ChatGPT

EnZIM: Offline ZIM Reader & Knowledge Assistant – PRD & Architecture

Executive Vision

EnZIM is envisioned as an **offline knowledge companion** that allows users to explore large encyclopedic archives (like Wikipedia) entirely without internet. It combines a **high-fidelity ZIM file reader** (for offline web content) with a rich visual navigation system and AI assistance. The goal is to empower educators, students, and knowledge seekers in bandwidth-limited environments with a responsive, modern interface to vast information repositories [1] [2]. The application not only offers traditional article viewing and search, but also an interactive "semantic mesh" of concepts to facilitate discovery of related topics. A built-in AI **assistant agent** provides natural-language Q&A, summaries, and guidance using the offline content, essentially bringing an AI-powered experience to offline knowledge bases. All of this is delivered as a **clean-room, MIT-licensed** solution to avoid any GPL licensing constraints, ensuring the platform remains open and permissive [3]. In short, EnZIM's vision is to be the **go-to offline wiki reader and AI tutor**, fusing robust content access with innovative navigation and AI-driven help.

Feature Set

The core features of EnZIM are summarized below:

Feature	Description
Offline Article Reader	Open and render content from ZIM archives (e.g. Wikipedia) with full fidelity (text, images, links, styling) [4]. Provides a clean reading view and basic in-article navigation (scroll, hyperlinks, table of contents, etc.).
Library Management	Manage a library of multiple ZIM files. Users can import/remove archives and view metadata (title, size, description) [5]. Includes an archive list UI and the ability to switch between loaded archives. Optionally, integrate an online catalog to download new ZIMs.
Full-Text Search	Fast full-text search within the current ZIM archive. Supports title and content queries with result ranking [6]. If the ZIM has a built-in index, use it; otherwise, generate a lightweight index on the fly. Displays search results with titles (and snippet previews if possible).
Semantic Mesh Navigation	An interactive knowledge graph of article relationships ("concept mesh"). Articles are nodes and connections represent links or semantic relations [7]. Users can explore related topics by interacting with this 2.5D graph (zoom, pan, click nodes) to navigate through the encyclopedia in a visual way.
Bookmarks & History	Save articles as bookmarks and automatically record recently viewed articles [8]. Users can view their bookmark list and reading history (with timestamps), and clear or export these as needed. Persists across sessions for quick return to favorite or last-read content.

Feature	Description
Annotations & Highlights	Annotate articles with notes, highlights, or tags. This could include text highlights, sticky notes, or even voice notes [9] . An annotation panel allows reviewing all annotations. These are stored locally (per archive or globally) so the user can build a personal knowledge base on top of the content.
"Chat with ZIM" Assistant	An AI-powered assistant to query and explore the archive in natural language. Users can ask questions about the content or request summaries, and the assistant will find relevant info in the ZIM and respond contextually [10] . This chat interface can guide navigation ("Which article should I read on quantum mechanics next?") and provide answers with citations from the offline content.
Text-to-Speech (TTS)	Listen to articles with integrated text-to-speech playback. Each article (or selected text) can be read aloud using system or custom voices, enhancing accessibility (aligned with screen-reader support) [11] . A playback control (play/pause, voice selection) is provided for convenience.

(Table: EnZIM key features and their descriptions.)

Semantic Mesh Engine Specification (UI, Data Model & Navigation)

One of EnZIM's standout features is the **Semantic Mesh**, which presents a visual network of related concepts to the user. The system constructs a local knowledge graph centered on the current article or query context:

- **Data Model:** Internally, content is represented as a graph of nodes and edges. Each **GraphNode** corresponds to an article (with its title and reference in the ZIM) and each **GraphEdge** represents a relationship [12] . Relationships primarily come from wiki hyperlinks (Article A links to B), but may also include category membership or other semantic links. The mesh is generated on-the-fly to avoid loading the entire wiki graph at once [13] – for example, when viewing an article, the engine gathers that article's outgoing links and a subset of incoming links from other articles [14] . It can also pull in second-degree connections (links of links) to enrich the context. *(In future, a semantic analyzer could use NLP or embeddings to add "similar topic" edges beyond explicit links [15] .)*

- **Graph Generation:** Upon activating the mesh view, the **Semantic Engine** fetches related articles and builds a subgraph in memory: *Current Node* (center) → *Neighbor Nodes* (direct links or related topics) → optional *Secondary Nodes*. For example, if the user is reading about "Quantum mechanics," the mesh might include directly linked topics (like "Photon," "Heisenberg Uncertainty") as primary neighbors, plus a few second-tier nodes like related experiments or scientists [14] . Each node knows its coordinates in the visualization layout. The layout can be computed dynamically (e.g. a force-directed or radial layout) or use a preset algorithm for stability.

- **UI Visualization:** The mesh is presented as an interactive 2D/2.5D network graph. In the **Synaptic Veil** theme, for instance, the central node is highlighted and connected by glowing lines to neighbors on a dark canvas [16] [17] . Nodes are rendered as floating labels or bubbles; the current focused article might appear larger or with a special highlight [18] . Edges (connections) are drawn as lines (SVG or Canvas) that can animate or highlight when a node is hovered [19]. The term "2.5D" indicates the graph may use layering or subtle depth effects (like node scaling or shadows) to give a sense of depth, without being a fully 3D environment. Users can drag or pan the view and zoom in/out to explore. Clicking on a neighbor node will focus that node (it becomes the new center), which triggers regeneration of the mesh around that topic and loads the article in the reader panel. This way, the mesh enables an exploratory browsing mode beyond linear hyperlink clicking.

