ChatGPT

PatchBay App: MIDI 2.0 Implementation and Visual Regression Testing

Overview of PatchBay and MIDI 2.0 Instruments

PatchBay Studio is the initial user interface of FountainAI – a visual patcher for MIDI 2.0 instruments that appears right after first boot. It presents an infinite canvas where *nodes* (instruments) can be placed and connected. Under the hood, each visual element (canvas, nodes like synth modules, etc.) can operate in an **"instrument mode"**: it publishes a CoreMIDI 2.0 virtual endpoint and exposes its tunable properties via MIDI-CI **Discovery** and **Property Exchange** [1]. In other words, PatchBay's UI components are also *MIDI 2.0 devices* with self-describing properties. This is enabled by FountainAI's **MetalViewKit** framework, which provides *visual instruments* that map their internal state to MIDI 2.0 messages. For example, **MetalViewKit** includes views (e.g. `MetalTriangleView`, `MetalTexturedQuadView`) that present as MIDI 2.0 instruments – their visual uniforms (rotation, zoom, color tint, etc.) map to MIDI 2.0 *Channel Voice* messages, and they advertise identity & tunable parameters via Property Exchange [2]. This unified MIDI 2.0 layer ("MIDI 2.0 everywhere") ensures every interactive component can be discovered and controlled programmatically. Crucially, the use of MIDI-CI (Capability Inquiry) with Property Exchange means instruments introduce themselves and can be configured without guesswork [3]. In summary, **the PatchBay app is built from modular MetalViewKit Instruments**, each acting as a MIDI 2.0 endpoint with discoverable state – a design that makes the UI highly deterministic and machine-operable.

MIDI 2.0 Implementation in FountainAI

FountainAI's MIDI 2.0 stack underpins PatchBay's interactive audio-visual engine. It includes support for multiple transports (CoreMIDI virtual endpoints, ALSA on Linux, and an in-process loopback for testing) and a MIDI-CI scaffolding for device discovery and property exchange [4]. Every instrument or service module speaks in **Universal MIDI Packets (UMP)** format (the MIDI 2.0 message format), which allows high resolution control (32-bit velocities, 16-bit pitch bends, per-note controllers, etc.). For example, when a PatchBay visual node is enabled in instrument mode, it creates a virtual MIDI 2.0 device with a unique **MetalInstrumentDescriptor** (including manufacturer, product, and instance ID). Through this interface, the instrument can send and receive MIDI **Channel Voice** messages (e.g. Note On/Off, CC, PB) to reflect user actions or control visual parameters [5] [6]. It also responds to **System Exclusive 7** messages for MIDI-CI: if a controller sends a Discovery Inquiry, the instrument automatically replies with its identity and supported features; if a **Property Exchange** GET arrives, the instrument returns a JSON snapshot of its current property values (e.g. canvas zoom level, a node's page size, etc.), and if a SET arrives, it applies the new property values to the UI state [7] [8]. This MIDI 2.0 implementation essentially bridges the UI and the audio/logic layer – even audio DSP engines and ML components are wired via MIDI 2.0. In fact, Fountain's "Instrument Bridge" integrates an in-process sampler and ML-to-MIDI pipelines so that audio DSP and visual instruments all speak the same MIDI 2.0 language [9] [10]. The result is a cohesive system where **visual state changes, audio engine events, and external controls are all unified as MIDI 2.0 events**. This design not only provides expressivity and precision, but also **determinism**: an instrument can publish a JSON property snapshot of its state and respond to the exact same inputs in a repeatable way, which is crucial for automated testing and reproducible creative sessions [11].

