ChatGPT

Archon: Comprehensive Product Specification

Introduction and Overview

Archon is a cross-platform desktop application for **managing complex equipment configurations** across various domains such as scientific labs, test sites, museums, game development setups, farms, and more. It provides a structured way to record the hierarchy of components making up equipment setups and track changes to those configurations over time. In other words, Archon is a configuration management tool that **maintains a consistent record of a physical asset's configuration and properties through any changes it undergoes** [1]. By leveraging a modern technology stack – **Svelte 5** for a reactive front-end, **Tailwind CSS 4** for UI styling, **TypeScript** for type-safe code, and **Vite** for fast bundling – Archon delivers a snappy and intuitive user experience. The application is packaged as a native desktop program using **Wails (v3)**, which wraps a Go backend runtime with a WebKit-based frontend, allowing Archon to be distributed as a single small binary on each platform [2] [3]. This modern stack (similar to other recent cross-platform apps [4]) ensures Archon is efficient, responsive, and runs seamlessly on **Windows, Linux, and macOS**.

Key Goals of Archon:

- Enable input and visualization of **hierarchical equipment configurations** (with nested sub-components).
- Allow attaching **arbitrary key–value properties** (metadata) to any item in the hierarchy for rich documentation.
- Support full CRUD operations (create, edit, delete) on components and properties in the hierarchy.
- Provide robust **version control via Git**: capture **snapshots** of configurations as Git commits with tags, view history, and push/share configurations through remote repositories (e.g. GitHub).
- Offer powerful **comparison (diff) tools** to visualize differences between any two configuration snapshots.
- Use **JSON** as the canonical format to store all configuration data, ensuring data is human-readable and easily integrable.
- Support extensibility through a **plugin system for importing** configurations from external formats or systems.
- Enable **exporting data** in various formats (JSON, CSV, PDF) and generate **visual reports** to communicate the configurations clearly.
- Ensure a **user-friendly UI/UX**: intuitive navigation of the hierarchy (tree views, column navigation), a project dashboard for snapshots, and easy editing of properties.
- Include **authentication** (local or Git-based) and identity tracking to attribute changes to specific users.
- Adhere to **best practices** in software design: modular components, clean separation of concerns (UI vs logic vs data), error handling, and extensibility.

The rest of this specification details Archon's features and design, organized into core functionality, UI/UX design, authentication, cross-platform considerations, user personas, and implementation best practices.

Core Features

Hierarchical Configuration Management

Archon's primary function is to let users **input and explore hierarchical equipment configurations**. Users can represent an overall system (for example, a lab bench or an exhibit) as a root node, with nested child nodes for each sub-component or equipment piece. This hierarchy can be as deep as needed to mirror real-world assemblies – for instance, a "Microscope" node might have children for "Optical Head", "Stage", "Camera", etc., each of which could have further sub-parts. The UI will display this structure in an expandable tree or columnar view, allowing users to drill down into sub-components easily.

Each item in the hierarchy can have an associated **name/label** and an **optional description**, but more powerfully, Archon allows attaching **arbitrary key–value properties** to any item. These properties let users record metadata like serial numbers, manufacturer, firmware version, location, calibration data, or any custom attributes relevant to their domain. Properties are stored as flexible key–value pairs (string keys and values by default, with potential support for other data types like numbers or dates in the future). The application does not impose a fixed schema – users are free to add any keys that make sense for their equipment. For example, a node representing a sensor could have properties like `{ "Manufacturer": "Acme Corp", "Range": "0-5V", "CalibrationDate": "2025-05-01" }`. This flexibility ensures **Archon can adapt to many domains and use cases** without requiring code changes.

Basic **CRUD operations** are supported on the hierarchy and properties:

- Users can **create** new components (nodes) at any level. For instance, adding a new device to an existing setup or building out a new configuration from scratch. New nodes can be added via an "Add Component" action, which could be available as a toolbar button or a context menu option in the hierarchy view.
- Users can **edit** components to rename them or modify their properties. This includes editing property key–value pairs (changing values, adding new properties, or removing properties). The UI will provide an **editor panel or modal** (see UI/UX section) that displays all details of a selected item and allows in-place editing.
- Users can **delete** components from the hierarchy. Archon will handle removals carefully: deleting a node will also remove its child subtree, after confirming the action with the user (to prevent accidental large deletions). Deletions will be tracked as part of the version history (so a snapshot before deletion still contains the removed items, and the diff will show the removal).
- Users can reorganize the hierarchy, for example by dragging and dropping nodes to a different parent (if applicable) or changing their order among siblings. This reorganization is effectively a combination of delete + create operations under the hood (moving a node from one parent to another).

All changes to the hierarchy and properties are immediately reflected in the application's state and are considered part of the **current "working" configuration**. Archon uses **JSON** as the underlying storage format for the hierarchy. The entire configuration can be serialized to a JSON document (or a set of JSON files). For simplicity, Archon may use a single JSON file (e.g. `archon_config.json`) stored in the project's Git repository to represent the root node and its nested structure. Each node can be an object with a unique ID, name, properties, and an array of children. For example, a simplified JSON representation might look like:

```json
{
  "id": "proj-1234",
  "name": "Lab Bench 1",
  "properties": {
    "Location": "Building 1, Room 202",
    "Owner": "Alice",
    "LastAudit": "2025-05-01"
  },
  "children": [
    {
      "id": "dev-100",
      "name": "Microscope",
      "properties": {
        "Manufacturer": "Acme Co.",
        "Model": "X100",
        "SerialNumber": "SN12345"
      },
      "children": [
        {
          "id": "dev-101",
          "name": "Objective Lens",
          "properties": {
            "Magnification": "40x",
            "NA": "0.65"
          },
          "children": []
        },
        {
          "id": "dev-102",
          "name": "Camera",
          "properties": {
            "Resolution": "5MP",
            "Interface": "USB3"
          },
          "children": []
        }
      ]
    },
    {
      "id": "dev-200",
      "name": "Pump Controller",
      "properties": {
        "IP": "192.168.0.50",
        "Firmware": "v2.3"
      },
      "children": []
    }
  ]
}
```

(Example: JSON data structure of a simple lab bench configuration. Each component has an `id`, `name`, optional `properties`, and `children`.)

Storing configurations in JSON has multiple benefits: it's human-readable (users could manually inspect or even edit the file if needed), it's easy to version with Git (diffs can be shown per line or parsed structurally), and it's widely compatible with other software (for import/export).

Archon will ensure that **all hierarchy and property data is persistently saved in JSON** format on disk (within the project's Git repository directory). Auto-save mechanisms can be employed so that as users make edits, the JSON is updated (and can be staged for commit). A user may also trigger a manual "Save" which simply ensures the in-memory state is flushed to disk. However, given Archon's integration with Git, saving might often coincide with making a commit (see Snapshots below).

