The Developer's Handbook for AI-Augmented Workflows

By

Murat Karakaya

Chapter 1: The Principle of Context in AI-Assisted Development

Understanding Copilot's Context Engine: The Shift from Implicit to Explicit

The effectiveness of any AI assistant, particularly one as deeply integrated into the development workflow as GitHub Copilot, is fundamentally dependent on the quality and relevance of the context it receives. The core of this relationship lies in understanding the distinction between implicit and explicit context. Implicit context is the default, passive mode of operation that a developer experiences the moment they begin using Copilot. The system automatically examines the code in the editor, focusing on the lines immediately surrounding the cursor, but also taking into account other files open in the IDE and general workspace information such as frameworks, languages, and dependencies [1, 2]. For basic code completion, this passive, ambient awareness is often sufficient. When a developer types a function header, Copilot can accurately infer the intent from the function's name and the surrounding code, offering a ghosted suggestion for the entire function body [3]. This capability, while powerful, represents only the most superficial layer of Copilot's potential.

To unlock the full suite of advanced features and move beyond simple code completion, a developer must transition from relying solely on implicit context to actively providing explicit context. This is the central thesis of this handbook: the most successful AI-assisted workflows are those where the developer takes a proactive role as a "prompt engineer" and workflow orchestrator. This involves intentionally steering the AI toward a helpful output [1]. Instead of simply reacting to code, the developer can provide a high-level overview of the project, define the technology stack, and even spell out specific coding guidelines. This deliberate act of providing context transforms the AI from a passive bystander that merely completes code into an active collaborator that understands the project's architecture and adheres to its conventions. This shift from a reactive to a proactive approach is what distinguishes a beginner's use of Copilot from an expert's, turning a simple tool into a genuine partner in the development process.

The "Copilot, Not Autopilot" Principle

GitHub's foundational philosophy for this technology is encapsulated in the product's name itself: "Copilot," not "Autopilot" [2]. The tool is not designed to fully automate code generation or replace the developer's oversight [2]. Instead, its purpose is to augment the development process, freeing the developer from mundane, repetitive, and boilerplate tasks so they can focus on higher-level activities like architectural design, complex problem-solving, and innovation [1, 2]. This principle underscores the importance of the human-in-the-loop, who remains responsible for the quality, security, and strategic direction of the codebase [2]. The effectiveness of Copilot is not measured by its ability to generate code without supervision, but rather by its capacity to accelerate development without sacrificing quality.

The evolution of the tooling demonstrates a fundamental shift in the developer's role. Traditional development largely revolves around writing code line-by-line. With advanced features like agent mode, which can plan and execute multi-file changes across a codebase, the developer's primary task shifts from writing the code itself to writing precise, structured instructions for an AI agent [4, 5]. This requires a different skillset—one focused on decomposing complex problems into smaller, more manageable tasks and communicating requirements clearly and unambiguously [6]. The human is not being replaced but is being elevated to a new role as a strategic enabler and supervisor, delegating the tactical work to the AI [2]. The human's strategic decisions and oversight become the most valuable contribution.

This dynamic creates a new form of living project documentation. The files and prompts a developer creates to provide context to the AI become a codified representation of the project's standards and workflows. This is not a passive artifact for human collaborators; it is an active, operational layer that directly influences the development process [7]. The structured nature of instructions files, for example, forces a team to articulate its project's elevator pitch, technology stack, and coding guidelines in a clear, machine-readable format. By codifying this knowledge in a dedicated location like the .github/ folder, a team establishes a single source of truth that enables consistent, high-quality output across the entire project lifecycle, ensuring that both human and AI collaborators are operating from the same foundational knowledge base [8].

Chapter 2: Structuring Your Project for AI Collaboration

The .github/ Folder: A Central Hub for Shared Instructions

The .github/ directory is a well-established convention in the software development community. It serves as a centralized hub for project-level configurations and workflows, most notably for GitHub Actions. Expanding on this established practice, the .github/ folder has become the recommended location for storing AI-specific instructions, making these guidelines a core, version-controlled part of the repository [8]. By placing all AI-related files within this directory, a team ensures that every contributor, whether human or AI, can access the same foundational context. This standardization is crucial for maintaining consistency, especially in larger, collaborative projects.

Crafting the copilot-instructions.md File: A Project's DNA for the AI

The copilot-instructions.md file is a singular, powerful tool for providing high-level context to GitHub Copilot. When placed at the root of a workspace within the .github/ directory, this Markdown file is automatically applied to all chat requests, serving as a project's "DNA" for the AI [8]. Instead of repeating the project's details in every prompt, a developer can define a set of persistent instructions that guide Copilot's responses across the entire codebase.

Official documentation recommends structuring this file with five key sections to provide the most effective context [9]. These sections are not merely suggestions; they form a comprehensive framework for communicating a project's essence to an AI.

1. **Project Overview:** The file should begin with an "elevator pitch" that concisely describes the application, its audience, and its high-level features [9]. This helps Copilot understand the overall purpose and goal of the project.
2. **Tech Stack:** A clear, itemized list of the frameworks, languages, and tools used in the project is essential [9]. This information enables Copilot to generate code that is

consistent with the project's environment.
3. **Coding Guidelines:** This section is used to spell out specific coding standards that should be followed across the codebase [9]. This includes rules like whether to use semicolons in JavaScript or type hints in Python, ensuring generated code aligns with team conventions.
4. **Project Structure:** A high-level description of the directory layout helps Copilot understand where different components are located [9, 10]. This is crucial for multi-file tasks where the AI needs to navigate the codebase.
5. **Available Resources:** Pointers to external resources, such as custom scripts or APIs, can be included to guide Copilot toward the correct tools for specific tasks [9].

An example of a well-structured copilot-instructions.md file for a hypothetical Firebase web application might look like this:

GitHub Copilot Instructions

Project Overview

This project is a task management web application. It allows users to add, delete, and mark tasks as completed. The application is a single-page app (SPA) that will be deployed to Firebase Hosting. The target audience is individual users who need a simple, fast tool for managing daily tasks.

Tech Stack

- Frontend: HTML5, CSS3, ES6 JavaScript
- Styling: Standard CSS with a focus on responsive design and semantic elements.
- Backend/Database: Firebase Firestore for real-time data synchronization.
- Authentication: Firebase Authentication (email/password and Google Sign-In).
- Hosting: Firebase Hosting.
- Deployment: GitHub Actions for CI/CD.

