ChatGPT

Expert Recommendations for Obsidian Vault Intelligence Plugin Success

Graph View Integration Strategy

Obsidian does not provide a public API for extending its core graph view, so you face a strategic choice: build a **custom graph view** as a separate `ItemView`, or attempt to **enhance the core graph** via unofficial hacks. The experience of existing plugins suggests a custom view is the safer and more powerful route. For example, the Juggl plugin implemented a completely separate graph interface using the Cytoscape.js library [1]. This gave Juggl complete control over features (like custom node styling, images, and even adding "text nodes") without being limited by Obsidian's built-in graph engine [2]. In contrast, the new Extended Graph plugin tries to overlay features on the core graph, but the developer acknowledges it is *not* built on the core and thus certain advanced features (like free-floating text nodes) aren't possible with that approach [2]. The custom view approach (à la Juggl) clearly enables richer functionality.

Performance and maintenance considerations: Using a custom graph view means you manage the rendering and data yourself, but it avoids breaking changes in Obsidian's internal code. Plugins that hack into Obsidian's internal graph (e.g. by manipulating the PIXI canvas or internal state) can be brittle – a core update might break your hacks. A custom `ItemView` is insulated from such changes, as it relies only on official APIs (to retrieve vault data) and your own rendering code. Indeed, Extended Graph's developer hints that Juggl "might also be more optimized" since it isn't constrained by the core graph [2]. User reports back this up: Extended Graph had to introduce a failsafe to disable itself on large graphs (by default it turns off if more than 20 nodes are present!) to prevent performance degradation [3]. In testing, Extended Graph started to lag during initialization with a vault of just 600+ notes when many visual enhancements (images, curved links, etc.) were enabled [4] [5]. Juggl, on the other hand, was designed for potentially large local graphs by letting users load only relevant portions ("workspace" mode) instead of everything at once [6]. This suggests that a well-designed custom view can handle larger vaults more gracefully, whereas piggybacking on the core graph may require limiting features for performance reasons [4].

User adoption and UX implications: It's worth noting that many Obsidian users did adopt Juggl despite it being a separate view. Juggl gained popularity by offering capabilities far beyond the default graph (custom layouts, selective expansion, saved workspaces) and accumulated a lot of community interest (over 700 stars on GitHub and integrations with other plugins like Breadcrumbs) [7]. This indicates power users don't mind opening a separate graph panel if it delivers value. On the other hand, there is also demand for improving the core graph – Extended Graph's release generated interest for those who want the familiar global graph just with extra features. The question is which approach aligns better with your target users. Given that your plugin's value is in advanced analysis (AI insights, network metrics), users are likely willing to go into a specialized view or panel to use it. The **risk of relying on internal hacks** (high maintenance overhead, potential breakage) likely outweighs the convenience of using the built-in graph canvas. In an ecosystem where Obsidian updates frequently, maintaining a fragile PIXI injection could become a nightmare. Therefore, **I recommend implementing a custom graph view** (as an Obsidian `ItemView`) for your plugin. This gives you full control to integrate 3D layouts, AI overlays, and network science visuals

without fighting the limitations of the core graph. You can still make it feel integrated – for instance, allow opening your view from a toolbar button or a right-click menu on notes (Juggl did this via a "Open in Juggl" command on notes [8]). This way, users can seamlessly switch to your enhanced graph when needed. Overall, a custom view is more future-proof and easier to optimize for your specific needs, whereas enhancing the core graph is only advisable for small cosmetic tweaks.

Large Vault Performance & Architecture

Handling vaults of 1,000–5,000 notes (or more) requires careful architecture to keep the plugin fast and avoid freezing the UI. The good news is that your current core analysis (74k connections computed in ~2-5 seconds) is quite promising. To scale this up while keeping Obsidian responsive, consider the following strategies for performance:

