ChatGPT

Writer's Toolkit (Gemini) – Overview

Introduction and Purpose

The **Writer's Toolkit** is a comprehensive desktop application for creative fiction writers, providing a suite of AI-powered analysis and writing tools to improve their manuscripts [1]. Built on Electron, it runs locally on the user's computer, storing all project files (manuscripts, outlines, world-building notes, etc.) in a `~/writing` directory for privacy and control [2]. The toolkit's purpose is to assist writers through all stages of the writing process – from brainstorming ideas and outlining, to drafting chapters and performing in-depth edits – by leveraging advanced AI models (Google's **Gemini** and formerly Anthropic's **Claude**) for intelligent feedback and content generation [1] [3]. By keeping everything local and user-managed, authors maintain ownership of their work while still benefitting from powerful AI analysis. In summary, the Writer's Toolkit empowers authors to organize projects and **use AI as a writing assistant** for creative and editing tasks, all within a convenient desktop UI [4].

Project Structure and Architecture

This project is organized as an **Electron application**, comprising a **main process** (Node.js backend) and a **renderer process** (frontend UI). Key files in the repository include `main.js` (Electron main process orchestrator), `index.html` and `renderer.js` (the primary UI and its logic), and various script files for tools and dialogs. The Electron main process (`main.js`) is responsible for initializing the app, managing windows, and handling backend operations like file I/O and AI API calls [5]. It loads configuration and state via `state.js` (which uses an `electron-store` for persistent settings) [6], and sets up the **Tool System** by loading all tool modules on startup [7] [8]. The app defines a **dedicated directory structure** for writing projects: each project is a folder under `~/writing`, containing text files (e.g. `manuscript.txt`, `outline.txt`, `world.txt`, etc.) for that project [9] [10]. All tool outputs are saved as timestamped text files in the project directory (to avoid overwriting previous results) [4] [11], reinforcing a versioned, local-first approach.

Electron Main vs Renderer: In the main process, `main.js` uses Electron's APIs to create the application windows and handle inter-process communication. It opens the main window (`index.html` in a BrowserWindow) and additional modal windows for certain dialogs (e.g. project selection or text editor) as needed [12]. The main process also manages integration with the AI APIs: it initializes an **AI API service** client (`client.js`) for Gemini and passes this to tools for making requests [13]. The renderer process (frontend) comprises HTML/CSS/JS that define the user interface. The **main UI (`index.html` + `renderer.js`)** presents a dashboard where the user selects projects and tools (with separate dropdowns for AI-based vs non-AI tools) and initiates tool runs [14] [15]. There are also small HTML/JS components for dialogs, like **project selection** (`project-dialog.html/js`) and **tool configuration/run** (`tool-setup-run.html/js`), and an **editor dialog** (`editor-dialog.html/js` along with an HTML/CSS/JS under `renderer/editor/`) which is a built-in text editor to view or edit the results [12].

Internally, the toolkit is built in a **modular, extensible way**. Each analytical or creative function is implemented as a **Tool class** (in its own `.js` file) that extends a common base class. The `BaseTool` class (defined in `base-tool.js`) provides shared functionality such as file reading/writing helpers and a standardized `execute()` method contract [16] [17]. It also defines `emitOutput()` callbacks to stream tool output back to the UI, and utility methods like `readInputFile()` and `writeOutputFile()` which handle resolving project paths, reading files, and writing results to disk [16] [18]. All tools are registered in a central `tool-system.js` using a **Tool Registry**. This registry maintains an array of tool definitions (TOOL_DEFS), where each tool has a unique `id`, a human-readable title, a description, the Class to instantiate, and a schema of expected options/parameters [19] [20]. On application startup, the Tool System dynamically loads each tool module by file name and makes them available in the UI's dropdowns (sorted into AI or non-AI categories) [21]. This design allows new tools to be added relatively easily – the repository even includes a developer guide ("**how_to_add_a_new_Tool.txt**") with steps to create a new tool script, register it in `tool-system.js`, and include it in the UI lists [22] [21]. The UI uses the registry info to populate the tool selectors and display each tool's description and configurable options (inputs, settings) before running [23].

