A Model Context Protocol for VCV Rack 2: Feasibility, Design, and Proposal

1. Introduction: The Case for a Model Context Protocol (MCP) in VCV Rack 2

1.1. Defining the Model Context Protocol (MCP)

VCV Rack 2 provides a powerful and flexible virtual Eurorack environment, enabling complex sound design and musical expression through interconnected modules.[1] Communication between these modules primarily relies on established paradigms like Control Voltage (CV) and Gate signals, mirroring hardware modular synthesis.[2] However, as module complexity and patch sophistication increase, the need arises for richer, more structured data exchange between modules that goes beyond simple voltage levels.

This report introduces and explores the concept of a Model Context Protocol (MCP) for VCV Rack 2. An MCP is envisioned as a standardized, robust mechanism allowing VCV Rack modules to publish, discover, and subscribe to arbitrary, structured data or contextual information. This information could range from musical metadata (chord names, sequence labels, scale information) and large datasets (sample information, wavetables) to complex internal states requiring synchronization between multiple, potentially non-adjacent modules.

Crucially, the MCP is not intended to replace existing, highly efficient mechanisms like CV, Gate, or the Expander/Message system used for adjacent module communication.[4] Instead, it aims to *supplement* these methods, providing a dedicated channel for higher-level information exchange that is currently difficult or impossible to achieve cleanly and reliably within the VCV Rack 2 ecosystem.

1.2. The Problem Statement: Limitations of Current Inter-Module Communication

VCV Rack 2 offers several built-in mechanisms for module interaction, largely based on Eurorack hardware standards:

- **CV/Gate/Triggers:** Analog voltage signals transmitted via virtual patch cables form the backbone of control. Pitch is typically controlled via the 1V/Octave standard, while Gates/Triggers (binary on/off signals, often 0V to 5V or higher) control timing and events.[2] Polyphony allows multiple channels of CV/Gate over single cables.[4] While essential for real-time control, these signals are inherently limited to single floating-point values per channel and lack the structure needed for complex data.

- **Lights & Widgets:** Visual feedback is provided via Lights [4], and user interaction occurs through Widgets like knobs, buttons, and switches.[4] These are primarily UI elements, not general-purpose data transfer mechanisms.
- **Serialization & Storage:** Modules can save and load their state, including parameter values and custom data (via dataToJson/dataFromJson), ensuring patch persistence.[4] For large data like samples, dedicated Patch Storage directories can be used to avoid UI lag during autosave.[4] However, these mechanisms are designed for persistence, not for real-time communication between active modules in a running patch. The timing and potential latency associated with saving/loading make them unsuitable for dynamic context sharing.
- **Expanders & Messages:** VCV Rack provides an official API for direct communication between *adjacent* modules.[4] Expanders allow one module to directly access the parameters, ports, or even internal variables of its immediate neighbors, while the Message system offers a safer, double-buffered approach with a one-frame latency.[4] This system is effective for tightly coupled modules (e.g., a main module and its dedicated expander [13]), but the strict adjacency requirement severely limits its applicability for general-purpose communication across a patch.[4]

The limitations of these existing methods become apparent when developers attempt to implement more sophisticated interactions. Sharing complex state information (like the settings of a multi-stage envelope across different modules), synchronizing parameters between non-adjacent modules, transmitting large datasets dynamically (e.g., sample metadata for a sampler controlled by a separate browser module), or establishing dynamic routing based on patch context are scenarios not well-served by the current tools. This necessitates workarounds involving complex CV/Gate encoding schemes, reliance on external tools like OSC [15], or plugin-specific internal communication methods [17], leading to non-standard, often brittle solutions.

The very proposal of a Model Context Protocol stems from this observable gap in VCV Rack's current inter-module communication capabilities. Developers seeking to implement complex interactions, such as sharing detailed sequence metadata or synchronizing intricate states between non-adjacent modules, encounter limitations with the established CV/Gate and Expander paradigms [User Query Point 2]. This indicates a need within the development community for a more sophisticated, standardized data exchange mechanism. Furthermore, the focus of an MCP is distinct from low-level signal transfer; it targets the exchange of higher-level *information* or *context*, differentiating it from core signal path communication methods like CV and

Gate.[2]

1.3. Report Objectives and Structure

This report aims to provide a comprehensive analysis of the feasibility and design considerations for implementing an MCP within VCV Rack 2. The objectives are:

1. Analyze the limitations of current VCV Rack 2 communication mechanisms for complex data sharing.
2. Survey existing community practices, unofficial protocols, and prior art within VCV Rack for advanced module interactions.
3. Investigate relevant protocols and standards from other software and hardware modular environments.
4. Explore the VCV Rack 2 SDK and technical considerations for implementing an MCP, including architecture, data serialization, discovery, API design, performance, and thread safety.
5. Propose a potential framework and API structure for a standardized MCP.

The report is structured as follows:

- **Section 2:** Analyzes the capabilities and limitations of existing VCV Rack 2 communication mechanisms.
- **Section 3:** Surveys community-driven solutions, interaction patterns, and discussions related to advanced data sharing in VCV Rack.
- **Section 4:** Examines communication standards and protocols in other relevant modular synthesizer environments.
- **Section 5:** Delves into the technical design considerations crucial for developing a robust and efficient MCP.
- **Section 6:** Outlines a proposed framework and API for the MCP based on the preceding analysis.
- **Section 7:** Provides concluding recommendations and discusses future directions.

This report is intended for technically proficient VCV Rack module developers, plugin architects, and potentially the VCV Rack core development team, providing a foundation for discussion and potential implementation of a standardized context-sharing protocol.

2. Analysis of Existing VCV Rack 2 Communication Mechanisms

A thorough understanding of VCV Rack 2's existing inter-module communication capabilities is essential before proposing a new protocol. These mechanisms form the

foundation upon which modules interact and define the baseline against which an MCP's necessity and design should be evaluated.

2.1. Core Signal Path: CV, Gate, and Polyphony

The primary method of real-time interaction mirrors hardware modular synthesis: analog voltage signals transmitted via virtual patch cables.

- **Control Voltage (CV):** Typically ranges from -10V to +10V, although common usage often involves +/-5V or 0-10V ranges.[2] Pitch control adheres predominantly to the 1 Volt per Octave (1V/Oct) standard, allowing for precise musical control.[2] Other parameters like filter cutoff or VCA level are modulated using CV, often mapped across a specific voltage range.[6]
- **Gate and Trigger Signals:** These are binary signals representing on/off states. Gates typically remain high (e.g., +5V or higher) for the duration of an event (like a key press), while Triggers are brief pulses used to initiate events.[2] They are fundamental for sequencing, envelope generation, and synchronization.
- **Polyphony:** VCV Rack extends the basic CV/Gate concept with polyphony, allowing a single cable to carry multiple independent channels (up to 16) of CV or Gate signals.[4] Modules access these channels using indexing.[8]

Strengths: These mechanisms provide the standard, low-latency, real-time control essential for modular synthesis. They are well-understood by users and developers familiar with modular concepts.
Weaknesses for MCP: The fundamental limitation is data capacity and structure. Each polyphonic channel carries only a single floating-point voltage value (float).8 This is insufficient for transmitting structured data (like text strings, complex state objects, or metadata) or large binary blobs. Communication requires explicit patching by the user, making dynamic or context-aware routing difficult without additional complex modules.

2.2. Module State Persistence: Serialization and Patch Storage

VCV Rack provides mechanisms for modules to save and restore their internal state, ensuring patches can be reliably saved and loaded.