Offload heavy computation to background threads: Obsidian plugins run in an Electron (Chrome V8) environment, which supports Web Workers for multi-threading. Using a Web Worker will prevent blocking the main UI thread during intensive analysis. Some developers initially ran into issues using workers (e.g. getting "Worker is not a constructor" errors by using Node's worker_threads in the wrong context [9]), but the solution is to bundle your code for the **browser** environment and use the standard Web Worker API [10]. In practice, this means configuring your build (Rollup, Webpack, etc.) to output a separate worker script and using `new Worker(URL, {type: "module"})` or similar. By doing this, you can delegate tasks like parsing 5,000 markdown files or computing network metrics to a worker. The main thread can then listen for progress or completion messages and update the UI accordingly. Using a worker will ensure that even if analysis takes several seconds, the Obsidian UI (typing, clicking around) stays smooth. A community developer discussing CPU-intensive plugins noted that without multithreading, heavy tasks *"halt the entire UI"* [11]. So this is crucial for good UX. If for some reason Web Workers become difficult (though they are supported if done right), at least break work into chunks and use `setTimeout`/`requestAnimationFrame` to yield periodically to the event loop [12] – but a true background thread is preferable for large-scale analysis.

Efficient memory management and incremental updates: Loading 5k notes into memory for analysis is doable on modern systems, but be mindful of memory use. Only keep the data structures you need. For example, if you build a graph of all paragraphs or concepts, store them in efficient maps/arrays, and periodically purge or serialize anything not needed for immediate results. One effective pattern is to maintain an **in-memory index** of your vault's data (e.g. a map of note -> list of important words/concepts, or paragraphs -> vector representation, etc.). Construct this index once (lazily or at plugin load in a worker) and update it incrementally when notes change, rather than re-scanning everything on each run. Obsidian provides events for vault changes – for instance, you can register for `vault.on("modify", ...)` or `metadataCache.on("resolve", ...)` to catch when a note is edited or its links updated. Using these, your plugin can *react* to changes: re-analyze just the modified note (and maybe its neighbors) and update the stored connections, instead of recomputing all 5,000 notes. This dramatically reduces workload for continuous use. It's the same principle that Obsidian's index and Dataview plugin use – they don't re-read every file on each query, they cache results and update incrementally. Design your analysis engine with this in mind from the start, as retrofitting it later is harder. Also consider memory lifecycle: if you maintain large arrays or maps, clear or replace them on vault swaps or plugin unload to avoid leaks.

Real-time vs on-demand analysis: You should decide when analyses run. A spectrum of options: real-time (on every keystroke or file save), scheduled background runs, or explicitly user-triggered. **Real-time on**

every keystroke is not practical for heavy analysis – it would overwhelm the system. Even Obsidian's own search and graph view don't update *every* character for large vaults. A better approach is a hybrid: perform analysis **on-demand or at specific triggers**, and potentially in the background at idle times. For example, one model: do an initial full analysis pass in a Web Worker when the plugin first activates or when the user opens the analysis view. After that, use incremental updates as noted above to keep the data in sync. The UI panel can then fetch the latest analysis results on demand very quickly. Another model is to analyze "just in time" – e.g., when the user opens the graph view or requests suggestions, then spin up the analysis. This might mean a short wait ("Analyzing vault…") but it happens only when needed. Many users will accept a 2-5 second pause for a one-time analysis on a large vault, especially if it's clearly communicated.

In fact, consider Obsidian's startup performance ethos: *"Lag is the mind-killer."* One forum post by a developer highlights that Obsidian now measures plugin load times and encourages delaying heavy work until after startup [13] [14]. The takeaway: **don't** bog down the app at launch by crunching thousands of notes before the user can do anything. Instead, defer that work. Perhaps let the plugin fully load (so Obsidian considers it "started"), then either run analysis in the background after a slight delay or wait until the user opens your plugin's UI for the first time [15]. This way, normal vault opening isn't slowed, and you only consume resources when the user is about to benefit from the results. If you do want to keep analyses continuously up-to-date (for the AI suggestions to always be ready), an idea is to perform **background batching**: e.g. every X minutes or when the system is idle, have the worker update some portion of the analysis. This is more complex but can give a real-time feel without actually computing everything in real-time. For a first version, on-demand analysis is perfectly fine and probably the easiest to implement and test.