- **Navigation & Integration:** The semantic mesh is tightly integrated with the reader. For example, a user reading an article can toggle on the "Explore" mode to slide in the mesh view (either side-by-side or overlay). In the **Prismatic Swiss** UI, a "Neighbors" list is also provided alongside the visualization [20] [21] – this list shows textual titles of related nodes (like a see-also list) for quick scanning or navigation by title. The mesh engine uses the same underlying data (the ZIM's link structure or precomputed indices) to populate both the graph and the neighbors list. The graph can be refreshed or expanded by user action (e.g. an "Expand network" button or gesture), which would fetch next-level connections. Through the mesh, EnZIM offers an **intuitive, discovery-driven navigation**: users can traverse the encyclopedia in a associative way, which is especially powerful for learning and research.

UI View Hierarchy & Multi-View Layouts

EnZIM's interface is designed with a **multi-panel layout** that adapts to different tasks. The UI can be thought of as a hierarchy of views or panes that are shown/hidden depending on user context:

- **Global Header:** At the top of the app is a persistent header bar. This contains the application branding and version, a search input, and status indicators [22] [23]. For example, the header might display "EnZIM" (or "AnZimmerman" as a code-name) with an icon, a search box for querying the archive, and status lights showing index status or offline mode (e.g. an "Index Online" dot, and a label like "DESKTOP" to denote platform [23]). The header's search bar allows quick jumping to articles or search results from anywhere.

- **Navigation Sidebar:** On larger screens (desktop/tablet), a left sidebar is used for primary navigation. In the **Brutalist Monolith** concept, this sidebar contains a list of navigation options (e.g. "Browse Archives", "Downloads", "History", "Settings") with bold icons/text [24] [25]. In other layouts, the left panel is dedicated to **Library Management** – showing the list of ZIM archives loaded. For instance, the Synaptic and Prismatic designs have a section titled "Archives" listing available ZIM files as selectable cards [26] [27]. Selecting an archive here loads it and updates the main view to that archive's content. On mobile or smaller windows, this sidebar may collapse into a menu or separate screen to save space.

- **Main Content Area:** The central part of the interface is context-dependent:

- **Library View:** If no archive is loaded (or when managing archives), the main area can show a library dashboard. In the Brutalist design, this appears as a grid of archive "cards" (monolithic blocks representing each ZIM, with title, description, size, and an action button) [28] [29]. The library view allows adding new archives (e.g., a floating "+" button [30] to import or download) and possibly filtering/sorting them.
- **Reader View:** When an article is opened, the main area (or a portion of it) becomes the Reader. The article content is rendered in a scrollable view with appropriate styling for headings, paragraphs, images, etc. The UI uses a webview or HTML renderer with the article's HTML, applying the selected theme's typography. For instance, headings might be styled with specific

font and prefix (the Synaptic theme prefixes `h2` with a “#” symbol for a wiki-like look [31]). The reader view also includes an article header section showing the article title and some actions [32] [33] – e.g. bookmark, share, or open in browser. Navigation within the article (via hyperlinks) is captured and handled by the app to load the new content internally.

- **Search Results View:** When a search is performed, the main area may display a list of results (likely similar to a library or list view). Each result item shows the article title, maybe a snippet, and on click will open that article. The search results could appear either in the main panel (replacing the article view until an article is chosen) or as an overlay modal if design prefers.

- **Semantic Mesh View:** In some layouts, the concept graph can take over the main content area. For example, the Synaptic Cartography Veil prototype places the mesh visualization at the center (with the article reader moved to the right side as a secondary panel) [34] [35]. In Prismatic Swiss, the mesh is shown in a right-side panel while the article remains in the center [36] [37]. The UI is designed to be flexible: on a large screen, one could even view the article and mesh side by side; on smaller screens, there might be a toggle to switch between reading mode and graph mode.

- **Secondary Right Panel:** Many designs incorporate a right-hand side panel for auxiliary content or tools. Depending on context, this panel can serve different purposes:

- In reading mode, it could show an outline or table of contents of the article (if available), or the user’s annotations for that article.
- In Synaptic’s layout, the right panel **is** the Reader (showing article text) when the mesh occupies the center [38] [39].
- In Prismatic’s layout, the right panel holds the Semantic Mesh visualization and a “Neighbors” list [40] [41].

- We also anticipate using the right panel for the **Chat Assistant** interface: for example, when the user activates “Chat with ZIM,” a chat sidebar could slide over or replace the mesh panel, showing the conversation thread with the AI. The assistant could thus live in a panel that can be toggled on demand, without leaving the article view.

- **Adaptive Layouts:** The view hierarchy is designed to collapse gracefully on smaller devices. On mobile, the header might shrink to a top bar with a menu button, the sidebar might turn into a swipeable drawer, and panels like the mesh or chat might become full-screen modal views. The use of responsive design (flexbox/grid and Tailwind CSS utilities) ensures that, for example, a 3-column desktop layout can turn into a stacked layout on mobile [42] [43]. Key views (Library, Reader, Search, Mesh, Chat) might be accessible as tabs or via a bottom nav on mobile.

