Ideal Scope and Purpose of the Electron Forge + Gemini-CLI Template

Core Responsibilities vs. What to Exclude

Core Responsibilities: The template's primary role is to set up a **minimal, functional Electron + Gemini CLI project** that works out-of-the-box. It should handle all the boilerplate wiring needed to integrate the Gemini CLI into an Electron app, while following Electron best practices. Key responsibilities include:

- **Project Structure & Config:** Provide a clear project structure with a main process (Electron's main.js), a preload script, and a renderer stub. Include an Electron Forge configuration for packaging (with sensible defaults for makers on different platforms) and any necessary plugins (e.g. fuses for security). This ensures the app can be built and distributed without additional setup.

- **Gemini CLI Integration:** Include the necessary code to invoke the Gemini CLI. For example, a minimal backend mechanism (like a small Express server or direct child process spawn) that can run Gemini CLI commands and stream results to the front-end. The template should demonstrate how the Electron app communicates with Gemini (e.g. using ipcMain/ipcRenderer or HTTP fetch to a local endpoint) as a reference implementation. This is the unique "core" of the template since its purpose is to combine Electron with Gemini CLI.

- **Security Best Practices:** Set sane defaults for security. For instance, enable context isolation, disable remote modules, disable Node integration in the renderer, and use content security policies by default. The template should also apply Electron Fuses (as shown in the POC code) to turn off features like NODE_OPTIONS or running as Node.js, making the app more secure by default. These configurations are part of the template's responsibilities to guide developers towards a secure app structure.

- **Basic Scripts and Tools:** Include npm scripts for starting, packaging, and making the app (as provided by Electron Forge) so a developer can run npm start and see the app in action immediately. Also ensure dependencies like Electron and Forge CLI are set in *devDependencies* and any required runtime deps (e.g. Express, if used for Gemini integration) are included. The idea is a beginner can clone or init this template and have everything needed to run or build the app.

What to Deliberately Exclude: To keep the template broadly useful and **unopinionated**, it should **not include heavy frameworks or full business logic** that are not universally needed. In practice, this means:

- **No Prescribed UI Framework:** Avoid bundling React, Vue, Angular, etc., or choosing a specific UI library in the base template. The template's UI should be framework-agnostic (e.g. plain HTML/CSS/JavaScript) by default. This keeps it lightweight and lets developers decide their own UI stack. Including a major framework by default would make the template more opinionated and possibly overwhelming for those who don't need that complexity. (Notably, Electron Forge's official templates also start minimal – e.g. the default Webpack template just sets up Webpack and TypeScript, but doesn't force React or others unless you specifically choose a React template.)
- **No Full App Features or Domain Logic:** The template should not attempt to provide a complete application's logic (for example, no sample note-taking app or large feature set baked in). It should exclude any code that isn't generic to most apps. For instance, **omit any placeholder business logic** not related to demonstrating Gemini CLI. The goal is a **"canvas" rather than a pre-painted picture** – developers will build their app on this canvas. Any included code should serve only to demonstrate the structure or the Gemini integration, not to impose how an app's functionality should be written.
- **Avoid Database or Auth Stubs:** Don't include stubs for things like database connections, authentication flows, or other back-end services that aren't strictly necessary. Those are highly application-specific and can bloat the template or confuse beginners. A clean template would leave such things out, or just mention in documentation how one *could* add them. (For example, Create React App also provides **no back-end or database by default**, just the files needed for a front-end. This principle keeps the starting point focused and easy to understand.)
- **Minimal Dependencies:** Refrain from adding dependencies that aren't needed for the core functionality. Each extra library (UI kits, utilities, etc.) should be scrutinized. A lean dependency list means less potential for security issues and easier maintenance. The template in the POC, for instance, only included express and cors because they were directly used for the Gemini CLI API route – it doesn't bundle anything like a UI library or database ORM, which is good.

By limiting scope in this way, the template remains **extensible and adaptable**. It provides a solid foundation (Electron + Gemini integration, packaging setup, security configs) but deliberately leaves room for the developer to choose their own frameworks and add their own features, rather than imposing those choices upfront.