Performance benchmarks: It's useful to define targets. For instance, *acceptable* performance might be: initial full analysis of a 5,000 note vault in under, say, 5-10 seconds in a worker thread (so as not to freeze UI), and any per-note incremental update in under 500ms. These are rough, but setting expectations helps. Users of community plugins generally understand if a complex operation takes a few seconds, as long as the UI remains responsive (or a progress indicator is shown). What they won't tolerate is the entire app stalling. So prioritize non-blocking behavior over absolute speed. Also, be sure to test on a range of devices – what is 2 seconds on a high-end desktop might be 10 seconds on an older laptop or mobile device. If possible, try simulating a lower-powered environment to ensure it's still okay. And as a fail-safe, consider offering **user controls** for performance vs accuracy. For example, you could let users choose to analyze only a subset of notes (specific folders or recent notes) if their vault is huge, or turn off very expensive analysis features. It's better to give an option than to be unusable on a giant vault. The Extended Graph plugin took the extreme approach of auto-disabling itself beyond a small node count to "prevent performance degradation" [3] – in your case you might not need to disable entirely, but you might, say, skip paragraph-level scanning if the vault has over N million words, etc. In summary, aim for intelligent scaling: **background processing, incremental updates, on-demand triggers, and user-configurable limits** will ensure your plugin remains smooth even as vault size grows.

Production Plugin UX/UI Integration Patterns

Designing a professional and user-friendly UI in Obsidian is crucial so that your powerful features actually get used. Based on successful plugin patterns, here are recommendations for various aspects of UX:

Sidebar panel and views: It's generally a good idea to present your plugin's information in a **dedicated sidebar panel or view**. Many high-profile plugins do this – for example, the Graph Analysis plugin adds a

view in the right sidebar by default [16] to display its findings. You'll likely want a similar always-available panel where network insights, suggestions, and controls reside. Using Obsidian's API, you can register a new `ItemView` (with a unique view type) that can live in the sidebar. This gives you a contained space to render graphs, lists of suggestions, etc., and it will have the standard Obsidian UI chrome (it can be pinned, collapsed, or moved by the user like any other pane). Be sure to handle the view's lifecycle properly: Obsidian can restore open views on reload, so **do not forcibly detach your view on unload** – let Obsidian reopen it after an update so the user doesn't lose their layout state [17]. In practice, that means you register the view and leave it; when the plugin unloads for an update, don't manually close the leaf. Also, avoid keeping a global reference to your view object; instead, retrieve it via `workspace.getLeavesOfType` when needed [18]. This prevents memory leaks or issues if multiple instances open. These details aside, the main point is to **make your panel easily accessible** – perhaps auto-open it once on first install (with a welcome message or analysis ready to run) so the user discovers it. After that, consider providing a toggle command (so users can open/close the panel via a hotkey or command palette). A sidebar panel is preferable to pop-up modals for something like ongoing insights, because the panel can stay open alongside the user's notes, updating live as needed.

Live updates vs manual refresh: In the UI, decide whether your panel updates automatically or requires user action. A good pattern is **contextual updates** – e.g., if the user navigates to a different note, the panel could show analysis relevant to that note. Graph Analysis does exactly this: when you switch the focused note, its suggestions panel recomputes for the new note [16]. You could do similarly, running a lighter weight analysis for the active note on the fly (using precomputed data) to show "related notes" or "paragraph connections" relevant to what's currently open. This makes the plugin feel responsive and integrated into the workflow. However, balance this with performance – if such on-the-fly computation is heavy, you might gate it behind a manual action (like a refresh button or it only runs when the panel is visible). An alternative is to allow the user to "pin" an analysis in the panel. For example, if they want to see the whole vault overview instead of context switching with the note, you could allow locking the panel to a certain view (like "show global clusters" until they toggle back to per-note mode). Providing a few modes (current note vs whole vault analysis, etc.) can be very useful.

AI suggestions intrusiveness: When integrating AI-powered suggestions or prompts, err on the side of *non-intrusive.* Obsidian users tend to prefer tools that quietly offer help rather than aggressively interrupt. So, avoid modal pop-ups or toast notifications that appear unasked to tell the user "I found a connection!" – that can get annoying fast. A better approach is to surface these insights within the UI the user is already looking at. For instance, your sidebar panel could have a section like " AI Suggestions" that the user can glance at when desired. You could also use subtle indicators: maybe an icon in the status bar or a small badge on the sidebar icon that lights up when new suggestions are available after an analysis run. This way, the user is *informed* of available insights but can choose when to look at them. Graph Analysis plugin's model is instructive: it doesn't spam the user with alerts; it simply shows a list of related notes in its panel, and users incorporate that into their workflow as they see fit [16]. You might consider a similar philosophy: the intelligence is "always on" in the background, but the user pulls it in when they need it. For example, if your plugin detects two clusters of notes that are disconnected (a "gap"), you could highlight this in the panel and perhaps provide a one-click action: "Ask AI to suggest how to bridge these topics." But it's the user's choice to click it. By keeping suggestions contextual and user-driven, you'll avoid disrupting those who are in a focused writing flow while still providing valuable insights to those actively seeking them.