- **JSON Serialization (dataToJson/dataFromJson):** Rack automatically saves standard parameter values (knobs, switches). For additional internal state (e.g., sequence data, mode settings, buffer contents), modules must override dataToJson() to serialize their state into a JSON object and dataFromJson() to restore it when the patch loads.[4] The Jansson library is used for JSON manipulation.[4] Rack ensures these methods are never called concurrently with the module's process() method, providing basic thread safety for this specific operation.[4]
- **Patch Storage:** For large data (>100KB, e.g., audio samples, wavetables),

serializing directly into the patch JSON can cause significant UI lag during autosave.[4] The createPatchStorageDirectory() method allows a module to create a dedicated directory within the patch file structure.[4] Modules can then read/write arbitrary files in this directory. When the patch is saved, Rack compresses these directories into the .vcv file.[4] File I/O should ideally happen off the main audio thread to prevent glitches.[4]

- **Plugin Settings:** For settings shared across all instances of modules within a plugin (e.g., global themes, API keys), developers can implement settingsToJson() and settingsFromJson() in plugin.cpp. This data is stored in <Rack user folder>/settings.json.[4]

Strengths: These methods provide robust mechanisms for module state persistence, handling both small configuration data and large asset files effectively.
Weaknesses for MCP: These systems are designed for saving and loading, not real-time inter-module communication. The latency involved, especially with file I/O for patch storage or the periodic nature of plugin settings saves, makes them unsuitable for dynamic context updates during performance. While serialization handles data structure, it doesn't facilitate live exchange between modules. This mismatch in execution context and latency profile underscores why serialization mechanisms cannot fulfill the role of a real-time MCP.

2.3. Adjacent Communication: Expanders and Messages

VCV Rack offers a specific API for direct communication between modules that are physically placed next to each other in the rack.

- **Expander Mechanism:** A module can query its left and right neighbors using getLeftExpander() and getRightExpander().[4] If an adjacent module exists and belongs to an expected Model, the querying module can directly access the adjacent module's public members, including parameters (getParam()), inputs (getInput()), outputs (getOutput()), and lights (getLight()).[4] The expander module can react to connection changes by overriding onExpanderChange().[4]
- **Message System:** To mitigate potential timing and thread-safety issues associated with direct access (especially if modules directly manipulate each other's internal state variables), the Message system provides a synchronized, double-buffered communication channel with a fixed one-engine-frame latency.[4] The sending module writes data to the receiving module's producerMessage buffer (which must be allocated by the receiver along with an identical consumerMessage buffer) and calls requestMessageFlip().[4] The engine swaps the buffers at the start of the next frame, and the receiving module reads the data from its consumerMessage buffer during its process() call.[4] Messages can be simple types or complex structs.[4]

Strengths: This system provides an officially supported, low-latency (Messages) or zero-latency (direct access) communication path ideal for tightly coupled modules, such as main modules extending their functionality via dedicated expanders (e.g., MindMeld MixMaster/AuxSpander 13, EQMaster/EqSpander 14).
Weaknesses for MCP: The absolute requirement for physical adjacency is the critical limitation.4 This prevents communication between modules separated by others or in dynamically changing layouts. It cannot serve as a general-purpose protocol for arbitrary module interaction across the patch. Furthermore, direct access requires careful implementation to avoid race conditions or timing inconsistencies 4, and the Message system requires explicit setup and handling by both participating modules. This adjacency bottleneck is arguably the primary technical justification for investigating a more general MCP.

2.4. Other API Elements (Lights, Widgets, Events)

Other elements of the VCV Rack API facilitate interaction, but not typically for general inter-module data transfer:

- **Lights:** Configured via configLight() [10] and controlled via setBrightness() [9], lights provide visual feedback to the user. They are outputs, not data channels.
- **Widgets:** Knobs, buttons, sliders (configParam, configButton, configSwitch) [4] and custom widgets [4] are the primary means of user interaction with a module. They manage parameters or trigger actions within their own module.
- **Module Events:** Methods like onReset(), onRandomize(), onSampleRateChange() allow modules to react to engine events or user actions directed at that specific module.[4] They are not designed for module-to-module communication.

These elements are crucial for module usability but do not offer mechanisms for the type of structured, potentially non-adjacent data sharing an MCP would enable.

The analysis reveals a clear gap: while VCV Rack provides excellent mechanisms for signal-level control (CV/Gate), state persistence (Serialization/Storage), and tightly coupled adjacent communication (Expanders/Messages), it lacks a standardized, built-in method for arbitrary, non-adjacent modules to discover each other and exchange complex contextual information in real-time. While functions like APP->engine->getModule(moduleId) [26] exist, the practical difficulty for one module to reliably obtain the correct moduleId of another, unrelated module without user intervention or non-standard techniques [27] highlights the absence of a crucial discovery and addressing layer needed for generalized communication. This missing piece is precisely what an MCP aims to provide.

3. Survey of Current Practices and Prior Art in VCV Rack

Despite the lack of an official, generalized inter-module communication protocol

beyond adjacent expanders, the VCV Rack developer community has explored various techniques and built modules demonstrating advanced interaction patterns. Examining these practices reveals common needs, potential solutions, and challenges inherent in the VCV Rack environment.

3.1. Community-Driven Protocols & Concepts

Several approaches have been discussed or implemented by the community to bridge the communication gap:

- **OSC (Open Sound Control):** Primarily used for communication *between* VCV Rack instances on different computers or between Rack and external software/hardware (e.g., TouchDesigner, Arduino, DAWs).[16] Modules like trowaSoft's cvOSCcv or TheModularMind's OSCelot facilitate this by translating between CV and OSC messages.[15] More ambitiously, community members have proposed using OSC *within* a single patch for general inter-module communication.[15] This involves defining OSC address spaces to target specific modules and parameters (e.g., /Rack/<slug>/<id>/param/<name>) [15] and necessitates common libraries for message parsing and creation.[15] The potential for point-to-point or broadcast messaging and its established data types make it an attractive option.[15]
- **Text-over-Cable ("Tipsy"):** An extensive community discussion explored encoding text or structured data (like JSON) into CV or audio signals sent over standard VCV patch cables.[15] Various encoding schemes were debated, from simple byte-to-float scaling (e.g., casting a byte to a 0-10V float) [18] to more complex ideas involving modem protocols [18] or using ASCII control characters for framing.[32] Key considerations included choosing a character encoding (UTF-8 preferred) [18], data framing (start/end markers, checksums) [32], and handling VCV Rack's signal constraints (e.g., NaN/INF suppression).[32] This led to the "TipsyProto" proof-of-concept implementation [32], demonstrating the feasibility of sending text messages this way, and modules like BASICally using it for debugging output to a TTY module.[34] While creative, this approach essentially treats CV cables as low-bandwidth serial data lines, potentially impacting performance and requiring careful encoding/decoding logic.
- **Scripting Modules:** Modules like ModScript Lune (using Lua) [35] and BASICally [34] empower users to write scripts that interact with the patch environment. These scripts can often read and write parameters of *other* modules by referencing their unique module ID.[34] Helper modules are sometimes provided to assist users in finding these IDs.[35] While offering immense flexibility for custom control and automation, these modules provide a user-driven scripting layer rather than a

standardized, programmatic protocol for direct module-to-module data exchange. They rely on introspection capabilities to function.

- **MIDI-based Communication:** Developers have used MIDI, often via VCV Rack's loopback MIDI driver [36], as a communication bus. This can involve standard MIDI mapping modules [37], custom protocols like sending JSON over serial to control Arduino hardware that then sends MIDI back to Rack [28], or using modules like MIDI-CAT [28] or MIDI-KIT [36] for advanced processing and routing. Proposals for using SysEx for detailed control have also been made, though recognized as complex to standardize.[40]

3.2. Advanced Module Interaction Patterns

Beyond specific protocol proposals, certain module designs showcase sophisticated interaction techniques:

- **Stoermelder Modules (MAP, CV-MAP, STRIP, etc.):** Ben Stoermelder's modules are renowned for their meta-control capabilities. MAP allows mapping UI controls (knobs, buttons) from one module to parameters on others, while CV-MAP allows CV control of parameters, even those without dedicated CV inputs.[28] STRIP saves and recalls the state of entire rows of modules.[41] These modules demonstrate powerful introspection – the ability to find other modules, query their parameters, and manipulate their values programmatically.[28] They primarily focus on controlling existing parameters rather than exchanging arbitrary contextual data.[41]
- **MindMeld Internal Communication:** The MindMeld suite (MixMaster, EQMaster, AuxSpander) exhibits seamless communication between non-adjacent modules, such as EQMaster automatically fetching track names and colors from a linked MixMaster.[13] Community discussion and developer comments indicate this is achieved using a plugin-internal mechanism, likely a global class acting as a singleton broker or message bus, rather than standard VCV cables or expanders.[17] This serves as a prime example of successful, complex context sharing *within* a single plugin's ecosystem.
- **Singleton/Broker Patterns:** The MindMeld example aligns with broader discussions about using singleton patterns within a plugin. Developers have used plugin-wide singletons to manage shared resources like MIDI device claims (ensuring only one module instance uses a specific hardware port) [29] or potentially to synchronize settings like themes across all module instances of that plugin.[27] Thread-safety is a key concern when implementing such global shared state.[46]
- **Interface Casting (dynamic_cast):** A recurring pattern discussed in developer forums involves using C++ dynamic_cast.[15] One module can iterate through other

modules in the patch (or check the module at the other end of a cable), attempt to cast the Module* pointer to a specific custom interface type (e.g., IMyCommunicationInterface*), and if successful, use the methods defined by that interface to communicate.[15] This allows for capability checking and establishing communication channels without prior knowledge of the exact module type, provided modules adhere to the defined interface.

