ChatGPT

Building a MacOS Native AI Browser (Atlas-Like): Strategy & Architecture

Vision: An Atlas-Like Native App for FountainAI

You propose creating a **MacOS native** browser-style application (using AppKit/SwiftUI) that integrates an LLM assistant and the `midi2.js` control-plane library. The goal is to replicate the concept of OpenAI's **ChatGPT Atlas** – an AI-enhanced web browser – but tailored to the FountainAI ecosystem. In this app, the **LLM interface** (chat/agent) would work in tandem with the **MIDI2-based real-time control layer** and FountainAI's reasoning stack. This means users could chat with an AI that not only browses or fetches information but also **orchestrates real-time computation or devices** using the MIDI2 API (e.g. scheduling GPU tasks, controlling instruments, etc.). The result is a platform where high-level reasoning (via an LLM) meets low-level timing and control (via `midi2`), all within a polished native Mac experience.

Lessons from OpenAI's Atlas (AI-Integrated Browser)

OpenAI's *ChatGPT Atlas* demonstrates the power of embedding an AI agent directly into a browsing environment. Atlas is essentially a **Chromium-based browser with ChatGPT built-in**, enabling users to "chat with a page" and even have the AI perform actions on websites on their behalf [1]. Key takeaways from Atlas:

- **Seamless Chat Interface:** Instead of a traditional address bar, Atlas features a chat field, making conversation the primary interface for navigation and tasks [2]. Users can ask the AI to summarize pages, fill forms, or execute multi-step web tasks via natural language.
- **Agent Mode Automation:** Atlas's advanced *Agent Mode* lets the LLM browse the internet and complete tasks like copying data between apps or adding items to a cart without direct user clicks [3]. The AI becomes an active agent in the browsing context, not just a passive Q&A bot.
- **Cross-Platform Delivery:** Atlas launched first on macOS as a standalone app and is expanding to Windows, iOS, Android [4]. This suggests that even though it's web-centric (Chromium), it's delivered natively per platform for better integration (e.g. using native UI shells and system resources).

Applying these insights: Your proposed app would similarly provide a conversational UI and agent capabilities, but extend them beyond web browsing. The LLM in your app could not only navigate information (like Atlas does) but also control *non-web* processes – for example, scheduling GPU computations, driving audio/visual outputs, or coordinating other services. In essence, it's **Atlas meets real-time orchestration**: a chat-driven interface for both web content and timed computational tasks.

Moving Beyond “Web-First”: Why Native App Makes Sense

Earlier approaches may have considered a purely web-based client (or Electron-style app) for this system. However, you’ve recognized that a **“web-first” model is not required** – and going native per platform has distinct advantages. Key points in this strategic shift:

- **Leverage of WebGPU Standard (Cross-Platform by Design):** The `midi2.js` library was designed around Web technologies, specifically positioning MIDI 2.0 as a **“real-time control plane for WebGPU”** and other subsystems [5]. Crucially, WebGPU is a spec-defined, cross-platform API for high-performance graphics/compute. By **targeting the WebGPU API**, your core logic gains a **uniform, platform-agnostic layer** for GPU and parallel computation. This means the same MIDI2-driven scheduling and control code can run on different platforms with minimal changes, since WebGPU abstracts the underlying Metal/Vulkan/DirectX calls. In short, the **WebGPU spec provides portability**, so we can embrace native apps without losing cross-platform compatibility – it’s “write once, run anywhere” for GPUs and real-time events [5]. This is a great advantage: we get broad compatibility **without being constrained to a browser tab**.

- **Performance & Real-Time Guarantees:** A native AppKit/SwiftUI application can offer better performance and timing control than a browser environment. Browser-based apps suffer from extra layers (JavaScript engines, sandboxing, unpredictable event loops) that can introduce latency or jitter. In contrast, a purpose-built native app can use optimized system APIs and threads for scheduling. This is critical since one of the MIDI2 stack’s strengths is precise timing (32-bit timestamps, jitter reduction, etc.). In a native macOS app, we can run the MIDI2 scheduler on high-priority threads or use Metal directly for GPU tasks, achieving more **deterministic real-time behavior** than in a general browser. The result is a more **responsive and reliable orchestration layer** (for example, scheduling a sequence of GPU animations or MIDI events will be rock-solid in timing, which aligns with MIDI2’s design for microsecond accuracy).

