Gemini-CLI Template Design: In-Depth Analysis and Recommendations

Prerequisites & Environment

Beginner-friendly CLIs must clearly define and handle all prerequisites to avoid user frustration. **Electron Forge** requires Node.js (v16.4+), Git, and a Node package manager (npm, Yarn, or pnpm). **Tauri** has a heavier setup: developers must install Rust and OS-specific build tools (e.g. Visual C++ build tools and WebView2 on Windows); if using a JavaScript frontend, Node.js is needed too. **NW.js** is more lightweight – it runs on Node and Chromium, so having Node.js (LTS version) installed is recommended. **Create T3 App** and **Create React App (CRA)** both simply need Node.js (recent LTS, e.g. Node 14+ for CRA) and npm/Yarn; these CLIs assume a basic Node environment. All tools support cross-platform development (Windows, macOS, Linux), though building installers for another OS may require running on that OS or using CI services.

Comparison of Key Prerequisites:

Tool/CLI	Key Prerequisites & Environment Setup
Electron Forge	Node.js ≥16.4, Git, and npm/Yarn/pnpm. Cross-platform, but packaging for Mac/Windows requires respective OS for code signing.
Tauri	Rust toolchain (cargo) + OS build deps (e.g. Xcode CLT on Mac, VC++ Build Tools + WebView2 on Win). Node.js if using a JS frontend.
NW.js	Node.js (latest LTS) installed in PATH; NW.js SDK binary for dev. Fewer special requirements, but native Node modules need build tools per OS.
Create T3 App	Node.js (LTS) and npm (or yarn/pnpm). Uses npx (comes with npm 5.2+) to run the CLI. No other global deps.
Create React App	Node.js ≥14 on dev machine. npm/Yarn (npm 6+ for npm init or use npx). Git recommended but not required.

Recommendations: The Gemini CLI should proactively ensure the environment is ready. On invocation, it can perform pre-flight checks: e.g. verify the Node.js version (and Rust, if applicable) and give a clear error if not met ("Gemini CLI requires Node.js 18+, please update your Node version"). Document all prerequisites up front in README (mirroring Forge's

clarity) and provide links to installation instructions. If the template targets desktop apps, include notes on OS-specific needs (for example, “For Mac builds, install Xcode Command Line Tools” similar to Tauri’s docs). For beginner friendliness, the CLI could detect missing dependencies and print guidance – for instance, if cargo is not found but the user chooses a Rust-based option, prompt “Rust not found. Please install Rust (e.g. via rustup) before proceeding.” (Tauri’s create-tauri-app does something similar by listing missing packages and how to install them). Additionally, use environment variables or configuration only if necessary; avoid requiring beginners to set env vars just to get started. Keep the initial environment setup minimal and clearly explained.

Init & Customisation

How the CLI initializes a project greatly affects user experience. Modern CLIs use either one-shot commands with sensible defaults or interactive wizards for customization. **Electron Forge** uses a non-interactive initializer (npx create-electron-app) with optional flags for template selection. By default it creates a project using a Webpack or Vite template (developers can specify --template=webpack/vite or their TypeScript variants). **Tauri** provides an interactive scaffolding tool: running npm create tauri-app@latest (or cargo create-tauri-app) will prompt for the app name, identifier, front-end framework (React, Vue, Svelte, etc.), language (TS/JS or Rust), and package manager. This guided setup helps beginners choose technologies step-by-step. **NW.js** does not have an official CLI scaffold; typically one would initialize a web app (e.g. with CRA or Vite) then add NW.js – for example, create a React app and install nw and adjust configs manually. **Create T3 App** is highly interactive: npx create-t3-app asks for your project name, whether to use TypeScript or JavaScript, then lets you select which stack features to include (Prisma, NextAuth, Tailwind, etc.) via checkboxes. It even initializes a git repo and installs dependencies with one confirmation. **Create React App** opts for simplicity: npx create-react-app my-app generates a project with no questions asked, though you can pass a template name for TypeScript or other templates. CRA thus sacrifices customization for a quick, one-command setup; any further configuration (like switching to TypeScript) is done via flags or by ejecting later.