- **Non-Adjacent Communication Examples (Code):** While many complex modules achieve interactions through clever CV/Gate patching [50], some likely employ internal mechanisms similar to the singleton or interface patterns discussed above. Analyzing open-source examples like ImpromptuModular [50], Nonlinearcircuits [51], Admiral [52], or Mahlen Morris's BASICally [34] might reveal specific implementations, although these are often plugin-internal rather than standardized protocols.

3.3. Community Discussions & Feature Requests

Numerous threads on the VCV Community forum reflect the ongoing interest in and challenges of inter-module communication.[15] Common themes include:

- Desire for scripting capabilities to control other modules.[28]
- Need to control parameters without dedicated CV inputs or even visible widgets (e.g., menu options).[41]
- Interest in sending structured data like text or JSON between modules.[15]
- Requests for API enhancements, such as a standard way to iterate through all modules in the patch (reportedly planned for Rack v2 [27]) or programmatic access to context menu actions (considered difficult and potentially unsafe [41]).

The independent emergence of similar solutions and recurring discussions across the community strongly suggests a persistent, unmet need for standardized ways to achieve complex, non-adjacent module interactions. These efforts point towards desirable characteristics for an MCP, including mechanisms for module/service discovery, support for arbitrary structured data, and operation independent of physical rack placement.

Furthermore, many of these advanced patterns, particularly meta-controllers like Stoermelder's MAP or scripting modules, fundamentally rely on introspection: the ability to programmatically discover other modules, query their properties (like parameters and types), and potentially interact with them. This implies that any successful MCP must either provide robust introspection capabilities or be built upon existing or future introspection features within the VCV Rack API.

Finally, observing that the most successful existing examples of complex communication (like MindMeld's suite [17]) operate *within* a single plugin highlights a key challenge. Defining a standard protocol for *cross-plugin* communication introduces significant hurdles related to API stability, ensuring agreement on data formats between different developers, versioning, and maintaining compatibility over time – complexities not faced when communication is entirely internal to one developer's set of modules.

4. Protocols and Standards in Other Modular Environments

Examining how other modular synthesizer environments, both software and hardware, handle communication can provide valuable context and inspiration for designing an MCP for VCV Rack 2.

4.1. Software Modular Systems (e.g., Reaktor Blocks)

Native Instruments Reaktor, particularly with its Blocks framework, offers a relevant software modular comparison.

- **Standardization:** Reaktor Blocks introduced a level of standardization aimed at improving interoperability between modules (called "Blocks") created by NI and third parties.[54] This standardization covers:
  - **Signal Conventions:** Standardized scaling for signals, differentiating between audio and control rates.[54] This ensures that connecting outputs to inputs generally produces musically meaningful (or at least predictable) results without requiring constant attenuation or scaling.
  - **Structural Conventions:** Prescribed ways to structure Blocks internally, dividing work into Panel and Structure layers.[54]
  - **Panel Sizes:** Standardized panel dimensions, akin to hardware Eurorack width/height constraints.[54]
- **Interconnection:** The goal was to allow users to freely interconnect Blocks in real-time, facilitating experimentation and "happy accidents," much like hardware patching.[54] This suggests internal handling of signal types and rates to ensure compatibility at connection points.
- **Beyond Cables:** While visual patching is central, software environments can implement communication mechanisms beyond explicit cables. Reaktor allows complex internal structures and potentially hidden connections or data sharing between modules, enabling sophisticated interactions.[54]
- **Limitations:** Early versions of Blocks faced limitations, such as modules being primarily monophonic, which might hinder certain polyphonic patching approaches common in VCV Rack.[54]

The key takeaway from Reaktor Blocks is the deliberate effort to standardize communication conventions (scaling, rates, signals) to foster a plug-and-play ecosystem, even for modules from different developers.[54] This directly parallels the goal of an MCP in VCV Rack for higher-level data exchange.

4.2. Hardware Modular (Eurorack)

Eurorack, the dominant hardware modular format, provides the foundational paradigm for VCV Rack. Its communication standards are primarily physical and electrical.

- **Core Standards:**
  - **Physical:** 3U height, variable width in HP (Horizontal Pitch, 5.08mm increments).[2]
  - **Power:** Standardized +/-12V and optional +5V DC power delivered via ribbon cables (10 or 16-pin).[3]
  - **Signal:** Analog voltages transmitted via 3.5mm mono patch cables.[3]
    - *CV:* Nominally +/-5V or 0-10V, with 1V/Octave as the common pitch standard.[2]
    - *Gate/Trigger:* Typically 0V (off) to +5V or higher (on).[5]
    - *Audio:* Typically around 10V peak-to-peak, significantly hotter than line level.[19]
- **Communication Beyond CV/Gate:**
  - **MIDI Integration:** MIDI is a digital protocol, fundamentally different from Eurorack's analog CV/Gate.[2] To integrate MIDI devices (keyboards, sequencers, DAWs), specialized MIDI-to-CV converter modules are essential.[5] These modules translate MIDI messages (Note On/Off, Velocity, CC, Clock) into corresponding analog voltages (Pitch CV, Gate/Trigger, auxiliary CVs, clock pulses).[59] Conversely, CV-to-MIDI modules exist but are less common.[59]
  - **Internal Bus Systems:** The Eurorack power bus connector sometimes includes pins designated for carrying CV and Gate signals across connected modules, intended for specific standardized functions.[5] However, most modules do not utilize these bus connections for signal transfer, relying instead on front-panel patching.[5] This concept is analogous to VCV Rack's Expander API but sees limited practical use in hardware for general communication.
  - **Audio as Control:** A unique characteristic of analog modular is that audio signals and control voltages share the same medium (voltage over cables). This allows for creative patching where audio-rate signals (like VCO outputs) can modulate parameters typically controlled by slower CVs (like filter cutoff), and LFOs can be used as audio sources if their frequency is increased into

the audible range.[6] This flexibility comes from the lack of strict distinction between signal types, but it doesn't facilitate structured data transfer.

The Eurorack world emphasizes a minimal set of standards focused on electrical compatibility and basic control signals.[61] Complex interactions are achieved through explicit patching and the creative use of voltage manipulation. The need for dedicated MIDI-to-CV converters clearly illustrates the challenge of bridging different communication paradigms.[59]

4.3. Lessons Learned and Applicability to VCV Rack MCP

Several key points emerge from comparing these environments:

- **Value of Standardization:** Both Eurorack's basic voltage standards and Reaktor Blocks' more detailed signal conventions demonstrate that clear standards are fundamental to enabling interoperability and fostering a diverse ecosystem.[2] An MCP would provide this standardization for complex data exchange in VCV Rack, lowering barriers for developers creating cooperating modules.
- **Abstraction in Software:** Software environments like Reaktor can abstract away low-level implementation details behind standardized interfaces or automatic connections [54], offering potentially cleaner workflows than the explicit, manual patching required for all connections in hardware.[3] An MCP could similarly abstract the complexities of data serialization, transport, and discovery behind a simpler API for VCV Rack developers.
- **Bridging Paradigms:** The necessity of MIDI-to-CV conversion in Eurorack highlights that bridging mechanisms are often required when systems with different communication methods need to interact.[5] An MCP design should consider if and how it needs to interface with existing VCV Rack signals (CV/Gate) or external protocols (OSC, MIDI), potentially requiring dedicated bridging modules or functions.
- **Potential of Software:** Software platforms inherently offer possibilities for more sophisticated, structured communication protocols that go far beyond the simple voltage standards feasible in hardware. VCV Rack, being software, is well-positioned to leverage this potential through a well-designed MCP.