Local Storage and Configuration: All writing data remains in local text files. The current project's path is tracked in the global app state (in memory and saved via electron-store), so that the application always knows which project folder to read from or write to [24] [25]. Users can easily switch projects or create new ones through the UI, which simply corresponds to changing that folder. An **environment file** or system environment variable provides the **AI API key** (e.g. `GEMINI_API_KEY`) needed for the AI services [26]. The `client.js` will flag if the API key is missing and the UI warns the user to set it [27] [28]. Other settings (like window preferences or last used project) are persisted in an Electron Store config file (named `writers-toolkit-config`) in the user's home directory [29]. The application can be packaged for distribution: **Electron Forge** configuration is provided via `forge.config.js` to create installers for Windows (Squirrel) and macOS (with a DMG, aided by the `create-dmg.txt` instructions) [30]. The packaging is configured to include all necessary files and resources (icons, etc.) and not to bundle the Node modules into an archive (asar disabled for easier file access) [31] [32].

Key Features and Tools

One of the standout aspects of Writer's Toolkit is the **breadth of tools** it offers. These tools range from deep manuscript analysis to content generation utilities, all tailored for novelists and storytellers. Below is an overview of the core tool categories and their functionality:

Manuscript Analysis & Editing Tools

For improving and editing existing writing, the toolkit provides a rich set of analytical tools that scrutinize a manuscript from many angles. Key examples include:

- **Token & Word Counter** – Counts the total words and estimates AI **tokens** in a text, giving writers a sense of length and potential AI usage [33]. This doubles as a quick check that the AI API is functioning [19].
- **Narrative Integrity Checker** – Checks for **consistency** and logic within the story. It can compare the manuscript against the world-building file and outline to find contradictions in characters,

settings, or plot continuity [34] [35]. Different modes focus on world consistency, internal logic, character development, unresolved plot threads, or all of the above [36] [37].