Feature Comparison: Initialization and Customization Options

Tool/CLI	Initialization Command & Options	Customization Features at Init
Electron Forge	`npx create-electron-app [--template=webpack	vite
Tauri	npm create tauri-app@latest (or cargo tauri init for existing project). **Interactive** wizard.	Prompts for project name, app identifier, frontend framework (React/Vue/etc.), package manager, and TS vs JS. Adapts generated template to choices (e.g. sets up chosen UI library).
NW.js	*(No official initializer CLI).* Use a web project generator (CRA, Vite, etc.), then install nw SDK manually.	No built-in customization – developers manually configure package.json (adding NW.js fields like "main" and "node-remote") and create a startup script (main.js).
Create T3 App	npm create t3-app@latest (or yarn create t3-app). **Interactive** CLI.	Prompts for app name, TypeScript vs JavaScript, and which optional stack components to include (each of tRPC, Prisma, Tailwind, NextAuth, etc.). Tailors the project to selected options (each piece is optional by design).
Create React App	npx create-react-app <name> [--template <name>]. Non-interactive quick start.	Minimal customization: can append --template typescript (or use official/community templates) to start with TS or a specific setup. No interactive feature selection; advanced customization requires ejecting or manual changes later.

Recommendations: Gemini CLI should aim for a balance between **ease-of-use and flexibility**. For absolute beginners, a single-command default (like CRA's) is comforting – e.g. npx create-gemini-app myApp should scaffold a working app with sensible defaults. However, to accommodate varying needs, implement an **interactive mode** (triggered automatically or via a --interactive flag) that asks key questions: project name (with default), programming language (JavaScript or TypeScript), and optional features or modules. This approach follows Create T3 App's success in guiding users through choices without overwhelming them.

Provide a few high-level template variants – for example, a "basic" template vs. one with extra example code – or allow skipping certain components. Flags should also be available for power users and CI automation (e.g. --no-git to skip Git init, --template=ts to force TypeScript non-interactively). By default, choose beginner-friendly options: use TypeScript if the user is comfortable (TypeScript improves maintainability, as T3 advocates, but perhaps ask first) and include minimal sample code that demonstrates structure without clutter.

After initialization, the CLI should not tightly control the project (avoid a heavy post-install scaffolding runtime). Instead, clearly explain how to adjust settings. For instance, if the template uses an Electron backend, include a config file (or comments in code) where users can enable/disable features. If environment-specific configuration is needed, generate .env.example files to guide users. Keep initial customization simple and avoid requiring newbies to immediately edit complex config files. In summary, **make the "happy path" zero-config**, but offer an interactive setup for customization, similar to T3 and Tauri's approach, to cater to those who need it.

Directory Structure & Conventions

A well-organized directory structure is crucial for maintainability and for helping new developers orient themselves. Each framework has its own conventions: **CRA** creates a minimalist structure – a /src folder for all React code, a /public folder for static assets, and configuration hidden in build scripts. Everything unnecessary is stripped away to avoid confusion. **Electron** apps typically split code for the main process (Electron's back end) and the renderer (front end). For example, an Electron Forge project (Webpack template) will have an entry point for the main process (like src/main.js) and a separate bundle for the UI (e.g. src/renderer.js and an index.html in a static folder). Forge's default templates put source files under a /src directory and include an index.html along with preload scripts, etc., mirroring best practices of separating concerns. **Tauri** projects have a distinct structure: the front-end (web code) resides in a usual /src (for React/Vue/etc.) or a framework-specific structure, and there is a **src-tauri** folder which contains Rust source code, a tauri.conf.json config, and other native resources. This clearly delineates the UI and the back-end (Rust) part of the app. **NW.js** projects often follow web app structure too (if using a framework like React, you have the same /src and /public as CRA), with the addition of NW-specific files such as a package.json manifest (defining NW.js settings) and maybe a main.js to launch the app. There isn't a strict convention enforced by NW.js beyond that – developers are advised to organize code as a normal Node/

web project, which means it's on the developer to keep things orderly. **Create T3 App** yields a structure derived from Next.js conventions: you'll have a pages/ (or src/pages) for Next.js routes, an api/ route directory (for serverless functions or tRPC endpoints), and typically a /prisma directory for the database schema, etc. T3's template also includes configuration files like next.config.js, and maybe a env.mjs or similar to load environment variables. It strives to scaffold a *modular* monolithic project, so front-end and back-end code (routers, procedures) live in one codebase with clearly separated folders (often src/server for backend logic, src/pages and src/components for front-end).