5. Technical Design Considerations for an MCP

Implementing a Model Context Protocol (MCP) in VCV Rack 2 requires careful consideration of various technical aspects to ensure it is robust, efficient, flexible, and integrates well with the existing architecture.

5.1. Protocol Architecture & Communication Patterns

The fundamental architecture dictates how modules discover each other and exchange information. Several patterns are viable:

- **Central Broker/Registry (Singleton):** This approach involves a single, globally accessible object within the Rack process.[46] Modules register the contexts they provide or subscribe to contexts they need with this broker. The broker manages discovery (mapping topics/contexts to provider modules) and potentially routes messages.
  - *Pros:* Centralized discovery simplifies lookup for modules. Managing global context or state might be easier. Successful examples exist within plugins like MindMeld [17] and for managing resources like MIDI devices.[29]
  - *Cons:* The broker becomes a potential performance bottleneck if heavily used. It's a single point of failure. Implementing a truly thread-safe singleton in C++ across potentially different module boundaries (if the broker lives outside individual plugins) can be complex, especially concerning initialization order and lifetime management.[48] Careful locking is required to prevent race conditions.[64]
- **Peer-to-Peer Discovery (Interface-based):** Modules could discover each other directly. This typically involves iterating through all modules currently in the patch (using APP->engine->getModuleIds() and APP->engine->getModule() [26], possibly facilitated by future API improvements [27]) and using C++ dynamic_cast to check if a module implements a specific, predefined MCP interface (e.g., IMCPProvider).[15] Once discovered, communication could be direct via interface methods or through established channels.
  - *Pros:* Decentralized architecture avoids a single bottleneck. Potentially more scalable for a very large number of communicating modules.
  - *Cons:* Discovery can be computationally expensive, especially if performed frequently by many modules (iterating through potentially hundreds of modules). Requires strict adherence to the defined interface. Managing connections and communication channels becomes the responsibility of individual modules.
- **Publish/Subscribe (Pub/Sub):** A common messaging pattern where modules "publish" messages or context updates tagged with a topic (e.g., "global_key_signature", "sequencer_1_state"). Other modules "subscribe" to topics of interest and receive relevant updates, often via callbacks. This pattern inherently decouples publishers from subscribers.
  - *Pros:* Highly flexible, promotes loose coupling between modules. Excellent for event-driven architectures where modules react to changes in context.

    - *Cons:* Requires a mechanism (either a central broker or a distributed protocol) to manage topics and route messages. Can introduce latency depending on the routing implementation.
- **Direct Addressing:** Once a module discovers the unique ID [66] of a target module (perhaps via a broker or user configuration), it could potentially send messages directly to that module's MCP endpoint.
    - *Pros:* Conceptually simple for point-to-point communication once discovery is solved.
    - *Cons:* Tightly couples sender and receiver. Requires a robust discovery mechanism as a prerequisite. Less flexible than Pub/Sub for broadcasting context changes.

A hybrid approach might be optimal, using a lightweight broker primarily for discovery and topic registration, while actual data transfer could occur via more direct channels (like thread-safe queues or interface calls) once modules have found each other.

5.2. Data Representation and Serialization

The MCP needs to transmit potentially complex data structures. Choosing the right format is critical for performance and usability.

- **Data Structures:** The MCP should ideally support transmitting arbitrary data, but defining standard structures for common use cases (e.g., musical scale/key, sequence transport state, parameter mappings, file metadata) could improve interoperability.
- **Serialization Formats:**
    - *JSON:* Used internally by Rack for saving patches.[4] Human-readable and well-supported via the Jansson library.[4] However, text parsing is relatively slow, and JSON can be verbose, making it potentially unsuitable for frequent real-time updates.[67] High-performance C libraries like yyjson exist but are not Rack's default.[67]
    - *OSC (Open Sound Control):* A well-defined binary protocol with support for various data types (int, float, string, blob, timestamp) and a flexible addressing scheme.[15] Already used for external communication in Rack.[16] Requires encoding/decoding libraries, adding some overhead.
    - *MessagePack:* A binary serialization format designed as a more efficient alternative to JSON.[67] Offers significant speed improvements and smaller message sizes.[67] Good cross-language library support makes it versatile.[68]
    - *FlatBuffers / Flexbuffers:* Binary formats developed by Google, known for very fast read access (potentially zero-copy deserialization).[67] FlatBuffers typically requires a schema definition compiled beforehand, while Flexbuffers offers

more flexibility.[67] Might be overkill if zero-copy access isn't strictly needed, and cross-language support might be less mature than MessagePack.[67]
    - *Custom Binary:* Offers maximum potential efficiency but requires defining the entire format, writing serialization/deserialization code, and maintaining it. High risk of creating incompatible silos if not universally adopted.

Performance is paramount in a real-time audio environment. Benchmarks consistently show binary formats like MessagePack and FlatBuffers/Flexbuffers significantly outperform text-based formats like JSON and YAML, especially for writing operations.[67] The trade-off often involves sacrificing human readability and potentially requiring schema definitions for optimal performance. Given VCV Rack's real-time constraints, binary formats appear more suitable for the core data transport of an MCP, although providing JSON as an option for less performance-critical contexts or debugging might be beneficial.

Table 5.1: Comparison of Potential MCP Data Serialization Formats

Feature	JSON (yyjson)	OSC	MessagePack	FlatBuffers/ Flexbuffers	Custom Binary
Speed (Read)	Moderate [67]	Moderate	Fast [67]	Very Fast [67]	Potentially Fastest
Speed (Write)	Moderate [67]	Moderate	Very Fast [67]	Fast [67]	Potentially Fastest
Size Efficiency	Low	Moderate	High	High	Potentially Highest
Schema Required	No	No	No	Yes/Optional [68]	Yes (Implicit)
Human Readable	Yes	Partially	No	No	No
Cross-Language	Excellent	Good	Very Good [68]	Good	Poor (by definition)
VCV	Internal Use	Existing Libs	Libraries	Libraries	Requires Dev

Context	[4]	[15]	Available	Available	

This comparison highlights the fundamental trade-off: binary formats like MessagePack and Flexbuffers offer superior performance crucial for real-time use, while JSON provides readability and existing integration but at a performance cost. OSC offers a balance with good features but moderate performance.

5.3. Module Discovery and Addressing

Modules need a reliable way to find the specific context or module they want to interact with.

- **Identifiers:** Module instances have unique 64-bit IDs assigned by the engine (getId() [66], accessible via APP->engine->getModule(id) [26]). Plugins and modules also have unique slugs defined in plugin.json.[9]
- **Discovery Mechanisms:**
  - *Broker Registry:* A central broker could map human-readable topic strings or context types to the Module ID(s) of the provider(s).
  - *Interface Query:* Using dynamic_cast during a scan of modules allows checking capabilities without knowing specific module types.[15]
  - *OSC-style Addressing:* A hierarchical path like /Rack/<plugin_slug>/<module_id>/context/<context_name> could uniquely identify a context endpoint.[15]
  - *User Configuration:* Allow users to manually link modules via context menus or dedicated mapping modules (similar to Stoermelder MAP [42]). Less dynamic but potentially simpler for some cases.

Designing an efficient and flexible discovery mechanism is a core challenge. Relying solely on Module IDs requires a mapping layer, while full patch scans can be resource-intensive. A combination of a broker for topic/service registration and interface checking for capability verification seems promising.

5.4. API Design

The C++ API presented to module developers should be clear, concise, and hide unnecessary implementation details.

