ChatGPT

AI-Powered Project Planner & Code Generator – Implementation Plan

This application will let a user describe a desired software project and then use an LLM to plan and build it automatically. The user's prompt goes into the LLM, which returns a detailed **feature list** (each with descriptions, key points, and tests). These features become cards in a **Kanban board** (Backlog, To Do, In Progress, In Review, Done). When the user moves a feature into **To Do**, the app triggers an AI agent (via a CLI) to implement that feature: it generates code, runs builds/tests, and pushes changes to GitHub as a PR. This plan outlines the architecture and steps to build this as a **cross-platform desktop app** (with at least web support), using local models and free AI CLI tools as much as possible.

1. Architecture & Tech Stack

- **Cross-platform Desktop/Web**: Use a web-based UI wrapped as a desktop app. For example, build the interface in **React** or another frontend framework and package it with Electron or Tauri so it runs on Windows, macOS, Linux (and can also be served as a web app). Electron (Chromium + Node.js) is a popular choice for cross-OS desktop apps [1]; Tauri is a lightweight alternative that supports any JS framework and produces smaller binaries [2].

- **Frontend**: Implement the UI with **HTML/CSS/JS** (e.g. React). Use an existing Kanban component or library (e.g. Syncfusion or react-beautiful-dnd) to display tasks in Backlog/ToDo/InProgress/Review/Done columns [3]. For example, a React Kanban board can visualize tasks by status (see figure below).

Example of a Kanban board UI (columns: To Do, In Progress, In Review, Done) showing tasks/cards in a project-management interface [3].

- **Backend/LGL Integration**: The app's logic can run in Node (Electron's main process) or a bundled Python process. Since we have a GPU, we can run LLMs locally. Options include the Hugging Face Transformers library, xenova/transformers in JS, or local inference engines like Llama.cpp or Ollama for Llama2-based models [4] [1]. For example, one could embed an ONNX model and use `xenova/transformers` in JavaScript as shown by De'Shaizer [1] [5]. Alternatively, a Python process with PyTorch/HuggingFace could run a model (e.g. StarCoder or Llama2) on GPU. The LLM will be prompted to generate feature lists and descriptions from the user's project description.

- **Data/State Management**: Maintain the project state (features, statuses) locally. This could be a simple JSON file or a lightweight local DB (SQLite via node, or `electron-store`, or `IndexedDB` in the renderer). Each feature card stores its description, test plan, and code status.

- **Local File System**: When coding begins, the app will create a local project folder (e.g. via `create-react-app`, `npm init`, or a CLI scaffolding tool) for the base project. Each feature implementation can create/modify files in that folder. The app can run shell commands (via Node's `child_process`) in that directory (e.g. to run builds or tests).

2. User Interface – Kanban Board

The UI will include:

- **Project Input**: A text input where the user describes the project goal in plain English (e.g. "Build a To-Do list web app with user login").

- **Feature Generation Trigger**: A button (e.g. "Plan Project") that sends the prompt to the LLM.

- **Kanban Board**: A drag-and-drop board with columns **Backlog**, **To Do**, **In Progress**, **In Review**, **Done**. Initially, all new features are added in **Backlog**. The user can drag features between columns. We can use a React Kanban component – for instance, Syncfusion's React Kanban component is specifically designed for task management [3].

- **Task Cards**: Each card on the board represents a feature and contains: the feature title, a brief description, and an icon or color indicating priority or owner. When clicked, a card can expand to show details and the test steps.

By clicking or dragging a feature card to **To Do**, the app knows to start coding it.

[3] *React Kanban components (like the one above) are widely used to organize features in software project management and support rich customization.*

3. LLM Feature Generation

When the user submits a project description, the app should call an LLM with a prompt like:

> *"I want to build a [type of project] with these requirements: [user description]. List the core features/modules needed to implement this project. For each feature, include: a short title, an elaborate description, key technical points or sub-tasks, and a brief test plan or acceptance criteria."*

The LLM (running locally) will respond with a structured list of features. The app parses this output and creates one Kanban card per feature (placing them in **Backlog**). For example, a feature entry might include: "User Authentication – allow users to register/login. Tests: try registering with invalid email, check password encryption."