MetalViewKit Instruments as Building Blocks

MetalViewKit-based Instruments are the fundamental building blocks for both app functionality and testability. Each instrument (e.g. the PatchBay canvas, a Stage node, a Grid overlay, etc.) registers itself with a transport that can be swapped for testing. In normal operation, instruments use system MIDI transports (e.g. CoreMIDI virtual ports) so that they can be controlled by external MIDI tools or internal services. However, in a test context, PatchBay can run with a special environment (e.g. `PATCHBAY_MIDI_TRANSPORT=loopback`) that forces all instruments to use an in-memory loopback transport [12]. The **LoopbackMetalInstrumentTransport** provides a hub where messages can be sent directly between a test script and the instrument, bypassing the actual OS MIDI stack [13] [14]. This means the test code can instantiate a *robot controller* and immediately find the instrument by name in the loopback registry. For instance, when the PatchBay canvas view is created in tests, it will register as an instrument (often named "PatchBay Canvas" or similar); the test harness can wait for this registration and then obtain a handle to send it MIDI messages [15] [16].

Once connected, the full power of MIDI 2.0 is available to drive the UI in a deterministic way. The **Robot Testing Harness** (sometimes called **MIDIRobot**) uses this to simulate user interactions at the data level. Instead of synthesizing low-level mouse events, the robot can send high-level MIDI2 commands – effectively **instructing the UI components via their instrument interface**. For example, the test robot can send a *vendor-defined JSON SysEx* message with topic `"ui.zoomAround"` to zoom the canvas around a point, or `"ui.panBy"` to scroll the view [17] [18]. It can send `"marquee.begin"` / `"marquee.update"` / `"marquee.end"` sequences to simulate click-and-drag selection boxes [19] [20]. It can even emulate connecting nodes: toggling PatchBay's "connect mode" and sending a sequence of port selection commands (this happens via the PatchBay service API or by simulating the sequence of port taps in the EditorVM, as seen in logic tests). Similarly, instrument uniform values can be adjusted by sending **Property Exchange SET** messages – for instance, a test can tweak the grid size by setting the `"grid.minor"` property via MIDI, or change a "Stage" node's page size by setting `"stage.page"` [21] [22]. Because each such message goes through the exact same code paths as a real user interaction (only at a higher semantic level), we get very robust test coverage. In summary, **the MetalViewKit instruments make every piece of the UI remotely controllable and observable**, forming a foundation for automating both functional behavior and rendering output.

Full-Stack Visual Regression Testing with the Robot Harness

Using the MIDI2-based Robot Harness, FountainAI has implemented a **full-stack visual regression testing** framework. The goal is to verify that interactive sequences produce the correct outcomes both *internally* (events, state changes) and *externally* (on-screen pixels), across the entire application stack. To achieve this, the testing system operates the PatchBay app like a robot user and checks for regressions at multiple layers:

- **UI Interaction Simulation:** The robot harness "plays" the app by issuing a scripted sequence of actions. It can place nodes, connect or disconnect cables, turn knobs, zoom/pan the canvas, etc., using the MIDI 2.0 instrument interfaces as described. This effectively simulates a human using the app, but with machine precision. For example, a test script might load a known scene graph, then *tap* an Oscillator node and *drag* a connection from `Oscillator.out` to `Filter.in` all via the robot, as conceptualized in design docs [23]. The robot actions are deterministic and timestamped, ensuring the sequence can be exactly reproduced.

- **Visual Snapshot Verification:** A custom **SceneRecorder** hooks into the Metal rendering pipeline of the app to capture rendered frames for comparison [24]. In practice, tests leverage utilities to

snapshot an NSView (the canvas) into an image and then compare it to expected output. The testing framework maintains **golden images** (baseline snapshots) for various UI states of PatchBay. After performing an action sequence, the test captures the resulting UI as an image and computes a diff against the baseline. If the pixel-level difference exceeds a small threshold, the test flags a regression. This provides extremely sensitive detection of unintended changes: *"The combination of pixel-level image diffs and semantic event validation offers unmatched precision."* [25] . In other words, even a minor rendering glitch or layout shift will be caught. The tests are run under consistent conditions to ensure pixel-for-pixel determinism (same window size, same initial state, etc.), leveraging Metal's deterministic rendering mode (Teatro engine) for reproducibility [26] .