Snapshots and Version Control (Git Integration)

One of Archon's standout features is its tight integration with **Git for version control**. Every significant state of a configuration can be saved as a **snapshot**, which under the hood corresponds to a Git commit in the project's repository. Users can create a snapshot at any time to record the current configuration state, providing a tag or label (and optional description) for that snapshot. This is analogous to recording a milestone or baseline of the equipment setup – for example, "Before reconfiguration for Experiment 42" or "Exhibit setup as of Opening Day".

When the user takes a snapshot, Archon will prompt for a **snapshot name (tag)** and an optional description or commit message. The application will then:

- Ensure all current changes are saved to the JSON files.
- Create a **Git commit** capturing those changes (the commit includes the updated JSON representing the configuration).
- Apply a **Git tag** with the provided snapshot name to that commit for easy reference (e.g., tag "v1.0" or "Baseline_Jan2025"). The tag helps treat certain commits as named configuration baselines.

Multiple snapshots (commits) form a **history of changes** for the project. Archon provides a **snapshot history view** (or commit log view) where users can see a list or timeline of all snapshots, including their tags, timestamps, author, and description. Each entry in the history corresponds to a Git commit. The user can click on a snapshot to view the configuration at that point in time (the app can load the JSON from that commit into a read-only view or a separate window, for example).

Because Archon uses Git, it inherently records **who made each change and when** (the commit author and timestamp). The user's identity (configured in Archon's settings or via Git credentials) will be used for commits, enabling **audit trails** of modifications. (See the Authentication section for details on user identity.)

Users can also integrate with remote Git repositories. Archon will allow configuring a **Git remote** (for example, a GitHub or GitLab repository URL). Once set up, users can **push** their commits/snapshots to the remote – this is useful for backup, for sharing configurations with colleagues, or for collaborating across multiple machines or users. Archon will provide UI controls for common Git operations: **Commit**, **Pull**, **Push**, **Clone**:

- **Commit:** As described, creating a snapshot in the UI will handle committing changes. Archon can also auto-commit behind the scenes (e.g., committing every change individually) but snapshotting explicitly allows grouping changes with a meaningful message. Uncommitted changes could be indicated in the UI (similar to a working directory state).
- **Push:** A button to push local commits to the remote repository. If credentials are needed, Archon will prompt for them or use stored credentials (possibly leveraging Git credential helpers or an OAuth token if using a service like GitHub).
- **Pull/Fetch:** If the remote has updates (e.g., if multiple users are pushing to the same repo or the user did work on another machine), Archon can pull changes. This introduces complexity of merge conflicts if two edits happened concurrently; handling merges is advanced, but Archon could at least detect and warn if the local JSON has diverged and attempt a basic merge or require user intervention.
- **Clone:** On the app's welcome screen or menu, users can clone an existing Archon project by providing a Git URL. Archon will then do a `git clone` under the hood and open the project, allowing quick start from an existing configuration repository.

Perhaps the most powerful capability this unlocks is the ability to **compare (diff) any two snapshots**. Archon will include a **Diff View** where a user selects two snapshots from the history (or a snapshot and the current working state) and the app will display the differences side-by-side or in a unified diff format. Since the data is JSON, Archon can leverage text diffs as well as structured diffs:

- **Textual Diff:** The raw git diff of the JSON file(s) could be shown, highlighting added, removed, and changed lines. This would show, for example, if a property value changed from X to Y, or if a whole component block was added or removed.
- **Semantic Diff:** Archon can also parse the JSON and present a higher-level comparison. For example, it can list which components were added or removed, which properties were changed (and their old vs new values), etc. This is more user-friendly for non-developers, as it avoids JSON syntax and focuses on the meaning of changes. A possible UI is a tree view of changes: e.g., "Microscope → Camera: property 'Resolution' changed from 5MP to 10MP" or "Added component 'Thermostat' under 'Pump Controller'".
- **Visual Diff:** For future enhancement, a side-by-side tree comparison could be displayed, or color-coded highlights in the hierarchy view to show differences.

The diff feature draws inspiration from similar configuration management tools – for instance, the Aligni system provides a comparison tool to visualize differences between two equipment configurations [5]. Archon will similarly allow users to pick any two snapshots and see exactly what changed between them, even if the snapshots are far apart in time. Every property change, component addition, removal, or move can be detected and shown. This is invaluable for users like engineers or curators who need to **track changes over time for troubleshooting, audits, or documentation**.

To illustrate: suppose a lab technician creates snapshots "Baseline" and later "AfterUpgrade" – the diff might show that under "Microscope > Camera" the firmware property changed, and a new child component "Cooling System" was added. Such a report could be exported or printed for record-keeping.

All version control operations will be handled behind the scenes via a Git library or Git CLI. From the user's perspective, they interact with straightforward concepts: *Project* (repository), *Snapshot* (commit/tag), *History* (log), *Sync* (push/pull). This integration of Git brings powerful versioning capabilities, but Archon will aim to simplify it for non-developers by using clear terminology (e.g., say "Snapshot" in the UI instead of "Commit", though in documentation we equate them). Advanced users, on the other hand, can rest assured that under the hood it's a normal Git repository – they could even manually clone or manipulate it outside Archon if needed.

Data Import via Plugin System

Archon is designed to be extensible. Many organizations or domains have existing sources of configuration data – for example, a lab might have an inventory list in CSV or Excel, a museum might have a database of exhibit items, or a farm might use an IoT platform. To facilitate getting data *into* Archon without painstaking manual re-entry, Archon will support a **plugin system for importing configurations**.

Plugin Architecture: A plugin in Archon is essentially a module that can **read some external data source or format and convert it into Archon's JSON structure** (the hierarchy with properties). The plugin system will allow third-party developers or advanced users to add new importers without modifying Archon's core code. Key design points of the plugin system:

- Plugins can be distributed as files or packages that Archon can load at runtime. For example, a plugin could be a JavaScript or TypeScript file (running within the Svelte/Node context) or perhaps a Go plugin loaded by the backend. Because Wails uses a Go backend, one approach is to allow Go-based plugins; however, a simpler approach is to allow JavaScript plugins running in the front-end context for portability.
- Archon will define a **Plugin API** (an interface) that plugins must implement. For instance, an import plugin might need to provide a function like `parse(inputData) -> ArchonConfigJSON`. Additionally, metadata about the plugin (name, version, author, what formats it supports, etc.) will be defined.
- Plugins could support various input **formats or sources**: e.g., CSV, XML, Excel, specialized JSON formats from other software, or even APIs. A plugin might prompt the user for a file to import or credentials to an API to fetch data.
- **Built-in Plugins:** Archon will likely ship with a few default import options implemented as plugins (or built-in modules using the same interface). For example, a **CSV Import plugin** can map CSV columns to Archon node names and properties, or an **Excel Import plugin** for structured spreadsheets. Another useful one could be **Import from existing Git repo** (if the user has an older config in a Git repository or some standard).
- The **Plugin Manager UI** will allow users to enable/disable plugins and trigger import actions. For example, an "Import Data" menu might list available importer plugins ("Import from CSV", "Import from LabEquipmentDB"...).
- When an import is executed, the plugin processes the input and returns data in Archon's internal JSON format, which Archon then merges into the current project (or creates a new project from it). For safety, Archon might do imports into a new branch or a draft state so the user can review before committing the imported configuration.

