ChatGPT

EnZIM – Product Requirements & Architecture Specification

1. Executive Summary and Vision

EnZIM is a next-generation **offline knowledge explorer** that evolves the AnZimmerman project's libraries into a production-ready application for reading and navigating ZIM archives (like offline Wikipedia) with advanced semantic and AI features. The vision is to provide users with a rich, encyclopedia-like experience entirely offline – combining a high-fidelity content reader, an interactive knowledge graph, and an assistant for intelligent querying. Built on clean-room **openZIM** implementations, EnZIM ensures there is no licensing contamination (no GPL code) while fully supporting the ZIM format for reading and writing content [1]. The product aims to **empower learners in low-connectivity environments** (e.g. students, travelers) by enabling fast, convenient access to knowledge without internet, and to introduce new ways of exploring that knowledge via semantic relationships and AI assistance.

Key Objectives: EnZIM focuses on an **offline-first** design (all features work without internet connectivity) and is optimized for speed and usability on a range of devices, from desktops to modest mobile phones [2] [3]. It emphasizes an intuitive UI that allows users to seamlessly switch between reading articles and visually exploring related topics. A built-in assistant ("Chat-with-ZIM") offers summarization of articles and answers user questions by drawing on the archive's content, simulating a knowledgeable guide. By integrating these capabilities, EnZIM transforms static offline encyclopedias into an **interactive, intelligent knowledge interface**, aligning with the mission of making offline information accessible and engaging. The application's cross-platform architecture (desktop, web, mobile) ensures this experience is consistent everywhere, leveraging a shared codebase and responsive design system [4] [5].

2. Core Features Table

The following table summarizes EnZIM's core features and capabilities:

Feature	Description
Offline Archive Reader	**High-fidelity ZIM Reader:** Open and display content from ZIM files (e.g. offline Wikipedia dumps) with full fidelity, including text, images, links, and styling [6]. Users can navigate within articles (scroll, follow internal links) and use an in-app table of contents if available. The reader presents content in a clean, responsive layout optimized for reading (akin to a browser reader mode).
Library Management	**Archive Library & Import:** Manage a library of multiple ZIM archives. Users can add or remove ZIM files and see metadata for each (title, size, description) [7]. A library view lists available archives (with friendly names and details) and allows selecting one to browse. (Stretch goal: an integrated downloader to fetch new ZIM files from online catalogs, with progress indication).

Feature	Description
Full-Text Search	**In-Archive Search Engine:** Perform full-text search within an open ZIM archive. Users can query article titles (and potentially content) with instant suggestions and get a list of matching articles [8] . Search results show article titles (and a snippet if possible) and allow jumping to the selected article. The app uses the ZIM file's built-in index if available [9] , or builds one on the fly for large archives, ensuring efficient queries even on multi-gigabyte files.
Semantic Mesh Explorer	**Knowledge Graph View:** Visualize relationships between articles in a "semantic mesh" or concept map. Each article is represented as a node, and connections (edges) represent hyperlinks or semantic relations (e.g. "See also" links) [10] . Users can toggle this 3D-inspired graph view to see the current article and its related topics as a network. It's an interactive explorer: clicking a node centers it (loading that article) and reveals its connections, enabling non-linear traversal of the knowledge web. This feature helps **knowledge explorers** discover content visually and intuitively.
Bookmarks & History	**Personal Archive & Recents:** Users can bookmark articles for later reference and view their reading history [11] . Bookmarked articles are saved persistently (with their title and reference) and listed in a Bookmarks view. A History view (or section) logs recently read articles (with timestamps), allowing users to retrace their steps. Both Bookmarks and History are stored locally per user, and can be cleared or exported.
Annotations & Notes	**In-Article Annotations:** Users can annotate content with highlights, comments, or voice notes. The annotation system supports multiple modes – text highlights/notes, voice recordings, or even ink drawings over content (useful on touch devices) [12] . Annotations are saved locally (e.g. as metadata linked to article or specific text) so users can revisit their notes. This feature turns EnZIM into a personal study tool, not just a reader.
Text-to-Speech (TTS)	**Audio Reading:** The app can read articles aloud using text-to-speech. This improves accessibility and enables hands-free usage. TTS uses offline-capable engines (e.g. OS-native TTS APIs or bundled voice models) so that users in offline environments can still have articles narrated. Users can control playback (play/pause, adjust speed) and the feature integrates with the bookmark system (e.g. continue from last listened position).
Chat Assistant ("Ask ZIM")	**AI-Powered Q&A and Summaries:** An integrated assistant lets users query the archive in natural language. For example, a user can ask *"What is quantum mechanics?"* and the assistant will provide a brief answer citing the relevant article [13] [14] . It can summarize a long article, help navigate by suggesting related articles, or answer questions by pulling facts from the ZIM content. The assistant runs locally or in a privacy-preserving manner (no external API calls by default) and uses the archive's text as its knowledge base. This feature turns the static content into an interactive conversation, aiding learning and discovery.

Feature	Description
Multi-Platform Support	**Desktop, Web, and Mobile:** EnZIM is available as a Tauri-based desktop app, a web application (or browser extension), and in the future a mobile app. All platforms share the same core features and UI principles [4] [5]. The UI is responsive and adapts to various screen sizes (widescreen desktop, tablet, phone) [15]. On desktop, users get native file access and performance boosts (via Rust), while the web version runs client-side in the browser (with IndexedDB for storage) [16]. Mobile versions will adjust navigation for touch (e.g. swipe gestures, bottom tabs) [5], focusing on one-handed use and offline storage optimizations.

(Sources: Core features derived from original AnZimmerman PRD [6] [10] and project notes [12].)

3. UI Flow Map and View Hierarchy

EnZIM's user interface is structured into modular views that together provide a cohesive experience. The app follows a **hub-and-spoke model**: a Library hub where users manage archives, and multiple primary views (Reader, Search, Graph, etc.) that stem from it. Below is an overview of the UI flow and view hierarchy:

- **Library View (Startup/Home):** When the app launches, it opens the **Library** view showing all available archives. This view presents a list or grid of ZIM files in the user's library, each displayed as a card with its title and details (e.g. "Wikipedia English – 4GB"). From here, the user can import new archives or select an archive to open [17]. The Library is the entry point and can be revisited anytime (e.g. via a "Library" button or sidebar).
- *Interactions:* Clicking an archive (ZimCard) in Library triggers `openArchive(id)` – loading that ZIM and navigating to the Reader view for its main page [18]. There is also an "Add Archive" action (button) to import a new .zim file into the library.

- **Reader View (Article View):** This is the core reading interface where a selected article's content is displayed. It consists of an article header (with title, maybe breadcrumbs for category) and the article content in a scrollable area. Within the Reader, users can click hyperlinks to navigate to other articles (within the same archive). Basic navigation controls (Back/Forward) allow moving through reading history. The Reader view also provides tools like **Bookmark** (to save the current article) and **TTS** playback controls. If the user opens an archive from Library, by default the Reader shows the archive's main page (e.g. Main_Page of Wikipedia) or a welcome page. Subsequent navigation stays in Reader view, updating the content.

- **Search (Overlay or Integrated):** A search bar is accessible at the top of the interface (persistent in the header) [19] [20]. When the user enters a query, a Search Results panel or overlay appears, listing articles matching the query. This could be a dropdown under the search bar or a full view overlay depending on design. Selecting a result will navigate the Reader to that article. In some designs, the search results may appear in the main content area or as a modal, but in all cases the flow is: user stays in context (Library or Reader) while viewing results, then jumps to Reader for the chosen article.

- **Semantic Graph (Mesh) View:** Toggling the "semantic mesh" brings up the interactive graph of related articles. In the UI, this can be presented in a couple of ways:

- **Overlay Mode:** The graph can overlay on top of or alongside the Reader content. For example, one concept design uses a three-panel layout: a left sidebar for the archive library, the center for the semantic graph, and a right sidebar for the article content [21] [22]. In this mode, the current article remains visible while its relationships are shown graphically in the center pane. The user can click another node in the graph, which will load that article (updating the Reader panel) and re-center the graph on the new node.
- **Full Graph Mode:** Alternatively, the semantic mesh could be a dedicated full-page view (replacing the Reader temporarily). In this case, switching to "Graph View" would show the network of nodes on the screen; clicking an article node from here would either open the Reader for that article or open a detail popover. A back button would return to the normal Reader layout.