In summary, EnZIM’s UI is composed of modular views: a persistent top bar, navigational sidebars, and dynamic content panels. By leveraging this hierarchy, the app can present a **unified interface** where users seamlessly switch between browsing archives, reading articles, exploring the mesh, and consulting the AI assistant, all within one cohesive window.

ZIM Parsing and Indexing (Clean-Room Library Stack)

At EnZIM’s core lies the **AnZimmerman** clean-room ZIM library, which handles all reading (and future writing) of the .zim files. This library is implemented in multiple languages (Rust, TypeScript, Python, Go, etc.) for flexibility [44], all based on the openZIM specification but written from scratch to avoid GPL license contamination [45]. For EnZIM: - **Desktop (Tauri):** We use the Rust native library for performance, allowing direct file I/O and multi-threading for heavy tasks. The Tauri backend can call into Rust code to load a ZIM, stream data, and perform search indexing [46]. This ensures efficient use of system resources and speed for things like decompression. - **Web (Browser/Extension):** We use the TypeScript ZIMReaderBrowser, which operates on an in-memory ArrayBuffer of the ZIM file [46]. Because web apps can't use the filesystem directly, the user will open a file via input (or the extension will have it in IndexedDB) and the JS library reads from that buffer. The same parsing logic is mirrored in JS (thanks to the common spec) – it can read the ZIM header, directory entries, and content blocks using browser APIs and WebAssembly for compression if needed. - **Mobile:** Likely follows the Tauri approach (Rust backend via Tauri Mobile for Android, and analogous for iOS once supported). The mobile app would bundle the Rust parsing code, or possibly use a thin WASM layer if needed, to handle ZIM files on-device.

ZIM File Structure: A ZIM file contains a header, a directory of articles, and clusters of compressed content. The library is designed for **random access**: it does not memory-map the entire file, but rather reads only what is needed on demand, which is crucial for huge archives [47]. When an article is requested (by article ID or URL), the engine: 1. Looks up the article's directory entry (which gives its title, namespace, and the pointer to where its content is stored) [48]. 2. Uses the entry's cluster number and blob index to locate the correct data chunk. It consults the ZIM header's cluster index to find the byte offset in the file [49]. 3. Reads that cluster from disk and decompresses it (supported compression formats include zlib, LZMA, or Zstandard depending on the ZIM) [50]. 4. Extracts the specific blob (the HTML or image) corresponding to the requested article, and returns that data to the UI for rendering. This process is very fast for individual articles (<2 seconds typical load for a large ZIM) and the engine may cache recently used clusters in memory to speed up sequential accesses.

Indexing & Search: Many ZIM archives come with a title index, but not all have full-text search built-in. EnZIM's search component will build an index if needed. On loading an archive, a background worker can iterate through article titles (and potentially content) to create an **inverted index** or use an existing lightweight search library. On the web/JS side, this could be as simple as scanning titles for matches (or using a WebAssembly port of a search algorithm), whereas on desktop we could use a Rust crate or even SQLite FTS module for faster full-text search [51]. The search workflow is: *User query → Normalize query → Query index → Rank results → Return matches*. We aim for interactive speeds (sub-second for title suggestions, a couple seconds for full-text on large archives) [52]. If indexing full text is too slow or memory-heavy, EnZIM can fall back to incremental search (e.g., first search only titles which is quick [53], then progressively search content in the background and update results). The indexing process itself can be optimized by skipping non-text media and possibly storing word frequencies or using trigram indexing for faster lookup [51]. All indexing is done client-side and stored in memory or a local database (IndexedDB or SQLite), and can be discarded when the archive is closed to save space.

Memory and Performance: Handling multi-gigabyte ZIM files means carefully managing memory and threading. The engine will utilize streaming reads and **lazy loading** – only reading clusters when needed, and releasing them if memory is tight. On desktop, heavy operations (like initial index build or large file decompression) can be offloaded to a separate Rust thread via Tauri (so the UI stays responsive) [54]. In a browser context, Web Workers could be used for similar background tasks. The library is optimized to avoid unnecessary copies; for example, it may use `ArrayBuffer.slice` or Rust `mmap` where appropriate. We adhere strictly to the ZIM spec for reading structure, and have robust error handling (e.g., detecting corrupt archives, gracefully handling missing images, etc.). The clean-room implementation gives us full control: we can extend it with features like custom metadata or alternative indexing in the future since we aren't bound by an external library's limitations.

Overall, the parsing and indexing layer provides a **unified API** to the rest of the app: the UI can call `getArticle(byTitle)` or `search(query)` on the loaded archive, and the engine abstracts whether that call goes to JS/TS code or to Rust native code depending on platform [55]. This modular core makes EnZIM's cross-platform architecture possible while keeping performance high and resources usage efficient.

Agent Prompt Design & Naming Conventions

EnZIM includes an integrated **AI assistant** ("Chat with ZIM") that lets users engage with the archive in natural language. Designing this feature involves both how the AI is prompted/trained and how the system integrates with it.