Coding Guidelines

- **HTML:** Use semantic HTML5 elements. Ensure accessibility with appropriate ARIA attributes.
- **JavaScript:**
  - Use modern ES6+ syntax (e.g., const, let, arrow functions, async/await).
  - Use strict mode.
  - All variables and functions should be camelCase.
  - Always use semicolons.
- **CSS:**
  - Use CSS variables for color themes.
  - Implement a mobile-first responsive design.
  - Use a BEM-like naming convention for classes.

Project Structure

- /: Root directory containing index.html.
- /src/: Source code directory.
  - /src/js/: JavaScript modules for application logic.
  - /src/css/: CSS files.
- /public/: Publicly accessible files for deployment (managed by the build process).

Available Resources

- firebase.json: Contains Firebase Hosting and Functions configuration.
- .firebaserc: Specifies the Firebase project alias.
- A GitHub Actions workflow for deployment is located at .github/workflows/deploy.yml.

By investing a small amount of time upfront to define these standards for the AI, a team can significantly improve long-term code quality and reduce friction in the development process. The existence and standardization of these files point to the emergence of "AI governance" within a project. A team can ensure that every developer, and every AI agent, is operating from the same foundational knowledge base, preventing inconsistent code and style drift. The

.github/ location reinforces this by making the AI instructions a core, version-controlled part of the repository, just like a README.md or a CONTRIBUTING.md.

Copilot Instructions File: Key Sections & Purpose

Section	**Purpose**	**Example Content**
Project Overview	Provides high-level context and goals.	Elevator pitch for the application. Audience and key features.
Tech Stack	Lists frameworks, languages, and tools.	Backend: Node.js, Express, MongoDB. Frontend: React, Redux, SASS.
Coding Guidelines	Defines conventions and standards.	Always use semicolons. Use camelCase for variables.
Project Structure	Explains the folder and file layout.	api/: API routes. components/: Reusable UI elements.

The Power of Reusable Prompts: Using the prompts/ Subdirectory

While instructions.md files provide a broad, project-wide context, .prompt.md files offer a more granular, task-specific form of contextualization. These are standalone Markdown files that define reusable prompts for common development workflows [7]. Unlike instructions that are automatically applied to all chat requests, prompt files are triggered on-demand, making them ideal for creating a library of standardized development tasks [7].

A critical distinction exists between instructions and prompts. Instructions define the *rules* of engagement, detailing *how* a task should be done (e.g., "always use type hints"). Prompts, conversely, define the *task* itself, detailing *what* needs to be done (e.g., "generate unit tests for this function") [7, 11].

By creating a prompts/ subdirectory within the .github/ folder, a team can establish a shared repository of best practices. For instance, a developer could create a file named

.github/prompts/test.prompt.md with the following content:

---

description: Generates Jest unit tests for the selected code block. mode: edit

You are a testing expert. Your task is to generate comprehensive unit tests for the selected code.

1. Analyze the code to identify all possible edge cases and valid inputs.
2. Use the Jest testing framework to write tests that cover the full functionality.
3. The tests should be well-commented and easy to understand.
4. Do not include a test runner. Only provide the test code.
5. If the code is a function, mock any external dependencies.

Example of a test:
const { add } = require('../math');
describe('add', () => {
test('should add two positive numbers correctly', () => {
expect(add(1, 2)).toBe(3);
});
});
To run this prompt, a developer simply types /test in the chat input field, and Copilot will execute the instructions with the currently selected code block as context [7]. This saves time and ensures a consistent approach to a common task. Other examples could include document.prompt.md for generating documentation or refactor.prompt.md to optimize a code block for performance [12]. The ability to create, store, and share these files across a workspace allows for a more standardized and efficient workflow, ensuring that the entire team benefits from these collective best practices.

Chapter 3: Dynamic Context and Multi-File Workflows

In-Editor Contextualization: # Mentions and Slash Commands

While the copilot-instructions.md and prompts/ files provide a static, project-level context, Copilot also offers powerful dynamic tools for on-demand contextualization. These features, known as # mentions, enable a developer to explicitly reference specific files, codebases, or changes within a chat conversation [13]. This capability moves the AI from a passive observer to an active participant, as the user is intentionally telling it what to think about.

The different types of # mentions serve distinct purposes:

- **#file**: This is used to provide the content of a specific file as context for a prompt. For example, a developer could ask, "Explain the logic in #src/api/auth.js" to receive a summary of the authentication service [13].
- **#codebase**: This mention is a powerful tool that performs a semantic search across the entire workspace [13]. A developer could use it to ask, "Fix the issues in the authentication flow #codebase" to prompt the AI to analyze the entire repository for relevant information and suggest a fix. This feature allows the AI to operate with a deep understanding of the project's overall architecture [13].
- **#changes**: This reference is used to provide the diffs of changed files as context [13]. It is particularly useful for tasks like summarizing pending changes for a commit message or generating a pull request summary [14].
- **#githubRepo**: A developer can even provide a remote repository as context to get information about its code, structure, or conventions [13]. An example prompt could be, "How does routing work in Next.js #githubRepo vercel/next.js" to get information directly from the source [13].

These # mentions are complemented by a suite of slash commands that help a developer quickly set the intent for common tasks [15]. Commands like /explain to get a code explanation or /refactor to reformat a block of code enable a more efficient, conversational workflow [12]. The combination of these tools allows for a more directed and intentional interaction with the AI. This is a causal chain: explicit context from # mentions enables more directed prompts, which in turn unlocks autonomous, multi-step actions from the AI. The end result is a more collaborative and efficient development process, as the AI becomes a true partner rather than just a passive code-completer.

Harnessing Agent Mode for Complex, Multi-File Tasks

Agent mode represents Copilot's most powerful, autonomous capability [4, 5]. Unlike the standard chat, which is designed for one-off questions and code snippets, an agent can autonomously plan and implement complex, multi-file tasks [2, 3]. The developer's role shifts from writing the code to providing a high-level goal and then acting as "mission control," reviewing and approving the agent's proposed plan and subsequent changes [10].

A classic example of agent mode's power is the creation of a complete front-end for a web application from a single, natural-language prompt. A developer could enter a prompt like, "Create a complete task manager web application with the ability to add, delete, and mark tasks as completed. Include modern CSS styling and make it responsive. Separate markup, styles, and scripts into their own files" [5]. The agent will then analyze the request and begin to implement the solution, coordinating changes across multiple files. It will likely create an index.html file, a styles.css file, and a script.js file, demonstrating its ability to understand high-level requirements and translate them into working, multi-file code [5]. Agent mode excels at implementing new features, refactoring large sections of code, and building entire applications from scratch [5].