Mobile compatibility considerations: Adapting to mobile (Android/iOS) is important if you expect users to occasionally check their graphs or suggestions on the go. Obsidian mobile does support community

plugins, but the UI is more constrained. First, ensure your plugin **does not crash or malfunction on mobile** – test it on a device or emulator. If certain features truly cannot work on mobile (due to performance or reliance on Node APIs), you might have to mark the plugin as "desktop only" in the manifest [19]. However, ideally you can make most features available. Graph visualization is the hardest part on mobile: small screen and limited GPU power. The Juggl plugin managed to make its graph view work on mobile, as noted in its documentation ("Works on mobile!") [20], so it's feasible. Follow some of Juggl's likely approaches for mobile UX: provide touch controls for zoom & pan (Obsidian's core graph already supports pinch-to-zoom on mobile – if you use a Canvas/WebGL library like Cytoscape or Three.js, ensure gestures are enabled). Avoid relying on hover events (which don't exist on touch devices) – any interactive element should use tap or long-press instead. You might need slightly larger UI buttons or hit areas, as fingers are less precise than mouse pointers. Also consider performance: maybe **limit the default graph size on mobile**. For instance, you could choose not to auto-render the entire vault graph on mobile; instead, start with a summary or require the user to choose a smaller scope. An idea: on mobile, your "graph view" could initially show just the currently open note's neighborhood or a particular query result, rather than thousands of nodes. If a user really tries to load the whole vault graph on mobile, warn that it could be slow or simply make that an intentional action ("Tap to load full graph"). In the worst case scenario that interactive graph manipulation is too clunky on phone screens, ensure at least the **analytical insights are accessible on mobile**. That might mean a textual list or summary of key findings (e.g., "Top clusters: X, Y, Z. Potential connections: Note A and Note B share topics."). Providing an alternate, simple view for mobile users will still deliver value when they're away from their PC. In short, plan for **graceful degradation**: the plugin should function and provide information on mobile, even if the fancy 3D graph or massive visualization is reserved for desktop.

Integration into daily workflow: To make your plugin essential rather than a novelty, integrate it at points in the workflow where it can assist without friction. Think about *when* a user would most benefit from a nudge or insight. Some ideas: when a user creates a new note, your plugin could (in the background) find related existing notes and gently indicate "You might link these with your new note" (perhaps by populating a suggestions panel section). Or while writing, if the user pauses, the plugin could update suggestions for that note (so they can check if they've mentioned this topic elsewhere). Always keep such features **opt-in or easily toggled**. Some users will love real-time suggestions; others might find it distracting and only want manual triggers. Providing settings to control the frequency or conditions under which suggestions appear is a good practice. For example, an option "Enable automatic suggestions for notes with few links" or "Offer to analyze new notes upon creation" lets power users turn it on, while others can leave the plugin more manual. Another powerful workflow integration is search and filtering: allow users to run your analysis on a subset of notes (say, a tag or a folder). Perhaps they could right-click a folder and choose "Analyze this folder's ideas" – your plugin could then open the graph focusing on just that subset. This aligns with how people organize projects in Obsidian, making your tool directly useful for focused research questions. Also, consider providing **commands** or hotkeys for key actions (for those who prefer keyboard-driven work). For instance, a command palette entry to "Show related notes (AI)" for the active note could quickly open the panel or a modal with those suggestions. This way, even users who hide the sidebar can access the intelligence features on demand. By embedding your features into existing flows (note-taking, reviewing, searching) rather than requiring a completely separate usage pattern, you increase the likelihood the plugin becomes part of the user's routine.