- **Core Interfaces:** Define abstract base classes or use C++ concepts for key roles:
  - IMCPBroker (if using a broker architecture)
  - IMCPProvider / IMCPPublisher: Interface for modules offering context.
  - IMCPSubscriber: Interface or mechanism for modules consuming context.
  - MCPMessage: Structure or class representing the data payload, potentially

including metadata (sender ID, topic, timestamp, format).

- **Key Functions:**
  - registerContext(topic, provider_interface)
  - findContextProviders(topic) -> returns list of provider info/IDs
  - subscribe(topic, callback_function) or subscribe(topic, subscriber_interface)
  - publish(topic, MCPMessage)
  - query(topic) -> returns current state (if applicable)
  - unregisterContext(topic)
  - unsubscribe(topic)
- **Asynchronicity:** Consider if operations like publishing or querying should be asynchronous to avoid blocking, especially if they involve potential cross-thread communication.

5.5. Performance and Real-time Safety

Performance is critical. The MCP must minimize its impact on the audio processing thread (Module::process()).

- **Audio Thread Constraints:** Avoid any operation that could block or take unpredictable time on the audio thread. This includes file I/O, network communication, complex serialization/deserialization, dynamic memory allocation (if possible), and waiting on locks contended by lower-priority threads.[20]
- **Benchmarking:** Performance of different MCP implementation strategies (e.g., broker vs. P2P, different serialization formats) should be benchmarked within the VCV Rack environment under realistic load conditions.[72] Comparing against existing mechanisms like Expander messages is important.[4]
- **Threading Model:** VCV Rack allows users to configure the number of engine threads.[64] The MCP implementation must function correctly and efficiently regardless of the thread count. Adding threads introduces synchronization overhead, which can sometimes negate performance gains or even worsen performance for certain workloads or under light load.[71] Using real-time thread priority requires caution, especially on Linux where it has reportedly caused issues.[64]
- **Strategy:** The recommended strategy is to perform MCP communication logic (especially complex serialization or discovery) on non-audio threads (e.g., a dedicated MCP thread or the GUI thread) whenever possible. Data transfer to/from the audio thread should use highly efficient, thread-safe mechanisms like lock-free queues.[47]

5.6. Synchronization and Thread Safety

This is arguably the most critical and complex aspect due to VCV Rack's multi-threaded nature.[4] Any shared data or communication channel accessed by multiple threads (audio, GUI, MCP worker) *must* be protected.

- **Primitives:**
  - std::mutex: Standard C++ mutexes provide exclusive access to shared resources. Essential for protecting complex data structures in the broker or shared state.[12] Must be used carefully to avoid deadlocks and performance bottlenecks caused by contention.
  - std::atomic: Suitable for simple flags, counters, or pointers that need to be updated or read across threads without full locking.[21]
  - rack::dsp::RingBuffer: VCV Rack provides a lock-free single-producer, single-consumer ring buffer, ideal for passing data between the audio thread and another thread (like GUI or an MCP worker).[47]
  - std::condition_variable: Useful for signaling between threads (e.g., notifying a worker thread that new data is available).[29] Requires careful use with mutexes to avoid lost wakeups or race conditions. Potential issues with reference invalidation if queue data is reallocated while waiting.[29]
- **Design:** The MCP API itself must be designed with thread safety in mind. Functions intended to be called from the audio thread must be non-blocking and real-time safe. Functions interacting with shared state (like a broker registry) must use appropriate locking. Clear documentation on the thread-safety guarantees (or lack thereof) for each API call is essential. Implementing thread-safe singletons requires careful attention to initialization and destruction order.[48]

Incorrect handling of concurrency is a primary source of instability (crashes, hangs) and data corruption in multi-threaded applications. Meticulous design and testing are required.

5.7. Versioning and Compatibility

An MCP intended for cross-plugin communication must handle evolution gracefully.

- **Protocol Versioning:** The protocol itself should include a version number (e.g., in message headers or during initial handshake).[32] This allows modules to check for compatibility and potentially adapt their behavior.
- **API Stability:** Once standardized, the core MCP API should remain stable across VCV Rack versions as much as possible. Changes should be backward-compatible or provide clear deprecation warnings and migration paths.
- **Data Format Evolution:** If data formats for specific contexts evolve, versioning

within the data itself (e.g., a version field in a JSON object) might be necessary.

5.8. SDK Integration

The MCP should feel like a natural part of the VCV Rack development experience.

- **Leverage Existing APIs:** Utilize engine::Engine [26] for accessing modules, engine::Module [66] as the base for participants, and the event system (onAdd, onRemove, etc.) [4] for lifecycle management (registration/unregistration).
- **Lifecycle:** Determine the appropriate points in the module lifecycle to initialize, register, unregister, and tear down MCP connections (e.g., constructor/destructor vs. onAdd/onRemove [21]). onAdd/onRemove are generally safer as the module is fully integrated into the engine at that point.[10]
- **Distribution:** Consider whether the MCP implementation should be part of the core Rack SDK or distributed as a separate helper library. Integration into the SDK ensures universal availability.

The design process must acknowledge that successful communication patterns have already emerged within VCV Rack (Expanders, internal brokers, thread-safe queues, interface casting).[4] An effective MCP should learn from these patterns, adopting or adapting proven techniques where suitable, rather than attempting to reinvent solutions in isolation.

6. Proposed MCP Framework Outline

Based on the analysis of VCV Rack's architecture, existing community practices, and technical considerations, this section outlines a potential framework for a Model Context Protocol (MCP). This proposal aims to balance flexibility, performance, and ease of use.

6.1. Recommended Architecture: Hybrid Broker/Interface Model

A hybrid architecture appears most promising, leveraging the strengths of different patterns:

1. **Singleton Broker for Discovery and Topic Management:**
    - A thread-safe singleton object, accessible globally within the Rack process (e.g., via APP->getMCPBroker()).
    - **Responsibilities:**
        - Manages a registry of available "topics" or context types.
        - Allows modules (IMCPProvider implementers) to register themselves as providers for specific topics.
        - Allows modules (IMCPSubscriber implementers) to query available topics

and find provider module IDs or interfaces.
    - **Implementation:** Requires careful thread-safe implementation using mutexes for registry access.[46] Provides a central point for discovery.
2. **Interface-Based Communication (Post-Discovery):**
    - Once a subscriber discovers a provider (via the broker), communication can potentially become more direct.
    - Providers implement a specific C++ interface (e.g., IMCPProvider_V1) defining methods for querying state or receiving direct requests.
    - Subscribers obtain a pointer (or weak pointer for safety) to the provider module (using the ID from the broker and APP->engine->getModule()) and dynamic_cast it to the provider interface.[15]
    - Direct interface calls can be used for synchronous queries (if real-time safe) or potentially for setting up dedicated communication channels (like dsp::RingBuffers).
3. **Publish/Subscribe for Updates:**
    - For broadcasting context changes, a Pub/Sub mechanism is ideal.
    - Providers publish updates for a topic via the broker (broker->publish(topic, message)).
    - Subscribers register callbacks or implement a subscriber interface (IMCPSubscriber_V1) with the broker (broker->subscribe(topic, subscriber)).
    - The broker dispatches published messages to all registered subscribers for that topic. Dispatching should likely occur on a non-audio thread to avoid blocking, with subscribers using thread-safe queues to receive data if needed in process().

This hybrid model uses the broker for efficient, centralized discovery and pub/sub eventing, while allowing for potentially more performant direct interaction via interfaces once modules are aware of each other.

6.2. Core API Definition

The following C++ interfaces and structures illustrate the proposed API (Version 1).