This plugin approach ensures **extensibility** – as new formats or integration needs arise, plugins can be written to handle them without changing Archon itself. It encourages community contributions (e.g., an

Indie game developer might write a plugin to import Unity scene hierarchies; a researcher might write one to pull configurations from a laboratory information system).

Security considerations: Running third-party code as plugins can be risky. Archon will isolate plugins appropriately. If using JavaScript plugins, they can be sandboxed in the front-end (with limited access, only allowed to produce data for import). If using Go plugins on the backend, those would be compiled code – likely we will stick to front-end scripting for simplicity. The plugin API will focus on data conversion tasks and not allow arbitrary destructive operations on the user's system (to maintain trust).

Data Export and Reporting

In addition to capturing and importing data, Archon will allow users to **export** their configuration data in various formats and generate **readable reports** for sharing with stakeholders who may not use the Archon app.

Export Formats:

- **JSON Export:** Since JSON is the native format, exporting to JSON is straightforward – essentially providing the raw data file. This is useful for interoperability (e.g., feeding the data to another system or for developers who want to use the config in code). The user can export the current state or a specific snapshot as a JSON file. By default, the entire project JSON (the same that is saved in Git) can be downloaded. If needed, Archon could also support splitting the JSON (e.g., one file per top-level component) for easier reading.
- **CSV Export:** For users who prefer spreadsheets (common in inventory management), Archon can export tabular data. One approach is to flatten the hierarchy into a list of components (with columns for the hierarchical path, maybe parent IDs, etc.) and their properties. Alternatively, multiple CSV files could represent different levels. However, a simple CSV of all components with columns like "Path, Property1, Property2, ..." might be easiest. The export module will likely let the user choose which properties to include as columns. This is especially handy for analysis in Excel or Google Sheets.
- **PDF Report:** Archon will support generating a human-friendly PDF report of a configuration. This report would include the hierarchy (possibly as an indented list or tree diagram), and the properties of each item. It can also include metadata like the project name, snapshot name/date, and perhaps the user who generated the report. The PDF could be styled to be clean and suitable for printing or emailing. For example, a museum curator might generate a PDF report of an exhibit's setup to include in their documentation, or a test engineer might include a PDF of a rig configuration in a test report.
- **Visual Diagrams:** A potential export/visualization feature is to generate a diagram (for example, a tree graph or network graph) of the configuration. Archon could integrate with a library to produce an **organizational chart-style** diagram where each node is a box containing the component name (and maybe key properties), connected to its children. This could then be exported as an image or included in the PDF. While not explicitly requested, this kind of visual report would greatly help users communicate the structure of their setups to others.

The user interface will provide an **Export dialog** where the user chooses the format (JSON, CSV, PDF) and any options (e.g., for CSV which properties/columns to include, for PDF maybe a choice of including certain sections or level of detail). After generating the export, Archon will allow the user to save the file to a location of their choice.

Because Archon’s data is versioned, users can also export historical snapshots. For example, they could generate a PDF report of an older snapshot to see how things looked at that time. This ties in with the Git history: the user would simply check out or select the snapshot in Archon, and then export, ensuring the exported data corresponds to that point in time.

Visual and Analytical Reports: Beyond static exports, Archon might include on-screen reports. For instance, a **summary statistics** report (how many items of each type, or a breakdown by some property), or an on-screen interactive graph of the configuration. These can be considered future enhancements, but the architecture will not preclude them. The plugin system could even be used to generate custom reports (e.g., a plugin that generates a maintenance schedule report from the data).

In summary, Archon’s import/export capabilities ensure that data does not live in a silo. Users can import existing information to jump-start their projects, and export out to standard formats for communication, analysis, or backup. This makes Archon not just a tool for internal use, but one that fits into broader workflows.

UI/UX Design and Interaction Suggestions

Archon’s user interface will be designed for clarity and ease of use, even when handling complex, deeply nested data. The UI will follow modern, minimalistic design principles, making use of **Tailwind CSS 4** for a clean look, and styling that can adapt to each OS (via Wails) for a **native feel**. Below are key UI/UX elements and how they will function:

Hierarchy Navigation and Browsing

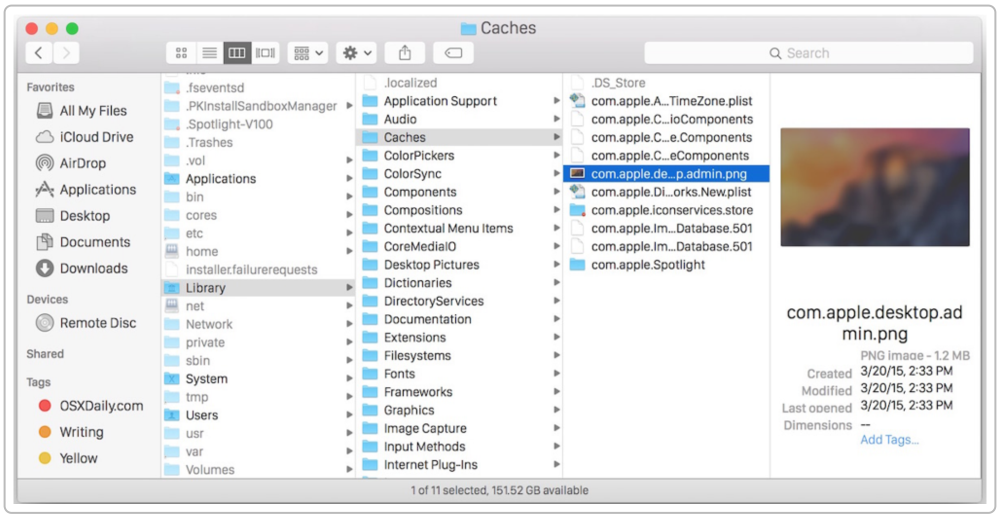

Figure: Example of a multi-column hierarchy navigation (macOS Finder’s column view). Archon may adopt a similar UI to allow users to navigate through nested configuration levels intuitively.

To represent and navigate the equipment hierarchy, Archon will offer either a **tree view** or a **macOS Finder-style column view**. The tree view is a familiar paradigm: a collapsible outline on the side showing nested items (with indentations or arrows to expand/collapse). This is straightforward and allows seeing multiple nesting levels at once. Alternatively (or additionally), the **column view** provides a series of side-by-side columns; selecting an item in one column reveals its children in the next column to the right, and so on. This approach (inspired by Finder, shown above) is very effective for deep hierarchies because it allows the user to trace a path and always see context at each level. Archon could implement a hybrid: a sidebar tree for quick jumping, and a main area with a columnar navigator for detailed browsing.