- *Interactions:* In either mode, the **centered node** represents the current article, visually highlighted [13]. Neighboring nodes (directly connected topics) surround it [14]. Users can pan/zoom the graph if needed and use controls to filter or expand connections (e.g. show more distant related nodes). The UI includes controls to toggle the mesh on/off and possibly a legend or status HUD showing graph metrics (e.g. number of nodes visible) [23]. This Semantic Graph view significantly enriches navigation by allowing serendipitous discovery of related topics.

- **Bookmarks & History Views:** Accessible typically via the library sidebar or a menu, these views list user-saved bookmarks and recent articles. In the UI, they might appear as filtered lists within the Library panel or as separate sections. For instance, a sidebar might have collapsible sections for "Local Archives", "Bookmarks", and "History" [24]. Clicking on a bookmarked article or a history item immediately opens that article in the Reader view (and if needed, opens the corresponding archive). These views help users quickly return to content of interest.

- **Assistant (Chat) Panel:** The "Chat-with-ZIM" assistant is invoked by a UI control, perhaps a chat icon or a dedicated panel. One approach is a slide-up panel or chat window that can overlay the bottom of the Reader. The user enters questions or commands here. The assistant responds with answers, summaries, or guidance, often including hyperlinks to relevant articles. For example, if the user asks for a summary of the current article, the assistant might display a short summary and offer to read it aloud or highlight key sections. If the user asks a general question, the assistant might identify one or more articles in the archive as sources and provide an answer with reference links. The assistant UI keeps a history of the Q&A exchange so the user can follow up with another question. It effectively adds a conversational navigation layer on top of the traditional UI. *(Design-wise, this was not fully illustrated in the initial prototypes, but it will be integrated in a minimalistic way to avoid clutter – e.g., an expandable chat bubble on the corner.)*

- **Settings & Theme:** A settings view allows configuration of preferences such as theme (light/dark or various color themes [25]), text size, and other options (e.g. toggling TTS voice selection or assistant mode online/offline). This might be a modal or separate page accessible via a gear icon.

UI Hierarchy Summary: The application's UI is modular and can be seen as layers: - *Global Shell:* Header with branding and search bar, plus maybe a global sidebar or navigation menu. - *Primary Views:* Library (archive selection), Reader (article content), Graph (semantic mesh), Assistant (chat) – some of which can be shown simultaneously in split panels depending on screen size. - *Secondary Panels:* Bookmarks, History, Settings (often overlaid or in sidebar).

Thanks to a consistent design system, these components share a common look and feel. Notably, we drew on three design concepts – **Brutalist Archive Monolith**, **Synaptic Cartography Veil**, and **Prismatic Swiss Utility** – when crafting the UI. The final interface is a synthesis of these: - *Brutalist*

Archive Monolith: inspired the Library view with a grid of archive "cards" and a bold, clean typography (monospace labels, strong contrast) [26] [27]. - *Synaptic Cartography Veil:* introduced the translucent, futuristic aesthetic for the semantic mesh with a dark background, neon accents, and a layered glassmorphic UI for panels [28] [29]. It informed the design of the Graph view and overall mood for exploration mode. - *Prismatic Swiss Utility:* contributed a lighter, modern Swiss-style layout with efficient use of space and subtle color accents (prismatic highlights). It demonstrated a balanced three-column layout (library – reader – graph) that influenced our responsive design [30] [31]. It also showed how bookmarks and other lists can be cleanly integrated into the sidebar navigation [32].

By combining these ideas, EnZIM's UI achieves **modularity** (each panel can be shown or hidden as needed), **clarity** (brutalist emphasis on content and hierarchy), and **vibrancy** (prismatic and synaptic visual cues to make exploration engaging). The flow between views is designed to feel natural: for example, a user can seamlessly go from reading an article to exploring related topics on the graph, then searching for something, and back to reading – all within one cohesive interface.

4. Semantic Graph Engine & Interaction Pattern

One of EnZIM's signature features is the **Semantic Mesh Explorer**, powered by an internal Semantic Graph Engine. This component is responsible for constructing and rendering the network of article relationships, and handling user interaction with that network.

Graph Construction: When a ZIM archive is loaded, EnZIM analyzes its contents to build a graph data structure: - **Node Definition:** Each article (content page) is a node in the graph. Key properties stored may include an identifier (or URL/path), title, and possibly metadata like article length or category. - **Edge Definition:** Edges represent meaningful relationships between articles. Primarily, these are explicit hyperlinks found in articles – if article A contains a link to article B, we create an edge A→B. This naturally forms a directed graph, but for the purpose of user exploration we treat it as an undirected connection (showing "A is related to B"). In future, other relationships could form edges too, such as belonging to the same category, or even semantic similarity via NLP. For now, hyperlink relations and *"See also"* references form the backbone of the mesh [10]. - **Algorithm & Performance:** The Semantic Analyzer module scans each article's content (or uses pre-parsed link indexes if available) to gather outgoing links [33]. Because archives like Wikipedia have thousands of links, we will likely limit the graph to certain scope (e.g., only links within main namespace, ignore trivial links, etc.). The analyzer might build the graph incrementally – for example, generating neighbors on the fly when an article is viewed, rather than storing the entire wiki graph in memory at once, which could be huge. Some caching is used so that if a node's neighbors were computed before, they are stored for quick reuse. - **Semantic Expansion:** As an enhancement, the engine can optionally compute content-based similarity. This could involve generating vector embeddings for article texts (likely too heavy to do on-device for very large archives) or leveraging precomputed data. Such similarity edges (like "articles on related topics") are a stretch goal – they are not explicitly in the ZIM, but could be precomputed offline and shipped with the archive or computed in a background thread. The system is designed to allow plugging in these additional edges in the future [33].

Graph Data Model: We maintain a structure like `Graph { nodes: Map<id, NodeData>, edges: Map<id, Set<id>> }`. For a given current article, we can retrieve its directly linked neighbors in O(1) via a lookup. Each node might also have a cached list of neighbors for quick iteration. There may be a limit to how many neighbors we display at once for usability (e.g., if an article has 100 links, perhaps show the 10–15 most relevant in the graph based on some ranking, to avoid visual clutter). The engine might rank links by importance (e.g., links in the introduction of the article, or links that are mutual between pages).

User Interaction Pattern: The semantic mesh is interactive: - When the user activates the Semantic Mesh view (for the current article), the engine generates the subgraph of interest: the **focus node** (current article) and its immediate neighbors (linked articles). These are passed to the UI for rendering in the mesh canvas. The focus node is highlighted distinctively (e.g., larger node, special color) [13], while neighbors are slightly smaller or translucent. - The user can click on any neighboring node. Upon clicking, that node becomes the new focus: the Reader view will load that article, and the graph view will update to show that node's neighbors. This is essentially following a link, but via the graph interface. The transition is smooth – it might animate the newly clicked node to the center. - Users may also navigate the graph using controls: for example, a *zoom* slider to adjust scale, or a *refresh/expand* button to fetch another layer of neighbors. If implemented, an "expand" control could fetch second-degree connections (neighbors-of-neighbors) and display them, possibly in a different style (e.g., smaller nodes) to indicate they are farther out [34]. However, we must guard against overloading the user with too many nodes; the UI likely limits the view to one degree at a time by default. - A **HUD (Heads-up Display)** or status overlay in the graph might show info like current zoom level and how many nodes are visible, and whether the mesh is fully active [23]. - **Edge visualization:** Edges are drawn as lines connecting nodes. In our implementation, these could be straight lines or curves; we may use subtle animation (e.g., pulsing or glowing lines) to draw the user's attention to active connections. The graph view in Synaptic Veil concept used subtle glowing connection lines on a dark background to evoke a neural network feel [21]. We'll ensure edges are distinguishable (perhaps highlighted when a node is hovered or focused). - **Graph Layout:** The positioning of nodes can be done with a simple force-directed layout or predefined circular layout. Given performance constraints, we may not run a heavy physics simulation in real-time; instead a fixed layout that looks good (like a starburst around the center) could be used for immediate neighbors. For instance, the prototype places neighbors in preset positions around the center node [14] [34]. This is computationally trivial and ensures a consistent appearance. We might randomize slightly or allow drag-and-drop repositioning if needed for clarity. - **Exiting Graph View:** Toggling the mesh off returns the user to the normal Reader-focused view (possibly closing the center graph panel or overlay). If the user had navigated via the graph, they will now be reading the article they ended up on. They can always re-open the graph for that article to continue exploration.