Assistant Role & Behavior: The assistant functions as an offline wiki expert that can answer questions, summarize articles, and guide navigation using only the content available in the loaded ZIM files. It does not have access to the internet, so it relies on the local knowledge base. When the user asks a question, the system will: 1. **Retrieve relevant content** from the ZIM archive – for instance, by performing an internal search for the topic or identifying the current article's text if the question is about the open page. 2. **Construct a prompt** for the LLM (Large Language Model) that includes the relevant extracted text (as context) along with an instruction to answer the user's question or fulfill the task using that context only. 3. The LLM (which could be running locally or via a local server, given offline requirements) generates a response. The system may post-process the answer to add citations or references to the source articles. 4. The answer is displayed in the chat UI, with references linking back to articles in the archive for transparency.

For prompt design, we define several **prompt templates** corresponding to different functions: - **Q&A Prompt:** Used when the user asks a factual question ("What is quantum mechanics?"). The system will include text from the "Quantum mechanics" article (and possibly related articles) in the prompt, and instruct the model to provide a concise answer with references. We might phrase the system message like: *"You are a helpful encyclopedia assistant. Using the provided article excerpts, answer the question and cite the article title for each fact you use."* The model's answer is thus an extractive summary with citations. This aligns with approaches like AskZim, which delivers answers with precise citations from the source content [56] [57]. - **Summarizer Prompt:** Used when the user requests a summary ("Summarize this article" or a section). The context will be the article text (or section text), and the instruction might be: *"Provide a summary of the above content in plain language."* The assistant will return a paragraph summary. If the user is in a specific article, this can be triggered via a "Summarize" button as well. - **Navigator/Guide Prompt:** Used for navigation or recommendation requests (e.g. "I want to learn about related topics to X" or "What should I read next?"). Here the agent might leverage the semantic mesh or search results. The prompt might include a list of related article titles or brief descriptions, and ask the AI to suggest where to go next or explain the connections. This is more generative, potentially with the assistant adopting a guiding tone ("Since you're reading about Quantum Mechanics, you might be interested in the article on the Schrödinger equation, which is directly related."). - **Conversational Mode:** In addition to factual Q&A, the assistant can operate in a chatty, explanatory mode where it can answer more open-ended questions or have a dialogue (while still drawing on the archive content). We plan to implement **two interaction modes** – one that is strict about citations (for verifiable answers) and one that is more conversational and user-friendly [56]. Internally, these might be two different system prompts or toggles: e.g., "Expert mode" vs "Friendly mode". In expert mode, every answer will include source references, whereas in friendly mode the assistant might omit the formal citations unless asked, providing a smoother conversational experience.

Prompt Naming Conventions: To manage these templates and to coordinate with the development process (especially with AI builder agents involved in coding), we adopt clear naming for prompts and agent tasks: - We use descriptive names like **WikiQA**, **WikiSummary**, **WikiGuide** for the user-facing assistant prompt templates. For example, a prompt file or constant might be named `PROMPT_WikiQA_v1` which contains the system message for Q&A mode. - When invoking the assistant, the system will specify which mode/prompt to use. This is analogous to AskZim's "ask" (Q&A with citations) vs "chat" (conversational) modes [56], but in our UI we might label them as "Answer with sources" vs "Discuss" for clarity. - On the development side, if using an AI agent to help build features, we also name those tasks in a consistent way (though this is more internal). For instance, when prompting a coding agent to implement a module, the prompt might be prefixed with the module name like "[UI/Search] Build SearchResultsList component". This ensures the agent knows the context of the task. The architecture documentation even suggests measuring success by how effectively AI builder agents can implement modules from these prompts [58]. Consistent naming (and tagging of prompts) helps track these tasks.

Agent Coordination Architecture: The assistant feature will likely employ a **mediator component** in the app that handles the flow: it takes user queries, uses the search/index module to fetch relevant text, applies the appropriate prompt template, and calls the LLM (which might be an external local service or an in-process model). This could be implemented as a Rust sidecar that loads a local model (if using something like GPT4All or llama.cpp), or as calls to a local API (e.g., if using Ollama as in AskZim [59]). The key is that the pipeline remains offline and fast enough for interactive use. Caching could be used: if the user asks multiple questions about the same article, we don't need to repeat the search step each time; recently used context can be cached or the conversation state can carry it.

In summary, the agent is designed to act as a *knowledgeable offline guide*. By carefully designing prompts for different tasks and maintaining clear conventions for those prompts, we ensure the AI's output remains accurate and relevant to the ZIM content. This design also aids any AI **development agents** working on the project, as each task (whether building the feature or using it at runtime) is well-defined and labeled, minimizing confusion in our human-AI collaborative workflow.

Modular Build Roadmap (Agent-Friendly Development)

To implement EnZIM in a structured way, we break the project into modules and phases. This modular roadmap not only aids human developers but is also **agent-friendly** – meaning that an AI coding assistant can be directed to build or assist with one module at a time with clear scope. Below is the staged development plan:

1. **ZIM Core Engine Integration:** *Goal:* Establish the ability to open a ZIM file and retrieve an article's content. Start by integrating the AnZimmerman library in the target platforms (Rust for Tauri, and/or TS for web). Develop a simple console or minimal UI to select a ZIM and display a specified article's HTML. **Agent task focus:** Ensure the low-level read (header parse, directory lookup, cluster decompress) works correctly. This forms the backend foundation for all other features.