Single-Project Interface: Archon will use a **"one project at a time"** model. That is, the main window will display the hierarchy of the currently open project (which corresponds to a single Git repository). This avoids UI complexity of managing multiple open projects simultaneously. Users can switch projects by using "Open Project" (which closes the current one and opens another, possibly in a new window). This model maps cleanly to Git repositories and reduces confusion – it's similar to how many code editors open one folder/repo at a time.

Given this single-project focus, the hierarchy view will typically occupy a significant portion of the window. Users will have controls to add or remove components from here (plus keyboard navigation for power users: e.g., arrow keys to move through tree, Del key to delete an item, etc.). Right-click context menus on items will provide quick actions like *Add child component*, *Rename*, *Delete*, etc., in addition to the main toolbar or menus.

Searching & Filtering: To assist with large hierarchies, Archon will include a search bar or filter. Users can type a keyword and the hierarchy view will filter to show matching items (and their parent context). This is useful if, for example, a lab setup has hundreds of items and the user wants to quickly locate "Pump Controller" by name or by a property value.

The UI will highlight the currently selected component and display its details in an adjacent panel (or a modal when editing, as described next). We will ensure that selection and expansion states are preserved as the user navigates, and possibly allow opening multiple columns or panels for different branches if needed (advanced).

Snapshot Management and Version History (Dashboard)

In addition to the main hierarchy view, Archon will feature a **Dashboard or sidebar for snapshot history and diffs**. This could be implemented as a secondary view or a modal:

- A **History Panel** could be a sidebar that lists snapshots (commits) in reverse chronological order. Each entry shows the snapshot tag (if any), commit message or description, author, and date. Selecting an entry could allow the user to see the configuration at that snapshot (e.g., the hierarchy view could switch to a read-only mode showing the historical state). There may also be a button to "Restore" or create a new branch from a snapshot (though by default, Archon operates in a linear history, advanced Git usage like branching might be hidden).
- The **Dashboard view** might be a dedicated screen (perhaps shown on application start or accessible via a top navigation) that provides an overview of the project: recent snapshots, a prompt to take a new snapshot, and possibly shortcuts to import/export. It might also show project metadata (project name, Git remote URL, last sync status, etc.). This dashboard acts as a home for project-level actions, whereas the main view is for component-level actions.

For performing **diff comparisons**, the UI might allow multi-selecting two snapshots in the history list and then clicking "Compare" (or a specific diff tool UI). Alternatively, an explicit "Compare Snapshots" feature can let the user pick “Snapshot A” and “Snapshot B” from dropdowns and then display the differences. The diff results could appear in a modal or a new tab/screen. For a polished experience, Archon might present diffs in a side-by-side tree view or a two-column text diff (with additions in green and deletions in red, etc.). Given that differences might be complex (especially if entire subtrees moved or changed), a clear textual summary will be provided as well: e.g., a list of changes such as “Added 3 components, Removed 1 component, Modified 5 properties” could precede the detailed diff.

Tagging and Notes: In the snapshot history, tagged snapshots (like named releases or important milestones) will be visually emphasized. Users can click to edit the tag or description of a snapshot (perhaps only for the latest commit or any commit – we have to consider Git’s immutability; likely allow editing the tag name or adding an annotation, but not changing historical commit messages easily unless we allow amend – probably keep it simple). Archon might also let users add a **note or annotation** to a snapshot aside from the commit message, for internal tracking (this could be stored in a separate file or Git notes).

Overall, the snapshot dashboard and history features will make Archon feel somewhat like a specialized version control client, but tailored to configuration data. This gives users confidence to experiment (knowing they can revert to an earlier snapshot) and to collaborate (knowing every change is logged).

Editing Components and Properties

When the user wants to add or modify information, Archon will provide **simple, focused dialogs or panels for editing**. Following a principle of keeping the main interface uncluttered, detailed editing will often happen in context-specific pop-ups:

- **Add Component Dialog:** When adding a new component, Archon can pop up a dialog or slide-out drawer asking for the new component’s name and initial properties. This could include choosing a type/template if we support templates (not required, but maybe future extensibility – e.g., a dropdown to select “Microscope” which pre-fills some properties). At minimum, the user enters the name, and optionally can add a set of key-value properties in this step or later. Once created, the new node appears in the hierarchy.
- **Properties Editor:** Selecting a component will show its properties, likely in a panel on the right side (in a two-pane layout: left side hierarchy, right side details). The properties could be displayed in a table or list form: each property key with its value in an editable field. Perhaps an “Edit” button turns the fields editable (or they’re always editable if the user has write mode on). Users can add new properties via an “Add property” button (which creates a new blank key/value row), or remove properties via a delete icon next to each. For complex data types, we keep it simple (treat values as text for now).
- **Inline vs Modal:** Simpler edits (like renaming a component) might be done inline – e.g., clicking on a component name could allow editing it in place (with validations for uniqueness if needed). However, property editing is probably easier in a dedicated area. If space permits, having a details panel always visible is nice; if the UI is constrained, we could have an “Edit Properties” button that opens a modal window listing the properties.

Using **modal dialogs or drawers** ensures that the user’s focus is on the editing task, and it prevents accidental background changes. For example, if a user opens the “Edit Component” drawer, they can change multiple fields and then hit “Save” or “Apply” to commit those changes (which then update the JSON and mark the project dirty for snapshot). If they cancel, changes are discarded. This pattern matches many native apps and will feel familiar.

Validation and User Guidance: The UI will validate inputs where possible. For instance, it will prevent creating two sibling components with the exact same name if that's something to avoid, or warn if deleting a component will delete its children. If a property key is extremely long or contains unusual characters, maybe warn. However, since keys/values are arbitrary, we mostly ensure the JSON stays valid. Real-time feedback (like highlighting fields in error) will be provided.

Multi-Select and Bulk Operations: Initially, Archon will focus on single-item editing, but we might consider allowing multi-selection of components for bulk deletion or bulk property editing (e.g., selecting 5 sensors and setting the same property on all). This can be a future improvement. The UI framework (Svelte) with Tailwind can handle multi-select lists and context menus for bulk actions.

Look and Feel (Using Tailwind & Native Conventions)

The aesthetic of Archon will be clean and utilitarian, in line with tools like VS Code or modern IDEs but simpler. We use **Tailwind CSS 4** to rapidly style components with a consistent design system. Tailwind 4 brings performance improvements and modern CSS features (like container queries and new utility classes), which we will leverage for responsive design (so the app works on various window sizes, high-DPI displays, etc.) [6] [7].