Recommendations: Adopt a **clear, conventional structure** that reflects the dual nature of the app (if Gemini CLI involves both front-end and back-end components, as implied by including "Back-end Extension"). For instance, one approach is:

- **Top-level directories**: use a src/ for all application source code. Under src/, have subfolders like renderer/ (or frontend/) and main/ (or backend/). For an Electron-based template, src/main/ can hold the main process (Electron entry, menu, preload, and any Node server code), while src/renderer/ holds UI code (pages, components, etc.). This separation matches how developers think of Electron apps and is beginner-friendly (it's obvious where front-end vs back-end code lives). In a web context (no Electron), you might use src/ for all client code and a separate server/ directory for an Express/Next.js API server.

- **Static assets and resources**: Provide a public/ or static/ directory for static files (HTML, images, icons). For example, CRA and Next use a public folder – Gemini CLI should do the same for things like an HTML shell or icon files. If using Electron, include an assets/ folder for things like icons or native binaries, and ensure the build process knows to include them.

- **Configuration and scripts**: Keep configuration centralized. For example, Electron Forge uses a single forge.config.js or a config.forge field in package.json to configure build, makers, and plugins. We should do similarly – e.g., a gemini.config.(js|json) that contains CLI-specific settings (ports, whether to enable auto-updates, etc.), so users have one place to look. Avoid scattering config across many files. For web frameworks, embrace their conventions (Next.js uses next.config.js, Tauri uses tauri.conf.json in src-tauri/). The Gemini template should generate those files with sensible defaults.

- **Naming conventions**: Use intuitive names and file extensions. If TypeScript is used, ensure files are .ts/.tsx accordingly. Name the main Electron entry main.ts (or .js) and the preload script preload.ts so their purpose is clear (as opposed to something ambiguous). The consistency will help beginners not get lost. Include comments in key files explaining their role (e.g., a comment at the top of main.js saying "// This is the Electron main process code – it runs in Node.js and controls the app lifecycle.").
- **Example content**: Provide a simple example component or route so users can verify everything works. For instance, CRA includes a simple App.js and index.html. Gemini CLI could include a basic UI (a welcome screen or demo component) and, if a backend is present, a sample API endpoint or IPC call. However, **keep examples minimal** and clearly marked, so users know they can delete them. Too much boilerplate can overwhelm beginners.
- **Documentation of structure**: In the README or project docs, outline the directory tree and what each folder is for. This helps new developers quickly locate things. For example: "/renderer: React front-end code; /main: Electron main process; /server: Node.js API server (optional)" etc. This mirrors the approach of some templates that explicitly list the generated files to acquaint users with the layout.

By adhering to common conventions (like separating source and static assets, grouping backend vs frontend logic), the Gemini template will feel familiar to those who have used similar tools, and it will guide newcomers on where to put new code. Consistency is key: decide on either a monorepo style (if the back-end might grow into a separate service) or a unified project, and structure accordingly, then use that structure in all examples and documentation.

Back-end Extension & Gemini CLI Hooks

This section covers two related aspects: extending the template's back-end functionality and providing hooks or plugins in the CLI for customization. In similar frameworks, **extensibility** is often achieved through plugin systems or generator commands. For example, **Electron Forge** supports plugins (for custom packaging, publishing, etc.) and defines lifecycle hooks where custom code can run during build processes. Forge's configuration allows defining functions for events like prePackage, postPackage, etc., enabling developers to inject behavior at those stages. This is a powerful way to extend build or release workflows without forking the tool. **Tauri**, on the other hand, emphasizes a plugin system for runtime features: developers can add official or custom **Tauri plugins** (for example, for HTTP, Filesystem, etc.) to extend

the Rust backend capabilities, all configured via the CLI or tauri.conf.json. Tauri also provides a GitHub Action to streamline CI builds, which is an extension in the CI context. **NW.js** is relatively bare-bones regarding extension; since it's essentially Node, developers extend it by using Node modules or writing their own code. There is no official NW.js CLI to add modules, but the flexibility of Node means you can integrate an Express server, database, or anything directly in your code (with care to ensure the NW context allows it). **Create React App** is not extensible via its CLI after creation – its philosophy is "eject" if you need to customize the build. That said, CRA does allow using different templates initially, and community tools like CRACO emerged to let advanced users tweak the config without full eject. **Create T3 App** similarly doesn't have a plugin system after scaffolding – it generates the project and hands you the code. The T3 CLI itself doesn't persist as a dev dependency to run further commands (as noted in their FAQ, it's a one-time scaffolder). Instead, all extension is manual: you add new tRPC routers, Next.js pages, etc., by writing code.