- **Deeper OS Integration:** By not being web-first, the app can integrate tightly with macOS features. This includes using **SwiftUI for a rich interface**, accessing local files/devices, using native windows/menus, drag-and-drop, notifications, etc. The user experience can be tailored to Mac conventions (keyboard shortcuts, touchbar, etc.), yielding a more polished and intuitive tool than a generic webpage. Moreover, if the app needs to interface with hardware (MIDI devices, sensors) or system services, a native app has the necessary permissions and frameworks. For instance, the app could interface with CoreMIDI or I/O Kit if needed (though your design wisely avoids direct CoreMIDI for portability [6], it’s good to have the option of lower-level access when absolutely necessary). Overall, **native apps can offer capabilities and UX refinements that web apps can’t easily match**.

- **Security and Offline Operation:** Running your own native application means you control the sandbox and network access. If needed, the LLM agent and reasoning engine could run offline or on a local network for privacy. Web-first solutions often depend on constant internet and expose more surface (since they run in a general browser). With a native architecture, you can decide which parts use cloud services (e.g. calling an OpenAI API for the LLM) and which run locally (e.g. the orchestration engine). This flexibility can improve security and resilience.

- **Platform-Specific Optimizations:** Going native per platform means on each OS you can optimize for that environment. For example, on macOS you might use Metal and Accelerate frameworks for ML or DSP tasks; on Windows, you’d use DirectX12 or DirectML, etc. Each app can still adhere to the WebGPU-level abstractions, but is free to tune performance or use

supplemental APIs where beneficial. Also, UI-wise, each platform's native toolkit (SwiftUI on Mac, maybe WinUI on Windows) can be used to deliver the best experience for that user base. In contrast, a web app would be stuck with one-size-fits-all UI and might not feel as "at home" on any platform.

In summary, embracing a **native-first strategy** doesn't sacrifice your cross-platform goals – thanks to WebGPU and similar standards, the core logic remains portable. It does, however, significantly enhance performance, real-time reliability, and user experience by utilizing each platform's full potential rather than lowest-common-denominator web tech.

Integrating `midi2.js` Natively (WebGPU Beyond the Browser)

A central piece of this architecture is the `midi2.js` library and its underlying philosophy. Recall that `midi2.js` views MIDI 2.0 not as a music-only API but as a **"timestamped, structured, vendor-agnostic event fabric"** for coordinating heterogeneous compute [5]. In practical terms, it provides: UMP packet encoding/decoding, high-res timing (JR timestamps), structured SysEx messaging, and scheduling – all of which can drive things like GPU tasks, physics simulations, or ML model parameters (not just synthesizers).

How do we use this in a SwiftUI Mac app? There are two paths:

- **Use the Swift MIDI2 Library:** The repository has a Swift implementation (the "Swift reference stack") that mirrors MIDI2 functionality. A Mac app could directly use this Swift library for encoding/decoding MIDI2 messages, scheduling events, and bridging to CoreMIDI or other outputs. This might be the most efficient route for MIDI messages and ties in nicely with SwiftUI. However, the Swift stack might be more focused on traditional MIDI (with CoreMIDI adapters for hardware I/O) and might not yet include the WebGPU-oriented abstractions present in `midi2.js`. You might need to extend the Swift code to add concept of a "WebGPU adapter" or any custom compute adapters similar to the JS version's WebAudio/Three.js demos [7] [8].

- **Embed `midi2.js` via JavaScript Engine:** Since `midi2.js` is written in TypeScript/JavaScript and designed for browsers, another approach is to embed a JS runtime in the Mac app. For instance, using WebKit's JavaScriptCore or a lightweight Node.js instance packaged in the app, you could run `midi2.js` within the native app. This would let you directly leverage its WebGPU integration and scheduling logic as-is. The native SwiftUI side could communicate with the JS engine (e.g. via message passing or calling into JS functions) to schedule events or get callbacks. Essentially, the SwiftUI UI becomes a host, and `midi2.js` runs behind the scenes orchestrating real-time tasks. This approach ensures you're using the exact same **spec-accurate, cross-browser code** in your native app, guaranteeing consistency with web environments [5]. The downside is added complexity of hosting a JS environment and potential performance overhead crossing between Swift and JS; but given Apple's JSCore is quite optimized, this can be manageable.

Regardless of approach, **WebGPU in a native context** needs consideration. On macOS, WebGPU (as a standard) is implemented by WebKit (Safari) and Chromium via Metal behind scenes. In a native app, you can't directly call "WebGPU API" unless you embed a browser view or use a library. One promising route is using the emerging **"WebGPU on native"** libraries (like Mozilla's `wgpu` in Rust, or Dawn from Google) which implement the WebGPU API for native apps. You could, for example, write a small native module (in Swift or C++) that uses Dawn or wgpu to create a WebGPU device/queue on Metal, so that `midi2.js` could potentially target it. However, this is cutting-edge and might involve significant work.