Demo UI vs. Headless Scaffold

Including a Basic Demo UI is Recommended. Most successful app templates include a minimal UI to prove that everything is wired up correctly. A small "Hello World" interface or example component helps beginners confirm that the app runs and the Electron<->Gemini integration works. It also serves as living documentation for how to call Gemini from the UI. Here's why a demo UI is beneficial:

- **Instant Feedback for Beginners:** When a new developer runs the scaffolded app for the first time, seeing *some* UI (even if very simple) reassures them that the app is working. For example, Create React App launches with a React logo and some instructional text on localhost, confirming the setup is correct. Similarly, an Electron Forge template typically opens a window with a minimal HTML page (like a title or message). Providing a basic window with a UI label or buttons (as in the POC which has "Repo Summary", "Key Files", etc. buttons) ensures the user isn't staring at a blank screen.
- **Demonstrating Gemini CLI Usage:** A demo UI can show how to trigger the Gemini CLI and display its output. For instance, the POC's UI allows the user to click a button, which calls window.api.runGemini('repo-summary'), and then streams the response to a <pre> element. This is an excellent illustration of the core feature. A beginner can click around and see real data (or at least placeholder data) coming from the Gemini CLI, which is far more instructive than a headless setup. It confirms that the plumbing (preload to main, main to back-end, etc.) is functioning.
- **Parity with Similar Templates:** Other frameworks provide example UIs. Tauri, for instance, includes a simple UI in its vanilla JavaScript template – it shows a window that says "Hello, World" (or in some cases a prompt to try the built-in greet command). Every create-tauri-app template ships with a default **greet** function in the backend and a corresponding UI element to call it, greeting the user from Rust. This approach helps developers see how front-end and back-end interact. Likewise, our template can include a few basic controls (buttons or menu items) to run sample Gemini tasks (like a dummy

"Analyze Repo" or "Hello Gemini" prompt) and display the result. It should be just enough UI to demonstrate functionality, but no more.

Avoid a Completely Headless Scaffold: A headless template (with no renderer content) would start the Electron app in the background or show an empty window with nothing to do. This can be confusing, especially for newcomers. They might wonder if something is wrong or how to proceed. A headless setup also misses an opportunity to demonstrate usage. Given that **usability for beginners is a priority**, it's better to include a simple interface than to expect users to wire up their own UI just to test the template. In the context of Electron, a "headless" app (no visible window or UI) is rarely useful as a starting point, since Electron's purpose is to create GUI applications.

Keep the Demo UI Minimal and Removable: While a demo UI is recommended, it should be **minimalistic and easy to remove or replace**. The template can include plain HTML/JS (or a very simple framework usage if absolutely needed) that the developer can discard when starting their real app. For example, showing how to capture Gemini's output in a <pre> tag is useful, but it should be clear what part of the code is "example" so they can delete it when scaffolding a new project. Providing comments in the code like "// Demo UI code below" or documenting how to remove the example can be helpful. This ensures that the demo UI serves its purpose (educating and confirming functionality) without locking the user into a particular UI structure.

In summary, **a small demo UI should be included by default** for clarity and completeness. It aligns with how other templates (Electron Forge's built-ins, Create React App, Tauri, etc.) handle scaffolding – they favor giving the developer something visible and interactive to start with, rather than an empty canvas. Once the developer understands the basics, they can then replace or expand that UI as needed.

Handling Back-End Routes and Database Stubs

Avoid Unnecessary Back-End/DB Boilerplate. It's important to keep the template lean, so include back-end routes or database helpers **only if they serve the template's core purpose**. In this case, the core purpose is integrating the Gemini CLI. That might require a minimal back-end component – for example, an Express server or a child-process handler – to invoke the Gemini CLI. Including that is reasonable, since it directly supports the Gemini integration. The POC code indeed uses a small Express app with a /run-gemini endpoint to execute the CLI and

stream results. This is a good pattern to include because it shows how one might structure calls to the CLI in a real app.

However, **the template should not include any generic or unrelated back-end/API scaffolding beyond that**. For instance:

- **No Stubbed Database Layer:** Don't include modules for connecting to a database (SQL or NoSQL) or an ORM with dummy config. Not every Electron app will use a database, and those that do might choose very different solutions. Including a placeholder db.js or sample queries would add bloat and potentially confuse beginners with unused code. If the end-user of the template needs a database, they can integrate one later according to their needs. The template's role is not to guess those needs.
- **No Unused API Endpoints:** Apart from the Gemini-related route, avoid adding, say, a /hello test endpoint or a bunch of commented-out example routes. These can clutter the code. It's better to document how to add routes if needed, rather than shipping them by default. Remember, every file or endpoint in the template should have a clear purpose. If it's not part of demonstrating Electron or Gemini CLI usage, it probably doesn't belong in the template.