- **Character Analyzer** – Compiles a list of characters and verifies their presence and portrayal across the manuscript, outline, and world files, highlighting any discrepancies (e.g. a character suddenly appearing with no introduction) [38] [39].
- **Plot Thread Tracker** – Identifies distinct plot threads in the story and tracks how they develop and interweave throughout the manuscript [40] [41]. It can even produce a simple visualization (ASCII-based) of plot convergence/divergence, and allows specifying certain threads to focus on (like "romance" or "mystery") [42] [43].
- **Tense Consistency Checker** – Scans the manuscript for shifts in verb tense (past vs. present) that might confuse readers [44] [45]. It reports any inconsistent tense usage, with options to adjust analysis sensitivity and define how chapters are recognized (to reset context per chapter) [46] [47].
- **Conflict Analyzer** – Analyzes scenes, chapters, and overall story arcs for **conflict**: the types of conflict (internal, interpersonal, environmental, etc.), their escalation, and resolution patterns [48] [49]. This helps ensure the story's tension has a satisfying progression.
- **Foreshadowing Tracker** – Identifies instances of foreshadowing (from blatant clues to subtle hints, including "Chekhov's gun" elements) and checks if and how they pay off later in the narrative [50]. It can organize results by chronological order or by type of foreshadowing, so the author can see if promises made to the reader are ultimately fulfilled [51] [52].
- **Dangling Modifier Checker** – Detects **dangling or misplaced modifiers** in sentences [53] [54]. Such grammatical issues (e.g. an introductory phrase that unintentionally modifies the wrong subject) can cause confusion or humor; the tool finds these and suggests fixes. Users can set the sensitivity and focus on specific types (dangling, misplaced, "squinting" modifiers, etc.) [54] [55].
- **Crowding & Leaping Evaluator** – Evaluates pacing based on Ursula K. Le Guin's concepts of "**crowding**" (densely detailed narration) versus "**leaping**" (jumps in time or narrative) [56]. It flags portions of text that are overly detailed or, conversely, transitions that feel too abrupt, helping the writer balance narrative rhythm. Options include adding a text-based visualization of pacing patterns and adjusting what to focus on (crowding, leaping, transitions, etc.) [57] [58].
- **Adjective & Adverb Optimizer** – Reviews the manuscript for overused adjectives and adverbs [59]. The tool identifies opportunities to replace weak descriptions (e.g. "very angry") with stronger nouns or verbs, following common writing advice (even referencing Le Guin's guidance on minimizing adverb overload) [59]. It can operate at different levels of detail and focus on specific categories (qualifiers, adverbs, adjectives, imagery) [60] [61].
- **Rhythm Analyzer** – Measures the sentence length and structural variety to gauge the **rhythm/flow** of the prose [56] [62]. Monotonous sentence patterns or a mismatch between sentence rhythm and the intended mood are reported. The analysis can be tuned by scene type (dialogue vs action vs exposition) and sensitivity [63] [64].
- **Punctuation Auditor** – Reviews the text's punctuation usage, catching things like run-on sentences, missing commas, or odd punctuation habits that could affect readability [65] [66]. It provides recommendations for improving clarity and adheres to chosen strictness levels for punctuation rules [67] [68].
- **Developmental, Line, and Copy Editing Tools** – These are AI-assisted editing passes at different granularity. **Developmental Editing** examines high-level structure: plot holes, pacing, character arcs, themes, point-of-view consistency, etc., across the whole manuscript [69] [70]. **Line Editing** focuses on style and wording on a chapter-by-chapter basis (the user specifies a chapter to edit) [71] [72], making intensive revisions to tighten prose. **Copy Editing** performs a thorough correctness pass on the entire manuscript – grammar, spelling, punctuation, and formatting consistency [73] [74]. These tools prepare the manuscript for publication quality. There are also two modes of

Proofreading: one for purely mechanical errors (typos, basic grammar) [75] and another for plot consistency issues [76], to catch any remaining continuity errors specifically. Both proofreading tools scan the full text and can be set to a language (e.g. English) for locale-specific conventions [77] [78].

(The toolkit's analytical features are notably exhaustive – covering everything from narrative logic to minute grammar details – giving writers a "second pair of eyes" on their work. Many of these analyses would traditionally be done by human editors, so it's unique to have them automated.)

Content Generation & Organization Tools

In addition to analysis, Writer's Toolkit assists with **creative generation** and planning. These tools help writers develop new content using AI:

- **Brainstorm Tool** – Helps generate story ideas and creative prompts. Given an initial idea or even just a request to brainstorm, it produces additional ideas or angles and appends them to an `ideas.txt` file for later use [79] [80]. It can either start fresh or continue building on existing ideas, and it allows the user to specify parameters like genre, desired number of characters, depth of world-building detail, etc., to tailor the brainstorming output [81] [82]. This is great for overcoming writer's block at the brainstorming stage.
- **Outline Writer** – Takes a high-level premise or concept (from a premise file, often an idea generated by the brainstorm tool) and produces a **plot outline** [83]. The outline can be guided by an optional skeleton provided by the user (ensuring the AI fills in details for a structure the author has in mind) [84] [85]. It also can incorporate an example outline or characters list if provided, and parameters like number of sections/chapters, language, genre, and whether to include detailed chapter summaries [86] [87]. The output is saved as `outline.txt`.
- **World Writer** – Using the generated outline and a list of characters, this tool creates detailed **world-building documentation** [88]. It "extracts and develops characters and world elements" by expanding on each character's backstory, the setting, lore, etc., ensuring consistency with the outline's events [89] [90]. It requires certain inputs (the story title, point-of-view style, and the existing `characters.txt` and `outline.txt` files) and produces enriched content for `world.txt` and updates to `characters.txt` with more details [91] [92].
- **Chapter Writer** – When it's time to draft the novel, this tool can generate actual **chapter text** for a rough draft [93]. It uses the outline, a specified list of chapters to write (with their titles from the outline), the world info, and any existing manuscript text to produce new chapter content [94] [95]. Essentially, it's an AI co-writer that fleshes out chapters based on the plan. It supports writing multiple chapters sequentially and appending them to the manuscript file, with a configurable delay between chapters (to avoid hitting API limits) [96]. There are options to control style, such as turning off enhanced dialogue (if by default the AI might emphasize dialogue) [97], or to avoid directly appending (if the user prefers to review each chapter first) [98]. The Chapter Writer thus accelerates the drafting process under the author's guidance.
- **Manuscript Extractor** – A reverse tool that helps writers who already have a draft but want to use the Toolkit's structured approach. The **Manuscript to Outline/Characters/World** tool analyzes an existing full manuscript (`manuscript.txt`) and generates an `outline.txt`, `characters.txt`, and `world.txt` from it [99] [100]. This effectively **onboards an existing novel** into the toolkit by extracting the key structural elements: it infers the outline of chapters/scenes, compiles a list of characters and their descriptions, and gathers world details mentioned in the text. It's especially