2. **Basic Reader UI:** *Goal:* Create a rudimentary article viewing interface. Implement a React component (or Tauri WebView) that can render HTML content of an article in a scrollable view. Add basic navigation controls – e.g. a back button and clickable links that load new articles via the engine. At this stage, the app should function as a simple offline wiki reader. **Agent task:** Build a "ArticleViewer" component and wire up link click events to the engine's `getArticle` calls.

3. **Library Management Module:** *Goal:* Implement the library/archives view. Create a UI to add ZIM files (file picker or drag-drop), list all loaded archives, and switch between them. This

involves a sidebar or a dedicated screen showing archive cards (with metadata like title, size). Include the ability to remove an archive from the list. **Agent task:** Develop a “LibraryManager” and “ArchiveCard” component, and handle persistence of the archive list (e.g., store in a JSON or IndexedDB so it remembers on next launch).

4. **Search Functionality:** *Goal:* Enable searching within an archive. This entails building the search index in the background when an archive loads (if needed), and creating a search UI (search bar in header, results list view). The search results view lists matching article titles; clicking a result opens that article. **Agent task:** Implement a `searchIndex` module (with functions like `buildIndex(ZIM)` and `queryIndex(query)`) and a “SearchResults” component to display matches. This stage will likely require tuning for performance and may involve using a third-party library or writing a simple indexer. We verify with large ZIMs to ensure acceptable speed [52].

5. **Semantic Mesh & Graph View:** *Goal:* Develop the semantic mesh visualization feature. This is a significant module including: extracting link data from articles (perhaps augment the index with link mappings), a Graph data structure, and a front-end component to render the network. Start with a simple version: e.g., when an article is open, fetch its immediate outgoing links and create nodes for each, connecting them to the central node. Render this with a basic force-directed layout or even a fixed radial layout. Focus on interactivity (click node to open article). **Agent task:** Build a “GraphView” component – possibly using a d3-force or a custom canvas for drawing nodes and edges. Also implement a function in the engine to get related nodes (outgoing links and maybe incoming via a reverse index). Iterate on layout algorithms for aesthetic results. This is a complex feature, so an agent might tackle sub-tasks like “computeGraph(article)” separately from the UI rendering.

6. **Bookmarks, History & Annotations:** *Goal:* Implement user data features. This includes a way to bookmark articles (e.g., a star icon to toggle bookmark on the article header) and a list UI to show all bookmarks. Similarly, maintain a history list (recently viewed articles) that updates on each navigation. For annotations, start with simple text-highlighting or note-taking on articles: perhaps allow the user to select text and add a comment (stored in local storage). The UI could include an Annotation sidebar or overlay and indicators within the article text. **Agent task:** Create a “BookmarkManager” (with functions add/remove list) and UI components for bookmark list and history list. Also an “AnnotationManager” to handle saving annotation data (which might be more involved if we support multiple types). Ensure these persist using a small local database or file as appropriate [60]. This phase greatly improves the user’s ability to personalize their experience.

7. **AI Assistant Integration:** *Goal:* Incorporate the chat/assistant interface. Begin by deciding on the local AI model or API to use (for development, this might start with calls to an online API like OpenAI for ease, but ultimately use a local model for offline). Implement a Chat panel in the UI with a text input for the user and a message list. Connect this to a function that orchestrates query → retrieve context → prompt LLM → get answer. Initially, focus on Q&A about the current article (e.g., user highlights text and asks a question about it). Then expand to general queries (using search). **Agent task:** Develop the “AssistantManager” which given a user query, performs the steps to generate an answer (possibly stub out the LLM call if needed). Also build the UI: a “ChatSidebar” component that can slide in. Use dummy data to test the UI flow (like pre-canned answers) before hooking up the real model. Once the pipeline is ready, test with a real local model or API in a controlled manner, verifying that the answers use only provided context. Iterate on prompt templates to improve quality (e.g., ensure citations are included as planned).

8. **Text-to-Speech & Audio UX:** *Goal:* Add the ability to play article content as audio. Leverage platform capabilities: on web, use the Web Speech API; on desktop, use a Tauri plugin or OS API for TTS. Provide a play/pause button in the reader (perhaps in the article header or a floating player). Show progress or highlight the sentence being read if possible. **Agent task:** Implement a “TTSController” that given text will invoke the speech API, and UI controls to start/stop it. Handle things like voice selection and queueing the speech so it reads in chunks. This feature improves accessibility, so also test with keyboard-only usage and screen readers to ensure it doesn’t conflict.

9. **Multi-Theme UI & Polish:** *Goal:* Finalize the theming system and overall UX polish. By this stage, we have several theme prototypes (Synaptic, Brutalist, Prismatic, Spectral). We integrate a **theme switcher** in settings, allowing the user to choose from a set of themes or create a custom one. This involves applying CSS variables as per theme definitions and ensuring UI components respond to theme changes (Tailwind can be configured to use CSS variable colors [61]). Also implement light/dark mode toggle and make sure it ties into the theme (some themes might have both a light and dark variant). Additionally, refine responsive behaviors and fix any UX issues. **Agent task:** Build a “ThemeManager” with a list of theme definitions (color palettes, maybe even layout differences for radical themes). Possibly use a Tailwind plugin or just swap classes on the root. Also, address any UI to-dos (e.g., transitions, loading spinners, error modals for file issues). At this stage, the product should look and feel coherent across its features.