Obsidian Ecosystem Strategy

Building a great plugin is not just about code and features – success also depends on navigating the ecosystem: getting the plugin approved, attracting users, and sustaining the project. Below are strategic tips:

Smooth approval and compliance: The Obsidian devs are quite welcoming to new plugins, but you must follow the submission rules. Ensure your plugin meets all **submission requirements** in the Obsidian Developer Docs (e.g. proper manifest with unique ID, a short clear description, version bumping) [21]. A few specific things to check before submission: if you use any Node.js or Electron-specific APIs (for example, if you were to use the `fs` module or OS-level calls for some reason), mark `isDesktopOnly: true` in your manifest [19]. This tells the review team that it won't try to run on mobile (preventing crashes). From your description, it sounds like everything is local but likely using web APIs or JavaScript libraries, so you might not need desktop-only – just be mindful of this rule. Another common review feedback is to **remove any leftover template code or console logs** [22] [23]. Since most plugins start from the sample plugin template, make sure you've renamed classes, settings, etc., to not have generic names like "MyPlugin" [24]. Also purge debug `console.log` statements or excessive logging – reviewers prefer a clean console so that users only see meaningful messages or errors [25]. Security-wise, avoid using `eval` or `innerHTML` with untrusted content, and generally follow good practices (the plugin guidelines explicitly warn against directly injecting HTML due to XSS risks [26] [27] – use DOM methods instead). If your plugin connects to external services (it sounds like it does not, since it's all local AI via presumably local models or just suggesting connections without calling an API), make sure to disclose that. In your case, the privacy angle is a selling point – *no* external calls – so you're good. Once you're confident in these areas, submit your plugin to the official repo. The review process typically takes a few days to a week. They might come back with minor suggestions – for example, "please use `this.app` instead of global `window.app`" (since the global may be removed in the future) [28] or things of that nature. Respond and fix those quickly. Demonstrating that you've adhered to their guidelines and care about quality will make the approval quick. Many plugin devs report that as long as everything is in order, their plugin was merged and published without issue on the first try.

Community adoption and visibility: To get users, you'll need to actively showcase the plugin's value. Start with a **forum post in Obsidian's "Share & Showcase" section**. Make it informative and enthusiastic: explain what the plugin does (in terms of user benefit, not just features), perhaps something like "This plugin uncovers hidden connections in your notes and visualizes your knowledge with AI assistance – effectively giving you an InfraNodus-like analysis *within* Obsidian, for free." Be sure to highlight the unique selling points: e.g. *privacy* (all analysis is local, no account needed, unlike InfraNodus which requires a cloud account [29]), *power* (advanced network science metrics, AI insights, interactive graph), and *integration* (fits into existing Obsidian workflow). Including a short **demo video or GIF** can dramatically boost interest – if people see a cool graph visualization or an AI suggestion example, it will draw them in. You don't have to have a polished marketing video; even a 1-minute screen recording posted on YouTube or GIF in the forum can do wonders. Additionally, share the news on Reddit (r/ObsidianMD) – there are weekly threads for showcasing work, or you can create a separate post if it's a substantial release. Many Obsidian users hang out on Reddit and might discover the plugin there. Twitter (or X) is another channel; if you have an account, a tweet tagging @obsdmd or using #ObsidianMD sometimes gets reshared by the community. Some community members write blogs or make YouTube videos about notable plugins – if your plugin solves a big problem (saving $600/year on a tool and adding capabilities others charge for), it's likely to get attention.

Documentation and onboarding: A key factor in adoption is how easy you make it for users to get started and see value. Consider creating a **GitHub Wiki or documentation site** for in-depth details (like how each analysis works, tips for usage). The Extended Graph plugin's author, for instance, set up a wiki and was actively improving documentation to help users understand how to use it [30]. You might also include a "Help" button in your plugin's UI that links to documentation or even opens an Obsidian note with instructions (some plugins ship an "Instructions.md" in their GitHub which users can read). Good documentation and quick responses to user questions (especially early on) will encourage positive reviews and word-of-mouth. Since your plugin has potentially complex features, *manage user expectations*. For example, explain that the first analysis might take a few seconds on a large vault, or that certain results (like AI-generated questions) are suggestions, not absolute truths. The more upfront you are in docs, the less confusion later. Also, prepare to offer **support**: have an issue tracker on GitHub for bug reports, and try to be responsive in the first few weeks of launch. Early adopters will forgive small bugs if they see the developer is actively fixing them. This builds a good reputation.