useful for “pantsers” (writers who don’t start with an outline) who later want to organize and analyze their draft using these tools [100].

Utility Tools

These are supplementary tools not directly about the story’s content, but useful for formatting and publishing tasks:

- **DOCX Text/Comments Extractor** – If an author has a Word document with editor comments, this tool will parse a `.docx` file and extract all the **comments alongside the text passages they refer to** [101]. The output is a text file listing each comment with context, so the writer can review feedback outside of Word. This is handy for incorporating beta reader or editor comments into the workflow.
- **EPUB to TXT Converter** – Converts an ebook file (`.epub`) into plain text [102]. This can be used to import content or to analyze someone else’s EPUB (for example, to run the analysis tools on a reference text, or to ensure your own exported ebook looks correct as text). The converter preserves basic structure (likely separating chapters) while producing a single `.txt` output.
- **KDP Publishing Preparation** – Assists in preparing for publishing on Amazon Kindle Direct Publishing. It analyzes the manuscript and suggests elements needed for KDP: it can generate **title ideas, a book description, categories, and keyword suggestions** based on the story’s content [103]. Essentially, it uses AI to draft the marketing copy and metadata for the book listing. The tool takes inputs about book type (fiction vs nonfiction) and target audience and can output a polished book description (optionally in HTML format ready for KDP) [104] [105], along with other publishing info.
- **“Drunk Claude” Critique** – A quirky and unique feature: this tool invokes the AI (Claude, in this case) to critique the manuscript while role-playing as if it were “drunk” [106] [80]. The idea is to get a brutally honest (and possibly humorously unfiltered) commentary on your writing. While partly tongue-in-cheek, this mode might surface candid insights or common-sense reactions that a more formal analysis could miss. It’s an example of the creative ways the toolkit leverages AI personas for useful feedback. (Under the hood, this likely uses a special prompt to Claude to adopt a less restrained, colloquial tone in its critique.)

All these tools are readily accessible from within the application’s unified interface. A writer can pick any stage of their workflow – brainstorming a premise, outlining, drafting chapters, or revising the prose – and find a tool in the Writer’s Toolkit to help with that task. Each tool runs in a controlled environment (often with user-set options) and produces an output report or file. The results can then be opened in the built-in editor or any text editor for further refinement.

User Interface and Workflow

Using the Writer’s Toolkit is intended to be straightforward. Upon launching the application, the user is greeted by the main window (a dark-mode interface by default) [107]. The typical workflow involves a few steps which the UI guides the user through:

1. **Select or Create a Project:** The top of the interface shows the current project name/path and a “Select Project” button [108]. Clicking this opens a project selection dialog (shown via `project-dialog.html`) where the writer can choose an existing project folder or create a new one [109]. Projects are simply subdirectories in `~/writing`, and each contains the text files for that project. Once a project is selected, the app knows where to read/write files. The UI also provides an

"Import .docx" button to convert a Word document to `manuscript.txt` (preserving chapters) and an **"Export .txt to .docx"** for the reverse, which helps integrate with MS Word if needed [110].