Some UI/UX considerations for look and feel:

- **Theming:** Archon will support both light and dark themes, to match OS preferences. Tailwind makes it straightforward to apply a dark theme palette. Wails can query the OS theme, so Archon's default theme can follow the system (with user override available).
- **Native Controls and Feedback:** Although the UI is web-based, Wails allows using native window frames and dialogs. Archon will use native file dialogs for opening/cloning projects and choosing export file locations, giving the user a familiar experience on each OS. Menu bars and context menus can also use native menus via Wails. For example, on macOS, Archon will have a proper menu in the menu bar (with About, Preferences, Quit, etc.), and on Windows/Linux it might use a custom menu or the standard window menu approach.
- **Icons and Graphics:** A set of intuitive icons will be used for actions (e.g., + for add, 🗑 for delete, ✏ for edit, commit/history icons, diff icons, etc.). We might use an icon library (like Heroicons, which pairs well with Tailwind) to maintain a consistent style. Icons help users quickly identify actions.
- **Layout:** The main layout likely consists of a sidebar (for navigation or snapshots) and a main content area (for hierarchy or details). We will use Tailwind's flex and grid utilities to create resizable panes. Users could be allowed to resize panels (e.g., drag to widen the sidebar).
- **Responsive Design:** While Archon is a desktop app, it might be used on various screen sizes including potentially tablets via frameworks like Wails (though Wails is desktop-focused). Regardless, the UI will adapt: on narrower windows, the sidebar might collapse into a tabbed interface or hide and overlay when needed.
- **Performance:** Svelte 5, being a compiler-based reactive framework, produces highly efficient DOM updates. Combined with Tailwind's atomic CSS (which is very fast to apply), the UI will remain responsive even for large data sets. We will use Svelte's reactivity to ensure only minimal updates

occur when data changes, and virtualization techniques if the hierarchy is extremely large (to render only visible parts).
- **Native-like Behavior:** Small touches make an app feel native: e.g., drag-and-drop support in the hierarchy (with appropriate cursor changes), keyboard shortcuts (like Ctrl+S to save/snapshot, Ctrl+Z for undo if we implement an undo for uncommitted changes, etc.), tooltips on buttons, and using OS-specific conventions (e.g., on macOS use Command key for shortcuts vs Ctrl on Windows/Linux). Wails gives low-level access to the OS (written in Go), so features like opening URLs in the default browser, integrating with system tray or notifications, etc., are possible. For instance, Archon might show a desktop notification when a long import completes, or have an option to minimize to tray if it's used as a background monitor (future possibilities).

By combining the flexibility of Tailwind CSS with careful attention to platform conventions, Archon aims to look and feel "at home" on each operating system. Users should not need to know it's built with web technologies – it should simply feel like a dedicated, polished configuration management app.

Authentication and Identity Tracking

While Archon is primarily a single-user desktop application, there are features that necessitate tracking user identity and possibly authenticating users, especially in collaborative scenarios or when syncing with remote services.

User Authentication: Out of the box, Archon can function without any login (just like a text editor). However, for teams or shared machines, having user accounts ensures accountability for changes. Archon will support a **basic user authentication** system. This could be as simple as maintaining a list of users who can log into the app (with username/password, stored securely). When enabled, the app would prompt for login at startup, and all actions would be tied to that user's identity. This is optional and can be turned off for a single-user mode.

Alternatively, Archon might integrate with Git-based authentication. For example, if a user connects a GitHub account for pushing snapshots, Archon could use that as the identity (federated login via OAuth). This might be overkill for an initial release, so the simpler approach is local accounts or just configuration of a "user profile" for commit attribution.

Identity for Changes: Regardless of login method, Archon will always attribute changes to an **author name and email** (just like Git). In the app settings or on first run, the user will set their name and email (similar to `git config user.name/user.email`). These are used in commit metadata so that each snapshot knows who made it. If multiple people collaborate by sharing the repo, the history will show each person's name on their commits.

Inside the app, we could show the last edit info for each component if we maintain such metadata. For instance, if a property was last changed by "Alice" on May 5, 2025, that could be shown in a tooltip or a "history" tab for that component. Achieving this requires recording extra metadata (either in Git commit messages or a separate log), so initially we might rely on the commit history for that information. However, a simple approach is: whenever a snapshot is made, link it to the author and timestamp (which Git does). Then a user can see "Snapshot X by Alice at time Y". For fine-grained per-item change tracking, that's an advanced feature not in core spec (it could be approximated by analyzing diffs by author).

Permissions: Archon by itself doesn't need complex permission levels (it's either you can edit or read-only). If multiple users use the same machine login, they could each have Archon accounts to identify themselves but they all can edit if they can log in. For shared projects via Git, the permission to push or not is handled by Git server side (e.g., GitHub access). Archon will not implement its own elaborate permission control – it relies on Git for collaboration control.

Git Credential Integration: When pushing to remotes like GitHub, Archon will need to handle authentication (SSH keys or HTTPS with username/token). The app will provide a place to enter these credentials or will use the system's Git configuration if available. Wails (Go backend) can call out to Git; for example, using `go-git` library or shelling out to system Git which might use SSH keys. We will make it as seamless as possible (e.g., for GitHub, personal access token input, with secure storage).

Audit Log / Change Log: For completeness, Archon could maintain an internal **audit log** of user actions (especially administrative ones like creating/deleting projects, enabling plugins, etc.). This could be useful for enterprise environments. Initially, the Git history is essentially the change log for config data. If needed, we can log other events (like "User X imported data from plugin Y at time Z") in a local log file.

In summary, Archon's authentication and identity tracking ensure that every change can be traced to a user. This not only helps in collaboration (knowing who did what) but is also important in some regulated environments (labs or museums might need records of who approved a change to an exhibit's setup, for instance). The design will remain simple unless extended for multi-user networked scenarios in the future.

Cross-Platform Support and Architecture

Archon is built to run on **Windows, Linux, and macOS**, and potentially other platforms supported by Go/Wails. Ensuring cross-platform functionality influences both technology choices and design:

- The choice of **Wails** as the app runtime means we get cross-platform support by default. Wails bundles a lightweight webview (WebKit on Linux/macOS, WebView2/Edge on Windows) and the Go runtime into a single package [3]. The frontend (Svelte) is essentially a web app, but it's loaded in a native window without the weight of a full browser. This yields a relatively small binary and fast startup, as noted by Sophie Au: Wails "wraps both Go code and a web frontend into a single binary" and feels like a slim alternative to Electron [2].
- **Compilation and Packaging:** We will use the Wails CLI to build installers or binaries for each OS. For Windows, likely an .exe (optionally with an installer or portable option), for macOS, a .app bundle (that can be distributed via disk image or zip, and potentially notarized for Gatekeeper), and for Linux, likely an AppImage or .deb/rpm packages. The build pipeline will produce these artifacts. Using Vite ensures the front-end is built optimized, and then Wails embed it.
- **Filesystem Paths and OS Integrations:** The app will account for differences like file path formats (Windows `C:\path\...` vs POSIX), default locations (e.g., where to store config or temp files – on Windows maybe `%AppData%/Archon`, on Linux `~/.config/Archon`, etc.). Wails/Go can help determine these. If the user selects directories in the UI (e.g., to clone a repo or export a file), we use the native dialog which naturally handles it.
- **Testing on each Platform:** We will test UI elements on all OSes to ensure nothing looks off (for example, font rendering or control sizes can differ). Tailwind should produce consistent styling, but we rely on system-provided scrollbars, title bars, etc., which differ. We might apply some conditional CSS or use Tailwind's platform detection if needed (or simply accept minor differences).

