3. write adequate commit messages describing actual changes – use the sparkle button to get help with that.

As a result of establishing these habits, you should push a new incremental version of your product to your repository each time. Ask `4o` on best practices for version numbering and follow them.

It might be handy to practice reverting both staged and committed changes, as well as other gitflow actions, but I don't suggest diving down in that rabbit hole unless absolutely necessary.



```
v1.3.6: implement for...
v.1.3.5: next attempt to fix DnD  Ant...
v.1.3.4: refactor ItemList and drag-...
v.1.3.3: update Item component pa...
v.1.3.2: refactor Item component: r...
v.1.3.1: add analytics for search  Ant...
v.1.3.0: add ItemsSearch compone...
v.1.2.1: removed unnecessary cons...
v.1.2.0: change primary click logic...
v.1.1.4: add EmptyListMessage co...
v1.1.3: update Item component to d...
v.1.1.2: small visual and copy tweak...
```

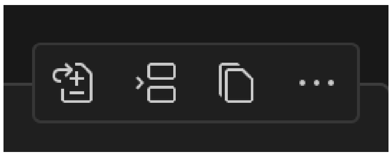
Stage 2. Iterative development

Step 2.1 First feature

Now, the first actual "development" task should be pretty easy: **display your app's version on the existing page.**

Ask `o1` or `4o` how to do that in GitHub Chat panel. Compare answers to better understand which model to use afterwards on which occasions. Follow the instructions.

You have three option to implement changes from Copilot, try them out.

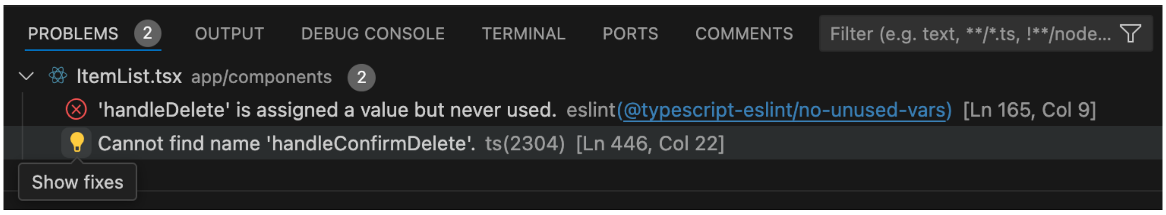


Fourth, slowest and most useful option, if you really want to understand what's going on, – copy particular part of the provided code, find related part of you file and insert it there. Repeat until all suggestions / all code is implemented.

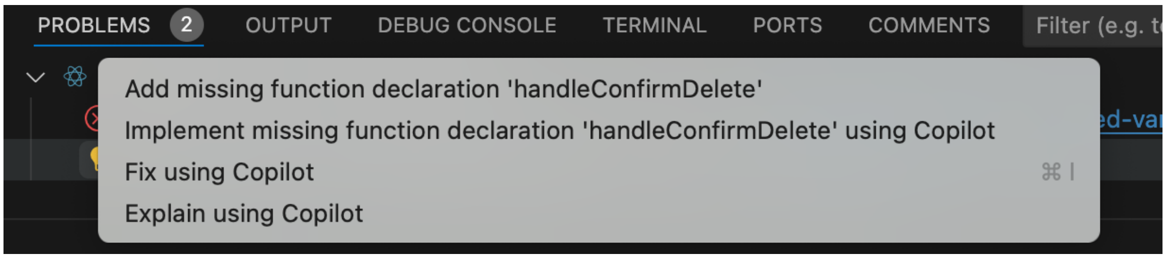
Interlude: Fixing code with LLMs

If you are using modern language like Typescript, the Problems panel is your best friend.

After each code change, pay attention to this panel, notice newly appeared errors and don't be shy of using "Fix using Copilot" option.