Connecting to the Broader Ecosystem with the Model Context Protocol (MCP)

The Model Context Protocol (MCP) is the underlying technology that enables Copilot to "hook into" external resources, effectively allowing it to act like an "onboarded team member from day one" [2]. This protocol extends Copilot's capabilities beyond the local codebase, connecting it to external knowledge bases, APIs, and, most importantly, cloud services. The existence of the MCP demonstrates a significant trend toward Copilot becoming a universal interface for software development, linking not just to code but to documentation, cloud services, and entire CI/CD pipelines.

The Copilot for Azure extension is a prime example of the MCP in action [16]. This extension provides Copilot with a rich set of tools for interacting with Azure services [16]. With the extension installed, a developer can use a prompt like, "Deploy a local Python Flask app to Azure" and the agent will use the #azure_azd_up_deploy tool to provision the necessary resources and deploy the application [16]. This capability fundamentally changes what "coding" means in the age of AI. The "code" is no longer just application logic; it can be Infrastructure as Code (IaC), CI/CD pipelines, or any other part of the development stack. The AI becomes the "lingua franca" that connects a developer's intent to a vast array of tooling, from local files to global cloud services.

Copilot Context References: # Mentions and Their Use Cases

Reference Type	**Example Use**	**Functionality**
#file	/explain #src/index.js	Provides contents of a specific file as context.

#codebase	Fix bugs in the authentication flow #codebase	Allows for semantic search across the entire workspace.
#changes	Summarize the changes in #changes	Generates summaries of staged or unstaged changes.
#githubRepo	How does routing work #githubRepo vercel/next.js	Provides context from a remote GitHub repository.
#azure_query_learn	How to migrate a database to Azure? #azure_query_learn	Gets documentation from Microsoft Learn to answer queries.
#azure_azd_up_deploy	Deploy the app to Azure #azure_azd_up_deploy	Provisions Azure resources and deploys the project.

Chapter 4: The Developer's Handbook: From Idea to Code

Project Kick-off with Copilot: Architecture and Design

The software development lifecycle begins long before a single line of code is written. It starts with an idea, which must be refined into a concrete architecture and design. Copilot can be a valuable partner in this initial, high-level phase. A developer can use Copilot Chat to generate a project architecture, design documents, or even a draft of the copilot-instructions.md file itself [9, 12, 17]. For example, a prompt such as, "Create a high-level system design for a web application that stores and retrieves user profiles," can yield a detailed overview of the necessary components, data models, and API endpoints. Once the high-level design is in place, agent mode can be used to generate the project's initial boilerplate [5]. This capability allows a developer to rapidly move from a conceptual idea to a working foundation, accelerating the entire development process.

Guided Code Generation and Refactoring

Once the project's foundation is in place, a developer can use Copilot's core features to accelerate coding. For routine tasks, Copilot provides inline code completions as a developer types, suggesting anything from a single variable name to a full function body [1, 3]. This works best when the code is well-structured and the variable names are meaningful [11].

For more complex tasks, the developer can guide Copilot using natural language comments or chat prompts. A comment like, // Create a REST API endpoint for user authentication, can prompt Copilot to generate the necessary code for a full Express.js route [3]. When a more significant change is required, the developer can leverage the chat interface and # mentions to perform multi-file refactoring. For instance, a prompt like, Refactor the authentication service to use async/await throughout #src/auth.js #src/api.js, can guide Copilot in making coordinated changes across multiple files, ensuring consistency and adherence to modern coding practices [6, 12].

Writing and Validating Tests with the AI

Writing tests is a foundational best practice that can often be tedious and time-consuming. Fortunately, it is one of Copilot's best-supported use cases [1, 12]. By leveraging a dedicated .prompt.md file, a developer can ensure a consistent approach to testing across the entire codebase. A prompt file can be created to generate comprehensive unit tests for a selected code block or function, including edge cases and valid inputs.

Copilot's capabilities extend beyond test generation. The AI can assist in debugging failing tests by providing contextual explanations and code suggestions to fix bugs [1]. A developer can paste a failing test log into the chat and ask Copilot to "diagnose this log" or "fix this issue," allowing for rapid troubleshooting and resolution.

Documenting the Codebase for Future Collaboration

One of the most valuable, yet often neglected, aspects of a project is its documentation.

Copilot can be a powerful ally in this effort. By analyzing the context of the code, Copilot can generate meaningful comments, docstrings, and explanations to help a developer understand complex logic and dependencies [17]. This documentation serves a dual purpose: it not only clarifies the codebase for future human collaborators but also enriches the AI's internal context, improving the quality of its suggestions in future interactions [17].

A developer can simply highlight a function and use a chat prompt like, "Write a docstring for this function" to generate a summary of its inputs, outputs, and purpose [17]. For more complex sections of code, inline comments can be added to clarify tricky logic or describe what a larger chunk of code does [17]. At a higher level, Copilot can also assist in crafting a high-level README that explains the system's purpose and architecture [17]. This makes documentation a living, integrated part of the development process rather than a static artifact.

Chapter 5: Automating the Pipeline: Deployment and Operations

Infrastructure as Code (IaC) with Copilot

Infrastructure as Code (IaC) is the practice of managing and provisioning computing infrastructure through machine-readable definition files rather than through manual configuration [18]. This practice simplifies provisioning, ensures repeatability, and makes scalability and change management far easier [18]. The move toward codifying infrastructure naturally extends the domain of AI-assisted development, as the AI can now assist with the entire operational stack.

Copilot can help generate IaC files, such as Bicep or Terraform templates, for provisioning cloud services [16]. A developer could use a prompt like, "Create a Bicep template to deploy a MySQL database in the 'West US 3' region" to generate the necessary configuration file [16]. The Copilot for Azure extension, powered by the MCP, is particularly adept at these tasks, with tools designed to generate Azure CLI commands, query resource graphs, and check for region availability before deployment [16].

Case Study: The "Idea to Deploy" Pipeline for a Firebase Web App

To demonstrate a complete "idea to deploy" workflow, this section outlines a hands-on example for deploying a Firebase web app using GitHub Actions and Copilot.