2. **Select a Tool:** The main panel is divided into two sections: **AI-based tools** and **Non-AI tools**, each with a dropdown menu [111] [112]. AI tools are those that call the Gemini/Claude API (e.g. the analysis and generation tools), while non-AI tools are utilities like the DOCX/EPUB converters. The user picks the desired tool from the appropriate dropdown. When a tool is selected, its description is displayed below the menu so the user knows what it does [111] [113].
3. **Configure and Run the Tool:** After selecting a tool, the user clicks "Setup & Run". This brings up a **Tool Setup dialog** (`tool-setup-run.html`) where the specific options for that tool are presented as form fields (for example, selecting the input file(s), adjusting analysis settings, specifying output preferences, etc., as defined in the tool's options schema) [19] [114]. The user fills in or confirms these parameters (often the defaults like `manuscript.txt` are pre-filled) [115] [116]. Once configured, the user hits **Run** to start the tool. The UI will then typically switch to a progress view where it might display log messages from the tool (using the `emitOutput` text streaming). During execution, most interface controls are disabled to prevent interference [117], and a timer may show how long the tool has been running [118]. There is a "Force Quit" button to abort a long run, though users are cautioned to use it only if necessary [119].
4. **View Results:** When the tool completes, its output (often a report or a generated text) is saved to the project folder (with a timestamped name). The Toolkit provides a built-in **editor window** to view these results conveniently. For example, after running a tool, the user can click an "Open in Editor" option which launches the Editor Dialog (`editor-dialog.html` and the rich text editor under `renderer/editor/`). This editor is a simple text editor with syntax highlighting (if needed) or at least formatting for the reports. Users can review the AI's suggestions or analysis and even make quick edits. Because all outputs are plain text, the user can also open them in external editors if preferred.
5. **Iterate or Toggle Settings:** The user can then close the editor and run additional tools. The interface also has a theme toggle (sun/moon icon) to switch between dark and light mode according to preference [120]. When finished, the "Quit" button will exit the app gracefully [121], remembering the last project and settings for next time.

This workflow lets writers cycle between writing and analysis. For example, they might generate an outline, then immediately run the Plot Thread Tracker on that outline to evaluate it, or draft a chapter and then use the Conflict Analyzer on just that chapter. The UI design emphasizes an **easy, step-by-step process**: pick project → pick tool → configure → run → see output. Each tool's operation feels like a plugin that slots into this same process, making the toolkit intuitive despite its wide functionality [122] [123]. The presence of separate AI vs non-AI sections in the UI is also user-friendly, since it signals which tools will consume API credits and which won't (for example, converting a file format is local and free, whereas developmental editing will call the AI service). Overall, the UI and workflow aim to be **author-centric**: writers can focus on their content while the app handles the heavy lifting of analysis/generation in the background.

Technical Design and Noteworthy Implementation Details

Under the hood, Writer's Toolkit demonstrates some noteworthy technical design choices:

- **AI Integration (Gemini and Claude):** The toolkit's AI features are primarily powered by Google's **Gemini** model, accessed through the `@google/genai` SDK [124]. The integration is encapsulated in a custom **API service class** (`AiApiService` in `client.js`), which abstracts the model's API. This