C++

```
#include <string>
#include <vector>
#include <functional>
```

```cpp
#include <memory> // For std::shared_ptr, std::weak_ptr
#include <atomic> // For potential internal use

namespace rack {
namespace mcp {

// Forward declarations
class IMCPProvider_V1;
class IMCPSubscriber_V1;
struct MCPMessage_V1;

// --- Message Structure ---
struct MCPMessage_V1 {
    std::string topic;
    int64_t senderModuleId = -1;
    // Consider adding timestamp, data format identifier (e.g., MIME type)
    std::shared_ptr<void> data; // Use shared_ptr for lifetime management
    size_t dataSize = 0;
    std::string dataFormat = "application/octet-stream"; // Default, e.g., "application/json",
"application/msgpack"

    // Helper to get typed data (requires knowledge of expected type)
    template<typename T>
   std::shared_ptr<T> getTypedData() const {
        return std::static_pointer_cast<T>(data);
    }
};

// --- Interfaces ---
class IMCPProvider_V1 {
public:
    virtual ~IMCPProvider_V1() = default;
    // Optional: Method for synchronous query, if applicable and safe
    // virtual MCPMessage_V1 queryContext(const std::string& topic) = 0;
};

class IMCPSubscriber_V1 {
public:
    virtual ~IMCPSubscriber_V1() = default;
    // Callback method invoked when a subscribed topic receives an update
    virtual void onMCPMessage(const MCPMessage_V1& message) = 0;
```

```
};

// --- Broker Interface ---
class IMCPBroker {
public:
    virtual ~IMCPBroker() = default;

    // Provider functions
    virtual bool registerContext(const std::string& topic, int64_t moduleId,
std::weak_ptr<IMCPProvider_V1> provider) = 0;
    virtual bool unregisterContext(const std::string& topic, int64_t moduleId) = 0;
    virtual void publish(const MCPMessage_V1& message) = 0;

    // Subscriber functions
    virtual bool subscribe(const std::string& topic, int64_t moduleId, std::weak_ptr<IMCPSubscriber_V1>
subscriber) = 0;
    virtual bool unsubscribe(const std::string& topic, int64_t moduleId) = 0;
    virtual bool unsubscribeAll(int64_t moduleId) = 0; // For module removal cleanup

    // Discovery functions
    virtual std::vector<std::string> getAvailableTopics() = 0;
    virtual std::vector<int64_t> findProviders(const std::string& topic) = 0;
    // Optional: Get provider interface directly (use with caution regarding lifetimes)
    // virtual std::weak_ptr<IMCPProvider_V1> getProviderInterface(int64_t moduleId) = 0;
};

// --- Accessor Function ---
// Needs to be implemented in the core Rack SDK or a helper library
// extern IMCPBroker* getMCPBroker();

} // namespace mcp
} // namespace rack
```

Table 6.1: Proposed MCP API Function Summary (V1)

Function Signature (C++)	Class/Interface	Purpose/Description	Key Parameters	Thread Safety Notes

bool registerContext(topic, moduleId, provider)	IMCPBroker	Registers a module as a provider for a specific topic.	Topic name, Module ID, Weak ptr to provider	Must be thread-safe (likely requires internal locking in Broker). Callable from non-audio thread (e.g., onAdd).
bool unregisterContext(topic, moduleId)	IMCPBroker	Unregisters a module as a provider.	Topic name, Module ID	Must be thread-safe. Callable from non-audio thread (e.g., onRemove).
void publish(message)	IMCPBroker	Publishes a message/update to a topic.	MCPMessage_V1 object	Must be thread-safe. Message dispatch to subscribers should happen asynchronously off the calling thread (especially if called from audio thread).
bool subscribe(topic, moduleId, subscriber)	IMCPBroker	Subscribes a module to receive updates for a topic.	Topic name, Module ID, Weak ptr to subscriber	Must be thread-safe. Callable from non-audio thread.
bool unsubscribe(topic, moduleId)	IMCPBroker	Unsubscribes a module from a specific topic.	Topic name, Module ID	Must be thread-safe. Callable from non-audio thread.
bool unsubscribeAll(	IMCPBroker	Unsubscribes a module from all topics (cleanup	Module ID	Must be thread-safe. Callable from

moduleId)		on module removal).		non-audio thread (e.g., onRemove).
std::vector<std::string> getAvailableTopics()	IMCPBroker	Returns a list of currently registered topics.	-	Must be thread-safe (read access to registry). Callable from any thread.
std::vector<int64_t> findProviders(topic)	IMCPBroker	Returns a list of Module IDs providing a specific topic.	Topic name	Must be thread-safe (read access to registry). Callable from any thread.
virtual void onMCPMessage(message)	IMCPSubscriber_V1	Callback function implemented by subscriber modules to receive messages.	MCPMessage_V1 object	Invoked by the Broker (likely on a worker thread). Implementation must be thread-safe if accessing shared module state. Use queues for audio thread.
template<typename T> std::shared_ptr<T> getTypedData() const	MCPMessage_V1	Helper to safely cast message data payload to the expected type.	Template type T	Safe to call from any thread.

6.3. Data Format Recommendation

- **Primary Format: MessagePack** [67] is recommended as the default serialization format (dataFormat = "application/msgpack") due to its balance of high performance (especially write speed, crucial for publishers), good compression, and lack of schema requirement, making it suitable for real-time exchange of

arbitrary data structures.

- **Secondary Format (Optional):** Support for **JSON** (dataFormat = "application/json") could be included for debugging purposes or less performance-sensitive contexts. Modules could potentially negotiate the format. The MCPMessage_V1 includes a dataFormat field to facilitate this.
- **Data Payload:** The data field uses std::shared_ptr<void> to hold arbitrary serialized data. The dataSize field is essential for interpreting the raw data. Publishers are responsible for serializing their data into a byte buffer managed by the shared_ptr, and subscribers are responsible for deserializing it based on the dataFormat and topic expectations.

6.4. Example Usage Patterns (Conceptual C++ Snippets)

C++

```
// --- Provider Module Example ---
#include "rack.hpp" // Includes VCV Rack headers
#include "mcp_api.hpp" // Assumed header for MCP interfaces
#include <msgpack.hpp> // Example using msgpack-c

struct MyContextProviderModule : rack::engine::Module, rack::mcp::IMCPProvider_V1 {
    std::string myContextData = "Initial State";
    int64_t myId = -1;
    std::weak_ptr<rack::mcp::IMCPProvider_V1> self_weak_ptr;

    MyContextProviderModule() {
        config(NUM_PARAMS, NUM_INPUTS, NUM_OUTPUTS, NUM_LIGHTS);
        //... other setup...
    }

    void onAdd(const rack::event::Add& e) override {
        rack::engine::Module::onAdd(e);
        myId = this->id;
        // Create a weak_ptr to self *after* construction is complete
        auto shared_this = std::dynamic_pointer_cast<IMCPProvider_V1>(
            rack::APP->engine->getModule(myId)->getWeakPtr().lock() // Example way to get shared_ptr
        );
```

```
        if (shared_this) {
            self_weak_ptr = shared_this;
            rack::mcp::getMCPBroker()->registerContext("my_topic", myId, self_weak_ptr);
        }
    }

    void onRemove(const rack::event::Remove& e) override {
        rack::mcp::getMCPBroker()->unregisterContext("my_topic", myId);
        rack::engine::Module::onRemove(e);
    }

    void process(const rack::engine::ProcessArgs& args) override {
        // Example: Publish update when a parameter changes significantly
        if (/* condition to publish */) {
            publishUpdate("New State");
        }
    }

    void publishUpdate(const std::string& newState) {
        myContextData = newState;

        // Serialize data using MessagePack
        msgpack::sbuffer buffer;
        msgpack::pack(buffer, myContextData);

        // Create message
        rack::mcp::MCPMessage_V1 msg;
        msg.topic = "my_topic";
        msg.senderModuleId = myId;
        msg.dataFormat = "application/msgpack"; // Or could use a constant
        // Manage buffer lifetime with shared_ptr
        msg.data = std::make_shared<std::vector<char>>(buffer.data(), buffer.data() +
buffer.size());
        msg.dataSize = buffer.size();


        // Publish via broker (fire and forget from module's perspective)
        rack::mcp::getMCPBroker()->publish(msg);
    }
```

```
};


// --- Subscriber Module Example ---
#include "rack.hpp"
#include "mcp_api.hpp"
#include "dsp/ringbuffer.hpp" // For thread-safe queue
#include <msgpack.hpp>

struct MyContextSubscriberModule : rack::engine::Module, rack::mcp::IMCPSubscriber_V1 {
    std::string receivedState = "N/A";
    int64_t myId = -1;
    std::weak_ptr<rack::mcp::IMCPSubscriber_V1> self_weak_ptr;

    // Thread-safe queue to pass data from MCP callback to process()
    rack::dsp::RingBuffer<std::string, 16> messageQueue;

    MyContextSubscriberModule() {
        config(NUM_PARAMS, NUM_INPUTS, NUM_OUTPUTS, NUM_LIGHTS);
        //...
    }

     void onAdd(const rack::event::Add& e) override {
        rack::engine::Module::onAdd(e);
        myId = this->id;
         auto shared_this = std::dynamic_pointer_cast<IMCPSubscriber_V1>(
            rack::APP->engine->getModule(myId)->getWeakPtr().lock()
        );
         if (shared_this) {
            self_weak_ptr = shared_this;
            rack::mcp::getMCPBroker()->subscribe("my_topic", myId, self_weak_ptr);
        }
    }

     void onRemove(const rack::event::Remove& e) override {
        rack::mcp::getMCPBroker()->unsubscribe("my_topic", myId);
        // Or broker->unsubscribeAll(myId);
        rack::engine::Module::onRemove(e);
    }
```

```
    // --- IMCPSubscriber_V1 implementation ---
    // This is likely called on a non-audio thread by the Broker
    void onMCPMessage(const rack::mcp::MCPMessage_V1& message) override {
        if (message.topic == "my_topic" && message.dataFormat == "application/msgpack") {
            try {
                // Ensure data pointer is valid
                if (message.data && message.dataSize > 0) {
                    // Access the raw byte data
                    auto vec_ptr = std::static_pointer_cast<std::vector<char>>(message.data);
                    const char* buffer_data = vec_ptr->data();

                    // Deserialize using MessagePack
                    msgpack::object_handle oh = msgpack::unpack(buffer_data,
message.dataSize);
                    msgpack::object obj = oh.get();

                    std::string deserialized_data = obj.as<std::string>();

                    // Push data to thread-safe queue for process() thread
                    if (!messageQueue.full()) {
                        messageQueue.push(deserialized_data);
                    }
                }
            } catch (const std::exception& e) {
                // Handle deserialization errors
                WARN("MCP Deserialization error: %s", e.what());
            }
        }
    }
    // ----------------------------------------


    void process(const rack::engine::ProcessArgs& args) override {
        // Check queue for new messages from MCP callback
        if (!messageQueue.empty()) {
            receivedState = messageQueue.shift();
            // Update internal state or UI based on receivedState
            //...
        }
```

```
        //... rest of DSP processing...
    }
};
```