- **Semantic Event and State Validation:** In parallel with image checks, the robot validates that the **semantic outputs** of each interaction are correct. This involves checking MIDI events and app state. For example, after a connection is made between nodes, the test can assert that a corresponding MIDI message (perhaps a Universal MIDI Packet encoding a property link or NoteOn) was generated, or that the application's internal graph data structure now contains the expected link. The harness includes a **UMPValidator** that inspects the stream of MIDI 2.0 messages produced during the test to ensure they conform to the MIDI 2.0 spec and to expected behavior [27] [28] . Additionally, an **OpenAPI Contract Checker** is integrated: as the PatchBay app talks to its backend service (PatchBay service over OpenAPI) during the test, all API responses are verified against the OpenAPI schema to catch any contract deviations [29] . This means the system's client-server agent interface is also under test, ensuring that an AI agent using that same API will not encounter unexpected results. The testing is truly *full-stack*: ***"Tests simultaneously exercise the GUI, rendering engine, and MIDI2 logic — detecting cross-layer regressions early."* [30] .

- **Corpus Logging and Analysis:** Unique to FountainAI's approach, test outcomes aren't thrown away – they are logged into the **FountainStore** corpus for long-term analysis [31] . Every robot test run can contribute data: the sequence of actions, the generated MIDI messages, and even the captured images can be saved as part of a special "robot-tests" corpus. This allows the AI side of Fountain (LLMs or analysis tools) to reflect on test scenarios. In essence, QA artifacts become training data or examples for the AI. *"By logging to FountainStore, Robot Testing aligns QA with the platform's AI reasoning fabric, turning tests into data for reflection."* [32] . This is a forward-looking design that blurs the line between testing and knowledge: an intelligent agent could study these stored test cases to understand how instruments behave, fostering **human-machine co-creativity** by having the AI learn the "language" of the app's interactions in a safe, deterministic context.

To illustrate the full-stack regression testing, consider a concrete example: a **Stage node margins test**. The Stage node is a visual instrument representing a page layout (with properties like margins and baseline spacing). A test can initially set the Stage's margins to a default (say all 18 points) and take a snapshot of the canvas [33] . Then it uses the MIDIRobot to send a Property Exchange message to increase all margin values to 72. After letting the UI update, it takes another snapshot [34] . The two images are diffed: the expectation is that changing margins significantly alters the Stage node's appearance (the content area becomes smaller), so a large number of pixels should differ. Indeed, the test computes the mean squared error (MSE) between images and asserts it is above a threshold (indicating a noticeable change) [35] . If this visual effect did not occur (e.g. due to a bug), the test would fail. Meanwhile, the same test could also verify that the `dashboard` data (which tracks instrument settings) has updated strings for the new margin values, ensuring the state change propagated to the app's model [36] [37] . This kind of holistic verification exemplifies how **visual regression tests catch UI rendering issues and logic errors in one sweep**.

PatchBay's Visual Regression Test Suite

The PatchBay app's test suite is a comprehensive collection of such scenarios, serving as a model for future FountainAI app development. The tests are organized under `PatchBayAppUITests` and include both functional checks and snapshot-based regression checks [38]. Key areas covered are:

- **Canvas and Grid Behavior:** Tests ensure that the infinite canvas and adaptive grid work as expected at different zoom levels and translations. For example, grid line decimation (hiding minor lines when zoomed out) and coordinate conversions are verified with numeric assertions [39]. Edge cases like negative or extreme zoom are clamped properly [40]. The grid snapping of node positions after drags is tested by simulating a drag and verifying the node's coordinates align to the nearest grid intersection [41] [42].

- **Node Linking Logic:** The suite tests the mechanics of connecting nodes. A logic test can programmatically call the same functions that a user action would (e.g. selecting an output port then an input port) to ensure an edge (wire) gets created in the view-model [43]. There are tests for the special "fan-out" connect mode (using Option key to connect one output to multiple inputs in sequence), and for breaking connections (via double-click on a port) – confirming the internal data (list of edges) updates correctly [44] [45].