service is responsible for sending prompts and receiving streaming responses. Notably, it sets all content safety filters to "off" to allow the AI to generate unrestricted feedback (relying on the user's discretion for appropriateness) [125]. It provides a method to **stream responses with a callback** (`streamWithThinking`), effectively feeding partial output chunks to the UI as they arrive [126] [127]. This can simulate a "thinking..." effect where the user sees the AI's answer being written out in real-time. The service also wraps a **token counting** endpoint: before sending a large prompt, the tool calls `countTokens()` to get an accurate token count of the input [128] [129]. This is used to ensure the prompt and expected output will fit within the model's context window limits. In fact, many tools calculate a **token budget** – how many tokens remain for the AI's answer given the prompt size and model limit – and they report these numbers to the user in the output [128] [130]. This prevents requesting outputs that are too large. Originally, the toolkit supported Anthropic's Claude model as well (the presence of files like `claude_client.js` and the "Drunk Claude" tool attest to this), but it appears to have transitioned focus to Gemini. The code still includes the **Anthropic SDK (**`@anthropic-ai/sdk`**)** as a dependency [131], which suggests Claude could be used as a fallback or for certain tools if configured. In the tool definitions, the **Drunk Claude** tool explicitly uses the Claude persona for stylistic reasons [80]. The dual-model support reflects a flexible design to accommodate different AI backends.

- **Prompt Structuring:** Each AI-driven tool implements its own `createPrompt()` (as suggested by the general tool template) [132] [133]. This means the prompt that is sent to the AI is crafted specifically for that tool's purpose. For instance, the Foreshadowing Tracker likely prompts: "List all instances of foreshadowing in this text and their resolutions...", whereas the Drunk Claude tool's prompt might instruct the AI to respond in a tipsy, brutally honest voice. These prompt templates are a key part of the toolkit's intelligence. They are not stored as separate files but embedded in the tool classes. The template file `tool-general-template.js` provides a starting point, encouraging developers to fill in a `createPrompt(inputText)` method with the right instructions for the AI [134] [133]. This separation of prompt logic per tool, combined with the token budgeting mentioned above, ensures each tool's AI query is optimized for its task.
- **Modularity and Extensibility:** Thanks to the **BaseTool/ToolSystem architecture**, adding new tools or modifying existing ones is relatively straightforward. Each tool class is self-contained and focuses on one job. The **Tool Registry** (`TOOL_DEFS`) holds the metadata, which the UI automatically reflects. This registry approach and the use of dynamic `require()` based on tool IDs (converting between snake_case IDs and hyphenated filenames) means the app can load only the tools that exist on disk and even potentially allow plugging in custom tools [135] [136]. The provided developer notes show the consistency required (tool ID strings must match across class, registry, and UI lists) [137], but otherwise the system is flexible. The presence of a `test.js` file (likely for quick testing) and the separation of AI vs non-AI tool lists in the UI (`nonAiToolIds` in `renderer.js`) further illustrate how the code is organized to accommodate growth.
- **File Handling and Caching:** The toolkit deals extensively with reading and writing files. The BaseTool's helpers ensure that relative file paths (like just a filename) are resolved against the current project directory automatically [138] [139]. This spares the tool implementer from worrying about path details. Many tools read multiple files (manuscript + outline + world, for example) and concatenate or cross-reference their contents for analysis. After generating output text, the BaseTool's `writeOutputFile()` is used to save the results, and the file is named with a timestamp and tool identifier (as seen in TokensWordsCounter saving `tokens_words_counter_<input>_<timestamp>.txt`) [11]. There is also a **File Cache** module (`file-cache.js`) that keeps track of recently created output files in memory [140]. Each time a tool runs, it clears its old cache and then adds new output file paths to this cache [141] [142]. This cache can

be used by the UI to easily list the results (for example, after running a tool, the UI could show a list of output files that were produced, enabling quick opening of those files). It's a simple in-memory map, but it's a nice touch to improve UX.