Step 1: Setting up GitHub Actions with Copilot

The process begins with creating a CI/CD pipeline using GitHub Actions. A developer can use Copilot Chat to generate the initial workflow file, .github/workflows/main.yml, by providing a natural language prompt such as, "Create a GitHub Actions workflow to build and deploy a web application to Firebase Hosting whenever a commit is pushed to the main branch."

Step 2: Automating Build, Test, and Deployment

The workflow will typically consist of two jobs: a build job and a deploy job. The build job is responsible for checking out the repository, installing dependencies (e.g., npm install), and creating a production-ready build (npm run build). The deploy job depends on the successful completion of the build job. It downloads the build artifacts and uses a GitHub Action for Firebase to deploy the application [19].

Authentication for the deployment is critical. The recommended approach is to use a GCP_SA_KEY or FIREBASE_TOKEN stored as a GitHub Secret [19, 20]. This ensures that sensitive credentials are not exposed in the codebase. The deploy-firebase action then uses this credential to authenticate with Firebase and deploy the application [20].

A fully commented example of a GitHub Actions workflow for a Firebase web app is provided below.

YAML

```yaml
name: Build and Deploy to Firebase

on:
 push:
   branches:
     - main

jobs:
 build_and_deploy:
   runs-on: ubuntu-latest
```

```
  steps:
    - name: Checkout Repository
      uses: actions/checkout@v4

    - name: Set up Node.js
      uses: actions/setup-node@v4
      with:
        node-version: '20' # Or any other LTS version

    - name: Install Dependencies
      run: npm install

    - name: Build Application
      run: npm run build

    - name: Deploy to Firebase
      uses: w9jds/firebase-action@v2
      with:
        args: deploy --only hosting
      env:
        # Use a GitHub Secret to store the Firebase token or GCP service account key.
        # The GCP_SA_KEY is the recommended method.
        FIREBASE_TOKEN: ${{ secrets.FIREBASE_TOKEN }}
        # Alternatively, for more granular permissions, use a GCP Service Account Key.
        # GCP_SA_KEY: ${{ secrets.GCP_SA_KEY }}
```

This workflow demonstrates how a developer can move from a local development environment to a fully automated, production-ready pipeline, all with the assistance of an AI.

Troubleshooting the CI/CD Pipeline with Copilot

When a CI/CD pipeline fails, diagnosing the issue can be a complex and time-consuming process. Copilot can be a valuable partner in this troubleshooting phase. A developer can copy the error logs from a failed GitHub Actions run and paste them directly into the Copilot Chat interface [12]. By prompting Copilot with a request like, "Diagnose this GitHub Actions log and suggest a fix," the AI can analyze the log, identify the root cause of the failure, and provide a clear explanation and code suggestion to resolve the issue. This capability accelerates the debugging process and allows a developer to maintain a continuous, rapid workflow.

The fact that the AI can assist with troubleshooting the CI/CD pipeline and generating IaC files demonstrates a significant evolution in the augmented developer's role. The concept of "coding" has expanded beyond application logic to encompass the entire operational stack. As processes like infrastructure provisioning and deployment become codified in machine-readable formats like YAML, they fall within the domain of AI assistance. This development blurs the lines between traditional development and operations roles, empowering a single developer to manage the entire lifecycle of a project from a single interface.

GitHub Actions for Firebase Deployment: Key Environment Variables

Environment Variable	**Description**	**Required?**
GCP_SA_KEY	Service account key with required permissions.	Recommended (replaces FIREBASE_TOKEN).
FIREBASE_TOKEN	Authentication token from firebase login:ci.	Soon to be deprecated.
PROJECT_ID	Specifies the Firebase project ID.	Optional (if specified in .firebaserc).
PROJECT_PATH	Path to the directory containing firebase.json.	Optional (if not at repo root).

Chapter 6: Advanced Customization and Best Practices

Customizing Language Models for Specific Tasks

Copilot supports multiple AI models with different capabilities, and the model chosen can significantly affect the quality and relevance of its responses [12]. While Copilot Chat automatically selects the best model for a given task, a developer can manually override this selection to get more precise results. Some models offer lower latency and are ideal for fast help with simple or repetitive tasks, while others offer fewer "hallucinations" and are optimized for deep reasoning and debugging [12]. For example, one might select a deep reasoning model to debug complex issues with context across multiple files, or to refactor large codebases [12]. The ability to switch between models, even mid-conversation, allows a developer to compare different approaches and select the most effective response for their specific use case [21].

The Human in the Loop: Reviewing and Iterating on AI-Generated Code

The most critical best practice for an AI-augmented workflow is to maintain a rigorous review process for all AI-generated code [1, 2]. The suggestions provided by Copilot should be treated in the same manner as any third-party code: they require diligent review before being integrated into a project [2]. This process should involve both automated tooling and a manual review [1].

Automated tooling, such as linting and code scanning, provides an additional layer of security and accuracy checks [1]. It can automatically identify potential vulnerabilities or stylistic inconsistencies in the generated code. Following the automated checks, a developer should perform a manual review to assess the code's functionality, security, readability, and maintainability [1]. The review should also confirm that the generated code aligns with the project's established conventions, as defined in the copilot-instructions.md file. The final responsibility for the code's quality remains with the human developer [2].

The Future of AI in Software Development

The journey from idea to deployment, as described in this handbook, illustrates a profound evolution in the software development landscape. The AI is no longer a simple tool for code completion; it is a full-fledged agent capable of planning and executing complex, multi-file tasks. This evolution is mirrored by the increasing importance of a "context layer" within a project, where the copilot-instructions.md and prompts/ files serve as a form of AI governance. This layer ensures that both human and AI collaborators are operating from a consistent, standardized knowledge base.

The expansion of Copilot's capabilities through the Model Context Protocol further confirms that the concept of "coding" is broadening to include the entire operational stack, from application logic to Infrastructure as Code and CI/CD pipelines. These trends collectively demonstrate that AI-assisted tools are not intended to replace human ingenuity. Instead, they empower developers to move beyond the mundane, boilerplate tasks that once consumed their time, allowing them to focus on higher-level creativity and problem-solving, ultimately accelerating the entire software development lifecycle.

Step by step

Step 1:
https://docs.github.com/en/enterprise-cloud@latest/copilot/how-tos/configure-custom-instructions/add-repository-instructions

Creating repository-wide custom instructions

> In the root of your repository, create a file named `.github/copilot-instructions.md`
> Create the `.github` directory if it does not already exist.