This Semantic Graph feature encourages users to explore the archive in a non-linear fashion, **transforming the encyclopedia into a web of concepts** rather than a set of isolated articles. It caters to the *Knowledge Explorer* persona (who enjoys seeing connections between topics) [35] [36] and differentiates EnZIM from basic archive readers. Technical note: all graph generation is done client-side, on the user's device, and stored in memory or local cache – there's no network call for computing relationships, as the data comes from the archive itself.

5. Backend Modules and Platform Split

EnZIM's architecture is composed of several modular backend components, with differences in implementation between platforms (desktop, web, mobile) to maximize performance and compliance with each environment's constraints. Below are the key modules and the cross-platform strategy:

- **ZIM Parsing Engine:** At the core is the **ZIM file parser** and content loader. This is implemented as a clean-room library in multiple languages (Rust, TypeScript, etc.) based on the openZIM spec [1] [37]. On desktop (Tauri), we leverage a Rust backend (`zimlib` crate) for efficient, multi-threaded I/O on large files [38]. In the web context (PWA or extension), we use a TypeScript implementation (`ZIMReaderBrowser`) that can read from an ArrayBuffer or blob [39]. Both expose similar APIs to the front-end. The parser can open a .zim file, read its directory entries, and fetch article content by title or index [40]. It handles decompression of data on the fly and ensures we **never load the entire file into memory at once** – instead we seek and stream the

needed parts (using IndexedDB for chunk storage on web if needed). This engine also provides low-level support for listing articles, getting the main page, retrieving images, etc., abstracting the binary format details away from the UI.

- **Search Engine & Indexer:** Searching within huge archives is non-trivial, so EnZIM includes a search module. On startup of an archive (or on first search query), an **Indexer** may run to build a lightweight index of article titles (and potentially content).

- *Desktop:* The Rust backend can build an on-disk or in-memory index (e.g. using a small embedded database or a custom trie) for fast lookup [41]. Full-text search might use an algorithm or library optimized in Rust, possibly in a background thread to avoid blocking the UI [42].
- *Web:* In the browser, we can use the JavaScript library to scan titles incrementally. For example, we might generate a list of titles and use a prefix search for suggestions. If full-text search is needed, we could compile a WebAssembly module of a search engine or use a wasm port of something like Lunr.js. We will likely limit full-text on web to smaller archives or rely on title search for speed. The web app can utilize Web Workers to offload indexing so the UI remains responsive [43].
- The search module provides an API: e.g. `searchArticles(query) -> [ArticleEntry]`. It returns a ranked list of results (title, snippet, etc.). The **architecture ensures consistent results across platforms**, even if the underlying implementation differs (Rust vs JS) [44].

- Because search is such a key feature for offline usability, we set a goal that even for large archives (millions of entries), a title search returns results within a second or two [45]. This may involve trade-offs (perhaps not indexing every word of content, only titles and redirects by default).

- **Semantic Graph Engine:** (Discussed in section 4 above.) This module can be considered part of the backend logic as well – it collates link data and (in future) performs any heavy NLP tasks for semantic similarity. On desktop, heavy computation (like generating vector embeddings for all articles) could be done in Rust or Python in the background if ever implemented. On web, we would likely not do that at runtime due to performance and memory constraints (we might precompute data offline). For link graph generation (lighter weight), both platforms can handle it by parsing HTML for links. The output is a data structure of connections that the front-end uses to render the mesh [46].

- **Assistant AI Module:** The Chat-with-ZIM assistant relies on AI capabilities. The architecture keeps this modular to allow different implementations:

- In an online scenario (if the user has internet and opts in), the assistant could call an external LLM service (with the ZIM content as context) to generate answers. But for offline use, we consider integrating a local inference engine. This could be a distilled language model running on-device or a rule-based Q&A that searches the archive. A possible approach is an **extractive QA**: the assistant performs a search in the archive for the user's question (like keyword search behind the scenes) and then formulates an answer from the found text.
- The assistant module likely uses the same search index to find relevant articles, then uses either a built-in summarization model or heuristic template to answer. For instance, if user asks "What is X?", the assistant can open article X (or the top result), take the first paragraph and present it as an answer, along with a "According to [Article Title]..." reference.

- This module will be implemented in Python or JavaScript. For example, we might include a small Python runtime in the app for AI tasks (if using something like a distilled transformer with ONNX). In Tauri/Rust, another possibility is to integrate `llama.cpp` or similar to run a small model CPU-only. The architecture leaves this as a plug-in service that communicates with the UI via a well-defined interface (maybe using Web Workers or Tauri commands). In summary, the assistant logic is separate from core browsing, and can be updated/upgraded independently (for example, swapping in a better model down the line).

- **Text-to-Speech Engine:** TTS is handled using platform capabilities wherever possible:

- On **desktop**, we can call out to OS-native TTS (e.g. using an API or rust crate that interfaces with system voices) – this ensures the use of built-in voices and offline functionality. Alternatively, we could bundle an offline TTS engine (but that increases app size and complexity).
- On **web**, the Web Speech API could be used. Modern browsers have an API for speech synthesis that works offline with cached voices. We will use that if available. If not, the web app might not support TTS fully offline (or at all, depending on browser support).

- On **mobile**, Android's TTS engine can be invoked via an Android plugin in Tauri; iOS similarly has AVSpeechSynthesizer. We unify these behind a single UI control. The architecture thus abstracts TTS such that the Reader view's "Play" button calls a generic `speak(text)` function, which under the hood calls the appropriate implementation for the platform.

- **Data Storage & Sync:** EnZIM stores user data like bookmarks, history, settings, and annotations in a local database:

- Desktop: likely using SQLite (via the Tauri backend) or even a simple JSON file if data is small. SQLite is preferred for robustness. The Rust side can expose queries or we use Tauri's IPC to communicate between front-end and a Rust data layer.
- Web: uses IndexedDB or localStorage. For example, bookmarks might be stored in IndexedDB with an object store keyed by archive ID.
- The data module ensures a consistent interface: e.g., a function `getBookmarks()` works on all platforms and returns a list of bookmark objects, whether behind the scenes it queried SQLite or IndexedDB [47]. This way, front-end components don't worry about where data is stored. We also ensure that data is kept separate per archive where appropriate (e.g., bookmarks might be segmented by archive).

- Currently, there is no cloud sync (all data stays local to device, aligning with offline-first). In the future, we may allow users to export their bookmarks/annotations or sync via a service, but that would be opt-in.

- **Platform-Specific Integrations:** Some features differ slightly:

- Desktop app can have a menu bar and system tray integration. For instance, we considered a tray icon for quick access to downloads or a mini search interface [38]. These are coded in Rust and only activated in desktop builds (the web app won't have a tray obviously). We ensure such codepaths are separated and do not affect the web build [48].
- Browser extension context (if we package EnZIM as a Chrome/Firefox extension) can integrate with browser features: e.g., a context menu "Save page to ZIM" or intercepting links to point to offline content [49]. These are considered part of a related project ("Zimmer" extension) but share the same core library. The architecture allows the extension to reuse EnZIM's components but with a different entry point (the extension popup UI instead of the full app UI).