Maintenance and long-term strategy: The Obsidian community values plugins that are maintained. Many users have been bitten by abandoned plugins that break after an update. To establish confidence, signal that you intend to maintain the plugin. You could mention in your README that you welcome contributions and will be actively developing it. Given the scope of your plugin, consider it an evolving project – you might get feature requests (e.g. "can it calculate XYZ metric" or "can it integrate with Dataview?"). You obviously don't have to implement everything, but be open to feedback. Also, keep an eye on the competition: for instance, if another free plugin arises or if InfraNodus lowers its paywall, know your unique value. Perhaps your advantage will always be the tight Obsidian integration and open-source nature. Lean into that. Engage with the community by perhaps writing a short tutorial on "how to use the vault analysis plugin to improve your notes" – education can drive adoption. You could collaborate with others too; for example, Breadcrumbs (for hierarchy) or Dataview might have synergies with your network analysis. In fact, Juggl's dev collaborated with the Breadcrumbs plugin [7] – those kinds of integrations can boost usage because you tap into each other's user base.

Competitive positioning: You are effectively bringing capabilities to Obsidian that previously required external or paid tools. Position this plugin as an **"essential power-user tool"** for knowledge management in Obsidian. It's not just a pretty graph; emphasize how it can actually enhance idea development (generate questions, find missed links, etc.). Users should feel that if they install this, their Obsidian vault becomes smarter and more valuable. Since Graph Analysis (the free plugin) already exists, clarify that your plugin is complementary or an advancement. Graph Analysis primarily does link-based algorithms and shows results in a list [31] [32]; it doesn't visualize a graph or use AI. You're combining analysis with visualization and AI-driven recommendations, which is novel. Some users who tried Graph Analysis might have wished for visualization or more automation – target that gap. Also, InfraNodus's Obsidian integration (if users are aware of it) has a high barrier – requiring an account and subscription [29]. You can explicitly say your plugin provides a *privacy-first, local* alternative to InfraNodus. That's a huge selling point for a lot of Obsidian users who intentionally seek offline solutions. In your communications, it's fine to mention InfraNodus as inspiration, but then highlight how your plugin is free and private. This positions it as community-friendly and not just a commercial product. Finally, **be confident but not arrogant** in positioning: encourage people to try it out and compare. If you can get a few vocal community members to test and love it, their endorsements will carry more weight than anything. Aim to make those first users delighted (perhaps even personally invite a few known Obsidian enthusiasts to beta test it and give feedback). Their buy-in will help drive broader adoption and establish your plugin as a must-have for serious Obsidian users.

Production Readiness Checklist

To ensure your plugin is truly production-ready for a wide release, review this final checklist of technical and user-experience items:

- **Performance & Optimization:**
- Profile the plugin on a large vault and make sure it meets acceptable speed targets. For example, full-vault analysis should ideally complete in a few seconds on a mid-range machine (if it's longer, ensure it's running in a worker with a progress indicator). The UI should never freeze during analysis – use workers or async strategies to achieve this [33].
- Implement fallbacks for extremely large vaults. If certain features won't scale (e.g. real-time 3D rendering of 10k nodes), disable or modify them gracefully. For instance, Extended Graph plugin sets a node limit and auto-disables features past that point to protect performance [3]. You might not need a hard cutoff, but consider a warning or an option to limit scope if a vault is above a certain size.

- Optimize memory usage: avoid holding onto large data in memory longer than needed. After finishing an analysis, if some huge structures (like full text of all notes or a massive graph adjacency list) are no longer required, let them be garbage-collected (e.g., clear references or use local scopes). If you use a web worker, release it or reboot it occasionally if it accumulates too much heap usage (some plugins spawn a fresh worker for each task to avoid memory bloat).

- **Code Quality & Maintainability:**

- Go through your code and remove any leftover sample code or irrelevant comments. Ensure class names and file names reflect their purpose (no `SamplePlugin` or similar) [24]. This not only helps reviewers but will make future maintenance easier for you and any contributors.
- Follow the Obsidian API best practices: use `this.app` (the app instance provided to your plugin) rather than global `window.app` [28] to future-proof your code. Use the provided `Vault` and `MetadataCache` methods to read notes, rather than manual file I/O, to leverage Obsidian's caching [34].
- Implement proper cleanup: dispose of any event listeners, intervals, or timeouts on unload. Utilize `this.registerEvent` and `this.registerInterval` which auto-clean on unload [35]. For example, if you added `vault.on('modify', ...)`, register it so Obsidian can remove it when the plugin is turned off. This prevents memory leaks and weird behavior when the plugin is disabled or updated.