A simpler near-term path: if your real-time needs involve GPU graphics or compute, you might directly use Metal in the Swift app. Since your MIDI2 events are high-level, you can manually map them to Metal calls. (For instance, a MIDI2 message might trigger a shader animation or update a buffer – the Swift code can call Metal functions accordingly.) This doesn't use the WebGPU API per se, but achieves the same goal (timed GPU control) using platform API. Because you have the **spec of WebGPU as a guiding interface**, you can design your Metal integration in a way that could later be abstracted to WebGPU common calls if needed. The key is that **the timing and structure from MIDI2** is the same, and you're simply swapping out the final execution layer per platform.

In short, **midi2's real-time scheduling and structured event stream will be central to orchestrating native tasks**. Your app will treat MIDI2 events as a universal currency for "timed commands", whether those commands are musical notes, shader parameter changes, or function calls. The Mac app can faithfully use this to drive native APIs. Thanks to the cross-platform nature of the MIDI2 schema and (ideally) WebGPU, you'll maintain compatibility – the same JSON schema or UMP messages can describe an action regardless of platform, which aligns perfectly with your cross-device strategy.

FountainAI Reasoning Stack Integration (LLM + Agents)

Perhaps the most exciting aspect is melding the **FountainAI reasoning/agent stack** with this new app. FountainAI's architecture (from what we know) unifies **OpenAPI-defined services and MIDI2-defined "instruments" into one agent ecosystem** [9]. In this model, every capability – whether a cloud API or a physical/virtual instrument – is described in a standardized way so that an LLM-driven *planner* can orchestrate them together. The cognitive layer (LLM) essentially acts as a conductor, reading the "score" of available functions and devices and coordinating them logically and temporally.

Integrating this into the app means the LLM in your Atlas-like browser isn't just blindly scraping or guessing how to use tools. Instead, it leverages the **formal contracts** of the FountainAI ecosystem. For example:

- An OpenAPI microservice might expose a `/generateReport` endpoint, and a MIDI2 instrument might expose a `playNote` capability via MIDI-CI. The FountainAI framework ensures both advertise themselves with machine-readable descriptors (OpenAPI schema or MIDI-CI profiles) so the LLM knows what it can do and how to call it. **"Every capability, whether exposed by a RESTful API or a physical instrument, must be discoverable, callable, and composable within the same reasoning graph."** [10]. This unified semantics is a game-changer – the LLM can reason about API calls and MIDI commands uniformly.

- The **LLM Orchestration layer** of FountainAI uses these descriptors to plan complex actions. According to the spec, LLM agents perform *Introspection* (fetching schemas), *Planning* (matching outputs to inputs between agents), *Reflection* (adjusting parameters in real-time), and *Learning* (saving successful compositions) [11]. All of this is done with a guarantee of safety and semantic correctness, because the LLM isn't hallucinating function names – it's using the exact operations defined by the agents [11].

In a native Atlas-like app, the **chat interface becomes the entry point for this orchestration**. The user might ask in natural language for a certain outcome ("Generate a harmonic visual pattern that reacts to this data and play a sound"), and behind the scenes the LLM (leveraging FountainAI's planner) will compose a solution: perhaps calling a data API (OpenAPI agent) to get data, then instructing a GPU instrument (via MIDI2 messages) to render a pattern, and a synth instrument to play sound – all coordinated in time.

Your app would facilitate this by: - Hosting the **Agent Registry and Planner** (possibly as local components or services). This registry keeps track of available agents/capabilities – some could be built into the app (e.g., a "WebGPU instrument" agent that uses `midi2` to control graphics, or a "Browser agent" that can fetch webpages), and others could be external services the app knows about. The planner (or Orchestrator) composes the graph of these capabilities when the LLM forms a plan [12]. - Providing the **LLM backend**. This could be an OpenAI GPT-4/GPT-5 via API, or a local model, augmented with a prompt/tool plugin that allows it to retrieve the OpenAPI/MIDI schemas and issue commands in a structured way. Based on the FountainAI spec, the LLM likely works in conjunction with a "Codex" or tool-usage layer that translates its high-level decisions into actual API calls or MIDI messages. Your app might embed this logic or call out to a FountainAI service that handles it. The key is that the LLM is guided by the formal capability descriptors, making it much more reliable in performing multi-step operations.