- **Performance on different OSes:** All modern OSes should handle Archon fine. On lower-end Linux machines or older Windows, the webview performance (for potentially large DOM if the config is huge) needs consideration. Svelte's efficient updates mitigate this, but we'll also implement paging or virtualization for extremely large hierarchies (so the app doesn't slow down rendering thousands of DOM nodes at once).
- **Native Features via Go:** One advantage of Wails is that we can write Go code for any platform-specific tasks. For example, if we want to use macOS's Touch ID for authentication (just as an idea) or integrate with Windows registry for some settings, we can – but these are likely unnecessary for Archon's scope. However, one concrete use is to access the local Git installation or manage credentials via OS. The Go backend can also handle heavy tasks (like large JSON diff calculations) in a separate thread, ensuring the UI stays smooth. This separation of concerns (frontend UI vs backend logic) follows a client-server architecture within the app.

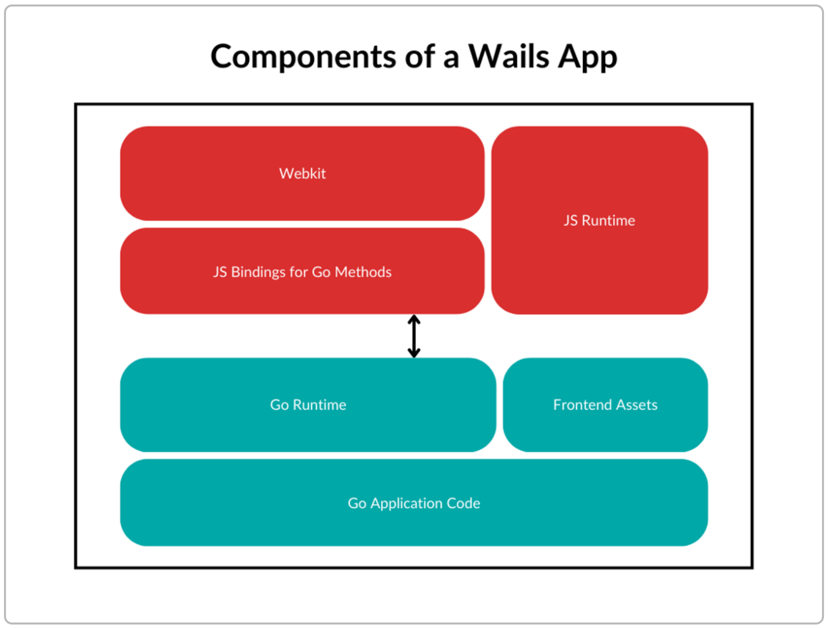

Figure: High-level architecture of the Wails-based application. Wails combines a Go runtime (for backend logic and OS integration) with a WebKit-based JS runtime (for the Svelte frontend), communicating via bindings. This allows Archon's Svelte+TypeScript UI to call Go functions (for Git operations, file access, etc.) and vice versa [3] .

As shown above, the components of the Archon app include the **Go Application Code** (handling things like invoking Git commands, file I/O, plugin loading, etc.) and the **Frontend Assets** (the Svelte compiled app, HTML/CSS/JS) running in a WebView. Wails provides the glue – **JS bindings for Go methods** – so the front-end can call functions like `createSnapshot()` or `pushToRemote()` which are implemented in Go, and get results back. This means complex operations or sensitive tasks can run in Go for performance and security, while the UI remains fluid and interactive [8] .

Cross-platform support also means we consider differences in Git availability. Wails can include a pure-Go Git implementation (like `go-git`) ensuring that even if Git is not installed on the system, Archon can perform version control. This is important for ease of use – a Windows user shouldn't have to install Git separately. However, using system Git could be an option for power users. All these details will be abstracted behind a simple interface in Archon.

Finally, our cross-platform philosophy will be validated by real-world use: we'll have some beta testers on each OS to ensure that Archon feels like a native citizen on that platform. For instance, on macOS, standard keyboard shortcuts (⌘+C, etc.) must work; on Windows, the app should respond to system dark mode toggle if possible; on Linux, ensure compatibility with major distros (Ubuntu, Fedora, etc.). Wails has been used in production for apps like Clave (authenticator app) which proved the viability of this stack on all platforms [4] . Archon will build on that foundation.

Supported User Personas and Use Cases

Archon is a general tool, but it has been envisioned with several specific **personas** in mind. These personas represent typical users in different domains who manage complex configurations. By considering their needs, we can ensure Archon's features are broadly applicable and truly useful in practice. The table below summarizes key personas, their context, and how they would utilize Archon:

Persona	Domain Context	Example Scenario	Key Needs & Features
Lab Technician	Scientific Laboratory	Manages equipment on a lab bench (microscopes, analyzers, etc.) and frequently reconfigures setups for different experiments.	- Hierarchical view to model an experiment's apparatus (bench -> devices -> sub-components). - **Snapshots** to save configurations before each experiment run and after changes, enabling reproducibility. - **Properties** to record settings (e.g., voltages, calibration dates) for each device. - **Diff** to compare "before vs after experiment" setups. - Possibly use import plugins to pull device lists from inventory databases.
Test Site Engineer	Industrial Test Facility	Sets up and maintains test rigs (assemblies of sensors, controllers, and DUTs (devices under test)). Configuration changes between test runs and must be tracked.	- **Version control** is crucial: each test configuration is snapshot-tagged (e.g., "Test #123 Config"). - **Git integration** to push configs to a central repo for team access. - **Diff** to see what changed between test runs (for troubleshooting when a test yields different results). - **Property metadata** like firmware versions, cable connections, etc., attached to components. - Uses **export** to PDF/CSV to include config details in test reports.

Persona	Domain Context	Example Scenario	Key Needs & Features
Museum Curator	Museum Exhibitions	Organizes an exhibit installation with multiple artifacts, display cases, lighting, and multimedia equipment. Changes exhibits seasonally and needs to document each setup.	- **Hierarchy** to represent an exhibit (exhibit -> sections -> cases -> items). - **Properties** for each artifact (e.g., accession number, condition notes, placement measurements). - **Snapshots** tagged by exhibit name/date (e.g., "Spring Gallery 2025") to record how everything was arranged. - **Visual report** (PDF with images/diagrams) to archive the exhibit configuration for future reference. - **Import plugin** possibly to bring in artifact data from the museum's collection management system.
Biomedical Researcher	Biomedical R&D	Configures complex lab setups for biomedical experiments (e.g., gene sequencers with attached PCs, custom analyzers). Needs to track configurations for compliance and repeatability.	- **Detailed properties** on each component to log calibration info, reagent lots, software versions – ensuring experiments can be audited. - **User authentication** important: knowing which researcher adjusted the setup last (for accountability). - **Snapshots** before each trial run; ability to revert if a new setup fails. - Possibly uses **plugins** to import experiment parameters from LIMS (Laboratory Information Management System). - **Cross-platform** usage: might use Archon on a Windows PC controlling instruments and on a Mac for analysis – data sync via Git.
Indie Game Developer	Game Development	Manages configurations of game development hardware and software – for example, multiple testing devices (PCs, consoles), engine configurations, or even in-game entity configurations stored as JSON.	- **One-project-per-repo** model fits game asset versioning; can use Archon to version certain game configuration files or hardware setups. - Uses **hierarchy** to organize assets or environment setups (e.g., "Level 1 -> NPC characters -> their inventory items", or dev hardware rigs). - **Git integration** aligns with existing developer workflows (push to GitHub, collaborate with team). - **Plugin** could be used to import/export data to the game engine (for instance, converting Archon's JSON to Unity YAML or vice versa). - **Diff** to see changes in a configuration between game patches.