6.5. Implementation Strategy Discussion

- **Broker Implementation:** The Singleton Broker needs careful implementation. It should manage its internal registry (e.g., std::map<std::string, std::vector<std::pair<int64_t, std::weak_ptr<IMCPProvider_V1>>>>) protected by a std::mutex for all read/write operations. Message publishing should likely queue messages internally and use a dedicated worker thread pool to handle deserialization (if needed for routing) and dispatching callbacks to subscribers, preventing the publish call itself from blocking. Subscriber callbacks (onMCPMessage) must be invoked off the audio thread.
- **Interface Discovery:** If incorporating direct interface calls post-discovery, the subscriber module would use APP->engine->getModule(providerId) to get the Module*, then dynamic_cast<IMCPProvider_V1*>(). This should be done cautiously, checking for null pointers and ideally using std::weak_ptr obtained from the module to manage lifetime issues if storing the pointer. Frequent scanning of all modules via APP->engine->getModuleIds() should be avoided in process(); discovery should happen less frequently or be event-driven.
- **Data Handling:** Using std::shared_ptr<void> for message data provides flexibility but requires careful casting and type knowledge at the subscriber end. Including a dataFormat string helps, but robust error handling during deserialization is crucial.

This proposed framework provides a starting point, acknowledging that the internal complexity, particularly around thread safety and efficient message dispatching in the broker, requires significant implementation effort. However, the aim is to present a relatively simple and intuitive API surface to the module developer, thereby encouraging adoption. Providing a reference implementation, perhaps as a header-only library or integrated into the SDK, would be vital for success, allowing developers to easily incorporate MCP capabilities without needing to implement the underlying complexities themselves.[32]

7. Recommendations and Future Directions

The analysis presented in this report indicates both a clear need for and the technical feasibility of a Model Context Protocol (MCP) within VCV Rack 2. To move forward effectively, the following recommendations and future directions are proposed.

7.1. Recommended MCP Approach

Based on the evaluation of VCV Rack's architecture, community needs, and technical constraints, the **Hybrid Broker/Interface Model (Section 6.1)** is recommended.

- **Architecture:** A thread-safe Singleton Broker should manage topic registration and discovery, providing a centralized point for modules to find context providers. It should also handle the dispatching of published messages for a robust Publish/Subscribe mechanism. Once discovered via the broker, modules *may* optionally establish more direct communication using defined C++ interfaces (IMCPProvider_V1, IMCPSubscriber_V1) for specific query/response interactions if performance dictates, though the Pub/Sub mechanism via the broker should be the primary mode for broadcasting updates.
- **API Structure:** The API outlined in Section 6.2, centered around IMCPBroker, IMCPProvider_V1, IMCPSubscriber_V1, and MCPMessage_V1, provides a clear and relatively simple interface for developers. Key functions include registerContext, subscribe, publish, findProviders, and the onMCPMessage callback.
- **Data Format: MessagePack** is recommended as the default serialization format due to its performance advantages in speed and size, crucial for real-time operation.[67] The MCPMessage_V1 structure should include a dataFormat field to allow for potential future support or negotiation of other formats like JSON for specific use cases.

Justification: This hybrid approach balances ease of discovery (via the broker) with flexible communication patterns (Pub/Sub via broker, optional direct interface calls). MessagePack offers the best performance characteristics for the core data transport layer. The proposed API abstracts much of the underlying complexity, particularly thread safety, from the average module developer.

7.2. Implementation Roadmap

A phased approach is recommended for implementing the MCP:

1. **Phase 1: Core Broker and API:** Implement the thread-safe IMCPBroker singleton and the core API functions (register, unregister, subscribe, unsubscribe, findProviders, getAvailableTopics). Implement basic, thread-safe message queuing and dispatch for publish and onMCPMessage (likely using worker threads). Define the MCPMessage_V1 structure.
2. **Phase 2: Serialization Integration:** Integrate MessagePack serialization/deserialization for the message payload. Provide helper functions or clear examples for packing/unpacking data into/from the MCPMessage_V1::data field.

3. **Phase 3: Reference Implementation & Documentation:** Develop a simple reference library encapsulating the client-side interaction with the broker API. Create comprehensive documentation with clear examples (similar to Section 6.4) demonstrating provider and subscriber implementation.
4. **Phase 4: Testing and Refinement:** Conduct thorough performance testing under various load conditions and thread counts. Perform rigorous thread-safety testing to identify potential deadlocks or race conditions. Refine the API and implementation based on feedback.
5. **Phase 5: SDK Integration:** Integrate the finalized MCP broker and API into the official VCV Rack SDK for broad availability.

7.3. Potential Challenges and Mitigation

- **Performance Overhead:** The broker and serialization/deserialization add overhead.
  - *Mitigation:* Optimize broker implementation (efficient locking, worker threads). Choose a fast serialization format (MessagePack). Encourage performing non-critical MCP operations off the audio thread. Continuously benchmark.
- **Developer Adoption:** Convincing developers to adopt a new standard takes effort.
  - *Mitigation:* Provide a simple, intuitive API. Offer excellent documentation and examples. Create a reference library. Engage the community during development. Highlight the new possibilities MCP enables.
- **API Stability and Versioning:** Breaking changes can fragment the ecosystem.
  - *Mitigation:* Design the V1 API carefully with future expansion in mind. Implement clear versioning in the protocol/API from the start. Follow strict deprecation policies for any future changes.
- **Implementation Complexity:** Building a robust, thread-safe broker is non-trivial.
  - *Mitigation:* Leverage existing C++ concurrency primitives and patterns carefully. Extensive testing is required. Consider making the reference implementation open-source for community review and contribution.[78]

7.4. Future Enhancements

Once a core MCP is established, future enhancements could include:

- **Advanced Querying:** Support for more complex queries beyond simple topic lookup (e.g., querying context based on metadata).
- **Standardized Contexts:** Define standard topic names and data formats for common use cases (e.g., rack/musical/key_signature, rack/sequence/transport_state, rack/modulation/matrix_v1).