- **UI Rendering Snapshots:** The most distinctive part of PatchBay's tests are the *image snapshot tests*. Many interactive features have corresponding golden images stored under `Tests/PatchBayAppUITests/Baselines`. For instance, there are baseline screenshots for the canvas in various states – with certain nodes connected, with certain grid settings, etc. The test runner compares new snapshots to these baselines to detect visual regressions. When a UI change is intentional (say a design improvement that changes the appearance), developers update the goldens and use a provided script to **rebaseline** the images [46]. The presence of this image-based testing means that even purely visual aspects (like a highlight glow when a link is added, or the alignment of labels) are under test. As noted in the project documentation, *"the canvas is snapshot-tested"* and existing golden images are treated strictly – any change in the rendering must be justified [47]. This gives the team confidence that refactoring the rendering engine or integrating new libraries (e.g. adopting AudioKit's Flow canvas) does not inadvertently alter the expected visuals [47].

- **Input/Output and Integration:** Some tests spin up the PatchBay service in a one-shot test mode to verify that the end-to-end integration holds. For example, they might call the suggestion API to retrieve auto-generated links and then apply them, checking that the UI reacts correctly. The OpenAPI schema compliance is also implicitly tested here (since any mismatch would cause either a test failure or a contract checker alert during robot tests). By exercising both the **PatchBay client (app)** and **PatchBay server (service)** in tandem under CI, the suite ensures that the agent-facing API remains stable and that the UI correctly consumes it [29] [48].

Together, these tests make the PatchBay app a **template for future app development** in FountainAI. New apps can be built as collections of MetalViewKit instruments, immediately gaining the ability to be controlled and tested via the same MIDI2 robotic harness. Developers are encouraged to write focused tests for each view's logic (as was done for PatchBay's grid, zoom, link operations, etc.) and to include snapshot tests for visual verification. The PatchBay project even provides tooling to assist with this process – for instance, environment flags to save out new baseline images on disk, and CI scripts to automate running visual tests and printing diffs [46]. This approach ensures that as the platform grows

(e.g. adding new instrument types or creative features), everything remains **verifiable in an automated, deterministic way**.

Agent Testability and Human–Machine Co-Creativity

One of the driving philosophies behind this MIDI2+Instruments architecture and robust testing is to facilitate **agent-based creativity**. By formalizing all interactions through open standards (MIDI 2.0, OpenAPI) and validating them rigorously, FountainAI enables AI agents (LLMs) to step in as reliable creative partners. The PatchBay app is essentially a "spec-first" control surface – every capability of the UI is also an API call or MIDI message, making it transparent to machines. The test suite's OpenAPI contract checks and corpus logging mean that an agent interfacing with PatchBay sees a well-defined, stable environment where it can suggest actions (like linking an LFO to a filter cutoff) and have high confidence in the outcome. In practice, the team has designed PatchBay so that *"for LLMs, it narrows the action space to typed, auditable changes and returns explainable diffs"*, allowing an AI to propose or apply changes on the canvas with minimal risk [49] [50]. The fact that every change is deterministic and recorded (with before/after states, diffs, and even image snapshots) means the AI can reason about its effects.

Crucially, the **Robot Testing framework doubles as an agent training ground**. The same mechanisms that let a test script connect nodes and tweak properties can be exposed to an AI agent in a controlled fashion. Because the system is thoroughly tested, any co-creative AI tool (such as an "auto-patcher" or generative assistant) can rely on the operations to be consistent with the human UI. This alignment frees the human and AI to collaborate – the human might add a synthesizer module on the canvas while the AI, via the API, immediately connects it to a sequencer and sets reasonable initial parameters. Both actions go through the same MIDI2 instrument interfaces, so they are functionally equivalent. PatchBay's design explicitly envisions *"a place where you can see, test, and evolve behavior with high confidence — and where LLMs can safely co-pilot rather than free-solo."* [51] In other words, the system provides guardrails (through determinism and testing) such that an AI helper can participate in creative patching without breaking things or doing the unexpected. This is the essence of **human–machine co-creativity**: the machine (agent) contributes ideas and actions in the creative process, and the human can trust these contributions because the entire stack (from GUI visuals to MIDI messages to backend logic) is validated and transparent.