The philosophy here is to **keep the template clean**. Many modern templates and CLIs follow this minimal approach. For example, Create React App doesn't spin up an Express server at all – it focuses purely on the front-end, leaving back-end decisions to the developer. In an Electron context, there is more need for a back-end (since Electron apps can bundle a local server or use Node APIs), but it should be limited to what's truly necessary. Our template's necessity is the Gemini integration, so one route or method to invoke it is enough.

If a Back-End is Included (e.g. Express), Keep It Simple: In the POC, the Express server.js serves one endpoint and includes CORS to allow the Electron renderer to call it. It doesn't, for example, set up any complex routing structure or templating engine. This is appropriate. The template can include that file (with comments explaining its role) to jump-start users who need an internal API. But it shouldn't go further – no session management, no extensive middleware aside from what Gemini needs (like CORS or file serving if needed for Gemini prompts).

Likewise, do not include placeholder functions like getUserData() or fake auth tokens, as those are not universally needed. If the template tried to include "everything you might eventually need" it would cease being a template and become a full (and overwhelming) boilerplate.

Instead, it's okay to **omit features to keep the template lean** and mention in documentation how to extend it. As the official Electron docs note, Electron development is very modular and unopinionated; developers are free to bring in what they need. A template should respect that by not over-providing.

In summary: Include only the back-end pieces that are **essential** (Gemini CLI invocation flow, basic IPC or HTTP for it). Omit any database hookups or multi-route setups that aren't absolutely needed. This keeps the template **clean, focused, and easy to understand** for a beginner. The result is a starting point that does one thing well – connecting Electron to Gemini – and leaves everything else (UI choices, databases, auth, etc.) up to the developer.

"Definition of Done" for the Template

For the template to be considered *complete* and **ready for use**, it should meet certain criteria. These criteria ensure that a developer can generate a new project from the template and have a smooth experience running and modifying it. Here's what the Definition of Done should include:

- **Complete File Set for a Running App:** The template should generate all essential files and configurations such that the app runs immediately. This includes:
    - package.json with proper scripts (start, make, etc.) and dependencies,
    - a main process file (e.g. main.js) that creates a BrowserWindow and ties in the preload script,
    - a preload.js that safely exposes necessary APIs (like runGemini) to the renderer,
    - a minimal index.html (or equivalent UI entry) that the BrowserWindow loads,
    - a renderer script (if needed) to handle UI interactions (as in the POC's renderer.js which listens for events and updates the DOM),
    - and any back-end file (server.js or similar) needed for Gemini CLI integration.
    - A default .gitignore file should also be included (to ignore node_modules, build artifacts, etc.), which is a standard practice for new projects.
- **Packaging & Build Configuration:** A working Forge config (forge.config.js) is a must, because users will want to package the app. The template's Forge config should list appropriate makers (Windows Squirrel, Mac .zip or DMG, Linux deb/rpm) with

default configs. It should also include any necessary Forge plugins (the POC uses @electron-forge/plugin-fuses and @electron-forge/plugin-auto-unpack-natives by default). Essentially, the template is "done" when the developer can not only run npm start successfully, but also run npm run make and get an installer or packaged app for their platform without errors.

- **Demonstration of Key Feature:** The template must demonstrate its headline feature – in this case, that means showing that Gemini CLI can be invoked. The Definition of Done includes having a sample prompt or action that uses Gemini CLI and returns output to the UI. For example, the template might include a gemini-templates folder with one example prompt file, and a UI button that triggers running that prompt via the CLI. When the user clicks it, they should see a result (or at least a success message) come back. This proves that all the wiring (IPC, child process spawning, etc.) is correct. If any part of that loop fails, the template isn't truly "done." (During development, one should test this end-to-end on all major OSes to ensure, for instance, that the PowerShell vs shell script logic works on Windows vs Linux/macOS.)
- **Documentation and Usage Instructions:** A template isn't done without documentation (more on documentation in the next section, but it's worth listing here). At minimum, the scaffolded project should include a **README.md** that explains: how to install dependencies, how to run the app in development, how to build it for production, and how the Gemini integration works at a high level. It should also mention any prerequisites (for example, if the Gemini CLI needs to be installed separately or if an API key is required for it, those details should be in the README). Create React App, for instance, generates a README with instructions for the newly created app by default – our template should do the same, tailored to Electron/Gemini specifics.
- **Quality and Safety Checks:** While a simple template might not include a full test suite, it's good if the template sets up basic tooling. This could mean:
    - a linting setup or at least a note in package.json (the POC has a placeholder lint script),
    - possibly TypeScript configuration if TypeScript is desired (or a clearly working JS configuration if not),

    - ensuring no critical errors or warnings on startup (e.g., no deprecation warnings from Electron or Node).
    - Optionally, one basic test (using a minimal framework like Jest or Mocha) could be included to show how to write tests for an Electron app, but this is optional. Many templates skip tests to stay simple, but they at least don't preclude adding tests.