- If you created a custom view (graph panel), do not manually detach it on unload (to allow Obsidian to restore it on reload) [17]. Also avoid storing the view instance in a global variable; access it via the workspace when needed [18]. These practices help avoid issues on plugin update (Obsidian will seamlessly reinitialize your view if you follow the conventions).

- **Testing (Functionality & Compatibility):**

- **Functional testing:** Test every major feature in a variety of scenarios. Does the graph view open and display correctly with a small vault? With a large vault? Test the AI suggestion generation with different notes (e.g., a very short note, a very long note, notes with code blocks, etc.) to see if it

handles all gracefully. Ensure suggestions are relevant and not nonsensical (some sanity-checking of AI output might be needed so you don't show completely off-base suggestions).

- **Cross-platform:** Test on at least Windows and Mac (and Linux if possible). Obsidian is cross-platform and your plugin should work everywhere. Watch out for any path issues or OS-specific quirks if you use file paths or OS operations (again, using Obsidian's abstractions mostly avoids this).
- **Mobile:** If your manifest is not desktop-only, install the plugin on Obsidian Mobile. Verify that core functions work – e.g., the analysis can run (might be slower), the UI elements are accessible (the panel opens, or alternative UI appears if panel isn't available on mobile). If something critical fails on mobile, either fix it or mark the plugin desktop-only to avoid user frustration. Keep in mind mobile hardware limits – maybe test on an older phone if you can to see how it holds up.
- **Backward compatibility:** Set your `minAppVersion` in manifest appropriately [36]. If you developed on the latest Obsidian, consider if it would break on slightly older versions. Generally, using the latest API is fine (people can update Obsidian), but ensure you don't require an Insider build unless necessary.

- **Edge cases:** Try things like: vault with no links at all (does your analysis handle a graph with isolated notes?), vault with lots of existing interlinks (does it still find "unlinked" connections or does everything show as one big cluster?), notes with non-English text (if you do any NLP, consider Unicode and multilingual support). Also test with different user settings that might affect behavior: for example, if using the graph, test both Light and Dark mode (color contrasts), different themes (some themes might have custom CSS for panels that could interfere, though that's rare).

- **User Experience & Onboarding:**

- **First-run experience:** It's often nice to greet the user with a brief intro. The first time your plugin runs, you could open your sidebar panel automatically with a "Welcome" message or an initial scan result. This way the user doesn't have to figure out how to start. If you do an initial analysis on first run, let the user know ("Analyzing your vault to build the knowledge graph…") so they aren't puzzled by background activity.
- **Documentation:** Ensure the README is thorough. It should include: what the plugin does, how to open the interface (e.g. "Activate the **Intelligence** panel from the ribbon or via Cmd+P command…"), and basic usage examples. Given the complexity, also document each major feature (maybe headings for "Graph View", "Connection Suggestions", "Community Detection", etc., with short explanations). The README is displayed in Obsidian's plugin browser, so make the first 250 characters a compelling summary (as per guidelines, one sentence action statement) [21]. You can then link to your GitHub wiki or a documentation site for more details.
- **Settings UI:** Provide settings for any tunable parameters – for instance, a toggle for automatic analysis vs manual, a slider for how many suggestions to show, or a threshold for community detection detail. Make sure settings are clearly labeled (use sentence case and avoid overly technical jargon in user-facing text) [37]. Organize them with headings if there are many options [38]. Good settings hygiene makes your plugin feel polished and lets users customize to their preference.
- **Visual polish:** Double-check that the plugin's visual elements (icons, panels, modals) match Obsidian's style. If you add a ribbon icon, use an appropriate Icon (perhaps the graph icon or a neural network-style icon if available) and give it an `aria-label` for accessibility. In your graph view, use sensible defaults for colors that work in both light and dark mode (or dynamically adapt if possible). Little touches like tooltips on buttons in your UI, or a legend explaining graph colors, can improve usability.

- **Error handling:** Anticipate things that could go wrong and handle them gracefully. For example, if the AI suggestion engine fails (say you integrate with an optional local model or something and it's not available), the UI should show a message like "AI suggestions are unavailable (check your settings or OpenAI key)" rather than just silently failing or throwing an error. Likewise, wrap any file-parsing in try/catch so one corrupt note doesn't crash the whole analysis – you could skip that note and perhaps log a warning. This way the plugin won't mysteriously break on certain user data edge cases.