- Mobile specifics: On Android, we have to handle permission to read files (for importing a ZIM) and possibly use a different file picker UI. Also, mobile might implement some native code to handle background tasks differently (for example, indexing might run as a background service). We anticipate using Tauri Mobile which is still evolving; if it's insufficient, we might consider Cordova/Capacitor as fallback. In any case, the core logic remains the same – only the shell differs.

Abstraction and Code Sharing: To manage these differences, we adopt a layering approach. The **front-end (React + TypeScript)** calls high-level functions for data (like `openArchive`, `getArticleContent`, `searchIndex`) without hardcoding the implementation. We have an **interface module** that detects the platform and delegates calls appropriately. For example, calling `getArticleContent(articleId)` might internally do:

```
if (window.__TAURI__) {
    invoke('get_article_content', { id: articleId }); // Tauri Rust command
} else {
    zimReader.getArticle(articleId); // JS library call
}
```

This pattern means 90% of the app code is platform-agnostic. Only the interface layer knows about Tauri vs browser differences [44]. Builder agents and developers can mostly write code once (targeting a generic interface) and trust it will run on all platforms.

Finally, **security** considerations are handled at the platform level: The app will sanitize or isolate any HTML content rendered, to avoid any malicious script from ZIM files from executing. On desktop, the Tauri WebView is configured with no remote script access and a strict Content Security Policy [50]. On web, if running as an extension or PWA, we may use a sandboxed iframe or DOMPurify-like sanitizer for the article HTML [50]. The clean-room library only deals with parsing, so it won't execute anything unintended; the rest is about how we render it safely.

6. Builder Agent Prompt Integration Templates

EnZIM is being developed in a **collaborative human+AI workflow**, where "Builder Agents" (AI coding assistants) help implement modules based on specifications. To facilitate this, we've established a consistent prompt format and naming conventions, ensuring that multiple agents (or developers) can work in parallel with minimal friction. This section outlines how we coordinate builder agents:

Standardized Prompt Format: All requests to the AI agents follow a structured template: - They begin with an explicit role cue, e.g., **"Builder Agent:"**, to clarify the task context [51]. - The request includes the component or module name and a concise description of what to build or do. For example: *"Builder Agent: Construct* `LibraryView` *component with a grid of ZimCard elements..."* [52]. - We then provide a breakdown of requirements: - **Description:** A high-level summary of the component or function's purpose. - **Inputs:** What data or props it should accept (for UI components) or parameters (for functions). - **Requirements/Constraints:** A list of specific details – UI layout specifics, state management expectations, performance constraints, etc., as bullet points. - **Output:** Specification of what the agent should deliver (usually code in a certain format, e.g., JSX for a React component, including styling with Tailwind CSS).

This structured prompt ensures the agent knows exactly what is expected. An example template for a UI component prompt might look like [53] [54]:

```
Builder Agent: Construct `SearchResultsList` component.
Description: Displays a list of article search results.
Inputs: `results` (Array of ArticleEntry objects) and `onSelect` (callback
when an item is clicked).
Requirements:
  - Each result item shows article title and snippet.
  - Highlight query terms in bold.
  - If no results, show a "No results" message.
  - Use `<ul>` with Tailwind classes for layout and spacing.
Output: JSX for the list and items, with event handling for selection.
```

The agent reading this prompt will then generate the corresponding JSX/TSX code fulfilling these criteria. We apply similar patterns for other kinds of tasks: - **Backend/Logic Module Template:** e.g., *"Implement* `buildSearchIndex()` *in searchWorker.ts* `"*, followed by description of the function, inputs (like an array of articles or path to ZIM), expected outputs (maybe a data structure or success flag), and any constraints (performance considerations, memory limits)` [55] `. The agent is asked to provide code (in the appropriate language) with comments.
- **Integration Task Template:** for tasks that involve wiring together components or hooking up state, e.g., *"Integrate` *BookmarksView* `with global state"*. The prompt would outline the steps the agent needs to take` [56] `(e.g., use a` *useAppState` hook to retrieve bookmarks, render them, handle click to open article, etc.) and specify which file to modify or produce. This ensures an agent doesn't just create isolated code, but knows how to fit it into the existing structure. - Testing/Validation Template: (We have not heavily used AI for testing yet, but in principle) a prompt can be structured to ask for unit tests or to have the agent review its output against criteria. For example,* "Builder Agent: Write unit tests for SearchResultsList component"* with requirements like covering cases for empty results, highlighting, etc. This is mentioned for completeness, though our main focus is on generation of the application code.

Naming Conventions & Consistency: We enforce strict naming conventions so that prompts and the resulting code remain consistent across the project [57]: - **Views & Pages:** Use PascalCase names ending in "View" for full-page components (e.g. `LibraryView`, `ReaderView`, `MeshView`, `BookmarksView`) [57]. These correspond to major sections of the app and often have their own route or container. - **Reusable Components:** Also PascalCase (e.g. `ZimCard`, `SearchBar`, `ArticleContent`). They usually live in a `components/` directory. - **State and Actions:** Use camelCase for state variables and actions (e.g. `currentZimId`, `openArchive`, `toggleBookmark`) [58]. We document a set of global actions so agents will call existing ones rather than invent new ones. For instance, when an agent implements the bookmark button, it knows to dispatch `toggleBookmark(articleId)` because we listed that in the spec (and perhaps in the prompt) [59]. - **Task Identifiers:** In documentation and commit messages, we might tag tasks like `[UI.LibraryView]` or `[State.Bookmarks]` [60]. This helps a supervisor (or a coordinating agent) to route the right context to the right agent. For example, an agent working on the UI can be pointed to the "UI Flow Map and View Hierarchy" section of this spec, whereas one working on state logic can be pointed to the "Developer UX – State store" section.

By adhering to these conventions, any code generated by different agents will fit together more coherently. An agent sees in the prompt exactly how things are named in the project, reducing chances of mismatched identifiers or styles [61]. It also means we can generate prompts semi-automatically: the spec document itself (this document) is written in a way that we can copy-paste relevant lines into a prompt. For example, if we have a section describing the `LibraryView` (like the stub prompt example above), we can literally use that as the basis of the agent prompt. This document thus serves as a **single source of truth** for both humans and AI builders [62].

Prompt Template Examples: We maintain a set of example prompts and stub templates to guide agents. As shown, a component prompt has sections for Description, Inputs, Requirements, Output. We have similar formatted examples for other typical tasks, so the agent always gets information in a familiar layout [53] [55].

We even include within this document an explicit example to illustrate usage:

```
Builder Agent: Construct `LibraryView`.
Description: The main library screen showing all available ZIM archives in a
responsive grid.
Inputs: `zimList` (Array of ZimArchiveInfo objects with {id, title, size,
description}) [17].
Requirements:
  - Display a grid of ZimCards. On desktop, use a 3-column grid; on mobile, a
single-column list.
  - Each ZimCard shows the archive title, a brief description or subtitle,
and size (e.g. "4200 articles, 3.5 GB").
  - Include an “Add Archive” button in the view (top-right) to import new
archives.
  - When a ZimCard is clicked, call `openArchive(id)` action and then
navigate to ReaderView [18].
  - Style: use Tailwind CSS for styling (cards with borders, hover highlight,
etc., matching design).
Output: Return a JSX component for LibraryView fulfilling the above
requirements.
```

We would feed such a prompt to the agent, and expect it to produce the corresponding `LibraryView.tsx` component code. Because the agent has been primed with our conventions (via earlier prompts or documentation references), it will know to output code consistent with our project structure. The example above demonstrates how the spec translates almost directly into a prompt format [63] [64]. We can repeat this for each module (“Construct `MeshView`…”, “Construct `SearchBar`…”, “Implement `zimSearch()`… etc.), effectively **modularizing development tasks** so that they can be tackled independently by builder agents. A lead developer (or coordinating agent) then integrates these pieces, with this document ensuring everyone (human or AI) is on the same page regarding how things should be named and behave.

In summary, the builder agent integration strategy in EnZIM is about **clarity and consistency**: clear templates for what to build, consistent naming so the generated code plugs in correctly, and a shared spec (this document) that all agents refer to for context. This has already improved development velocity, as measured by modules completed per iteration [65], and we plan to continue leveraging it throughout the project.