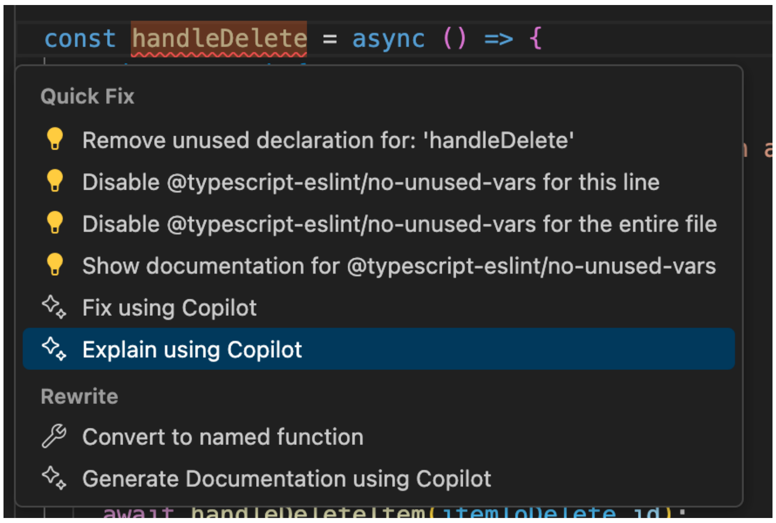
```
ItemList.tsx app/components 2
'handleDelete' is assigned a value but never used. eslint(@typescript-eslint/no-unused-vars) [Ln 165, Col 9]
Cannot find name 'handleConfirmDelete'. ts(2304) [Ln 446, Col 22]
```



PROBLEMS 2
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
COMMENTS
Filter (e.g. t
Add missing function declaration 'handleConfirmDelete'
Implement missing function declaration 'handleConfirmDelete' using Copilot
Fix using Copilot
⌘ I
Explain using Copilot

The biggest threat at this point is that Copilot might use the most simple, direct and *wrong* way of handling the problem. This will happen because of the lack of the context.

Two ways to deal with that are: “Explain” and “/fix” commands.



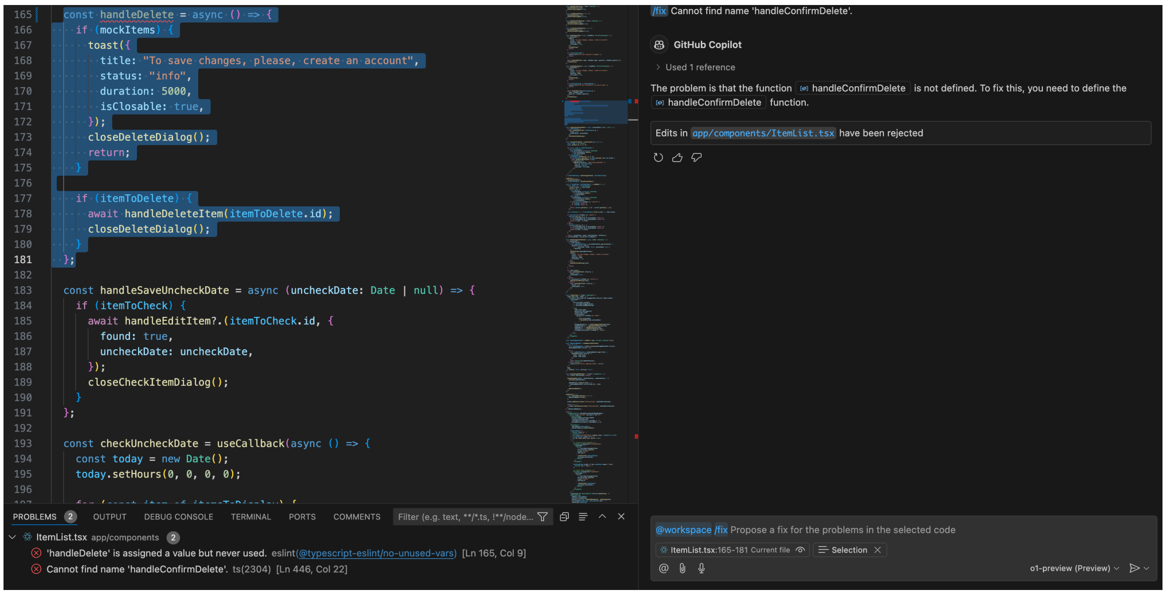
```
const handleDelete = async () => {
```

Quick Fix
Remove unused declaration for: 'handleDelete'
Disable @typescript-eslint/no-unused-vars for this line
Disable @typescript-eslint/no-unused-vars for the entire file
Show documentation for @typescript-eslint/no-unused-vars
Fix using Copilot
Explain using Copilot
Rewrite
Convert to named function
Generate Documentation using Copilot

First of all, use “Explain” option to understand what Copilot thinks about the problem and what does they know about it. This will help you to better understand your own codebase structure and what it actually does.