Each module can be tested and iterated independently, which is ideal for parallel development or AI-assisted development. For example, while the GraphView is being fine-tuned, another thread/agent can work on the Bookmark UI, since they touch different parts of the app. Throughout this roadmap, after each stage we will have a working application increment (e.g., after stage 4, we have a fully usable reader with search). This incremental approach ensures that even if some advanced features take longer (like the assistant), the core application is usable and can gather feedback. Moreover, by defining clear module boundaries and success criteria for each (e.g., “Search returns results within X seconds on a Y GB file” or “Assistant answers questions with citations”), we make it easier for an AI agent to understand the requirements and verify its work.

Finally, automated tests will be developed alongside (where feasible) to validate each module’s functionality – this also assists AI agents by providing immediate feedback. For instance, a set of unit tests for the search index or a snapshot test for the theme styles can signal if the agent’s code meets the expectations. This roadmap, therefore, doubles as a plan for an AI-collaborative development cycle.

Cross-Platform Strategy (Desktop, Web & Mobile)

EnZIM is planned as a cross-platform solution from the outset, targeting Desktop, Web, and Mobile with as much shared code as possible.

- **Desktop (Tauri):** The primary target is a Tauri-based desktop app for Windows, macOS, and Linux. We chose Tauri (Rust backend, webview frontend) to leverage native capabilities (file system access, native windows) while writing the UI in web technologies (React/TypeScript) [62] . The desktop app packages the Rust ZIM library for optimal performance on large files, and can handle multi-threaded operations (like indexing) without browser limitations [55] . Features like the semantic mesh rendering can use WebGL or Canvas in the webview, taking advantage of desktop GPUs for smoother visualization. We also plan desktop-specific enhancements: system tray integration for quick access, registering the .zim file type to open with EnZIM, and perhaps global hotkeys for the assistant. The desktop UI will be resizable, and we'll ensure it adapts to various window sizes (responsive design) rather than fixed dimensions.

- **Web App / Browser Extension:** EnZIM will have a variant that runs in the browser for users who cannot or do not want to install a desktop app. This will be similar to the Chrome extension "Zimmer" concept [63]. In this mode, due to browser sandboxing, the app cannot directly open arbitrary files on disk without user input. Users will load a ZIM by selecting it via an `<input type="file">` (or by using the extension's context menu to save pages into a ZIM archive). Once loaded, the entire archive lives in memory (or potentially in IndexedDB for persistence) [63]. The TypeScript ZIM library (ZIMReaderBrowser) is crucial here, as it can operate on the ArrayBuffer of the file without any server [55]. The web version might not support all features – for example, spawning a full local LLM for the assistant is not feasible purely in-browser, so the "Chat with ZIM" might require connecting to a local server or could be disabled in this mode. However, core features like reading, searching, and the semantic graph (via client-side JS) will work. We'll also explore making it a Progressive Web App (PWA) so that it can be "installed" in the browser and work offline after the initial load. Additionally, a Chrome/Edge extension build will be made: this could integrate with the browser to allow saving the currently viewed page into a ZIM (for personal archives) and using the EnZIM viewer to read it – effectively providing offline Wikipedia in the browser itself.

- **Mobile (Android/iOS):** Mobile support is planned, focusing first on Android using Tauri Mobile (which wraps a webview with Rust on mobile) [64]. The idea is to reuse as much UI code as possible; our React/Tailwind front-end can likely be used with minimal changes in a mobile webview context. We will adapt the layout for small screens: a single-column design, touch-friendly controls (larger buttons, swipe gestures to open panels), and maybe a bottom navigation bar for switching views (Library, Reader, Search, etc.) [64]. File access on mobile will rely on the OS's file picker (to import ZIMs) and we must handle storing large files on device (which could be several GB, so we need to stream from storage, not load fully into memory). Android's Tauri support will allow us to use the Rust backend similarly to desktop, but we'll watch memory usage closely given mobile constraints. For iOS, since Tauri support is emerging, we might alternatively package the web app in a WebView inside a Swift wrapper or use React Native with a native module for ZIM parsing. iOS has stricter sandboxing, so file import/export flows need to use the UIDocumentPicker, etc. In both mobile platforms, we'll enable the text-to-speech using native TTS APIs for a better voice (and because iOS Safari might not allow Web Speech without internet). The mobile UX will emphasize quick access: e.g., an initial screen to pick an archive, then a simplified reader with an always-visible search button and back navigation. We'll also integrate mobile-specific features like sharing an article (export to PDF or HTML, or share text) and possibly background download of ZIM files within the app.

- **Unified Codebase:** To maintain consistency, we use a single codebase for UI and a well-abstracted core. The React frontend will be shared across Desktop and Web (with conditional code where needed). The styling uses Tailwind with design tokens so we can easily adapt to dark mode or different themes across platforms [65]. The data layer (ZIM reading, search) is accessed via an interface that has two implementations: one calls Rust via Tauri's IPC (for desktop/mobile), the other calls the JS library functions (for web). By keeping these APIs the same (e.g. a function `openArchive(pathOrFile)` and `getArticleByTitle(title)`), the React components don't have to worry about platform differences [66]. We will run platform-specific testing (e.g., ensure the iOS build passes App Store guidelines and memory limits, ensure the web version works in various browsers, etc.).