7. Data Model Overview

EnZIM's data model defines how content and user data are structured in the application. Key entities include the Archive (ZIM file), Articles (and related content), and user-generated records like bookmarks and annotations. Here we outline the primary data types and their relationships:

- **ZimArchive (Archive):** Represents a loaded ZIM file (a collection of articles). It has properties such as:
- `id` – a unique identifier for the archive in the app context (could be derived from file name or a hash).
- `title` – a human-friendly name of the archive (e.g. "Wikipedia English 2023-08").
- `description` – a short description if available (some ZIMs include description strings).
- `size` – size on disk (for info display) [17] .
- `count` – number of articles (if we want to display that).
- Possibly `mainPage` – the URL or index of the main page (if defined in the ZIM header, e.g., Main_Page).

- The ZimArchive object is mostly used in the Library view to show what archives are available and to initiate opening one. Once opened, a reference to the current archive is kept (in state, like `currentArchiveId` or a pointer to the reader object).

- **Article:** Represents a single article or entry from the archive. In the context of the app's state, an Article object might include:

- `id` or `index` – a unique reference (could be a combination of ZIM archive ID + internal index). In ZIM, each entry has an index in the directory table.
- `title` – the article title (which is often also the display title).
- `url` or `path` – the internal path used to fetch it (ZIM uses a namespace + URL scheme).
- `content` – the HTML content of the article (which we retrieve on demand when the article is opened). We usually do not keep all contents loaded for all articles – only for the one being read and maybe a cache of last few.
- `mimeType` – typically "text/html" for articles, but could be images or other media for non-article entries.
- `links` – (optional) list of linked article IDs or titles (this could be populated by the Semantic Engine).

- In practice, the low-level `ZIMReader` gives us either the content directly or a handle to retrieve it. We often convert content to a React-renderable form (e.g., sanitized HTML).

- **ArticleEntry / SearchResult:** A lightweight representation used for lists (search results, etc.). For example, when performing a search, we might create objects with:

- `title`
- `snippet` – a short snippet of text showing the context of the search term [54] .
- `articleId` – to retrieve the full Article when needed. This is to avoid carrying full content for every search result.

- **Bookmark:** When a user bookmarks an article, we create a record like:

- `archiveId` – which archive the article belongs to (since bookmarks might be global across archives or per archive; we consider per archive listing in UI).
- `articleId` – or some reference to the specific article (could be title as key, but article IDs are more stable if available).
- `title` – stored for convenience to display in the bookmarks list.
- `note` (optional) – if we allow users to add a short note or label to a bookmark. Bookmarks are stored persistently (in a JSON or DB). They can be retrieved and displayed in the BookmarksView easily. In the state, we might keep a list of bookmark IDs for quick access.

- **History Entry:** Each time an article is opened in Reader, we log a history record:

- `timestamp`
- `archiveId`
- `articleId`
- `title` History might be stored in a simple list (most recent first). We may cap the history to a certain length (to avoid unbounded growth). This helps implement back/forward navigation as well.

- **Annotation:** If annotations are supported, an annotation record could be:

- `archiveId, articleId` to identify context.
- Location in text (maybe an offset or a quote of text, or a DOM element ID if available).
- `type` – e.g. highlight, text note, voice note.
- `content` – for text notes, the note text; for voice notes, a path or blob reference to an audio file; for highlights, maybe the highlighted text excerpt.

- These would be stored in a local database and associated with the article. When an article is loaded, the app would check if annotations exist for it and overlay them (e.g., highlight the text or show note icons).

- **Semantic Graph Data:** The semantic engine might maintain its own data structures:

- We don't necessarily persist the entire graph (it can be regenerated). However, we might cache some results in memory. For example, we could have a dictionary of `{articleId: [relatedArticleIds...]}` for quick lookup. This could be filled on the fly.
- If we enhance the graph with computed relatedness, we might store an `ArticleVector` (embedding) per article, but that's advanced/optional.

- For the purpose of the UI, the "nodes" that get rendered are essentially ArticleEntries (id + title, plus maybe an abbreviated title if long). We use the same identifiers as the main article data, so clicking a node can fetch the corresponding content easily.

- **Global State Store:** From a high-level perspective, much of the above data is managed via a global app state (if using Redux/Zustand):

- e.g., `state.currentArchiveId`, `state.currentArticleId` (which the ReaderView subscribes to in order to display content).
- `state.bookmarks` could be an array of bookmark records.
- `state.history` similarly.

- `state.searchResults` for transient search result list.
- `state.theme` or `state.settings` for UI preferences.
- We carefully define this state shape so that all components and agents use it consistently [66]. The actions (like `openArchive`, `openArticle`, `addBookmark`, etc.) update these state fields accordingly [59].

Consistency with ZIM Spec: Internally, the ZIM Parsing Engine deals with **DirectoryEntries** and **Namespaces** as defined by openZIM [67] [68]. For example, images are stored as separate entries (namespace 'I'), and the main articles are namespace 'A'. In our data model, we abstract that away: an Article in the app may include images by referencing their blob URLs via the ZIM loader on demand. We ensure images get loaded when needed (the HTML content has placeholders for images which we populate by fetching the image entry from the ZIM). The data model thus indirectly includes those relationships (article content includes references to image entries). We might have a caching mechanism for images (so if multiple articles use the same image, we don't reload it each time).

Data Lifecycle: When the user closes an archive or exits the app, we persist necessary data. Bookmarks, history, annotations, settings – all are saved. The archive content itself isn't stored permanently by the app (the user keeps the .zim file externally, or if they downloaded it through the app, it's stored in app data folder or a known location). If the user removes an archive, we'll also remove any associated bookmarks/history for cleanliness (or mark them as orphaned).

By organizing data in this way, we keep a clean separation between *content* (which is in the ZIM and accessed via the parsing engine) and *user data* (which is managed by our application state and storage). This separation ensures that, for example, updating or replacing a ZIM archive won't corrupt user data (worst case, some bookmarks might point to an article that's no longer present if they switch to a different version of an archive – a scenario to handle gracefully). The data model is designed to be simple and serializable, which also means it can be easily shared with AI agents in prompts (for instance, providing the structure of a `ZimArchiveInfo` or `Bookmark` object as part of a prompt, so the agent knows what properties to use) [17]. Consistency in these definitions is key for both correct functionality and seamless builder agent contributions.

8. Developer UX (State Management & Platform Distinctions)

EnZIM's development approach prioritizes a smooth Developer Experience (DX) for both human developers and AI builder agents. Two aspects are central to this: a unified state management system (to coordinate data flow across components) and clear handling of platform-specific distinctions (to minimize forks in code logic).

Unified Global State Store: We adopt a single-source-of-truth principle for application state. Whether we use Redux, Zustand, or Context + Reducer, the idea is that **all major app data is stored in one predictable structure** [69]. For example, a simplified state shape might be:

```
state = {
  currentArchive: <id or null>,
  currentArticle: <id or null>,
  archives: [<ZimArchiveInfo>, ...],
  bookmarks: { <archiveId>: [<Bookmark>, ...] },
  history: [<HistoryEntry>, ...],
  searchResults: [<ArticleEntry>, ...],
  settings: { theme: "...", ... },
```

```
  // ... etc.
}
```

All components subscribe to the parts of state they need. For instance, the ReaderView will listen to `state.currentArticle` and the content map to know what to display, and BookmarksView will use `state.bookmarks[currentArchive]` to list bookmarks. We expose helper hooks like `useAppState()` or specific selectors (e.g. `useCurrentArticle()` that returns the current article data) to make it easy for any developer or agent to access needed data.