Second, more reliable option is to use /fix command in Copilot Chat panel.

This is how the typical situation looks like: I’ve selected the erroneous part of the code, typed `/fix` command and added “selection” to the copilot’s context.



```
const handleDelete = async () => {
  if (mockItems) {
    toast({
      title: "To save changes, please, create an account",
      status: "info",
      duration: 5000,
      isClosable: true,
    });
    closeDeleteDialog();
    return;
  }

  if (itemToDelete) {
    await handleDeleteItem(itemToDelete.id);
    closeDeleteDialog();
  }
};

const handleSaveUncheckDate = async (uncheckDate: Date | null) => {
  if (itemToCheck) {
    await handleEditItem?.(itemToCheck.id, {
      found: true,
      uncheckDate: uncheckDate,
    });
    closeCheckItemDialog();
  }
};

const checkUncheckDate = useCallback(async () => {
  const today = new Date();
  today.setHours(0, 0, 0, 0);

```

```
ItemList.tsx app/components 2
  'handleDelete' is assigned a value but never used. eslint(@typescript-eslint/no-unused-vars) [Ln 165, Col 9]
  Cannot find name 'handleConfirmDelete'. ts(2304) [Ln 446, Col 22]
```

/fix Cannot find name 'handleConfirmDelete'.

GitHub Copilot

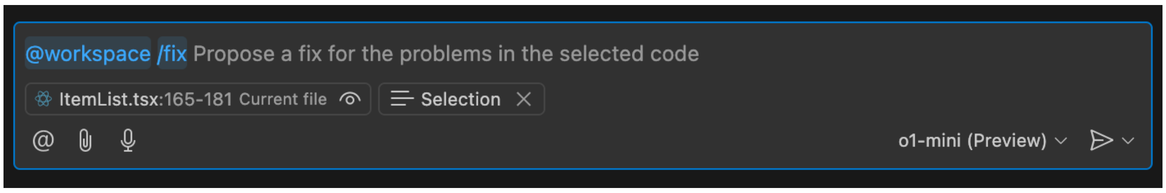
Used 1 reference

The problem is that the function handleConfirmDelete is not defined. To fix this, you need to define the handleConfirmDelete function.

Edits in app/components/ItemList.tsx have been rejected

@workspace /fix Propose a fix for the problems in the selected code

This time the `o1-preview` is selected, and you should already understand why for such easy case it is unnecessary.


@workspace /fix Propose a fix for the problems in the selected code
ItemList.tsx:165-181 Current file
Selection
o1-mini (Preview)

After Copilot produced the code,
read it slowly, understand each suggestion and implement them by choosing one of the options.

Your code is ready to be deployed only when there are no Problems. The tricky thing is that errors are displayed in the Problems panel only for files that are opened or directly affected by your recent changes.

To make sure that all codebase is errors-free, use build command for your project. (By this point, you should be already aware of it).

Do you need to understand the code? Tricky question! You don't need to understand the syntax, but you will have to understand what each part of your code does.

If the Copilot suggest you to change the name of your function and include some new parts in it – in most cases it is beneficial if you understand what's going on.

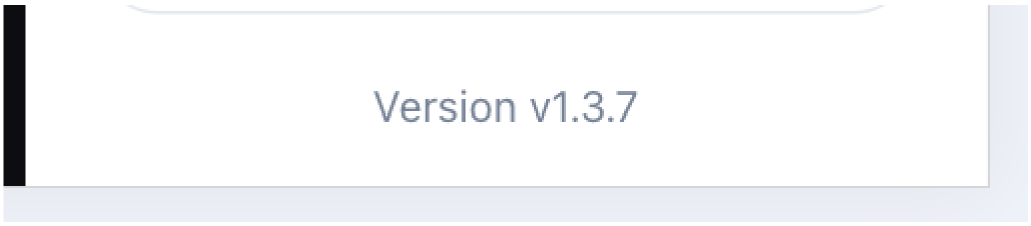
You can spend sometime implementing LLMs suggestions blindly, but at some point they won't understand context, will decide to reimplement something already existing or just drop something that you already have in our code out of the new version of the same file. You'll have to catch that as soon as it happens. **You have to be diligent in that regard. To some people, this point defeats the whole purpose.**

> As long as a model can't hold the whole context of your app (ideas, description, code, internal and external

> dependencies, and the current task) – you have to do that yourself.