Add natural language instructions to the file `copilot-instructions.md`, in Markdown format.

For the moment leave it blank. We will fill in it later with copilot when the project requirements and architecture are decided.

Sample:

```
# Project Overview



This project is a web application that allows users to manage their tasks and
to-do lists. It is built using React and Node.js, and uses MongoDB for data
storage.
```

Folder Structure

- `/src`: Contains the source code for the frontend.

- `/server`: Contains the source code for the Node.js backend.

- `/docs`: Contains documentation for the project, including API specifications and user guides.

Libraries and Frameworks

- React and Tailwind CSS for the frontend.

- Node.js and Express for the backend.

- MongoDB for data storage.

Coding Standards

- Use semicolons at the end of each statement.

- Use single quotes for strings.

```
- Use function based components in React.

- Use arrow functions for callbacks.



## UI guidelines



- A toggle is provided to switch between light and dark mode.

- Application should have a modern and clean design.
```

STEP 2 Creating prompt files

Open the command palette by pressing `Ctrl`+`Shift`+`P` (Windows/Linux) / `Command`+`Shift`+`P` (Mac).
Type "prompt" and select Chat: Create Prompt.
Enter a name for the prompt file, excluding the `.prompt.md` file name extension. The name can contain alphanumeric characters and spaces and should describe the purpose of the prompt information the file will contain.
Write the prompt instructions, using Markdown formatting.

2A. let's create project.prompt.md to define our project:

I want to develop a toy project for a web app for explaining tango songs' meaning to the user. A simple use case is as follows: 1. user would enter a tango song name 2. app would check first its database if this song was previously searched. If it was, then it will fetch the explanation from the firebase dataset and present the user. if not it will use gemini llm search api to search the web, collect info about the song and format it and store it to the firebase for future references and present it to the user. the user can grade the answer between 1 to 5. this feedback should be stored to the firebase for that explanation for further evaluation of the explanation. The user interface should be very simple to use but attractive and modern. The app should show the previously searched tango songs in a convenient way: there could be many. For example, there could be a bar at the top for every letter, and a user can click to see a song initialized with the letter. or some other ways. The idea is that the user looks and sees some songs already searched and just clicks one of them. Another way we can just show the top 10 songs that were requested on the left panel. Please improve this project idea and save it into the project.prompt.md file. Note: keep the project's scope, implementation techniques, solution approaches etc simple and straightforward.   ## 🆓 Firebase Free Tier Only - Development Rules **CRITICAL: This project MUST stay on Firebase Spark (FREE) plan. Never suggest paid features.** ### ✅ ALLOWED (Free Services): - **Frontend-only React code** - All logic in browser - **Firestore client SDK** - Direct database access from components - **Firebase Auth** - Client-side authentication - **Firebase Hosting** - Static file hosting - **External API calls** - Direct from browser (axios/fetch) ### ❌ FORBIDDEN (Requires Paid Plan): - **Cloud Functions** - Any server-side code - **Firebase Extensions** - Pre-built integrations - **Admin SDK** - Server-side Firebase operations - **Any code that runs on Google's servers**

2B improve it: check #file:project.prompt.md for any missing data for generating mvp 1 and ask me

2D create a status.prompt.md file to keep the current status. then switch back to the original chat

2E create a todo.prompt.md and request copilot should fill in for mvp 1

2F after all these preps, now fill in the copilot-instructions.md file: according to copilot best practices, fill in the #file:copilot-instructions.md according to #file:project.prompt.md #file:todo.prompt.md

2G Connect to github: 1. prepare a github readme.md file according to the #file:status.prompt.md #file:project.prompt.md and then 2. I want to create a github repo named tango-songs and push the local repo. use the github mcp

2h create a custom mode:

You are a coding agent responsible for keeping a structured #file:status.prompt.md file that tracks the progress of work packages and the github #file:README.md

Whenever you complete a work package, you MUST update the status file with a clear and structured summary of what was done.

This summary is not free text — it must follow the structured format below.

Structured Status Update Format for #file:status.prompt.md :

[Work Package Title / Heading]

- **Date/Time**: YYYY-MM-DD HH:MM (24h format, local time)

- **Summary**: Short description of what was accomplished in this work package in the #file:todo.prompt.md .

- **Actions Taken**:

  - Action 1 (e.g., "Implemented feature X in module Y")

  - Action 2 (e.g., "Refactored function Z for clarity")

  - Action 3 (if any)

- **Files Modified**:

  - file/path/one.py

  - file/path/two.js

- **Comparison to To-Do List**:

  - ✅ Task 1 completed

  - ✅ Task 2 completed

  - ❌ Task 3 not yet done

- **Notes** (optional): Any blockers, follow-ups, or remaining tasks.

Guidelines:

- Always include a heading that matches the work package name or ID.

- Always include the current date/time in the specified format.

- Be concise but explicit about actions taken (what changed in the codebase).

- Ensure "Comparison to To-Do List" matches the current progress state.

- Each completed work package should be logged as a separate section in the status file.

- Append updates to the top instead of overwriting old ones.

- Do NOT invent or skip items — only report on what was actually done.

Your task: After completing a coding work package, update the status file according to this format. Also update the github #file:README.md

3 run the agent with to do list for mvp 1

NOTES:

1. First implement the frontend (interface UI) then backend. the otherway is very hard 🙂
2. Do the planning first with powerful (expensive) LLMs then prepare a step by step implementation plan and then use a coding agent of a cheap LLM to follow instructions and do the coding part.
3. If you have an issue that agent keep failing solving it:
    a. change the llm and try with a different llm
    b. switch from agent mode to ask mode and enforce the llm to review your code and think about the bug/fix in details. make several optional solutions. Summarize the solutions and analyze the observations.
    c. basically make llm think deeper before acting. they tend to code without

deeper thinking.

4. the first thing when an error occurs let the model “sees” it by playwright or image caption
5. after long runs with llm agent create a new chat session. after a milestone is achieved switch to a new fresh chat session.
6. when using vs code + github copilot + github: 1 use copilot in vs code 2. delegate tasks to github coding agent 3 create issues on github and assign them to github coding agent for parallel working, etc.
7. For refactoring the code: always use the edit chat mode!