Persona	Domain Context	Example Scenario	Key Needs & Features
Agricultural Technician	Smart Farming	Oversees a network of sensors, irrigation systems, and machinery on a smart farm. Configurations include sensor calibration, network topology, and field equipment settings.	- **Hierarchy** to map the farm setup (farm -> fields -> devices in each field -> subcomponents). - **Properties** to store data like sensor thresholds, last maintenance date, location coordinates. - **Snapshots** seasonally or before/after changes (e.g., “Before Spring planting config” vs “After irrigation upgrade”). - **CSV Export** to share a list of all equipment with settings with stakeholders or for regulatory compliance. - **Import plugin** to integrate with farm management software or IoT platform data (pull current sensor list and statuses).

Each persona highlights different aspects of Archon’s functionality, but common themes emerge: the need to structurally organize complex setups, track changes over time reliably, and annotate everything with domain-specific info. Archon’s design fulfills these needs with its flexible hierarchy, robust versioning, and extensibility.

It’s worth noting that Archon’s interface should remain **simple for the casual user** (e.g., a single lab tech managing their bench) yet **powerful for advanced users** (e.g., an engineer managing hundreds of devices). By testing the app with these personas (perhaps via UX research or beta testing in each domain), we can refine default settings and terminology. For instance, a museum curator might not be familiar with the term “repository” or “commit”, so the UI in that context should use more general terms (“project”, “snapshot”) and have good onboarding tutorials. Meanwhile, the indie developer will appreciate seeing the Git underpinnings clearly.

In conclusion, supporting these personas ensures Archon is versatile and user-centered. The personas guide not only feature inclusion but also how we present features in the UI and documentation (speaking the language of the user’s domain when appropriate).

Best Practices and Implementation Notes

To build Archon in a maintainable and high-quality way, the development will follow several **best practices**:

Modular Svelte Components

The UI will be broken down into **reusable Svelte components**, each handling a specific piece of the interface or functionality. For example, we will have components such as `HierarchyTree` (or `ColumnNavigator`), `PropertyEditor`, `SnapshotHistoryList`, `DiffViewer`, etc. Each component will encapsulate its logic and use Svelte’s reactive stores or context for shared state where needed. This modular approach makes the code easier to manage and extend. It also allows testing components in isolation. We'll structure the project so that components are organized by feature (maybe a folder for hierarchy-related components, one for git-related UI, etc.).

Using Svelte's latest features (Svelte 5 likely introduces improvements in reactivity and possibly a new "runes" system for state management), we will ensure state flows are clear and avoid anti-patterns. For example, a central store might hold the current project data (the JSON hierarchy) and snapshots; components subscribe to just the parts they need. This prevents unnecessary re-renders and keeps performance snappy even as data grows. **Unidirectional data flow** will be encouraged: for instance, a user action in a component dispatches an event or calls an action that updates the store (or calls a backend function via Wails), and then the store update propagates to components.

We will also create UI components for generic use, such as modal dialogs, confirmation prompts, form inputs (with Tailwind styling). These can be reused across the app (e.g., the same confirmation dialog component is used for delete confirmations, snapshot confirm, etc., just with different text).

Clean Snapshot and Git Integration Logic

Interfacing with Git can be complex, but we will design the integration to be as straightforward as possible:

- The application will have a dedicated module (likely on the Go side, as part of backend logic) responsible for all Git operations. This module will provide high-level functions like `initRepo()`, `commitSnapshot(tag, message)`, `push(remote)`, `diff(commitA, commitB)` etc. By wrapping Git in our own interface, we can later swap implementations (for example, using a different Git library) without affecting UI code.
- We'll make sure to handle Git errors gracefully. For instance, if a push fails due to network issues or auth, the UI should catch that and display a user-friendly error (e.g., "Push failed: network error, please check your connection" rather than a cryptic Git error). Similarly, if a commit fails (unlikely, but e.g., no changes to commit), inform the user.
- The **snapshot tagging logic** will be carefully implemented to avoid collisions and maintain clarity. If a user tries to create two snapshots with the same tag name, Archon will warn or auto-increment a version. Tags in Git need to be unique unless forced – we'll keep it simple by enforcing unique snapshot names per project.
- **History management:** We plan a linear history model (no branching via the UI to keep things simple). If a user wants to experiment and then perhaps not keep changes, they can always revert to an earlier snapshot (which would effectively be a new commit that reverts, since we won't expose low-level reset). We should clarify this in documentation: Archon is not a full Git GUI, it's focused on sequential version tracking. This approach is a best practice to reduce user confusion and potential Git misuse.
- For the diff feature, if using text diffs, we might use existing diff libraries. If implementing semantic diff, we will write robust comparison functions for JSON object trees. Best practice here includes handling edge cases (property order differences, floating point vs int representation differences, etc.) so that diffs are accurate and meaningful.

In summary, by isolating Git-related code and treating snapshots as first-class entities, we ensure the version control features are reliable. We'll also **extensively test** snapshot creation, branching, and merging scenarios (even if we don't expose branching, user might do a Git operation outside the app).

Extensible Plugin System Design

The plugin system will be designed with **separation and safety** in mind. Key best practices:

- Define clear **interfaces/contracts** for plugins. For instance, an import plugin might need to implement a function `doImport(options)`. We will document what this function should do and how Archon will call it. Keeping this interface stable means plugins won't break when the app updates (unless absolutely necessary).
- Use a **sandbox** or restricted context for running plugins. If using JavaScript plugins, they could be run in an iframe or web worker, for example, to limit their access. They should communicate results back to the main app through structured data (the configuration JSON). This prevents a poorly written or malicious plugin from affecting the main app's internals or the user's system.
- Provide a **Plugin SDK or helpers**. For example, if many plugins will want to create Archon JSON nodes, we can provide a small library that helps plugin authors create nodes, assign IDs, etc., correctly. This reduces errors and speeds up plugin development.
- **Error handling in plugins:** If a plugin throws an error (say, it couldn't parse a file), Archon will catch that and show a message like "Import failed: [error message]". The system should not crash because of a bad plugin. Possibly maintain a log for plugin operations for debugging.
- Versioning of plugins: consider including a version field and maybe compatibility info (a plugin might specify it needs Archon v2.0+ or such). This is forward-looking; initially, with few plugins, not a big issue.
- **Security**: Only load plugins from user-approved sources. Perhaps only allow installing plugins by placing files in a specific directory or via an official registry (if we ever create one). Warn the user that plugins are third-party code.