Global Actions: We define a set of actions (or operations) that mutate the state. Examples: - `openArchive(archiveId)` – sets `currentArchive`, loads archive metadata (maybe triggers loading main page). - `openArticle(articleId)` – sets `currentArticle` (and maybe push to history list). - `toggleBookmark(articleId)` – adds or removes a bookmark for the current archive [58]. - `setSearchResults(results)` – update the searchResults list. - `setTheme(themeName)` – change the active theme, etc. These actions are the *only* way state is changed (if using Redux, they correspond to dispatched actions with reducers; if Zustand, they are functions that update state). This consistency means that any UI component or agent writing a component doesn't create new state variables ad-hoc – they must use the existing actions and state shape [70]. For example, if an agent is implementing the bookmarking UI, we make sure it knows about `toggleBookmark` action (through documentation/prompt), so it calls that rather than attempting something unsupported. The advantage is that all parts of the app speak the same "language" when it comes to state changes, reducing bugs and merge conflicts.

We document these actions in the spec (and code) so they're easily referenceable. In fact, an agent prompt might explicitly say: *"Use the* `openArchive` *action defined in the state module when an archive card is clicked."* This way, the agent won't invent `loadArchive` or some other name, it will use the correct one [59].

State and Agents: Because multiple AI agents might work on different features, having this single state logic prevents them from diverging. One agent might be tasked with implementing search and will see that to show results, they should call `setSearchResults` and maybe navigate to a Search view. Another working on the Reader will use `openArticle`. If both follow the same spec, their code will naturally integrate. We also guard against concurrency issues by possibly having a *single source agent or maintainer* review the combined state definition. If two agents propose changes to the state structure, the lead developer (or a supervising agent) will reconcile them before integration [71]. So far, by clearly pre-defining the state and actions, we've minimized such conflicts.

Platform Distinction Handling: To a developer, EnZIM appears mostly as one codebase. We hide platform differences behind abstraction layers: - We discussed in section 5 how file access or heavy computation are abstracted. Developers simply call high-level functions like `fetchArticleContent()` and get results, without worrying if it came from a Rust backend or JS. This is implemented by runtime checks for environment [44]. - We maintain separate configuration files or modules for platform-specific things (e.g., Tauri IPC handlers in Rust, versus service worker setup for the web). But those are clearly separated in directories like `src-tauri/` for Rust and perhaps `src/web/` for any web-only code. The React front-end stays the same. - Conditional logic in React components: In a few places, we need to render something only on a certain platform. For example, a "Open File" button might show on desktop (to pick a file from disk) whereas on web that might not exist or be replaced by a drag-and-drop area. We handle this with simple environment checks:

```
{isDesktop && <Button onClick={openFileDialog}>Open ZIM File</Button>}
```

where `isDesktop` is determined by checking if the Tauri API is available (or some env var at build time) [72] . We limit these conditions to UI elements and never fork large logic flows if possible. The spec notes these so that agents know to include them when relevant (we'd remind an agent in the prompt if needed: "wrap this in an {isDesktop && ...} block"). - Styling differences: Ideally the design is unified, but on mobile we might use slightly different layout (e.g., a bottom tab navigation vs a sidebar). We achieve this by CSS media queries or small component variations rather than maintaining two separate sets of components. For instance, the LibraryView grid adjusts columns based on screen width (Tailwind responsive classes or our own breakpoints) [18] . Agents are instructed to use responsive design (as per requirements in prompts) so that their output works on both mobile and desktop without separate code.

Development Workflow: From a developer UX standpoint, we ensure: - The app can be run in a browser for rapid development (since it's React). This hits the JS implementation of zim parsing, which is fine for testing moderate archives. - For testing the Rust backend, you run the Tauri app. We keep hot-reload for front-end and a quick Rust compile cycle for backend, to minimize friction. - Writing code once and deploying to all: a developer doesn't need to write a feature 3 times for desktop, web, mobile. They write it once in React/TS. Only if something doesn't work in web (e.g., file system access) do they need to consider an alternative path, but that's usually handled by the core libraries. - Documentation like this spec is kept up-to-date and serves as a guide during development. We treat the spec as a living document (in fact, part of the "canvas" approach) – when a design decision changes, we update it here so all agents and developers can see the new source of truth.

Example – State Coordination in Action: Suppose one agent built the `BookmarksView` to display bookmarks, and another built the logic for adding/removing bookmarks from the Reader. Thanks to the shared state: - The Reader's bookmark button dispatches `toggleBookmark(currentArticleId)` which updates the global `state.bookmarks` list [58] . - The BookmarksView component, which is subscribed to that list, automatically shows the new bookmark in the UI without needing further integration. If the agents followed the spec, the names match and it works. If by chance the second agent created a different action name or state field, our code review or tests would catch it – but so far, by **providing these conventions in the prompt**, we avoid the mismatch in the first place [61] .

Developer Tools: We also plan to integrate some developer conveniences: - A Redux DevTools or Zustand devtools integration for debugging state changes in real-time. - Logging in development builds that clearly logs when an action is called (so if something misfires, we see it). - Unit tests for critical utils (the ZIM parsing logic is covered by tests in its own library; we may add tests for say the search function or state reducers). - Storybook or styleguide for UI components could be considered to let designers (or AI agents with visual context) see components in isolation. This is more of a stretch goal, but since we have multiple style prototypes, a component gallery might be useful.

In essence, the developer (and agent) experience is crafted to be **cohesive**: one state, one set of components, one codebase for all platforms. This reduces cognitive load and errors, and it makes it easier to coordinate an AI-driven development process. Agents can be spun up to tackle different parts without worrying that their part only works on one platform or breaks global state – because the rules are laid out clearly here. Our success metric internally is that we can onboard a new AI agent (or human contributor) and have them productive in short order by reading this spec and following the established patterns [65] .

9. Launch and Build Roadmap (Modular Milestones for Agents)

To deliver EnZIM effectively, we've broken the development into modular milestones. Each milestone corresponds to a set of features or components that can often be built and tested somewhat independently – a perfect fit for assigning to builder agents or focused sprints. The roadmap not only helps timeline the project but also illustrates how the system can be incrementally constructed and integrated.

Milestone 1: Core Reader MVP – *"Offline Reader Foundation"*
Goal: Get the basic application running with a single archive.
- Implement the **ZIM file open/load** flow: the user can select a ZIM file (desktop: via file dialog, web: via file input). Use the ZIM Parsing Engine to open it and display the content of the main page in the Reader view.
- Build the **ReaderView component** to render an article's HTML content in a scrollable view, with proper styling (using our Tailwind design system). Ensure that images and links within the content are handled (images should load via the zimlib, links trigger navigation within the archive).
- Minimal **navigation controls**: at least a Back button (to go to the previous article in history) and perhaps Forward if easy. History tracking can be implemented in a simple way here (stack of visited pages).
- **Performance check:** Loading an article should be reasonably fast (<2 seconds for first page) [45] . Test with a medium-sized ZIM.
Agent tasks: Construct `ReaderView` UI, implement `openArchive` and `openArticle` logic, integrate a simple history stack.
Success Criteria: User can open a ZIM and read articles, clicking links to navigate, and use back button.

Milestone 2: Multi-Archive Library & Basic Search – *"Library and Lookup"*
Goal: Expand to support library management and find content easily.
- Implement the **LibraryView** to list multiple archives [17] . Users can add archives to the library (persist this list). Show archive metadata (title, size, etc.). The UI from the prototypes (grid of cards or list) will be used.
- Support **switching archives**: when one archive is open and user goes back to library to open another, ensure state resets appropriately (currentArchive changes, new Reader loads).
- Add the **Search bar** in the header and a basic search functionality. Initially, focus on title search. As user types, show a dropdown of matching article titles (this can be an auto-complete style list). If more convenient, a dedicated Search view can show results after hitting Enter.
- The search uses the indexing mechanism: for now, maybe load all titles into memory on archive open (if memory permits) or do prefix matching by scanning. Optimize later as needed.
- Implement a **Bookmarks** mechanism (back-end): allow user to bookmark the current article (e.g. star icon in header). Store it and show a basic list of bookmarks (could be a section in Library sidebar or a simple modal list).
- Implement **TTS basic**: a button that triggers the OS/browser TTS for the current article. Doesn't need a lot of UI beyond play/pause, but ensure it can read at least the article title and first paragraph as a proof of concept.
Agent tasks: LibraryView UI, Archive card component, integrate state for archives list; Search bar component and searchResult item component; simple bookmark toggling logic and UI indicator (e.g., star icon fill when bookmarked); wire up a call to Web Speech API or Tauri command for TTS.
Success Criteria: User can maintain a list of archives, open any, search for an article by title, and bookmark articles. These features should work on both desktop and web. The app at this stage should cover the basic use cases of an offline reader (read, navigate, search, bookmark).