For **Gemini CLI**, since it appears to target a full-stack (or twin) application (possibly an Electron app with a Node/Express back-end, given the mention of server files), we should enable two levels of extension:

1. **Back-end extension**: Make it easy to add new backend capabilities. This could mean if the template includes an Express or FastAPI server, provide a consistent pattern for adding routes or API endpoints. For example, include a routes/ folder with an index, and instruct that adding a new file there auto-registers a new route. Or if using something like tRPC/Next.js API routes, document how to add a procedure. We might consider providing sub-commands in the CLI like gemini add:route <name> to generate a boilerplate route handler, or gemini add:module <name> to scaffold a common feature (similar to how Angular's CLI generates components/services). This is advanced, but can greatly speed up development for those who want it. If implementing such generators is too much, at least supply clear examples in the template so users can copy-paste to extend the backend.

2. **CLI hooks and plugins**: Design the Gemini CLI to be extensible itself. This means allowing custom scripts or modules to hook into the CLI's lifecycle. Taking inspiration from Forge – where one can specify a function to run at certain events (start, package, etc.) – we can allow users to define, say, in gemini.config.js, some hooks like postInit (runs after project creation), preBuild (runs before bundling), postBuild (after build,

perhaps to perform custom packaging or run tests), and so on. These hook functions could simply be Node scripts that the CLI will import and execute. This gives power users a deterministic way to extend behavior (as opposed to having to rely on modifying the CLI's source or writing external scripts). Additionally, a **plugin API** could be offered for larger extensions: e.g., a way to drop in a Node package that augments the CLI with new commands. Electron Forge and Tauri both highlight extending core functionality via plugins. For instance, Forge lets you write custom "makers" to package in new formats, and Tauri's plugin system lets you add capabilities to the app easily. Gemini CLI could allow plugins that add new subcommands (for cloud deployment, for example) or integrate with external services.

If Gemini CLI is influenced by the AI agent context (the term "Gemini" hints at Google's Gemini AI), it's worth noting how **Gemini CLI (Google's)** uses hooks: an issue from its repository suggests a hooks system for custom automation, where users can register scripts to run on certain events (pre-tool use, post-tool use, etc.). While that context is AI-specific, the general idea is relevant: **shifting from hardcoded behavior to user-configurable hooks increases flexibility**. We should apply this principle to our CLI design.

Recommendations: Implement a **hook system** in Gemini CLI to allow custom logic during the dev/build lifecycle. For example, allow a gemini.config.js export like:

```
module.exports = {
  hooks: {
    preBuild: async () => { /* e.g., lint the code */ },
    postBuild: async () => { /* e.g., run electronegativity audit or notify */ }
  }
}
```

This resembles Electron Forge's hooks (e.g., generateAssets, prePackage) and gives developers a clear extension point. Document the available hook events and their parameters. Keep the API simple (async functions that the CLI will call).

Additionally, consider a **plugin interface** for larger extensions. For instance, a Gemini CLI plugin could be an npm package that exposes a function to extend the CLI (maybe adding commands or altering the config). The CLI can load plugins declared in the config. This way,

the community or internal teams could develop plugins for things like deploying the app, integrating with CI, or adding a new template.

For **back-end extension**, if the generated app includes a server (like an Express app listening on localhost for the front-end to call), scaffold it in a way that is easy to grow. Perhaps use a modular router setup (each route in its own file) and include one sample route as a template. Write clear instructions in the code or README: "To add a new API endpoint, create a file in /src/main/routes and it will be auto-loaded." This removes guesswork. If using a framework like Next.js or tRPC (as T3 does), highlight how to add a new procedure or page. Essentially, **provide patterns, not just code** – beginners benefit from having a model to follow when extending the back-end.

Finally, incorporate **guardrails and guides**: since back-end code can be complex, include hints for common needs. For example, if the template uses a database, include a note on how to add a new model/migration. If no database by default, at least structure the code such that adding one later (or connecting to an API) doesn't require restructuring the whole project.

In summary, **make the CLI and template extensible**. Allow the CLI to be extended (hooks/plugins for custom behaviors), and make the template code extensible by modular design and maybe CLI subcommands for code generation. This ensures that as users' apps grow, they aren't constrained by the initial setup. They can evolve the project without "ejecting" or starting from scratch, which is a strong point for advanced users and prevents beginners from hitting a wall as they learn more.