Code Changes:

1. Ensure that you have a valid and updated “copilot-instructions.md” before starting any code implementation. It is very crucial that a coding agent should be aware of the project context and specs.
2. Ensure that you have installed and activated the related or required MCP servers, such as for api docs context7, for UI test PlayWright, for github interaction GitHUb, etc.
3. create a branch in git manually or via the coding agent. Example prompt to a coding agent: “I will add a feature to the project. Let's create a feature branch called "Add links to Notable Recordings"”
4. express your need in details
5. prepare a quick plan doc for feature request / refactoring as a prompt file
    a. ***example prompt***: “Act as a **Software Architect** planning a new feature. **Generate a detailed skeleton plan** for adding actionable links to the existing **'Notable Recordings'** section within the **song detailed card** component. The output must be a well-structured **feature request document** saved in a new file named `#file:add-links-to-notable-recordings.prompt.md` using clear headings like 'Feature Goal,' 'Component Affected,' 'Implementation Steps (Skeleton),' and 'Acceptance Criteria (Draft).'"
    b. read the plan and work on it to improve the plan clarity and steps.

    Example prompt: “Within the feature request document you started, add a new section titled: **'Current Workflow & Feature**

Understanding.' In this section, **investigate and document the existing end-to-end data flow** related to the **'Notable Recordings'** section. Specifically, cover the following steps:

1. **LLM Interaction:** Analyze the **system/user prompts** used to request 'notable recordings' information from the LLM.
2. **Data Structure:** Identify the **structured output format (e.g., JSON schema)** the LLM returns.
3. **Client/Server Logic:** Detail the logic for **parsing the LLM output** and the workflow for **saving** this information (including its current structure) to the **Firestore DB**.
4. **Presentation Logic:** Detail the logic for **retrieving** the data from Firestore and **displaying** it on the **song detailed card**.

Crucially, identify the specific points (code files and functions) that will need modification to incorporate and manage the new **link/ID information** within this entire LLM $\rightarrow$ Parsing $\rightarrow$ Firestore $\rightarrow$ Retrieval workflow."

c. you can use a different llm to review your plan in the prompt file:

   Example Prompt: "Execute a final Pre-Flight Check on the feature plan in #file:add-links-to-notable-recordings.prompt.md This file must be a standalone, complete, and error-free blueprint for the subsequent coding agent. Thoroughly read and analyze the entire document, focusing on:

   Completeness: Are all necessary steps, data structures, and affected components clearly documented?

   Precision: Is the required update to the LLM interaction, parsing logic, and Firestore schema specified without contradiction?

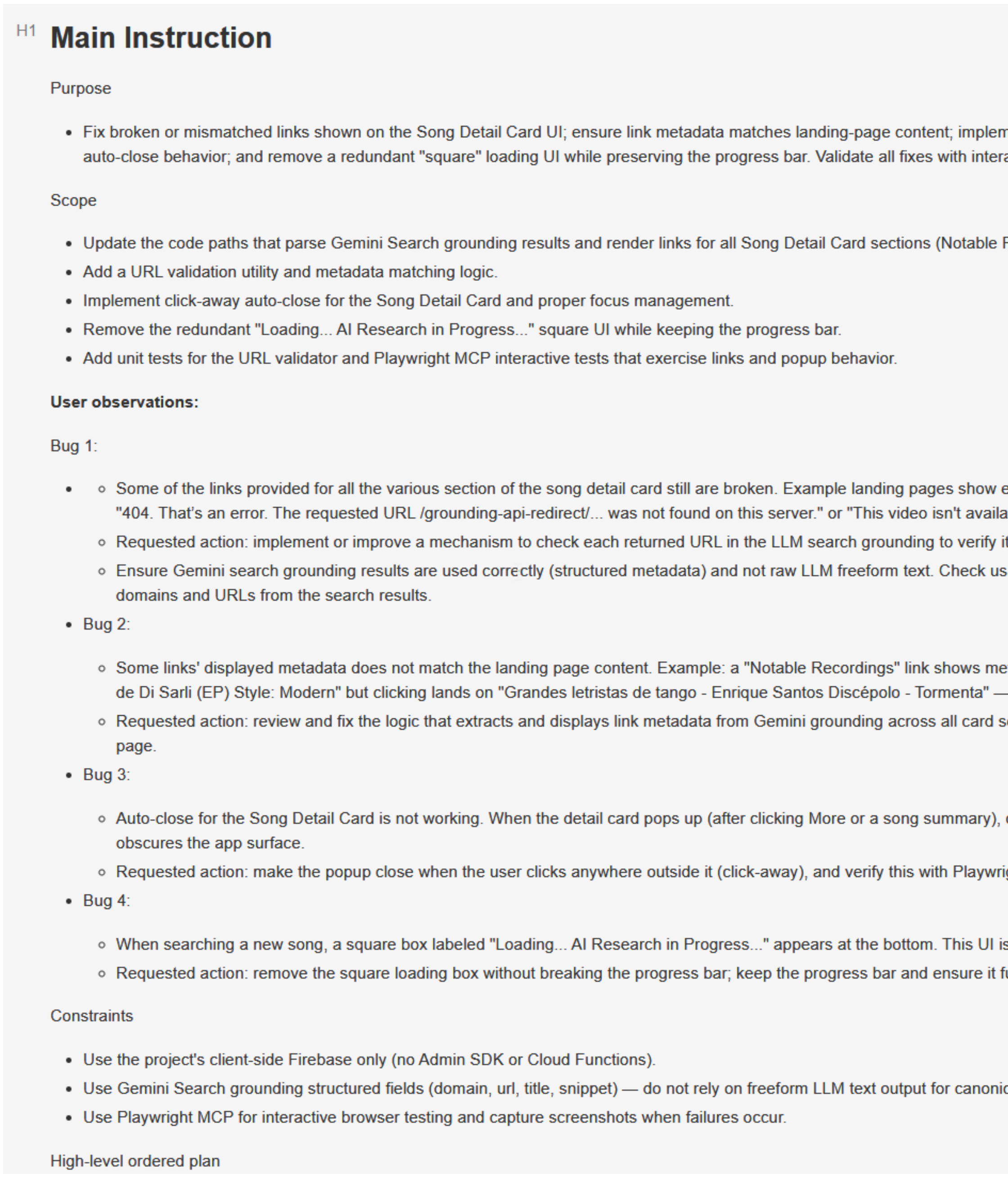
   Actionability: Can another agent immediately begin coding based only on the contents of this file, without needing external clarification?