Milestone 3: Semantic Mesh & Advanced Navigation – *"Semantic Explorer"*
Goal: Introduce the Semantic Mesh view and other advanced navigation aids.
- Develop the **MeshView component** that renders the semantic graph for a given article. This includes drawing nodes and edges in an overlay or pane [21] [22]. Start with just one-degree relationships (current article and its linked neighbors).
- Hook up the **Semantic Engine** to supply data: when an article loads, get its link targets (could parse the HTML for `<a>` tags). Create node objects for each linked article (title and id). The visual layout can be simple fixed positions initially.
- Make the graph interactive: clicking a neighbor node should navigate to that article (which triggers the Reader to load it and then the graph to update). Ensure this interaction updates history as well.
- Add a UI control (button) to toggle the mesh view on/off easily. For example, in ReaderView's toolbar (maybe the icon of connected nodes we saw in the prototype) [73]. When off, the center pane might revert to showing just the article (if we used a split-pane design).
- Additional nav aids: maybe a **breadcrumb trail** in the article header (e.g., show the archive name or category path if available) [74] to give context, and possibly a **Table of Contents** for long articles (if the ZIM has section anchors). The ToC can be a collapsible sidebar in Reader if time permits.
- Refine search: consider adding full-text search capability if performance allows. Possibly mark this as a stretch for now or ensure title search is very robust.
Agent tasks: Construct `MeshView` UI (possibly using an HTML canvas or SVG for drawing lines; the prototypes used SVG lines [75] and absolutely positioned divs for nodes [14] – we can follow that); implement the logic to extract links from article content (could be done via DOM parser in JS or via the zim index if any); integrate click handlers on nodes to call `openArticle`. Also, agent to implement breadcrumb component or ToC if included.
Success Criteria: While reading, the user can switch to a visual graph of the article's connections and click through related topics. This makes the app a true "knowledge explorer". The feature should feel responsive (generating the graph quickly on article load). Basic layout for ~5-10 nodes should be clear and not cluttered.

Milestone 4: AI Assistant Integration – *"Ask ZIM"*
Goal: Embed the conversational assistant and ensure it can handle useful queries.
- Implement the **Assistant UI panel**. Perhaps as a chat widget in the bottom-right or a full-height sidebar. Develop a simple chat interface: a text input box and a scrollable area for the conversation.
- Integrate a **QA system**: Initially, this could be rule-based – when the user asks something, perform a search in the archive for keywords, then either return the intro of the top article or a message "I found these articles...". If an open-source QA model is available (and can run locally in acceptable time), integrate it for a better answer formulation. If not, use templating: e.g. question "What is X?" -> find article X or search it -> if found, respond with first sentence of that article and "According to [Article]...".
- The assistant should also handle requests like "Summarize this article" (which it can do by taking the text and either using an algorithm like picking first sentence of each section or using a summarizer model if available). Or "Show me related topics" which ties into the mesh (it can basically list what the mesh view shows, or even trigger the mesh view to open).
- Focus on ensuring no internet needed: all answers come from the content we have. If an LLM is used, it should operate without calling external APIs (maybe a distilled model running via WebAssembly or via Rust CPU). We might test integration with a small model (for instance, a 100MB quantized model) to see if response times are okay (~ a few seconds per answer). If not, stick to deterministic methods.
- Voice integration (optional): allow user to click a mic and ask a question via speech (speech-to-text), then answer via TTS. This can leverage OS capabilities; it's an extension of the assistant feature but nice to consider if time permits in this stage.
Agent tasks: Build the ChatUI component; implement a function `answerQuery(query)` in the backend that does the search/summary; possibly another agent to integrate a small ML model (if chosen). Ensure the assistant has access to the search index or can call the existing search function. Possibly an

agent to fine-tune the formatting of answers (including citing article titles or sections).
Success Criteria: User can enter a question about the content (e.g., "When was Albert Einstein born?" while having a Wikipedia ZIM loaded) and get a relevant answer with reference. Or ask for a summary of the current page and get a concise summary. The assistant should handle at least a few types of queries reliably. This transforms the user experience from purely manual browsing to interactive inquiry.

Milestone 5: Polish, Optimization & Cross-Platform Release – *"Production Hardening"*
Goal: Finalize the product with performance tweaks, complete platform support, and prepare for launch.
- **Performance Optimizations:** Profile the app on large ZIM files. Optimize search (perhaps implement a better indexing if needed), ensure memory usage is within limits (e.g., use streaming where possible, limit caching). The Rust backend might get improvements like memory-mapped file access for huge archives [76], and the web might use IndexedDB chunk caching to handle large files without RAM explosion. Ensure semantic graph generation is not too slow (maybe cache link lists for articles once computed).
- **UI/UX Polish:** Revisit the UI against design prototypes and fix any inconsistencies. Ensure themes (light/dark and others) are working – possibly implement the palette of 9 themes (Brutalist, Minimal, Cyberpunk, etc.) as selectable options [77]. Check accessibility: keyboard navigation, screen reader labels, contrast (meeting WCAG guidelines as much as possible) [78]. Add any missing animations or feedback (e.g., loading spinners when opening a big file, etc.).
- **Cross-Platform Testing:** Test the desktop app on Windows, macOS, Linux – fix any file path issues or packaging quirks. Test the web PWA in Chrome, Firefox – ensure the service worker caching works and the app can load a file in each (with user selecting file). For mobile, if we have an Android build via Tauri, test on a physical device: adjust touch targets, maybe incorporate Android's back button handling to tie into app navigation. Address any mobile performance concerns (maybe disable heavy features if device is low on resources).
- **Documentation & Launch Prep:** Create a user guide (how to use features), and developer docs for contributors. Also, finalize licensing checks to ensure all components are MIT or similarly permissive [79] [80]. Prepare distribution: set up CI for building installers for desktop, and publish the web version (maybe on a static site or as a browser extension if applicable).
- **Modular Release of Builder Agents:** As a meta-step, document how the builder agent workflow succeeded and prepare prompts or templates for future maintenance tasks. This includes writing down any adjustments made to agent outputs and lessons learned (this might be an internal milestone artifact).
Agent tasks: Many of these are smaller polish tasks that could be done by specialized agents or scripts – e.g., an agent to optimize an algorithm, one to enforce a coding style pass, etc. However, final tweaking will be overseen by human developers (especially for UX feel). Possibly use an agent to generate additional unit tests for critical sections now that the code is mostly done, to ensure reliability.
Success Criteria: The app is **feature-complete and stable**. It can be released as v1.0 across platforms. All primary use cases (reading, searching, exploring graph, Q&A) work smoothly. Performance on a typical device is acceptable (no crashes with a 4GB Wikipedia ZIM, search within a second, graph toggle without lag, etc.). We have confidence to launch to real users (perhaps starting with a pilot group like some offline education initiative to gather feedback).

This roadmap not only guides development but also illustrates the modular nature of the project – each milestone delivers a slice of functionality that can be developed somewhat in parallel by different agents/teams and then integrated. For example, Milestone 3's Semantic Mesh could be developed while Milestone 2's search is being finalized, as long as the interfaces (e.g., how to fetch links) are agreed. The builder agent approach thrives in this environment: clear boundaries and specifications for each module. By following this plan, we ensure steady progress and a coherent final product.

(Reference: The development checklist from the AnZimmerman project indicates earlier phases like core library implementation and annotation features were completed in prior weeks [81], giving us a head start on EnZIM's foundation.)

10. Integration with Design & External Generators (Figma, HTML-to-Design, etc.)

EnZIM's development process embraces a **design-engineering loop** that leverages external tools for UI design and code generation. The goal is to ensure our UI implementation remains faithful to design intentions and to speed up the workflow by using automation where possible. This section describes how we integrate with design tools like Figma and Excalidraw, and how builder agents could utilize those in the workflow.