By planning the plugin system architecture up front and keeping it modular, Archon will remain flexible. We essentially treat the core application as a platform, and imports/exports as extensions. This also means we can add new official plugins (for popular formats) in future releases without touching core logic – a maintainability win.

Robust Error Handling and User Feedback

In an application that deals with file operations, version control, and user data, things can occasionally go wrong. Archon will include comprehensive error handling and communicate clearly to the user in such cases. Some practices include:

- **Contextual Messages:** Any time an operation fails (import, export, snapshot, push, etc.), catch the exception or error code and display a message that is both user-friendly and logs the technical details for troubleshooting. E.g., "Failed to import file. Please check that the file format is correct and try again. (Error: JSON parse error at line 5)". The user sees the friendly tip, and if needed, an "Advanced details" can show the technical error.
- **Non-blocking UI and Loading States:** When performing potentially long operations (cloning a repo, generating a PDF, computing a diff on a huge config), the UI will not freeze. We'll use asynchronous calls (promises or Go routines) and show a spinner or progress bar. If the operation can be canceled (maybe cloning), provide a cancel button.
- **Confirmation Prompts:** Destructive actions like deleting a component or an entire project will ask "Are you sure?" This prevents accidental data loss. For snapshot deletion (if we allow deleting a snapshot/tag), likewise confirm since that alters history.

- **Undo/Redo (Limited):** While full undo across all operations is complex (especially with Git in the mix), we can implement a simple undo for text edits in property fields, or even an “Undo last delete” by leveraging the snapshot system (for instance, Archon could automatically create a snapshot before a bulk deletion so user can recover). At minimum, a user who accidentally deletes something can choose not to snapshot those changes (or close the project without saving, if auto-save is off).
- **Feedback for Actions:** Provide visual or auditory feedback when actions complete. E.g., after a snapshot is created, show a brief toast notification like “Snapshot ‘Baseline’ created successfully.” After pushing to remote, “Changes pushed to origin.” Small confirmations help the user feel confident that their action took effect. Conversely, if nothing happened because there were no changes to commit, say so (“No changes to snapshot”).
- **Accessibility:** Follow accessibility best practices in the UI – ensure that the app is navigable via keyboard (tab order, ARIA labels on custom elements, etc.), and that color choices have sufficient contrast (Tailwind can help with this and we can offer a high-contrast theme if needed). This ties into error display too: e.g., not relying solely on color to indicate an error in a field (use icons or text).
- **Logging:** Under the hood, Archon will log errors (and possibly key events) to a log file (or console in dev mode). This is important for debugging user issues. If a user reports a bug, we can ask for their log file to see what went wrong (without exposing sensitive data).

Adhering to these practices ensures that Archon will be reliable and pleasant to use. Users will trust the tool if it behaves predictably and transparently. A configuration management tool must not lose data or leave the user guessing about state. Therefore, we will also implement **autosave or reminder to save**. For example, if a user closes the app with un-snapshotted changes, Archon should warn “You have unsaved changes. Do you want to create a snapshot or discard changes?” – akin to document-based applications asking to save. Perhaps Archon can even auto-snapshot on close with a generic message, but better to ask the user.

Additional Considerations

Aside from the major points above, a few other best practices and notes:

- **Performance optimization:** Use efficient algorithms for diff and large data handling. For example, do not perform deep copies of the whole config too often; instead diff could be computed by traversing once. Use web workers if heavy computation needed in JS. The Go backend can handle heavy lifting like generating PDF (using a library) or interacting with Git which is C under the hood – these are fast.
- **Testing and QA:** We will write unit tests for critical logic (especially the JSON diff, import/export transforms, and Git integration logic – using a temp repo in tests). UI end-to-end testing could be done with tools like Playwright or by simulating user flows in a test environment. Ensuring reliability in diverse scenarios (long hierarchies, unusual characters in properties, network failures during push, etc.) is part of best practices.
- **Documentation and Help:** The product spec wouldn’t be complete without mentioning that we should produce user documentation – a help guide or at least tooltips in the UI. Possibly include a “Tutorial Project” or sample configuration to demonstrate features on first launch (some users might appreciate an interactive guide).
- **Community and Support:** If plugins are a big part, we might maintain a repository of community plugins, and a forum for users to share tips (though this is beyond the app itself, it’s good to plan for facilitating user-driven improvements which is a best practice for longevity of the tool).

By following the best practices outlined, the development team will create an Archon application that is **maintainable, scalable, and user-friendly**. The modular design and plugin architecture ensure it can grow over time, the clean Git integration provides a robust backbone for all configuration management tasks, and strong error handling plus user feedback will foster trust in the tool.

Conclusion: This specification has detailed the envisioned features and design of Archon, covering everything from core functionality to UI/UX, multi-user considerations, cross-platform deployment, persona-driven use cases, and development best practices. Archon aims to fill a niche for a **universal configuration management tool** that is versatile enough to serve laboratory technicians, engineers, curators, developers, and beyond. By combining a powerful version-controlled backend with an intuitive front-end, Archon will empower users to manage their complex equipment setups with confidence, knowing that every change is tracked and every configuration can be reproduced. The use of Svelte, Tailwind, and Wails ensures that the app will be both high-performance and beautifully native on all platforms. As development proceeds, continual feedback from target users (our personas) will refine the product, but this specification provides a comprehensive blueprint to build from. With Archon, "complex configurations" become easier to tame, collaborate on, and preserve for the future.

---

1 Equipment (Configuration Management) - Aligni
https://docs.aligni.com/equipment/

2 Building a GUI App Using Go and Svelte | Sophie Au
https://sophieau.com/article/building-with-wails/

3 8 Introduction to Wails - Desktop Apps in Go - Project Structure - TheDeveloperCafe
https://thedevelopercafe.com/articles/introduction-to-wails-build-desktop-apps-with-go-project-structure-17ee3f7fcdf7

4 The Perfect Trio: Wails, Go & Svelte in Action - DEV Community
https://dev.to/ansxuman/the-perfect-trio-wails-go-svelte-in-action-50h1

5 Configurations - Aligni
https://docs.aligni.com/equipment/configurations/

6 Tailwind CSS 4.0: Everything you need to know in one place
https://daily.dev/blog/tailwind-css-40-everything-you-need-to-know-in-one-place

7 What's New in Tailwind v4 - DEV Community
https://dev.to/best_codes/exciting-updates-in-tailwind-version-4-40i0