Security Hardening

Security must be baked into the template to protect beginners (who may not be aware of all risks) and to follow best practices. Electron and similar tools have well-documented security guidelines that we should incorporate. **Electron** specifically urges developers to **disable remote code execution and dangerous APIs by default** – for instance, do not enable Node.js integration in any browser window that loads untrusted content, and do enable contextIsolation so the renderer and preload run in separate contexts. Modern Electron apps should start with contextIsolation: true and nodeIntegration: false in BrowserWindow settings. The Gemini template should ensure these are the default unless there's a specific reason otherwise. Also, set a **Content Security Policy (CSP)** for any loaded web content – e.g., only allow scripts from 'self' – to mitigate injection risks. Electron prints runtime warnings if security recommendations are not met (e.g., if Node integration is on), so by following the checklist (load secure content,

enable isolation, sandbox, etc.), our template will start with a clean bill of health. We can integrate Electron Forge's **Electronegativity** plugin to scan the app for common vulnerabilities – this tool checks for insecure patterns and misconfigurations automatically. Running such checks as part of npm run build or in CI can catch issues early.

For frameworks like **Tauri**, security is a selling point: Tauri uses the system webview and **does not serve an HTTP server** by default, eliminating a class of vulnerabilities (no local host server to potentially exploit). If Gemini CLI can leverage this (if using Tauri or similar), it's advantageous. In any case, the template should avoid practices like using eval() or loading code from remote URLs at runtime. If the template is Electron-based and needs to load remote content, explicitly document the risks and perhaps demonstrate using the <webview> tag with proper allowlist rather than BrowserWindow with Node integration.

NW.js historically allows a lot of leeway (Node and DOM in the same context), which can bypass many web security restrictions. If providing an NW.js option or similar, set the manifest fields conservatively: e.g., use "node-remote" in package.json to restrict which domains can access Node APIs (perhaps only 127.0.0.1 or none, if not needed). Encourage using nwdisable for iframes if loading external content. Basically, emulate Electron's stance: no untrusted code runs with Node privileges.

Beyond runtime settings, consider **supply chain and deployment security**: Lock dependency versions with a lockfile and encourage running npm audit. The template's default should have no high-severity vulnerabilities in its dependencies (choose well-maintained libraries). Enable GitHub's Dependabot or similar for updates by default (perhaps include a dependabot config). For back-end code, include basic security middleware if using Express (e.g. use Helmet for HTTP headers, use parameterized queries or an ORM to avoid SQL injection if a database is included).

Security Hardening Checklist (to implement in template/CLI):

- **Secure Defaults**: Configure Electron's BrowserWindow with secure defaults (no Node integration, contextIsolation on). Turn on sandboxing if possible. In Tauri, use the permission allowlist – only enable APIs that are needed (the tauri.conf.json allows fine-grained control). For web apps, ensure production builds have no developer backdoors (like CRA's dev server proxy isn't relevant in prod).

- **Content Security Policy**: Include a CSP meta tag or header in the default HTML (e.g., <meta http-equiv="Content-Security-Policy" content="default-src 'self';"> which can be adjusted as needed). Beginners might not know about CSP, so providing one out-of-the-box is valuable.
- **IPC/Bridge Security**: If our app requires communication between front-end and back-end (e.g., Electron's ipcRenderer), expose only narrowly-scoped APIs. Use a preload script to expose safe functions (e.g., window.api.send(channel, data) that only whitelisted channels can use). Validate all inputs coming over IPC. The template should illustrate this by example – e.g., show a secure IPC listener that checks message origin or includes an ACK mechanism. Also, encourage checking event.senderFrame.origin or similar to ensure messages truly come from your app.
- **Dependencies**: Avoid known risky dependencies. If using Electron, maybe include the Electronegativity plugin as mentioned, so running npm run make also prints any security warnings (Forge's plugin runs after packaging to spot issues). In documentation, remind users to keep Electron (or Tauri, etc.) up-to-date because framework updates often patch security issues.
- **Code Signing & Integrity**: While more of a distribution concern, it's worth treating as a security measure. Recommend code signing for production builds (and if possible, provide a placeholder in CI for where to add signing). This ensures the app isn't tampered with when distributed. Tauri has an updater that can check signatures of updates; if our CLI includes an auto-update mechanism, ensure it verifies signatures or hashes of updates to prevent malicious updates.