- **Dependency Usage:** The project makes good use of npm libraries to implement features rather than reinventing them. Some notable dependencies: `mammoth` is used for .docx to HTML/text conversion (supporting the DOCX extractor) [143], `jszip`, `xmldom`, **and** `xpath` are used to parse and convert EPUB files (reading the XML inside EPUB for the converter tool) [144], `docx` **(Package)** helps in creating Word documents from text (for the export feature) [145], and UI libraries like `chalk` (for colored console output in dev) and `cli-progress` (perhaps for progress bars in the console or terminal runs) are included [146]. The presence of `luxon` suggests date/time handling (likely used for timestamp formatting or scheduling chapter delays), and `marked` indicates the app can render Markdown (perhaps the reports are in Markdown and then rendered to HTML in the editor, or for removing Markdown in outputs) [147]. These choices show that the toolkit leverages existing tools to handle file formats and text processing, allowing the focus to remain on the AI and writing-specific logic.
- **State and Settings:** The `AppState` class handles global state like which project is active and whether a tool is currently running (to disable UI appropriately) [148] [149]. It also reads/writes settings via `electron-store`. This includes remembering the last opened project and possibly the window size or UI theme. The first thing the app does on startup is initialize this state (asynchronously) [150], so that any persisted configuration is loaded before the UI or tools are used. This ensures a smoother user experience (e.g., automatically re-opening the last project).
- **Cross-Platform and Distribution:** Using Electron Forge/Builder, the app is configured to build for multiple OSes. The `forge.config.js` and instructions like `create-dmg.txt` indicate the author packaged the app for distribution on macOS and Windows [30]. The product name is "Writer's Toolkit" and an app ID is set for the installer [151]. These details, while not directly affecting runtime, mean that an author using this toolkit can get it as an installable app, which is rather convenient compared to running scripts manually or using a command-line tool. It wraps up a complex system into a user-friendly application.

Notable Elements: Two aspects really stand out in design: (1) the **comprehensiveness** of the toolset and (2) the commitment to **local-first, user-controlled workflow**. The toolkit doesn't limit itself to one or two AI tricks; it attempts almost every type of writing analysis one could think of [152] [153]. This all-in-one approach is ambitious and quite unique – it's like having an AI developmental editor, copyeditor, and writing coach all in one app. Tools like *"Drunk Claude"* add personality and show the creative thinking in the project's design (it's an unconventional way to get feedback, likely included to spark new perspectives for the author). Additionally, referencing known writing techniques (Le Guin's crowding/leaping, Chekhov's gun for foreshadowing, etc.) shows the toolkit is informed by literary concepts, not just generic AI output, making it more tailored to writers' real concerns.

Secondly, the fact that all data stays on the user's machine (with only API calls to the AI service) addresses a major concern for writers: **privacy and ownership** [2]. Authors can use the AI to analyze their entire novel without uploading the actual text to some cloud storage or web app – the AI receives only the prompt and relevant excerpts as needed, and the integration presumably avoids retaining data (aside from the AI service itself, which is an external API). The local file structure (`~/writing`) and the decision to build as a desktop app underscore this philosophy. It means the toolkit can be used offline for non-AI tools, and when online, the user is only sending data to the AI model, not to any intermediary server controlled by the toolkit maker.

Finally, the **documentation and examples** included in the repo indicate an effort to help users and contributors. The `README.md` provides an excellent overview of all features (much of which has been cited above) and likely usage hints [154] [155]. There's a detailed **YouTube demo script** (`youtube.txt`) which walks through using the app step by step, which would be helpful for new users to watch or read to understand the workflow [156] [157]. The presence of developer docs for adding tools shows foresight for open-source contributors or future expansion [158] [22]. All of this rounds out the project as a polished toolkit intended not just for personal use, but for others to adopt and extend.

In conclusion, the Writer's Toolkit is a powerful amalgam of tools for fiction writers. Its structure (Electron app with modular tool classes) makes it maintainable and extensible, while its functionality addresses a wide spectrum of writing tasks. A writer using this toolkit can brainstorm a story, generate an outline and world-building bible, draft chapters, and then rigorously self-edit the manuscript – all in one application. The integration of cutting-edge AI (Gemini/Claude) provides a level of insight and assistance that can speed up the writing process or improve the quality of the prose in ways that would be hard to achieve alone. At the same time, the writer remains in control: the tools make suggestions, but do not overwrite the author's files unless explicitly asked, and all changes can be reviewed in the built-in editor. This design reflects the ideal of AI as a **co-pilot for creativity** – doing the heavy lifting of analysis and routine writing tasks, so that the author can focus on storytelling and higher-level revisions. The Writer's Toolkit exemplifies how AI and human creativity can work hand-in-hand in a practical software solution for writers.