- **Extensibility and Cleanliness:** The template is "done" when a developer can easily start building on it. That means the code should be clean and commented where non-obvious. For example, marking sections like // Electron Forge automatically generated code above or // Gemini CLI integration below could help. It also means that if the developer, say, wants to swap out the UI framework or add a new route, they shouldn't have to untangle messy or interwoven code. In practice, this criterion is met if the template code is modular (e.g., separate files for main, preload, renderer, server) and not too intertwined.

To put it simply, **Definition of Done = The template serves as a polished starting point**: it runs without additional tweaks, demonstrates the core integration, includes all necessary pieces (and nothing extraneous), and guides the user on next steps. If any required piece is missing (for example, forgetting to include the content security policy and then the fetch to localhost fails due to CSP – that would mean not done), it would not be ready. The template should be tested from a fresh perspective to ensure it's truly turnkey.

Documentation, Opinions, and Optional Features (Catering to Beginners)

When creating a template intended for both personal and public use – especially targeting **beginners or less experienced developers** – documentation and design decisions ("opinions") must be handled with care:

Comprehensive Documentation: Great templates come with great documentation. This includes:

- **README.md:** As mentioned, a README should ship with the template. It should be beginner-friendly, walking through setup and explaining the project structure. For example, it can have sections: "Getting Started" (how to run dev and build), "Project Structure" (listing key files like main.js, preload.js, etc., with a sentence on each), and "How the Gemini Integration Works" (explaining the flow: UI calls window.api, which

calls Express, which calls the CLI, etc.). If there are any steps required (like installing Google's Gemini CLI or setting environment variables), those should be clearly spelled out.

- **Inline Comments:** The code itself can teach. Beginners will read the template's code to learn how it works. Important sections of code (like the IPC event handlers, or the security settings) should have brief comments explaining their purpose. For instance, a comment in forge.config.js about what fuses do, or a comment in preload.js noting why contextBridge is used (to safely expose runGemini without exposing Node). This way, the template doubles as an example project.
- **External Links:** If relevant, link to external resources for further learning. For example, a link to Electron Forge docs, Electron's security tutorial, or Gemini CLI's documentation in the README can be helpful for those who want to dig deeper. Since this template might be many people's first exposure to Gemini CLI, a pointer to Gemini CLI usage docs or Google's guide would be very useful.

Managing Opinions: An "opinionated" template is one that makes a lot of choices for the developer (about structure, libraries, etc.), whereas an unopinionated one leaves more choices open. We need a balance:

- The template should be **as unopinionated as possible in areas unrelated to its purpose.** That means, aside from integrating Electron and Gemini, it should not enforce specific choices for unrelated things (UI library, state management, styling, etc.). This is why we exclude UI frameworks by default and keep the code vanilla. It gives newcomers a clean slate. Electron itself is quite unopinionated about structure, and our template should reflect that by not adding unnecessary constraints.
- However, the template will inevitably have *some* opinions – for example, using Express for the backend or using fetch/IPC to communicate results. These are reasonable defaults chosen for simplicity and clarity. It's fine to have these as long as they are documented and not too hard to change. For instance, if a developer later prefers using Electron's ipcMain.invoke instead of an Express server, they should be able to swap that without rebuilding the whole app. Our use of Express in the POC is an opinion (one could directly spawn the Gemini process in the main process instead). We should document why we chose that ("Using a small Express server to manage the AI process, which keeps the main process responsive and allows easy HTTP fetching from the

renderer") so that users understand the rationale. Good templates often explain their design decisions, helping learners grasp not just the "what" but the "why."

- If certain opinions might not suit everyone, consider making them **optional or easy to modify.** For example, if TypeScript is used, not everyone likes TS – but since Electron Forge provides separate templates (JS or TS), we could either provide two variants or document how to disable TS if desired. In many cases, though, it's acceptable to pick one path (perhaps TypeScript, since it's increasingly standard) and then just accommodate users via docs if they want pure JS. The key is not locking users in too tightly.