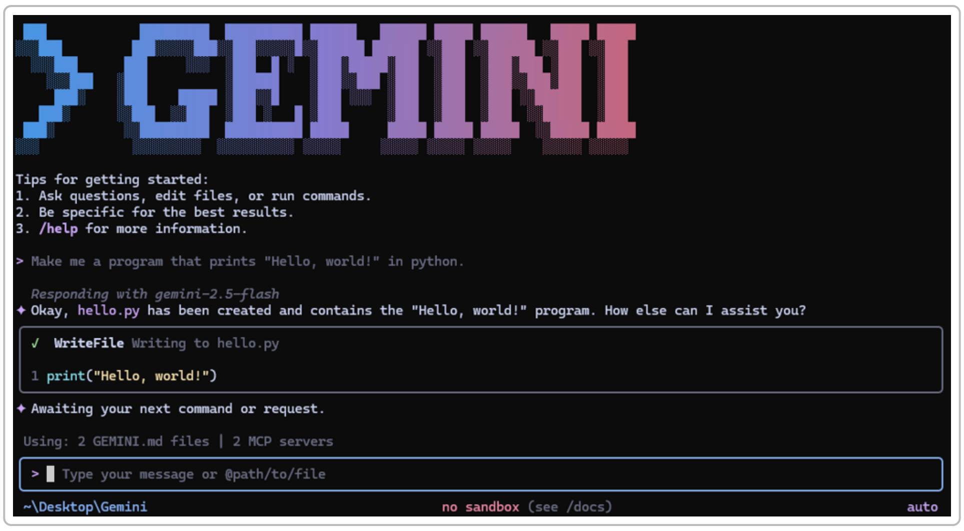
- **Implementation**: If using Transformers in Node, the app can use an async function to call `pipeline('text-generation', model)` [1]. If using Python, a Flask/Express server or a Python subprocess can serve as the LLM API. In either case, the output should be JSON or a parseable format (the prompt can request a Markdown or JSON list).

- **Open-source Models**: Models like StarCoder2, CodeGen, or Llama2-Chat can be used. Hugging Face has code-focused models (e.g. StarCoder) and allows onnx export [1]. Running large models (13B or 70B) requires GPU and enough VRAM. According to benchmarking, tools like Llama.cpp can run Llama2 models efficiently on high-end GPUs [4].

4. AI Coding Agents (CLI Tools)

For implementing features, we will employ an “AI coding agent” via the command line. These tools take natural language instructions and directly generate code, run commands, and even commit changes [6] [7]. Key options include:

- **Google Gemini CLI**: An open-source terminal AI agent for Google’s Gemini model [8]. It supports multimodal inputs and can run shell commands and edit files. It has a free tier (60 req/min, 1000/day) and a 1M token context window [9]. Gemini CLI can read/write files and execute code – for example, in the image below it was asked to create a “Hello, world!” program and it created the file.



```
> GEMINI

Tips for getting started:
1. Ask questions, edit files, or run commands.
2. Be specific for the best results.
3. /help for more information.

> Make me a program that prints "Hello, world!" in python.

  Responding with gemini-2.5-flash
✦ Okay, hello.py has been created and contains the "Hello, world!" program. How else can I assist you?
┌─────────────────────────────────────────────────────────────────────────────┐
│ ✓  WriteFile Writing to hello.py                                            │
│                                                                             │
│ print("Hello, world!")                                                      │
└─────────────────────────────────────────────────────────────────────────────┘
✦ Awaiting your next command or request.

 Using: 2 GEMINI.md files | 2 MCP servers
┌─────────────────────────────────────────────────────────────────────────────┐
│ > ▌ Type your message or @path/to/file                                      │
└─────────────────────────────────────────────────────────────────────────────┘
 ~\Desktop\Gemini                      no sandbox (see /docs)                auto
```

Example of Google’s Gemini CLI in action: the assistant confirms it created a “hello_world.py” with `print("Hello, world!")` *in response to a prompt* [8].

- **Anthropic Claude Code CLI**: A Docker-based CLI agent by Anthropic [6] [7]. It “lives in your terminal” and can build features from descriptions, debug code, and handle git [6]. It automatically plans steps, edits files, runs tests, and commits code [6] [10]. (Note: requires an Anthropic account.)

- **ForgeCode**: An open-source Node.js CLI agent. It supports multiple backends (OpenAI, Anthropic, etc.) and can scaffold code from prompts. It launches with `npx forgecode@latest` and requires minimal setup [11].

- **Aider**: A terminal-based assistant that builds a “map” of your repo and can generate code diffs. It supports many models and has built-in git integration to auto-commit changes with messages [12].