- Enabling **MIDI2 loopback and bridges**: According to the agent spec, CoreMIDI is avoided, and instead they use loopback or network MIDI2 transports for interoperability [6]. In your app, you might instantiate a loopback MIDI2 driver (perhaps the `midi2.js` library can act as one) so that any "instrument agents" run within the app can send/receive MIDI2 messages internally (or to other local agents) with no OS-specific MIDI driver needed. This keeps the system deterministic and portable. For example, a "virtual instrument agent" in the app could receive MIDI2 UMPs from the LLM's plan execution – those could drive a sound or visual module directly.

- Including a **Browser/Web agent** (if web browsing is a desired feature like Atlas). Since Atlas's hallmark is browsing web pages via AI, you likely want a component that can fetch webpages and allow the LLM to summarize or extract info. This could be done by an agent in the FountainAI sense: a microservice agent that given a URL can return the page content or perform searches. Because that agent would present an OpenAPI (HTTP-based) interface, it fits right into the ecosystem – the LLM could call `GET /web/fetch?url=...` or similar to get content, then maybe use other agents on that data. Integrating web browsing isn't mandatory for your app's vision, but it would certainly match Atlas feature-for-feature and beyond.

To ensure clarity, your app's **architecture** might look like this:

- **UI Layer (SwiftUI)**: Chat interface for user ↔ LLM conversation, and panels/canvases for output (e.g., a web view for displaying browsed pages or a custom Metal view for visualizations driven by MIDI2 events, etc.). Also UI for any direct user control of agents (perhaps a list of connected devices or services, etc.).

- **Agent & Orchestration Layer**: This is the core "brain". It could be embedded as part of the app or run as a local background service. It includes:

- The **Agent Registry** (catalog of capabilities from both cloud APIs and local instruments).
- The **Planner/Orchestrator** that uses the registry info to connect agents together at runtime (as per the Ensemble layer in the spec [9]).

- The **LLM integration** which actually interprets user requests and produces a plan. This might involve prompt engineering where the LLM is given the agent descriptors (so it knows what actions are available) and possibly few-shot examples of how to call them. The Orchestrator might double-check or simulate the plan before execution, given the importance of reliability.

- **Execution Layer**: When a plan is decided, the app must execute it. Some parts might be simple API calls (HTTP requests to services), others might be scheduling a series of MIDI2 events (for a

device or internal instrument). Execution will leverage `midi2` for anything time-sensitive. For example, if two actions must happen in sync or in a tight sequence, wrapping them in UMP packets with timestamps via `Midi2Scheduler` ensures they occur at the right moment [13] [14]. The execution layer would also handle streaming results back (e.g., if an agent emits real-time events, the app catches those and maybe feeds them into the LLM's context or displays to user).

- **Native Services/Adapters**: These are the actual "agents" the app hosts natively. This could include a **WebGPU/Graphics adapter** (to render visuals from MIDI2 events), a **WebAudio or CoreAudio adapter** (to produce sound from MIDI events or to route audio), and any domain-specific modules (physics engine adapter, etc., echoing the adapter idea in `midi2.js` for Three.js, Cannon.js, etc. [8]). Each adapter would conform to the agent contract: e.g., a "Physics agent" might advertise capabilities like `applyForce` which correspond to some MIDI2 control messages under the hood. By adhering to the same descriptors, the LLM can treat even complex subsystems uniformly.

Notably, this entire design embodies the FountainAI philosophy that *"APIs perform like instruments, and instruments document themselves like APIs, with LLMs conducting both through a single semantic score."* [15]. In other words, your native app will be the stage on which this orchestra of agents performs, with the LLM as the conductor and MIDI2 as the timing sheet music. The synergy between a powerful reasoner (LLM) and a precise sequencer (MIDI2) is extremely potent – the LLM brings adaptability and intelligence, while the MIDI2 stack ensures reliable execution in time.

Architectural Benefits and Trade-offs

Strategically, pursuing a native Atlas-like app for each platform (starting with macOS) appears sound and forward-looking:

- **Innovation & Differentiation:** This app wouldn't just be a copy of Atlas; it goes a step further by integrating *real-time control and multi-modal actions*. It positions MIDI 2.0 (via `midi2.js`) as a backbone for orchestrating anything in computing, which is exactly the vision of the library [16] [17]. No one else (to our knowledge) has an AI agent that can both talk to web services and jam with low-level instrument protocols in one package. This could be a flagship demonstration of FountainAI's unified architecture.

- **Cross-Platform Path:** Starting native on macOS makes sense (especially since Atlas did the same, likely due to Mac's strong developer audience). With the groundwork of WebGPU-abstraction and MIDI2, you can later port to Windows, Linux, etc., building native UIs there. If core logic is kept modular (perhaps even move it into a shared library in C++/Rust or use portable Swift where possible), each new platform app mainly implements the interface layer. This incremental rollout mirrors Atlas (macOS first, others next) and reduces risk by proving out the concept on one platform.