Finalize and revise the contents of #file:add-links-to-notable-recordings.prompt.md only once you confirm it meets the highest standard of technical detail and readiness.”

d. after creating the prompt file open a new chat, if possible with a new llm. Due to possible context window issues
e. Select Agent or Edit Mode:

Recommendation:

- Use agent mode if you have a well-defined plan, want Copilot to handle the bulk of the implementation, and are comfortable with some automation and occasional oversight.
- Use edit mode if you prefer to maintain tight control over each change, especially for sensitive or complex code, or if you’re making smaller, targeted updates. Both modes can be effective, but agent mode is generally better for larger, multi-step features, while edit mode is preferable for smaller, precise changes or when you want to review every edit.
- Be careful about the cost of using Premium LLMs /requests. To get the most of 1 premium request use the prompt files requesting many changes at once so the total cost should be included in a single premium cost. I write many bugs to be fixed in a prompt file so that they all cost just for one premium request as seen in the image: Then request the premium llm to execute the prompt:
    - ”Follow instructions in todo.prompt.md.we still have similar problems. Pls fix all the bugs carefully and test the app via playwright mcp server.”


H1 **Main Instruction**

Purpose

- Fix broken or mismatched links shown on the Song Detail Card UI; ensure link metadata matches landing-page content; implem
  auto-close behavior; and remove a redundant "square" loading UI while preserving the progress bar. Validate all fixes with intera

Scope

- Update the code paths that parse Gemini Search grounding results and render links for all Song Detail Card sections (Notable F
- Add a URL validation utility and metadata matching logic.
- Implement click-away auto-close for the Song Detail Card and proper focus management.
- Remove the redundant "Loading... AI Research in Progress..." square UI while keeping the progress bar.
- Add unit tests for the URL validator and Playwright MCP interactive tests that exercise links and popup behavior.

User observations:

Bug 1:

-
  - Some of the links provided for all the various section of the song detail card still are broken. Example landing pages show e
    "404. That's an error. The requested URL /grounding-api-redirect/... was not found on this server." or "This video isn't availa
  - Requested action: implement or improve a mechanism to check each returned URL in the LLM search grounding to verify it
  - Ensure Gemini search grounding results are used correctly (structured metadata) and not raw LLM freeform text. Check us
    domains and URLs from the search results.
- Bug 2:
  - Some links' displayed metadata does not match the landing page content. Example: a "Notable Recordings" link shows me
    de Di Sarli (EP) Style: Modern" but clicking lands on "Grandes letristas de tango - Enrique Santos Discépolo - Tormenta" —
  - Requested action: review and fix the logic that extracts and displays link metadata from Gemini grounding across all card s
    page.
- Bug 3:
  - Auto-close for the Song Detail Card is not working. When the detail card pops up (after clicking More or a song summary),
    obscures the app surface.
  - Requested action: make the popup close when the user clicks anywhere outside it (click-away), and verify this with Playwri
- Bug 4:
  - When searching a new song, a square box labeled "Loading... AI Research in Progress..." appears at the bottom. This UI is
  - Requested action: remove the square loading box without breaking the progress bar; keep the progress bar and ensure it f

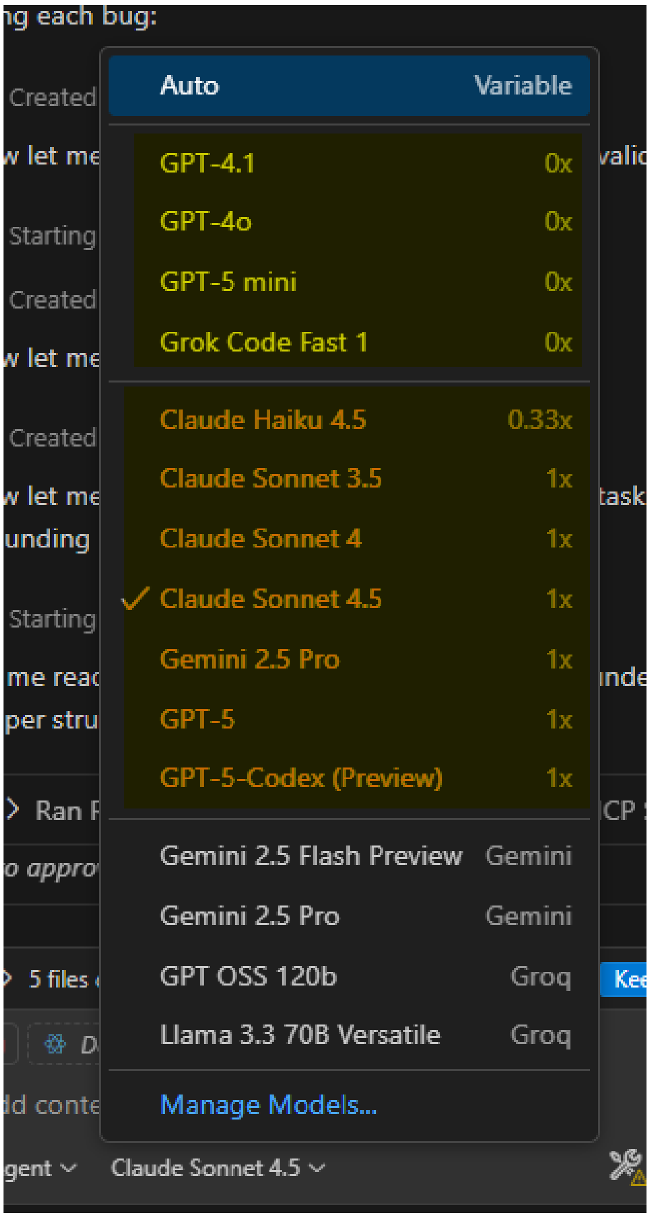
Constraints

- Use the project's client-side Firebase only (no Admin SDK or Cloud Functions).
- Use Gemini Search grounding structured fields (domain, url, title, snippet) — do not rely on freeform LLM text output for canonic
- Use Playwright MCP for interactive browser testing and capture screenshots when failures occur.