- **GPT Engineer**: A CLI tool that reads a prompt file and synthesizes a project. It can scaffold an entire app, write files, and run commands in one run [13]. This could be used to create the initial project structure or big features.

- **Hugging Face CLI / transformers-cli**: Hugging Face provides a command-line interface to run models locally (via `huggingface-cli`) [14]. This could serve as a lightweight code generator if you have an onnx model.

In practice, any of these can be invoked by the desktop app (via Node's `child_process.spawn`). For example, when a feature moves to To Do, the app runs something like:

```
gemini -p
"Implement feature: [feature description]. After coding, run tests: [test
instructions]."
```

The AI agent will edit files or output diffs. Many agents (Claude Code, Aider, etc.) automatically commit their changes to git [6] [12] . If not, the app can capture the output and write files itself.

5. Workflow: Feature Implementation

1. **Base Project Setup**: When the first feature is started, the app should ensure a project skeleton exists. This might mean running `npm init react-app` or similar. The AI agent can also scaffold this by prompt.

2. **Triggering Code Generation**: As soon as the user drags a feature to **To Do**, the app transitions it to **In Progress** and spawns the chosen AI agent CLI. The prompt given to the CLI should include:

3. The feature's **description** and **test steps** from the Kanban card.
4. Instructions to generate the code for that feature.
5. Commands to run tests or a build (e.g. `npm test`, `pytest`, etc.).
6. Instructions to commit/PR the changes with a message.

For instance:

> "Implement the feature *'User Authentication'* as described: [detailed description]. After writing code, run the test suite. If tests fail, fix the errors. Once passing, commit all changes with message 'Add User Authentication feature' and create a pull request."

1. **Automatic Testing and Debugging**: After code is generated, the app should automatically run the project's build and test commands. If errors or failed tests occur, the app can loop: it feeds the error messages back to the CLI agent (e.g. via another prompt) asking it to fix the code. Many AI agents support iterative debugging.

2. **Git Commit & PR**: When code is working, push it to Git. If the CLI agent already commits, great. Otherwise, the app can run `git add`, `git commit`. Then use the GitHub CLI (`gh pr create`) or GitHub API to open a pull request. The app should prompt the user to link their GitHub (via OAuth or a token). For example, using GitHub CLI:

   ```
   gh pr create --title "Add [Feature]" --body
   "Auto-generated feature code for [Feature]."
   ```

   The URL of the PR can be shown to the user [15] .

3. **Status Updates**: Once the PR is opened, mark the feature card as **In Review**. After merge/approval, mark it **Done**.

Throughout this workflow, the Kanban board is kept in sync. The user can also manually move cards or abort a feature if needed.

6. GitHub Integration

To integrate with GitHub: - **Authentication**: Ask the user to sign in or provide a personal access token. Use something like the GitHub CLI (`gh auth login`) or embed an OAuth flow. - **Pushing Code**: After local commits, the app pushes the branch to GitHub.
- **Pull Requests**: Use `gh pr create` (as above) or the GitHub REST API via a library like `@octokit/rest`. The CLI is easiest and is designed for automation [15].
- **Secrets/Config**: The app can remember which GitHub repo (or create a new one) and branch to use for the project.

In summary, GitHub becomes the remote deployment for the code. Initially the user can supply an existing repo URL (or one can be created), and the app treats that as the project repo.

7. Cross-Platform Packaging

Once the app logic is complete, package it for desktop distribution: - If using **Electron**: use `electron-builder` to generate installers (.exe, .dmg, etc.). Electron bundles the web UI and Node backend into one app.
- If using **Tauri**: write the core app logic in Rust (or spawn Node/Python processes) and bundle the frontend web app. Tauri produces very small binaries (~600KB) [2].
- Ensure the LLM models and any Python dependencies are bundled or installed. For example, include instructions to download model weights on first run, or package with `pip` requirements.

Also provide an option to run as a normal web app (for the web support requirement). The same React UI could be deployed to a web server; the difference is how the LLM is hosted (e.g. via a backend service or limited to user's API keys).

8. Development Plan

A step-by-step plan:

1. **Prototype LLM Feature Generation**: Write a simple Node/Python script that takes a project description and calls a local LLM (e.g. HuggingFace pipeline) to output a feature list. Ensure the output format is parseable (JSON or Markdown).

2. **UI and Kanban Board**:

3. Scaffold the frontend (e.g. `create-react-app`).
4. Integrate a Kanban component and create columns.
5. Build UI components for "New Project" input and feature cards.