To recap, after making minor changes, building you project, committing to the repository and triggering GitHub actions you should get a freshly updated functional page in production environment.

Only after that you should develop your first feature (display app version) and learn how to deal with versioning and caching along the way.



By the time you read this, my app will be well over v1.5

Only after succeding with that, proceed to next stages.

If you can't get through this part – give up and hire someone. Or use other tools. Or wait for next 6 months and try again (I used the last option last May).

Step 2.2 Develop the ~~first page~~ user authentication process

> This step aims to confirm that the technologies and libraries you've chosen – fit.

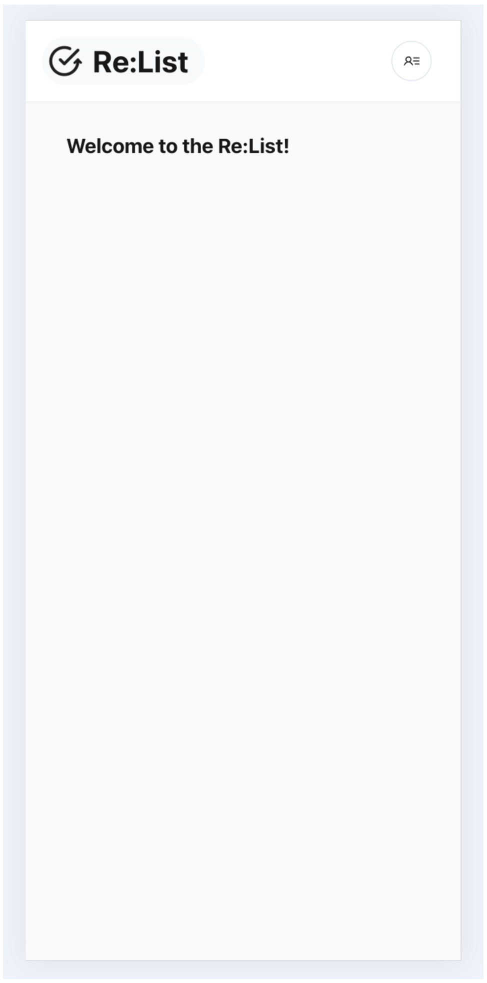
At this step, you should have switched to VScode/GitHub Copilot, so return to ChatGTP only when you need to use images as inputs.

Now it's time to see if the setup you've chosen works for your particular task. The easiest way to do that is to start by developing user authorization process. It might sound counter-intuitive for a product manager (since it doesn't directly connected with actual value of your product), but, please, read the following rationale.

Your goal at this point is still to verify that you can build something functional and practical using the only tools that are available to you and your current skills. You also still need to verify that libraries you've chosen are compatible with your projects goals and desired esthetics.

The smallest increment you can ship at this moment to verify both – is user authorization. It will be challenging enough so you can confidently say afterwards that you are capable of developing more complicated stuff. It will be challenging enough to understand limitations of your tech stack and components library. It will be visual enough for you to understand if the library you've chosen is agile and versatile to implement the look you were aiming for.

So, instead of implementing the home page or the primary screen of your app/website – start by implementing a very simple login form and empty landing page that will be shown to authorized users.



Welcome to the Re:List!

An imitation of the end result of this step. Yours should be even simpler.

Overall the result of this step should produce either "hell, yeah!" or "ffffk, no…" type of reaction out of you.

That will be an actual hard evidence if you should continue down this path at all. Believe me, it's soul crashing to spend hours building you "actual" product (or rather it's visual representation), and ending up not understanding how to bring it to live (btw, you will most definitely need user auth as part of that). So, always start with something actually working, even if it's not visually appealing or "valuable" – if you will need it later, and you can polish it later, if you keep going at all. Here is when your grit and perseverance comes in handy.

As always, ask `o1` to produce a comprehensive plan first. Read it, ask questions.