Handling Optional Features: A template can't include every possible feature, but it can guide users on how to add them:

- **Don't include, but document:** For features like database integration, custom menus, or auto-update, it's often best not to include them by default (to avoid clutter), but perhaps mention in a README section "Possible Extensions" or comments like "// If you need X, you can do Y." For example, "If your app needs to save data, you can use a database or local file. Electron's official documentation on storage can be found here…". This way, beginners know the template is a starting point and they have directions for common next steps without those features being baked in.

- **Provide configuration hooks:** If something might be commonly toggled, ensure the template isn't preventing it. For instance, if the template is JavaScript but a user wants to add TypeScript, is it straightforward? (Maybe include a commented tsconfig.json or at least no code that would break if renamed to .ts files.) Or if a user wants to use a different port for the local server, can they change one constant easily? These little touches make the template more **extensible**.

- **Community and Examples:** For public templates, it's good to reference how others do it. For example, Electron Forge's own template uses Webpack and includes an example TypeScript config, which is a light opinion that many accept. Create React App was highly opinionated about using Webpack and hiding the config, which was great for beginners (no config needed) but advanced users sometimes chafed at it. We likely want to lean toward the Electron Forge philosophy of providing the common setup but still allowing extension. The Electron team explicitly mentions that CLIs can "enforce guidelines on structure" which is helpful for beginners. In our case, the

Gemini-Electron template will enforce a certain project layout and method of integration (that's our guideline), and that's fine – it provides a learning path. We just don't want to enforce things unrelated to that goal.

Usability for Beginners: Ultimately, the template should be *approachable*. That means:

- It runs with one command and does something visible (we covered this).
- It doesn't overwhelm with too many files. Every file included should have a reason that a beginner can grasp. (If the template is minimal, the number of files is small anyway. Contrast with a very heavy boilerplate that might have dozens of files – that can scare off newcomers. Our target is closer to the "small canvas" approach: e.g., Create React App's initial structure is just a few folders and files needed for a basic app, not a full enterprise project.)
- The tone of documentation should be friendly and explanatory. Possibly anticipate common questions in comments (e.g., "Why use an Express server instead of calling Gemini CLI directly? Because X, Y…").
- If the template is to be used by a broad audience, consider versioning and updates. A changelog or release notes might be overkill for a template, but any breaking changes or required environment setup (like "Gemini CLI version X is needed") should be communicated if someone updates the template.

Comparisons in Summary: To put this into perspective, here's a brief comparison of how similar tools handle these aspects:

Template / Tool	Default UI	Back-end Included	Opinions	Beginner Aids
Electron Forge (base)	Basic HTML window with minimal content (no framework by default).	No built-in back-end beyond Electron's main process (unless using specific template).	Low to moderate – structure and build are set, but you choose frameworks (or use official React/Vue templates for those).	Documentation on electronforge.io, comments in config. No interactive tutorial but straightforward CLI.
Create React App	Yes – a sample single-page React app (logo, some text) is generated to illustrate a working UI.	None – it's front-end only (no server or DB included out-of-box).	High in front-end choices (forced to use React + its build config), but zero concern for back-end (user adds if needed). Aimed at quick start with zero config.	Excellent README and script hints, hides complex config to not overwhelm. Very beginner-friendly; you run it and see results immediately.
Tauri (vanilla template)	Yes – a minimal HTML/JS UI ("Hello, World" type content, plus an example button to call a Rust command).	Yes – includes a minimal Rust back-end with an example greet command, exposing how front-end calls back-end.	Moderate – enforces Rust for back-end and a certain project structure, but lets you choose your front-end framework or vanilla JS.	Good official docs, the template itself prints messages on how to run, and code comments are provided. Multiple template options (vanilla, React, Svelte, etc.) to suit preferences.
Electron + Gemini	**Yes – minimal UI** (e.g., a couple of buttons and an	**Yes – minimal Node/Express server** for	**Low** – focuses only on Electron and Gemini integration. Does	**Beginner-focused**: Includes a README with usage instructions, code comments explaining key

Template / Tool	Default UI	Back-end Included	Opinions	Beginner Aids
Template (ours)	output area to demo Gemini CLI responses). This shows the integration in action and can be easily replaced.	Gemini CLI. Only the routes needed for Gemini (e.g., /run-gemini) are included, nothing more.	not force any front-end framework or extra features. Opinions limited to technical necessities (e.g., using fetch/Express for IPC).	sections, and a clean layout. Optional features (UI frameworks, DB, etc.) are not included but guidance is given on how to add things. The aim is a **clean, minimal, extensible** starting point that one can understand and build upon.