Recommendations: Gemini CLI's generated projects should pass a "secure by default" baseline. Apply the Electron security checklist items 1–5 from the start – e.g., only load local files or secure HTTPS URLs, disable Node integration in any UI that isn't absolutely trusted, enable contextIsolation, and define a strong CSP. Provide comments in config files explaining these settings so beginners learn *why* they're there ("// Node integration is off for security: the UI cannot directly access Node APIs"). This educates users not to simply turn things on without understanding the risk. Use tools to automate security checks: we can integrate Doyensec's Electronegativity for Electron – possibly as an npm script npm run audit:sec – so users can scan their app after making changes.

For any back-end component, enforce secure practices – e.g., if an Express server is included, enable HTTPS in production (maybe provide a sample self-signed cert for localhost testing or instructions to configure one). If the app stores sensitive info, advise using OS keychains or secure storage instead of plain text files. These might be beyond the template's implementation, but mentioning them in README or code comments adds value.

Lastly, consider **hardening CI/CD** as part of security: if GitHub Actions is used to build releases, recommend adding a step to verify the binaries (e.g., checksum comparison) and to not expose secrets (like signing keys) unnecessarily. The template's GH workflow (if we include one) should use secure practices – for example, use encrypted secrets for certificates and not log them. This level of detail shows a commitment to security from development to distribution.

Build, Packaging & Distribution

A hallmark of a good template CLI is smoothing out the build and release process. Each compared tool has a strategy for packaging apps and integrating with CI/CD, particularly GitHub in our assumption. **Electron Forge** provides a one-stop build pipeline: running npm run make will package the app for the current OS (producing an installer like .exe, .dmg, .deb etc. depending on configured "makers"). Forge also has a **publish** command that can automatically create a GitHub release and upload artifacts to it (via its GitHub publisher). This tight integration means a developer can, with the right config and a GH token, push a new version easily. **Tauri** similarly has a robust bundler – it can output app bundles for all major desktop platforms (app, dmg, msi, etc.) using Rust's build system under the hood. The Tauri team even provides a **GitHub Action (tauri-action)** that builds the app for Windows, macOS, and Linux in parallel and attaches them to a release. This greatly simplifies cross-platform distribution for maintainers. **NW.js** doesn't have an official bundler in the base project, but developers use community tools like nw-builder (or manual zipping) to create executables. NW.js documentation lists how to package apps either by copying files alongside the NW runtime or zipping them into a .nw file – it's more manual, but it works. **Create React App** doesn't deal with installers, of course – instead "build" creates an optimized static bundle (npm run build produces static files) which you then deploy to a web server or static hosting. The CRA docs and community provide guidance for deploying to GitHub Pages, Vercel, Netlify, etc., but the tool itself doesn't handle it. **Create T3 App** is similar: it's essentially a Next.js app, so building yields a .next production directory or a Node server bundle. Deployment is up

to the user (often Vercel for Next.js). There's no built-in "release" command since it's not a desktop app template.

Given our context (likely desktop or full-stack local app), the focus is on **packaging executables** and releasing them via GitHub Actions. We assume Gemini CLI apps will be delivered to end-users, so we need to cover building installers for Windows (.exe or MSI), Mac (.app or DMG), and Linux (.AppImage, .deb, etc.).

Recommendations: Provide a consistent build script that produces distributables for all target platforms. We suggest integrating Electron Forge's or Tauri's approach:

- If Electron-based, use Electron Forge under the hood for packaging. This means the template should include Forge config (defining makers like Squirrel for Windows, DMG for Mac, etc.). Running npm run make in the project should generate installers for the current OS by default. For cross-platform builds, document how to do it via CI (since building a Windows installer on Linux requires some hacks or using Wine/VM – easier is to run the build on each OS environment). Given GitHub is the default CI/CD, we can provide a **GitHub Actions workflow** template in the project. For example, include a .github/workflows/release.yml that uses a build matrix (jobs on ubuntu, windows, mac runners). In each job, check out code, install dependencies (including devDeps like Electron), and run the make/package command. Then have the workflow combine artifacts and create a GitHub Release draft with those files attached. We can take inspiration from Tauri's official Action which simplifies this. In fact, if using Tauri, simply instruct users to use tauri-action with minimal config to get cross-platform releases.