The ever going loop of "think-change–fix–check–fix...deploy–check–fix–..." starts here (and never ends, it could be only halted).

Use fixing code with LLMs techniques, make sure to work on one thing at the time and structure your commits and deploys accordingly, but for the sake your own sanity, ask for a plan first (sorry for repetition, but get used to it if you want to succeed), understand suggestions and only then implement changes.

Optional interlude: refactoring

In my practice, LLMs aren't actually thinking like an experienced programmer at this point (December 2024). Hence, they have a tendency to not follow actual best practices (as I imagine them) and as a result you might end up with overcomplicated oversized code files.

It's up to you to decide, if and when you will split your code into smaller interconnected chunks (modules). I'm not an software engineer, so I don't have a proper opinion about that, only practical one.

While you rely on an LLMs to produce changes in your code it's practical to have somewhat long files – when you feed them to the copilots they better understand the context.

At the same time, it take considerable amount of time for LLMs to suggest changes for big files, since they have to process them first, and then spell out changes accordingly (especially if you instruct them to spell out the whole file code – prompt I heavily relied upon at the first stages).

> Two practical advices here:
>
> – occasionally, you should ask a model to review the code and suggest how to refactor it into separated modules/components. The actual prompt doesn't matter that much, any copilot will understand your intention and will suggest something. Take that suggestion with a grain of salt and decide for yourself, if you'd like to spend extra time implementing them.
>
> – you should do that only after you get a working increment in the product environment (hence, committed to repository). This is you only chance to stop yourself midway in a major refactoring implementation and easily revert all half-finished changes back. Trust me, sometimes you will need that option.

Stage 3. Follow the plan, step by step. Get feedback early

Remember the readme file you've compiled early? What are the actual steps written in it?
Have you divided your product into sizable pieces?

My point here is simple: build your product by small, but usable increments. If you can use something yourself, ask someone to use it as well. Here their feedback, you will be surprised.

The early you start showing `0.2` or `0.3` versions of your product to actual potential users, the quicker you learn. And you will learn a lot.

> After being a product manager for well over a decade, I cannot express how fresh was the feeling of putting something half-baked in the open and immediately hearing actual unfiltered feedback.

This problem is well known, but rarely really understood until experienced personally: *you actually fall in love with your ideas*, and you are so blinded by them, you can't see their weaknesses. Even if those weaknesses can be described as "subjective" – if you build your product for people out there – bring your product to people out there. This is truly the only way to get a feedback from reality – which should be your utmost priority at any point.

So, now, after building and successfully using user authorization, start by implementing the most basic feature there is in your product. This will be the second, not biggest, but probably most important hump in your journey.

If you overcome it, the sky is the limit.

Stage 4. Pre-release: proper refactoring & optimizations

If you reach this stage, it's time to set up a dedicated staging environment, feature flags, and beta users program. You can even decide to conduct a *proper* code refactoring.

At this point, you should be skillful enough that you don't need my instructions on how to set up all of that.

If you need to understand how to set up analytics, implement monetization options, write T&C, or implement a user help bot powered by ChatGTP – you don't need my advise – there is an LLM for that.

My last suggestion will echo something I've alluded to many times before in this guide: don't be caught in a rush of implementing, changing, developing something right away – read slowly, consider thoroughly and act only after that. Contrary to that – release as quickly as possible!

Seeing the actual results of your work is much more important than feeling good about the process.

P.S.

As you can see just by reading this guide – the bigger part of the content, the most useful and generally applicable information – it's all about starting your development and getting first, actual, real-world results. After that, even though they all start the same way, each path leads to it's own end.

I've written this guide to encourage you to try. My first two attempts went sideways, but the third one produced https://relistapp.app – shared shopping list for recurrent purchases (or, if you prefer, shared recurrent tasks app).

I've started using it on the first day of actual development, when it was just a list of checkboxes. I changed core mechanics at least twice since then – and not for the lack of pre-considerations, but, actually, because of having to many of them.

Now, I know that my next attempt will produce something functional and useful, and I'm confident that yours will to.

If you have any questions regarding coding with LLMs or suggestions on how to improve this guide, connect with me on LinkedIn and we'll figure something out.

Now, Go Forth and Create the Art!

Good luck!