Sources: The above overview is based on the project's repository documentation and code (cleesmith/writers-toolkit-gemini), including the README.md which describes features and design [152] [159], the defined tool set and configuration in code [34] [11], and accompanying usage notes [160] [12].

---

[1] [2] [3] [4] [5] [6] [9] [10] [12] [13] [30] [33] [69] [79] [83] [88] [93] [99] [101] [102] [103] [106] [124] [152] [153] [154] [155] [159]
README.md
https://github.com/cleesmith/writers-toolkit-gemini/blob/04ec8af1f60c5a7b620750e40aa45cd64a9a6a52/README.md

[7] [8] [27] [28] [150] main.js
https://github.com/cleesmith/writers-toolkit-gemini/blob/04ec8af1f60c5a7b620750e40aa45cd64a9a6a52/main.js

[11] [141] [142] tokens-words-counter.js
https://github.com/cleesmith/writers-toolkit-gemini/blob/04ec8af1f60c5a7b620750e40aa45cd64a9a6a52/tokens-words-counter.js

[14] [15] [23] [108] [110] [111] [112] [113] [120] [121] index.html
https://github.com/cleesmith/writers-toolkit-gemini/blob/04ec8af1f60c5a7b620750e40aa45cd64a9a6a52/index.html

[16] [17] [18] [138] [139] base-tool.js
https://github.com/cleesmith/writers-toolkit-gemini/blob/04ec8af1f60c5a7b620750e40aa45cd64a9a6a52/base-tool.js

[19] [20] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] [60] [61] [62] [63] [64] [65] [66] [67] [68] [70] [71] [72] [73] [74] [75] [76] [77] [78] [80] [81] [82] [84] [85] [86] [87] [89] [90] [91] [92] [94] [95] [96] [97] [98] [100] [104] [105] [114] [115] [116] [135] [136] tool-system.js
https://github.com/cleesmith/writers-toolkit-gemini/blob/04ec8af1f60c5a7b620750e40aa45cd64a9a6a52/tool-system.js

[21] [22] [137] [158] how_to_add_a_new_Tool.txt
https://github.com/cleesmith/writers-toolkit-gemini/blob/04ec8af1f60c5a7b620750e40aa45cd64a9a6a52/how_to_add_a_new_Tool.txt

24 25 29 148 149 state.js
https://github.com/cleesmith/writers-toolkit-gemini/blob/04ec8af1f60c5a7b620750e40aa45cd64a9a6a52/state.js

26 125 126 127 129 client.js
https://github.com/cleesmith/writers-toolkit-gemini/blob/04ec8af1f60c5a7b620750e40aa45cd64a9a6a52/client.js

31 32 131 143 144 145 146 147 151 package.json
https://github.com/cleesmith/writers-toolkit-gemini/blob/04ec8af1f60c5a7b620750e40aa45cd64a9a6a52/package.json

107 109 117 118 119 122 123 156 157 160 youtube.txt
https://github.com/cleesmith/writers-toolkit-gemini/blob/04ec8af1f60c5a7b620750e40aa45cd64a9a6a52/youtube.txt

128 130 132 133 134 tool-general-template.js
https://github.com/cleesmith/writers-toolkit-gemini/blob/04ec8af1f60c5a7b620750e40aa45cd64a9a6a52/tool-general-template.js

140 file-cache.js
https://github.com/cleesmith/writers-toolkit-gemini/blob/04ec8af1f60c5a7b620750e40aa45cd64a9a6a52/file-cache.js