- **Deployment and Updates:** The desktop app will be distributed via installers (and possibly the Microsoft Store / Mac App Store if feasible). The web extension will be published to the Chrome Web Store and Mozilla Add-ons for Firefox if we adapt it (Firefox might need some tweaks if not Chrome-specific). Mobile apps will go to Google Play and Apple App Store when mature. We will consider how to provide archive content to mobile users – possibly pointing them to Kiwix's archive library for downloads, or even bundling a small sample archive to start. Cross-platform also means keeping UX consistent: we maintain the same visual style and iconography across all platforms for familiarity [65], but also respect platform conventions (for instance, on iOS, use the native share sheet icon, on Android use Material-like toggles where appropriate).

In essence, EnZIM's cross-platform strategy is **"write once, adapt as needed"**: use the web tech for maximum reuse, use native code only where necessary (ZIM parsing, TTS, etc.), and design the UI to be responsive and flexible. This way, users get a similar, high-quality experience whether they are on a laptop offline in a library, on a phone during travel, or in a web browser at home. By leveraging Tauri and modern web capabilities, we maximize code sharing and minimize having to maintain separate codebases for each platform.

UX Themes & Palette Integration

EnZIM embraces a design system that supports multiple **themes** or skins, both for aesthetic preference and practical readability (light/dark modes). Early prototypes explore four distinct theme concepts:

- **Synaptic Cartography Veil:** A futuristic, neon-infused dark theme inspired by neural networks. It features a black/indigo background with an animated generative mesh pattern, giving the impression of a "digital brain" in the background. The UI elements are glassy panels with blurred translucent backgrounds [67]. Highlight colors are electric cyan and blue for interactive elements [68], complemented by magenta and gold accents for secondary highlights [68]. This theme uses a lot of glow effects and subtle animations (pulsing dots, etc.) to evoke a high-tech feel. Font is mainly sans-serif (Inter) with monospace for code-like elements, and text is colored in a soft white/gray on the dark background [69]. This theme is ideal for users who enjoy a visually striking, immersive UI. Despite the flair, it maintains readability through high contrast text and judicious use of color for links and nodes.

- **Brutalist Archive Monolith:** A bold, print-like theme that draws from brutalist web design. It uses a light gray "concrete" background texture [70] with stark black text and heavy black borders around UI elements (giving a "printed card" appearance). The typography is a mix of a grotesque sans-serif and monospaced font for a utilitarian feel [71]. Bright accent colors like construction orange and blue are used sparingly for highlights or active indicators [72]. This theme's article cards in the library view appear as literal cards with drop shadows and solid outlines [28] [73]. On hover, they shift position as if physically lifted (a brutalist interactive touch). Overall, the Brutalist theme appeals to those who prefer a no-nonsense, grid-aligned interface with a touch of retro (it almost feels like reading a printed encyclopedia page under glass). It ensures high legibility with black-on-light text and is often favored in bright environments.

- **Prismatic Swiss Utility:** A clean, modern theme with a Swiss-design influence (think clear layouts, grid systems, and helvetica-style clarity). It has a mostly white and light gray interface [74], with splashes of color drawn from a **prism palette**: cyan, magenta, and golden yellow as accent lines and hovers [74]. The name "prismatic" comes from these multi-color details (for example, section separators or border underlines might have a subtle gradient line going from cyan to pink to yellow [75]). The typography uses Inter for normal text and a stylish display font

(Space Grotesk) for titles [76] [77], giving it a contemporary but slightly artistic tone. UI panels are white with a slight translucency or shadow (mimicking paper on a light table). This theme likely is the most neutral and business-like, great for daytime use or presentations. Interactive elements like menu items highlight with a light cyan or pink background on hover [78] [79], reinforcing the polished feel. Importantly, Prismatic Swiss is very *utility-focused*: everything is crisp, minimalistic, and oriented toward functionality with just enough color to guide the eye.

- **Spectral ZIM Reader:** A theme that combines dark mode with vibrant "spectral" highlights. It has an almost sci-fi aesthetic with a backdrop of subtle radial gradients in neon cyan and pink that appear like lens flares or spectral refractions on a dark void background [80]. The UI elements (sidebars, headers) are semi-transparent black with blur, similar to Synaptic, but Spectral uses a unique **"filament"** motif: for example, the app's vertical sidebar is referred to as the "filament spine," a slim panel with glowing edges [81] [82]. Active elements glow in cyan or sometimes amber, and there's use of rotation or motion (a floating action button might rotate on hover with pink highlight [83]). Fonts are Inter and JetBrains Mono, giving it a techy feel, with text colors in cool whites and gray-blue for secondary text [84]. Spectral seems to emphasize the reading experience: the main article view is centered with ample padding, using relatively large text for comfortable reading in the dark [85] [86]. This theme would appeal to users reading at night or those who enjoy a sleek dark UI with some personality.