- Embrace GitHub Releases as the distribution mechanism. With Forge, for example, one can configure the GitHub publisher in forge.config.js and just call npm run publish to upload artifacts. We can simplify it further by automating in CI: e.g., when a version tag is pushed, the Action runs packaging and uses the GitHub API to create a release. Provide the necessary pieces (maybe use environment vars or a dummy token in config that the user must replace with a real GH token in CI secrets). The idea is that a beginner can follow a step-by-step to set up automated releases without having to research it from scratch.

- Ensure code signing steps are documented or stubbed. For Mac .app signing and Windows signing (if relevant), include placeholders in the workflow (e.g., steps

commented out with “Use signtool here with your cert” or mention Electron Forge’s signing guides). Even if the beginner doesn’t use it initially, it’s good to signal how to integrate it later, as unsigned apps on Mac/Win can be problematic for end-users.

- If the template is full-stack (with a back-end server), consider containerizing or packaging that as well. For instance, if the idea is to deploy the back-end separately (say, a cloud API), provide a Dockerfile or at least instructions to build one. However, if the back-end is only local (for an Electron app), then packaging the entire app covers it.

- For web deployment (in case some Gemini CLI users build purely web apps), integrate a simple deployment. Possibly include an npm script for deploying to GitHub Pages (CRA often had gh-pages for this), or advise how to deploy to a platform like Vercel. It might be beyond scope to automate that in the template, but a mention in README like “After npm run build, upload the build/ directory to your static host or use GH Pages” would help.

- **Artefact naming and versioning**: The template should use the project’s version from package.json for naming releases. Forge and Tauri do this (Tauri’s GH action creates releases with the app version tag automatically). We should ensure the version is managed in one place (probably package.json or a config file) and the build process uses it to name output files. This avoids confusion and ensures users increment versions properly for updates.

- **Size and optimization**: Guide users on trimming the app for distribution. For example, Electron apps often can be slimmed by asar packaging (Forge does this by default) and excluding devDependencies. Tauri produces very small binaries by default. We can mention techniques like tree-shaking, electron-builder pruning, etc., if relevant. Perhaps integrate something like electron-forge-plugin-auto-unpack-natives and other Forge optimizations out of the box so that the app isn’t bloated.

- **Updating**: If feasible, have a plan for app updates. This might be as simple as “when you publish a new release on GitHub, users can download it; consider using electron-updater or tauri’s built-in updater in the future.” Tauri includes an auto-updater that checks GitHub Releases JSON by default, which is great. If using Electron, we might not implement auto-update in the template by default (as it needs a feed URL or service), but at least we can mention it as a next step.

In sum, **automate as much of the packaging and release as possible**. Provide a ready-to-use GitHub Actions workflow that: on a tag or release branch, builds the app for all platforms, runs tests (if any), packages the binaries, and creates a GitHub Release with those binaries attached. This aligns with what developers expect today (CI-driven releases) and saves beginners a lot of time configuring their own.

For example, a simplified snippet in the documentation or comments could be:

```
# .github/workflows/release.yml

jobs:

  build:

    runs-on: ${{ matrix.os }}

    strategy:

      matrix:

        os: [ubuntu-latest, windows-latest, macos-latest]

    steps:

      - uses: actions/checkout@v4

      - uses: actions/setup-node@v4

      - run: npm install

      - run: npm run make   # or tauri build etc.

      - uses: actions/upload-artifact@v3 ...
```

And then another job to create release and download artifacts, etc. We don't need to provide the full thing here, but the template could include it.

Moreover, highlight any **notable differences** between our approach and others: for instance, if using Tauri, the CI can produce all targets from one runner using cross-compilation (Rust can cross-compile Linux <-> Windows in some cases). The official Tauri action actually builds on each OS for native modules, which is the robust approach. We can lean on those community solutions.

Finally, keep **beginner-friendliness** in mind: provide simple commands to execute. In package.json, include scripts like "build:win", "build:mac", "build:linux" that set up the appropriate flags if needed, so a novice can run them locally if they happen to have those OS environments. And produce helpful output – for instance, after packaging, print a message like "Build complete! Installers are in the out/ folder." This kind of guidance makes the experience nicer.

By learning from Electron Forge (which combines packaging and publishing) and Tauri (with its CI integration and small binary focus), Gemini CLI can deliver a seamless build->release pipeline. The user should feel that going from "I wrote some code" to "I have an installer my friend can download" is straightforward and well-documented, rather than a mysterious process.