In conclusion, the PatchBay app demonstrates how FountainAI's **MIDI2 implementation and MetalViewKit instruments** create a powerful synergy between user interface, automated testing, and AI integration. The full-stack visual regression suite not only safeguards against regressions, it also produces a rich dataset of interactions. This approach ensures that as new instruments or features are added, they come with built-in testability and observability. By systematically testing everything from pixel outputs to MIDI event streams, FountainAI has built a development model where creative tools can be evolved rapidly **without sacrificing reliability or determinism**. This paves the way for complex creative applications (in music, visual arts, etc.) that remain stable under continuous changes and even **invite AI agents to co-create** in a safe, testable environment. The PatchBay app's testing architecture is thus a blueprint for future apps – proving that with the right abstractions (MIDI 2.0 instruments) and rigorous regression tests, we can achieve a harmonious collaboration between human creativity and machine assistance, all while maintaining a rock-solid user experience.

References and Sources

- FountainAI PatchBay Robot Testing Framework [52] [53]
- FountainAI MIDI2 Instrument Implementation (MetalViewKit) [5] [7]

- FountainAI Instrument Bridge Design 2 3
- PatchBay App Agent & Test Guide 1 38
- PatchBay Visual Regression Tests (Examples) 34 42
- PatchBay Studio Vision and Agent Integration 51 47

---

1 38 46 47 48 49 50 51 AGENTS.md
https://github.com/Fountain-Coach/FountainKit/blob/f038578cfdca20a484f27bbfe627a7f7199f532e/Packages/FountainApps/Sources/patchbay-app/AGENTS.md

2 3 4 9 10 11 INSTRUMENT_BRIDGE.md
https://github.com/Fountain-Coach/FountainKit/blob/f038578cfdca20a484f27bbfe627a7f7199f532e/Design/INSTRUMENT_BRIDGE.md

5 6 7 8 17 18 MetalInstrument.swift
https://github.com/Fountain-Coach/FountainKit/blob/f038578cfdca20a484f27bbfe627a7f7199f532e/Packages/FountainApps/Sources/MetalViewKit/MetalInstrument.swift

12 21 GridInstrumentTests.swift
https://github.com/Fountain-Coach/FountainKit/blob/f038578cfdca20a484f27bbfe627a7f7199f532e/Packages/FountainApps/Tests/PatchBayAppUITests/GridInstrumentTests.swift

13 14 MetalInstrumentTransport.swift
https://github.com/Fountain-Coach/FountainKit/blob/f038578cfdca20a484f27bbfe627a7f7199f532e/Packages/FountainApps/Sources/MetalViewKit/MetalInstrumentTransport.swift

15 16 19 20 MIDIRobot.swift
https://github.com/Fountain-Coach/FountainKit/blob/f038578cfdca20a484f27bbfe627a7f7199f532e/Packages/FountainApps/Tests/PatchBayAppUITests/MIDIRobot.swift

22 36 37 StageRobotInstrumentTests.swift
https://github.com/Fountain-Coach/FountainKit/blob/f038578cfdca20a484f27bbfe627a7f7199f532e/Packages/FountainApps/Tests/PatchBayAppUITests/StageRobotInstrumentTests.swift

23 24 25 26 27 28 29 30 31 32 52 53 Robot-Testing.md
https://github.com/Fountain-Coach/FountainKit/blob/f038578cfdca20a484f27bbfe627a7f7199f532e/Plans/Robot-Testing.md

33 34 35 StageMarginsStressTests.swift
https://github.com/Fountain-Coach/FountainKit/blob/f038578cfdca20a484f27bbfe627a7f7199f532e/Packages/FountainApps/Tests/PatchBayAppUITests/StageMarginsStressTests.swift

39 40 41 42 43 44 45 CanvasLogicTests.swift
https://github.com/Fountain-Coach/FountainKit/blob/f038578cfdca20a484f27bbfe627a7f7199f532e/Packages/FountainApps/Tests/PatchBayAppUITests/CanvasLogicTests.swift