- **Release & Support:**

- Before tagging a 1.0.0 release, update your `version` in manifest and in your `versions.json` (if you maintain one) to match. Tag the release on GitHub and follow the community plugin submission steps (pull request to the `obsidian-releases` repo). Double-check that the manifest JSON and README are correctly formatted, as these are what the plugin store uses.
- Once released, be prepared to **iterate** quickly if needed. It's common to get a few bug reports or suggestions within days of launch. Plan a bug-fix update (e.g. 1.0.1) soon after if necessary. This will show the community that the plugin is actively maintained and any rough edges are being smoothed out.
- Set up channels for feedback: GitHub issues is standard. You might also get posts on the forum thread – respond there to show engagement. If issues start repeating, consider improving documentation or adding a FAQ. For example, if many ask "Does it work with plugins X or Y?" or "How do I interpret this graph metric?", you can add that info to your documentation.
- **Long-term maintenance:** Keep an eye on Obsidian's updates (subscribe to the announcements or check the insider builds) to catch any upcoming changes that might affect you. For instance, if in the future Obsidian introduces an official graph API (one can hope!), you might want to adopt it. Or if there are breaking changes in the plugin API, plan to update accordingly. Users appreciate when a plugin stays compatible over time – it encourages them to invest time in using it deeply.
- Take care of yourself as well – it's a free plugin, and while the community is appreciative, don't burn out trying to please everyone. It's fine to set expectations in your README about what the plugin will and won't do. That said, a well-supported plugin with strong unique value can even be monetized (donations, sponsors) if it becomes essential to many – since you mentioned it's free and replacing a $600/year tool, you might attract companies or power-users who'd sponsor its development. This could be a nice bonus, though keeping the plugin free and open is itself a huge contribution that will earn you goodwill in the community.

By following this checklist and the guidance above, you'll be well on your way to delivering a **production-ready, highly impactful** Obsidian plugin. With the right technical choices (robust architecture, thoughtful UX) and a savvy ecosystem strategy (clear positioning and community engagement), your "vault intelligence" plugin can become an indispensable tool for Obsidian power users – essentially supercharging their knowledge graphs without any cost or privacy trade-offs. Good luck, and we're excited to see this in action! 2 29

---

1 6 7 8 20 GitHub - HEmile/juggl: An interactive, stylable and expandable graph view for Obsidian. Juggl is designed as an advanced 'local' graph view, where you can juggle all your thoughts with ease.
https://github.com/HEmile/juggl

2 4 5 30 New plugin: Extended Graph - Share & showcase - Obsidian Forum
https://forum.obsidian.md/t/new-plugin-extended-graph/98850

3 Performances · ElsaTam/obsidian-extended-graph Wiki · GitHub
https://github.com/ElsaTam/obsidian-extended-graph/wiki/Performances

9 10 12 33 How to Speed Up CPU-Intensive Tasks in an Obsidian Plugin? Workers Not Supported - Developers: Plugin & API - Obsidian Forum
https://forum.obsidian.md/t/how-to-speed-up-cpu-intensive-tasks-in-an-obsidian-plugin-workers-not-supported/103392

11 Can plugins use Web Worker? - Developers: Plugin & API - Obsidian Forum
https://forum.obsidian.md/t/can-plugins-use-web-worker/81040

13 14 15 Call for plugin performance optimization - especially for plugin startup - Developers: Plugin & API - Obsidian Forum
https://forum.obsidian.md/t/call-for-plugin-performance-optimization-especially-for-plugin-startup/32321

16 Obsidian Graph Analysis Plugin – Curtis McHale
https://curtismchale.ca/2023/07/19/obsidian-graph-analysis-plugin/

17 18 22 23 24 25 26 27 28 34 35 37 38 Plugin guidelines - Developer Documentation
https://docs.obsidian.md/Plugins/Releasing/Plugin+guidelines

19 21 36 Submission requirements for plugins - Developer Documentation
https://docs.obsidian.md/Plugins/Releasing/Submission+requirements+for+plugins

29 InfraNodus Obsidian Graph View Plugin with AI Insights & Chat
https://infranodus.com/obsidian-plugin

31 32 GitHub - SkepticMystic/graph-analysis: Analyse the structure of your Obsidian graph using various analysis techniques
https://github.com/SkepticMystic/graph-analysis