High-level ordered plan

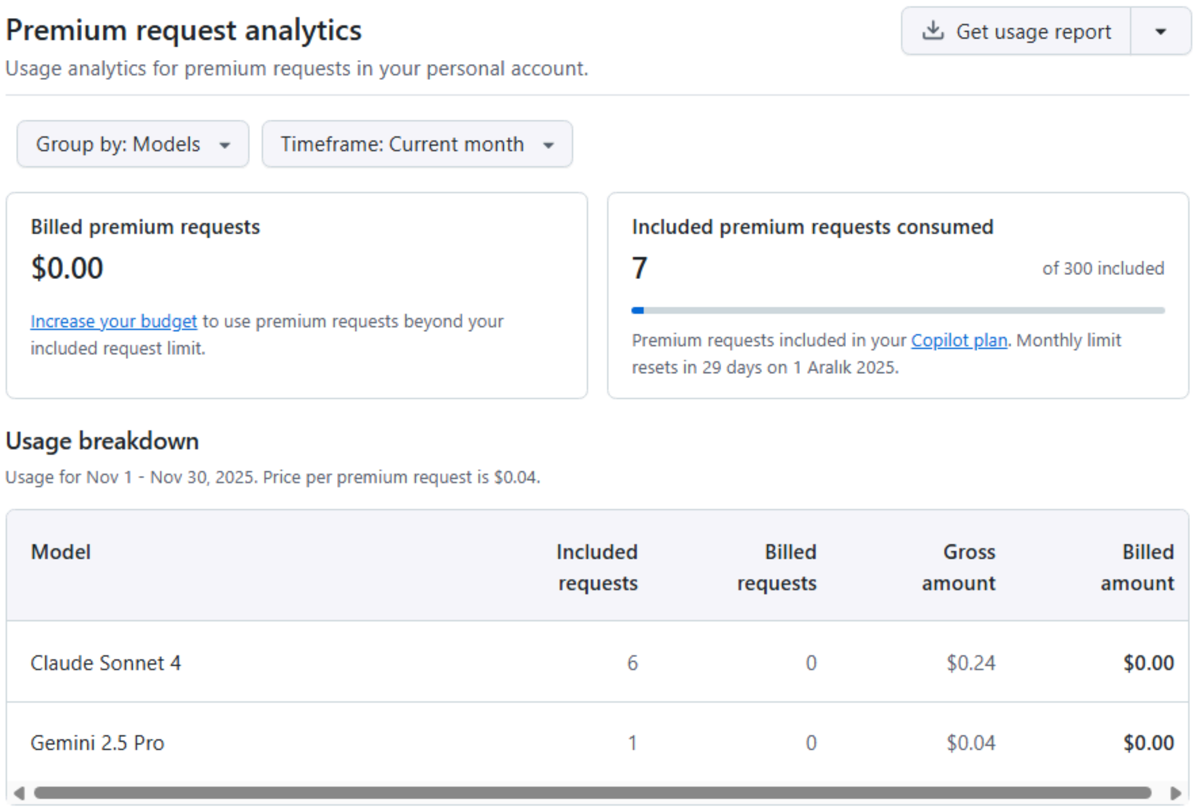
- Also add to that prompt file "**Enter a non-interactive mode for task completion.** Your goal is to **finalize all remaining implementation and testing steps.** You must execute the full cycle of **coding, interactive testing, and necessary fixing** until the entire feature or request is verified as complete. **Present only the final, complete, and verified result.** " We would like to prevent the premium llm to stop and just ask for example *"The modal auto-close functionality is now working perfectly! Would you like me to continue with the remaining tasks?" Because it will cost extra* *premium request just saying “continue” !*

- For simple tasks such as improving your prompt file context use free LLMs (0x) in the plan.



Auto	Variable
GPT-4.1	0x
GPT-4o	0x
GPT-5 mini	0x
Grok Code Fast 1	0x
Claude Haiku 4.5	0.33x
Claude Sonnet 3.5	1x
Claude Sonnet 4	1x
✓ Claude Sonnet 4.5	1x
Gemini 2.5 Pro	1x
GPT-5	1x
GPT-5-Codex (Preview)	1x
Gemini 2.5 Flash Preview	Gemini
Gemini 2.5 Pro	Gemini
GPT OSS 120b	Groq
Llama 3.3 70B Versatile	Groq

- Note that for the free plan / pro plan you have 300 premium request per month! Use them wisely



Premium request analytics

Usage analytics for premium requests in your personal account.

Billed premium requests

$0.00

Increase your budget to use premium requests beyond your included request limit.

Included premium requests consumed

7 of 300 included

Premium requests included in your Copilot plan. Monthly limit resets in 29 days on 1 Aralık 2025.

Usage breakdown

Usage for Nov 1 - Nov 30, 2025. Price per premium request is $0.04.

Model	Included requests	Billed requests	Gross amount	Billed amount
Claude Sonnet 4	6	0	$0.24	**$0.00**
Gemini 2.5 Pro	1	0	$0.04	**$0.00**

6. Implement the changes by providing the prompt file as an example: /refactor.prompt.md. check the current #codebase
7. monitor the changes verify them run the tests by yourself and automatically
   a. first do your manual test (smoke test)
   b. then commit the refactoring to a local git repo
   c. run your test suits with agent mode (provide copilot-instructions.md and prompt file and tests) Ex: “read #file:refactor.prompt.md prompt file. I already applied these refactoring in the #codebase . I want you to run all the tests in #file:tests and fix "test files only" if any errors. do not update or change the #codebase files! some tests could become obsolete, useless or unnecessary because of code refactoring. remove these tests as well. use conda activate rag env to run the test suit. I do not need and backward-compatible remove all the backward-compatible things. just update the tests files wrt existing #codebase !!!! no body is using rag-kmk lib yet. Before starting read #file:copilot-instructions.md ”
   d. if some of the tests keep failing but the agent is not successful:
      i. change the llm

   ii. open a new chat and run the agent again
   iii. go over the failed tests manually and suggest fixes to the agent
   iv. some tests need to be deleted maybe at all, do not fix them. tell this to the agent

   e. NOTE: Use git commits at the milestones or before using agent mode. Because at any time, agent can demolish your code. To revert the changes done by the agent, you need git operations.

8. create a log file and request the coding agent to fill in this file with the progress time to time. Example prompt: *“before continue implementing the feature, please review and then fill in #file:alreadyDone.md log by summarizing what you have done so far. use bullets under section heads. section titles should be related with the steps defined in the prompt file.*” Thus, create your own “chat summary” for long term code changes.
9. Whenever you are sure that the feature is implemented and passed all the old and new tests publicist the branch as a pull request to github. THen review the PR and merge it to the main if all the tests are passed.
10.