Export to Design Tools: We provide mechanisms to export the current state of our UI or components into formats that designers can work with: - *Figma Export:* We plan to use the html.to.design tool or similar, which can convert live web layouts into Figma frames [82]. EnZIM could include a developer-only feature (or a CLI script) that takes a snapshot of a rendered view (the HTML/CSS structure of, say, the ReaderView or an entire screen) and feeds it to html.to.design. This generates a Figma document that mirrors our implementation. Designers can then open that in Figma to make refinements or annotations. By exporting actual code output, we ensure pixel-perfect fidelity – the spacing, colors, and typography in Figma will match the code [83]. This helps catch any discrepancies and allows designers to propose adjustments in a familiar environment. - *Excalidraw Integration:* Excalidraw is an open-source sketching tool great for diagrams and wireframes. For features like the **semantic mesh**, which is essentially a graph visualization, we can export the nodes and connections as an Excalidraw diagram [84]. This means a user (or designer) could click "Export Graph" and get a .excalidraw JSON or an SVG of the current semantic network. They could then tweak the layout or style in Excalidraw (for a presentation, for example). Conversely, if a designer sketches a new UI idea in Excalidraw (say a new layout for the Library screen or a concept for a 3D version of the graph), we can import that as a reference. While not direct code, having a hand-drawn concept in a sharable format (JSON/SVG) that we keep in the repo can inform builder agents or developers. - *How Export Works:* These exports might be triggered via a menu in a development mode or via command-line. For Figma, since direct API requires auth, the simpler path is an intermediate file: e.g., the developer runs a script that generates an HTML representation or uses the html.to.design CLI, producing a `.figma.json` or similar, which is then loaded into Figma manually [85]. For Excalidraw, since it's open, we could generate the file directly in the app (e.g., compose JSON with our node positions). This could even be an automatic snapshot saved whenever we do a release, to document the state of the UI.

Import / Codegen from Design: The reverse – using designs to inform code – is also part of our approach: - *Figma to Code:* If designers make changes in Figma (after an export or on their own), we leverage tools or plugins that translate those into code suggestions. For instance, Figma has an API to fetch the design structure. A builder agent can be given the Figma data (or a simplified description of it) and instructed to generate JSX/CSS that matches [86]. We maintain a mapping between design components and code components – e.g., a Figma component named "ZimCard" corresponds to our `<ZimCard>` React component [87]. This naming parity is deliberate; it means the agent can more easily align the design with the existing code structure. - In practice, we might not fully automate code generation from Figma, but even partial help is useful. For example, if a designer adjusts spacing and colors in Figma, an agent could read those values and update our Tailwind config or CSS variables accordingly, since our design tokens are shared [88]. - *Layout Stubs for Generation:* Sometimes we conceptualize a new UI in plain text or simple diagrams. We maintain **layout stubs** – textual outlines of screens (as shown in some sections of this spec). For example:

```
LibraryView:
  Header: Title + AddButton
  Grid:
    ZimCard * N (for each archive)
```

This can be given to a builder agent as a scaffold to generate actual JSX with appropriate components [89]. We've effectively done this in our prompt examples. Such stubs can also be derived from design wireframes. It ensures an agent understands the hierarchy of elements to create, even without a pixel-perfect design image. - *AI-assisted Design Refinement:* Our builder agents can also be prompted to adjust code to match design guidelines. For instance, "Make the spacing in LibraryView match the Figma spec: 24px margins, 16px gutter" – since our UI uses utility classes, an agent can straightforwardly change classes (like `p-4` to `p-6`) as needed [90]. This is more efficient when the spec (like a Figma measurement) is provided.

Maintaining Design-Code Consistency: Key to this process is ensuring that changes in design reflect in code and vice versa: - We centralize style definitions (colors, fonts, spacing scales) in one place (Tailwind config and CSS custom properties). If a color is updated in Figma style guide, we update the Tailwind palette to match, so the next export from code will already reflect that color, and any agent generating new UI will use the updated token [88]. - We version control design artifacts. For example, if we export an Excalidraw of the semantic mesh, we commit it in a `/design` folder. If a designer modifies it, we can see diffs. Similarly, if we have a Figma file link, we note the version. This allows builder agents (in the future) to be aware of design changes by checking these artifacts. - Each section of this spec is tagged or labeled in a way that both developers and designers can discuss it. For instance, if the designer says "The SearchResultsList should have 8px padding according to design", we can update the spec and that flows into prompts for agents. Essentially the spec and the Figma are kept in sync by diligent updates.

External Generators and Future Integration: Beyond Figma and Excalidraw: - We keep an eye on tools that convert HTML to design or vice versa. The html.to.design we use is one, but also there are AI-driven tools that can generate React code from screenshots or design files. Because we have a set architecture and naming scheme, if we ever plug such a tool in, we'd train it on our conventions to get usable output. - The **builder agent** itself is a kind of generator. We consider the possibility of meta-prompts: using an agent to analyze a Figma file and produce prompt templates for another agent. This hasn't been fully explored yet, but conceptually: - Agent A: "Read this Figma JSON, output a structured description of the UI." - Agent B: "Take this structured description and generate code." - By splitting tasks, we might handle complex inputs better. This is speculative but aligns with our multi-agent orchestration philosophy.

Use Cases Example: Suppose we want to redesign the Bookmarks list UI. A designer makes a new layout in Figma. We can: 1. Export the current BookmarksView code to Figma (as a starting point). 2. Designer adjusts it (maybe adds icons, changes grouping). 3. Use the Figma API to get the updated design data. 4. Prompt a builder agent: *"Here is the JSON of a Figma frame for BookmarksView. Update our BookmarksView component's JSX/CSS to match this design, using existing components as needed (like use the* `<ZimCard>` *for each bookmark if appropriate)."* 5. The agent returns updated code. Developers review and tweak. 6. We then perhaps export back to Figma to verify visually.

While not fully automated, this tight loop significantly **reduces manual translation work**. It also means our design and code are never too far apart – avoiding the common problem of the "design drift" where the implemented app slowly diverges from the Figma mockups. Here, because we continually exchange info between the two mediums, we catch drifts quickly and correct them [91].

In conclusion, EnZIM's integration with external design and generation tools is about bridging the gap between **creative design** and **structured development**. By exporting our real UI into design tools, we invite visual refinement and ensure accuracy. By importing designs back into our development process (with AI assistance), we accelerate implementation and maintain fidelity. This synergy supports our collaborative builder-agent workflow, where designers, developers, and AI all participate in creating a polished product [92]. EnZIM is not just an app but a showcase of how modern development can tightly weave together specification, design, and automated building for a faster, smarter workflow.

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [15] [16] [17] [18] [25] [33] [35] [36] [38] [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] [69] [70] [71] [72] [76] [78] [79] [80] [82] [83] [84] [85] [86] [87] [88] [89] [90] [91] [92] AnZimmerman ZIM Reader – PRD & Architecture Document (1).txt
file://file-8gCqWmBmr49VQ3J7eubBut

[12] [77] [81] CHECKLIST.md
https://github.com/rebots-online/EnZIM/blob/2a9721ed046b4162fe2192bf9b84ace01b005b5d/CHECKLIST.md

[13] [14] [19] [20] [21] [22] [23] [28] [29] [34] [73] [74] [75] synaptic_cartography_veil_1.html
file://file-HHEuB6bpRf1w3mNp98gjUh

[24] [30] [31] [32] prismatic_swiss_utility_0.html
file://file-Qm7gxxxHEaz6sVqPc24h2U

[26] [27] brutalist_archive_monolith_0.html
file://file_00000000062871f5a36413ba077b2928

[37] [39] README.md
https://github.com/rebots-online/EnZIM/blob/2a9721ed046b4162fe2192bf9b84ace01b005b5d/README.md

[67] [68] ARCHITECTURE.md
https://github.com/rebots-online/EnZIM/blob/2a9721ed046b4162fe2192bf9b84ace01b005b5d/ARCHITECTURE.md