(The table above compares our intended template with similar scaffolds. It underscores that a small demo UI and minimal necessary back-end are common, and heavy opinions are typically left out to favor extensibility and beginner-friendliness.)

Actionable Recommendations

Based on the analysis, here are concrete recommendations to ensure the Electron Forge + Gemini CLI template stays **clean, minimal, and extensible** while being beginner-friendly:

- **Keep It Lean:** Every file and dependency should earn its place. Include only what a majority of users need to start. If something is nice-to-have but not essential (like a fancy UI library, a database utility, etc.), leave it out. This will result in a project that's easier to navigate for first-timers. Remember the Create React App principle: "only the files you need to build your app".
- **Provide Clarity through Example, Not Complexity:** Do include a tiny working example of the core functionality (Gemini query -> response) so users can learn by example. But make that example simple. For instance, one template prompt (maybe a hardcoded prompt asking Gemini "Hello, are you working?") and a button to trigger it is enough to showcase the flow. This gives users a pattern to follow for their own prompts. Avoid the temptation to add multiple different examples or a complex demo app.
- **Document Everything Relevant:** Ensure the generated project has a **well-written README** and that the code has helpful comments. The README should also highlight the next steps a user might take (e.g., "To add your own Gemini prompt templates, put them in gemini-templates/ and call window.api.runGemini('<name>') with your template name"). It should clarify any environment setup (like "Make sure you have @google/gemini-cli installed or accessible via npx"). Good documentation will make the template useful for a public audience, not just the author. Beginners especially appreciate step-by-step guidance.
- **No Lock-In of Choices:** The template should welcome extensions. For example, if a user wants to use React, they should be able to install React and adjust the renderer accordingly without fighting the template. This might mean structuring the code so that the renderer logic is decoupled (perhaps all in a renderer.js that can be replaced with a React entry point later). Similarly, if someone doesn't want the Express server and instead wants to use direct IPC, they should be able to remove server.js and tweak a few lines in preload.js and still have things work. In short, favor a modular design.

- **Test on Fresh Eyes:** Before considering the template "done," test it as if you are a new user: generate a new project, follow the README, and see if everything makes sense. Have a beginner (if available) try it and give feedback. If they're confused by any file or step, refine the template or docs. The goal is that a newcomer can use this template to successfully create an app and feel empowered, not lost.
- **Draw from Proven Templates:** Continue to mirror the aspects of well-regarded templates:
    - Use Electron Forge's built-in templates as a baseline (for structure and config) since those are officially recommended.
    - Emulate Create React App's friendliness – for instance, CRA prints a message in the console after creation with next steps. Our CLI could do similarly (e.g., "Project created! Run npm start to launch the app. This will open a window where you can press the buttons to see Gemini in action.").
    - From Tauri's approach, note how they allow choosing frameworks. We might not implement an interactive prompt in Gemini CLI (though it could be a future enhancement – e.g., gemini-cli init could ask "do you want to use React? Y/N"). For now, at least ensure the base template doesn't prevent those additions. Possibly in documentation, mention "You can integrate this with frameworks like React or Vue; see [link or guide] for an example."
- **Maintain Extensibility:** As the template will be used for both individual projects and possibly by a community, make sure it's easy to maintain. Keeping it simple not only helps users but also makes the template easier to update when Electron or Gemini CLI versions change. Fewer moving parts means fewer things to break with updates.

By implementing these recommendations, the Electron Forge + Gemini-CLI template will fulfill its ideal scope: it will **focus on the essentials (Electron platform + Gemini AI integration)**, and provide a crystal-clear, beginner-friendly starting point. It will avoid the pitfalls of over-engineering or excessive opinions, instead embracing a **"minimal but complete" philosophy** that invites developers to make it their own.

Ultimately, the template's success will be seen in how comfortably a newcomer can go from "zero" to a running Electron app with AI capabilities. If they can do that quickly, understand how it works, and then smoothly expand on it to create their dream application, then the

template has done its job. Keeping the scope tight and the purpose clear is the surest way to achieve that success.