All themes support a common layout structure, but they can adjust certain metrics (for instance, the Brutalist theme might use larger padding and a grid background [87], whereas Synaptic and Spectral focus on absolute-positioned visual backdrops). We use a **CSS variable approach** to define theme colors and styles, which allows dynamic switching without reloading CSS [88]. Tailwind is configured to use these CSS variables for colors, so the components automatically pick up the new theme's palette when variables change. We also provide a **light vs dark** toggle where applicable: some themes are inherently dark (Synaptic, Spectral) or light (Brutalist, Prismatic), but we could create a light variant of Synaptic or a dark variant of Prismatic by adjusting the palette. Additionally, a set of preset themes (likely the 9 themes from the Zimmer extension) will be available [89] – the four above are prominent examples, but others might include variations like Sepia mode, High-contrast mode for accessibility, etc. Users can preview and select themes in a Settings panel, where each theme is shown with a small thumbnail.

From a UX perspective, theming is not just about color but about **mood and usability**. We ensure that in every theme: - The content remains highly legible (meeting contrast guidelines). E.g., in dark themes, text is slightly tinted away from pure white to avoid eye strain, and in light themes, we avoid too-bright white backgrounds by using off-whites. - Interactive elements are clearly indicated. Each theme has a designated accent color for links and buttons (cyan in Synaptic, orange/blue in Brutalist, etc.), so the user always knows what is clickable or selected. - The personality of the theme aligns with its purpose: for deep focus reading at night, a darker Spectral theme; for an academic/library setting, maybe Brutalist or Prismatic for print-like clarity.

Finally, the theme system is implemented in a way that is **extensible**. Developers or even users (if we expose custom theming) could create new themes by defining a set of CSS variable values (and perhaps some background images or patterns). Since EnZIM is MIT-licensed and oriented to community use, we expect that theme contributions could come from the community, expanding the palette. Internally, we keep theme definitions in a structured format and maintain consistency (for example, we ensure each theme defines a `--color-bg`, `--color-surface`, `--color-primary`, etc., even if the values differ) [61]. This also made it easier to implement the 9-theme rotation in the Chrome extension earlier [89], and we carry that flexibility into EnZIM.

By integrating these varied themes, EnZIM caters to a broad range of user preferences and contexts, all while keeping the core experience uniform. Whether someone prefers an energetic, graphically rich interface or a simple, scholarly one, they can tailor the reader's appearance without losing functionality – a key aspect of our UX strategy.

References:

1. EnZIM Product Requirements & Architecture Document [90] [65] – *Core feature scope, theming, and platform targets.*
2. AnZimmerman Clean-Room ZIM Library – README [91] [92] – *Overview of the ZIM library implementation (MIT-licensed, multiple languages).*
3. Architecture – System Components [46] [93] – *Integration of parsing engine and search/semantic analyzer components.*
4. UI Prototype – Synaptic Cartography Veil (HTML/CSS) [94] [95] – *Neon dark theme with mesh network UI elements.*
5. UI Prototype – Brutalist Archive Monolith (HTML/CSS) [96] [97] – *Light theme with bold lines, archive card grid layout.*
6. UI Prototype – Prismatic Swiss Utility (HTML/CSS) [74] [40] – *Minimalist theme, multi-color accents, semantic mesh side panel.*
7. UI Prototype – Spectral ZIM Reader (HTML/CSS) [80] [81] – *Dark theme with spectral glow, compact nav bar, focus on content.*
8. AskZim Project – README [10] [56] – *Example of an offline AI Q&A system using ZIM files (inspiration for assistant mode and citations).*
9. EnZIM UI Development TODO [98] [99] – *Notes on implementing multi-theme support and a built-in ZIM viewer with search, bookmarks, etc.*

---

[1] [2] [3] [4] [5] [6] [7] [8] [11] [12] [13] [14] [15] [42] [43] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [58] [60] [61] [62] [63] [64] [65] [66] [88] [90] [93] AnZimmerman ZIM Reader – PRD & Architecture Document (1).txt
file://file-8gCqWmBmr49VQ3J7eubBut

[9] [89] [98] [99] UI_TODO.md
https://github.com/rebots-online/EnZIM/blob/2a9721ed046b4162fe2192bf9b84ace01b005b5d/TOOLS/UI_TODO.md

[10] [56] [57] [59] GitHub - KaboomFox/askzim: askzim is an offline, AI-powered command-line and API backend for querying the contents of .zim files.
https://github.com/KaboomFox/askzim

[16] [17] [18] [19] [22] [23] [26] [27] [31] [32] [33] [34] [35] [38] [39] [67] [68] [69] [94] [95]
synaptic_cartography_veil_1.html
file://file-HHEuB6bpRf1w3mNp98gjUh

[20] [21] [36] [37] [40] [41] [74] [75] [76] [77] [78] [79] prismatic_swiss_utility_0.html
file://file-Qm7gxxxHEaz6sVqPc24h2U

[24] [25] [28] [29] [30] [70] [71] [72] [73] [87] [96] [97] brutalist_archive_monolith_0.html
file://file-AP4y3SiAbiUxgR5o4GSDYR

[44] [45] [91] [92] README.md
https://github.com/rebots-online/AnZimmerman/blob/5465cf4816753cd9e7054a77617e7dfcbec9b0cd/README.md

[80] [81] [82] [83] [84] [85] [86] AnZimmerman-.html
file://file-GLh24AysoHug2c1kvLWDxo