- **User Experience and Adoption:** Many users (especially professionals or power users) prefer native apps for performance and integration. A SwiftUI app can offer a smooth, responsive experience that might feel more trustworthy than a web app running in Chrome. Also, distributing via app stores or direct download gives you control over updates and possibly monetization (if that's a concern). Users could run the app offline for certain local tasks (e.g., controlling MIDI hardware or local files through AI), which is a unique selling point compared to cloud-only solutions.

- **Complexity:** On the flip side, this approach is complex. You'll be developing essentially both a *browser* and an *agent orchestrator* plus integrating a *real-time engine*. That's a lot of moving parts. Ensuring the LLM plans actions correctly via the agent contract will require careful design (probably extensive prompt tuning and fallback rules). There's also the matter of resource usage – running an LLM (possibly via API calls) alongside heavy GPU tasks could tax system resources, so you'll want to optimize scheduling (perhaps pausing/unpausing certain processes). However, these challenges are surmountable with your deep understanding of the stack, and the payoff is high.

- **Maintenance:** By not using a single web codebase for all, you accept maintaining multiple codebases (one per platform UI). This is mitigated by maximizing shared code: for example, the agent logic and MIDI2 core can be shared. Using automated generation from your JSON schemas (as you do for Swift and JS libraries) can help keep implementations in sync. Also, consider open-sourcing parts of this; there might be community interest in an AI+MIDI orchestration framework.

Conclusion

Yes – building a **native, platform-specific Atlas-style application** for FountainAI makes a lot of sense. It aligns with the strengths of your technology stack: the WebGPU-focused design of `midi2.js` gives you cross-platform, high-performance roots [5], and your FountainAI agent architecture provides the blueprint for integrating AI reasoning with those capabilities in a robust way [11]. By forgoing a purely web-first approach, you gain better control over performance, timing, and user experience, all while still adhering to open standards that ensure compatibility across environments.

In essence, this strategy lets you **"go native" without losing the web's universality** – truly the best of both worlds. A MacOS SwiftUI app can showcase the full potential: imagine an AI that can open web pages to gather info, then trigger a MIDI2-driven visualizer or IoT device in perfect sync, all through one intuitive interface. This would be a compelling demonstration of how AI assistants can move from just text-based help to *real-time, multimodal co-creators*.

Architecturally, as long as you keep the layers modular (UI vs. core logic vs. adapters) and use the WebGPU/MIDI2 specs as the connecting tissue, the system will remain scalable and portable. It is a bold undertaking, but it directly capitalizes on the "spec-defined cross-platform compatibility" you identified and the rich **FountainAI philosophy of merging APIs and instruments into one semantic ecosystem** [15].

In summary: Going native per platform is not only feasible – it's strategically advantageous. It will allow the FountainAI + MIDI2 vision to shine on each device with optimal fidelity, much like Atlas did for web browsing but even more ambitiously for general computation. You have the right building blocks; now it's about orchestrating them (quite literally!) into a unified product. The roadmap ahead is exciting, and this approach positions you at the forefront of AI-driven interactive computing.

---

[1] [3] [4] OpenAI launches ChatGPT Atlas AI browser, LLM can browse the internet for you and even complete tasks — initial release for macOS, with Windows, iOS, and Android to follow soon after | Tom's Hardware
https://www.tomshardware.com/tech-industry/artificial-intelligence/openai-launches-chatgpt-atlas-ai-browser-llm-can-browse-the-internet-for-you-and-even-complete-tasks-initial-release-for-macos-with-windows-ios-and-android-to-follow-soon-after

2 Thoughts, Observations, and Links Regarding ChatGPT Atlas
https://daringfireball.net/2025/10/thoughts_observations_and_links_regarding_chatgpt_atlas

5 8 13 14 README.md
https://github.com/Fountain-Coach/midi2/blob/d6af5d1916c6258cb4619d3f1080097c7b5c0e89/midi2.js/README.md

6 9 10 11 12 15 AGENTS.md
https://github.com/Fountain-Coach/FountainKit/blob/9e6509c91fafcc780903c413e66492079de4136a/specs/AGENTS.md

7 AGENTS.md
https://github.com/Fountain-Coach/midi2/blob/d6af5d1916c6258cb4619d3f1080097c7b5c0e89/midi2.js/AGENTS.md

16 17 paper.md
https://github.com/Fountain-Coach/midi2/blob/d6af5d1916c6258cb4619d3f1080097c7b5c0e89/midi2.js/paper.md