6. Store state in React (or a global store) and persist to disk (localStorage or file).

7. **Connect UI to LLM**: Implement the backend (Node/Electron main or Python server) that receives the project prompt from the UI, runs the LLM, and returns features. Populate the Kanban with the results.

8. **Implement CLI Agent Calls**: Decide which CLI tool(s) to use (e.g. Gemini CLI and/or Claude Code). In code, listen for status-change events (card moved to ToDo) and spawn the CLI process with the appropriate prompt. Capture the process output.

9. **Code Integration and Testing**: Have the app apply the generated code to the project folder. Then run build/test commands (`npm run build`, `npm test`, etc.) and display any errors. Loop back errors to the agent if needed. Automate the success path (commit and PR) on passing tests.

10. **GitHub Linking**: Add a settings section for GitHub auth. Use the GitHub CLI or API to link the local folder to a remote repo. After coding each feature, push and create a PR automatically.

11. **UI Polishing**: Display progress to the user (e.g. show logs of the CLI agent, build results). Allow canceling a task. Show PR URLs in the Kanban card.

12. **Packaging and Cross-Platform Builds**: Configure `electron-builder` or Tauri build scripts. Test the app on Windows, macOS, and Linux. Ensure models and dependencies are included (or easy-to-install).

13. **Iteration**: Test end-to-end with a sample project (e.g. a simple blog app). Refine prompts to the LLM/CLI agents to get reliable code. Add error handling (e.g. if the model fails).

14. **Documentation**: Write a README and user guide explaining how to install/run the app, connect GitHub, and interpret the generated features and PRs.

Throughout, keep components modular so one can swap out the LLM model or CLI agent if needed. For instance, start with an open model like GPT4All or Mistral-Instruct for local testing, then later allow using commercial APIs for better results.

9. Considerations & Challenges

- **Quality of AI Output**: The features and code generated by LLMs will not be perfect. The user may need to review and refine the prompts or the generated code. Always include manual override options.
- **Security**: Running AI-generated code on the local machine can be risky. The app should run in a controlled environment (e.g. a sandboxed process) and warn users.
- **Resource Usage**: Large models require heavy GPU/CPU. Provide options to use smaller models or API fallbacks.
- **Synchronization**: Ensure the Kanban state, local files, and git state stay in sync even if the app crashes.

10. Conclusion

By combining a web-based UI with powerful AI coding agents, this desktop app can automate much of the software development workflow. Users get a high-level project plan from the LLM, see features organized in a Kanban board, and watch as AI handles the tedious coding, testing, and Git workflow. Leveraging existing CLI tools like Google's Gemini CLI or Anthropic's Claude Code allows the app to "think and act" like a developer in the terminal [6] [8]. With careful integration and iteration, this tool can dramatically speed up prototyping and development of web applications, while remaining fully cross-platform and leveraging local compute resources.

Sources: Documentation and articles on Claude Code [6], Google’s Gemini CLI [8], CLI coding tools [11] [13], React Kanban components [3], and desktop app development (Electron/Tauri) [1] [2] were used to inform this plan.

---

[1] [5] How I Put a Large Language Model Inside a Desktop App Using Javascript | by Michael De'Shazer | Medium
https://medium.com/@mikedeshazer/how-i-put-a-large-language-model-inside-a-desktop-app-using-javascript-838bd46ae811

[2] Tauri 2.0 | Tauri
https://v2.tauri.app/

[3] @syncfusion/ej2-react-kanban - npm
https://www.npmjs.com/package/@syncfusion/ej2-react-kanban

[4] 6 Ways to Run LLMs Locally (also how to use HuggingFace)
https://semaphore.io/blog/local-llm

[6] [10] Claude Code overview - Claude Code Docs
https://code.claude.com/docs/en/overview

[7] [11] [12] [13] Top 10 Open-Source CLI Coding Agents You Should Be Using in 2025 (With Links!) - DEV Community
https://dev.to/forgecode/top-10-open-source-cli-coding-agents-you-should-be-using-in-2025-with-links-244m

[8] [9] GitHub - google-gemini/gemini-cli: An open-source AI agent that brings the power of Gemini directly into your terminal.
https://github.com/google-gemini/gemini-cli

[14] 20 Best AI Code Assistants Reviewed and Tested [August 2025]
https://www.qodo.ai/blog/best-ai-coding-assistant-tools/

[15] GitHub CLI | Take GitHub to the command line
https://cli.github.com/manual/gh_pr_create