- **UI Integration:** Visual cues in the Rack interface to indicate MCP connections or activity.
- **Bridging Modules:** Dedicated modules to bridge MCP topics to/from standard CV/Gate signals, OSC messages, or MIDI data, further integrating MCP with existing workflows.[59]
- **Schema Support:** Optional support for schema validation (e.g., using FlatBuffers schemas or JSON Schema) for specific topics requiring strict data contracts.

Developing and standardizing an MCP represents a significant opportunity for VCV Rack. It requires careful technical design, particularly regarding performance and thread safety. However, the potential benefits – enabling new forms of complex module interaction, fostering innovation, and providing a robust alternative to current ad-hoc solutions – are substantial. Collaboration between the VCV Rack core team and the active developer community will be crucial for designing, implementing, and successfully adopting such a protocol.[78] A well-executed MCP could unlock entirely new categories of modules focused on patch-wide orchestration, intelligent meta-control, advanced data visualization, and deeper inter-module collaboration, pushing the boundaries of what is possible within the virtual modular environment.

Works cited

1. Virtual Eurorack Studio - VCV Rack 2, accessed April 26, 2025, https://vcvrack.com/Rack
2. How to start with Modular Synthesis? Modular Synth 101 - Sine Community, accessed April 26, 2025, https://sinecommunity.com/modular-101/
3. Eurorack 101 - Intellijel, accessed April 26, 2025, https://intellijel.com/support/eurorack-101/
4. Plugin API Guide - VCV Rack Manual, accessed April 26, 2025, https://vcvrack.com/manual/PluginGuide
5. Building a Synthesizer: Glossary and Electrical Connections - Craig Stuntz -, accessed April 26, 2025, https://www.craigstuntz.com/posts/2023-02-23-building-a-synthesizer-glossary.html
6. CV/gate - Wikipedia, accessed April 26, 2025, https://en.wikipedia.org/wiki/CV/gate
7. Starting Out In Modular | Noise Engineering, accessed April 26, 2025, https://noiseengineering.us/blogs/loquelic-literitas-the-blog/starting-out-in-modular/
8. VCV Rack API: rack::engine::Port Struct Reference, accessed April 26, 2025, https://vcvrack.com/docs-v2/structrack_1_1engine_1_1Port
9. Plugin Development Tutorial - VCV Rack Manual, accessed April 26, 2025, https://vcvrack.com/manual/PluginDevelopmentTutorial
10. Migrating v1 Plugins to v2 - VCV Rack Manual, accessed April 26, 2025, https://vcvrack.com/manual/Migrate2

11. Module Panel Guide - VCV Rack Manual, accessed April 26, 2025, https://vcvrack.com/manual/Panel
12. Module expanders tutorial? - Development - VCV Community, accessed April 26, 2025, https://community.vcvrack.com/t/module-expanders-tutorial/4868
13. MarcBoule/MindMeldModular: Modules for VCV Rack - GitHub, accessed April 26, 2025, https://github.com/MarcBoule/MindMeldModular
14. MindMeldModular | Modules for VCV Rack - GitHub Pages, accessed April 26, 2025, https://marcboule.github.io/MindMeldModular/
15. Interest in a module+protocol for sending text over VCV cable? - Page 2 - Development, accessed April 26, 2025, https://community.vcvrack.com/t/interest-in-a-module-protocol-for-sending-text-over-vcv-cable/20155?page=2
16. Question about communicating between 2 different computers. - VCV Rack, accessed April 26, 2025, https://community.vcvrack.com/t/question-about-communicating-between-2-different-computers/18967
17. MindMeld Mixmaster + EQMaster + Rackwindows Console = Awesome - VCV Community, accessed April 26, 2025, https://community.vcvrack.com/t/mindmeld-mixmaster-eqmaster-rackwindows-console-awesome/12240
18. Interest in a module+protocol for sending text over VCV cable? - Development, accessed April 26, 2025, https://community.vcvrack.com/t/interest-in-a-module-protocol-for-sending-text-over-vcv-cable/20155
19. Eurorack voltage standards : r/synthdiy - Reddit, accessed April 26, 2025, https://www.reddit.com/r/synthdiy/comments/8iex2l/eurorack_voltage_standards/
20. best practices - store huge data - Development - VCV Community, accessed April 26, 2025, https://community.vcvrack.com/t/best-practices-store-huge-data/13424
21. Load Sample - Thread-Safe - VCV Best practice - Development, accessed April 26, 2025, https://community.vcvrack.com/t/load-sample-thread-safe-vcv-best-practice/22670
22. DaveBenham/VenomModules: Venom VCV Rack Modules - GitHub, accessed April 26, 2025, https://github.com/DaveBenham/VenomModules
23. Module expanders - sharing a minimal example - Development - VCV Community, accessed April 26, 2025, https://community.vcvrack.com/t/module-expanders-sharing-a-minimal-example/23101
24. MindMeld - VCV Library, accessed April 26, 2025, https://library.vcvrack.com/MindMeldModular
25. Plugin API Guide - Development - VCV Community, accessed April 26, 2025, https://community.vcvrack.com/t/plugin-api-guide/20046
26. VCV Rack API: rack::engine::Engine Struct Reference, accessed April 26, 2025, https://vcvrack.com/docs-v2/structrack_1_1engine_1_1Engine

27. Touching other Instances of Module in Rack - Development - VCV Community, accessed April 26, 2025, https://community.vcvrack.com/t/touching-other-instances-of-module-in-rack/11933
28. Would it be Possible to Manipulate VCV Rack using Python Script? - Development, accessed April 26, 2025, https://community.vcvrack.com/t/would-it-be-possible-to-manipulate-vcv-rack-using-python-script/12895
29. Save/restore of module with multiple audio interfaces on MacOS - VCV Community, accessed April 26, 2025, https://community.vcvrack.com/t/save-restore-of-module-with-multiple-audio-interfaces-on-macos/20863
30. OSC (OpenSoundControl) modules in v2? - VCV Community, accessed April 26, 2025, https://community.vcvrack.com/t/osc-opensoundcontrol-modules-in-v2/15114
31. scripting language / interface to send values to module parameters ? - VCV Rack, accessed April 26, 2025, https://community.vcvrack.com/t/scripting-language-interface-to-send-values-to-module-parameters/20541
32. Interest in a module+protocol for sending text over VCV cable? - Page 4, accessed April 26, 2025, https://community.vcvrack.com/t/interest-in-a-module-protocol-for-sending-text-over-vcv-cable/20155?page=4
33. Interest in a module+protocol for sending text over VCV cable ..., accessed April 26, 2025, https://community.vcvrack.com/t/interest-in-a-module-protocol-for-sending-text-over-vcv-cable/20155?page=3
34. mahlenmorris/VCVRack: Set of modules for use with VCV Rack 2.0 - GitHub, accessed April 26, 2025, https://github.com/mahlenmorris/VCVRack
35. ModScript: Using scripts for a better and richer integration with MIDI controllers (beta soon!), accessed April 26, 2025, https://community.vcvrack.com/t/modscript-using-scripts-for-a-better-and-richer-integration-with-midi-controllers-beta-soon/18197
36. MIDI-KIT - script processor for MIDI - by stoermelder - Announcements - VCV Community, accessed April 26, 2025, https://community.vcvrack.com/t/midi-kit-script-processor-for-midi-by-stoermelder/18989
37. VCV Rack gets physical - A DIY project 19" Hardware Controler, accessed April 26, 2025, https://community.vcvrack.com/t/vcv-rack-gets-physical-a-diy-project-19-hardware-controler/3538
38. Two way communication between an Arduino board and a VCV Rack module. - GitHub, accessed April 26, 2025, https://github.com/EMC23/arduino-vcvrack
39. [BETA] RSBATechModules plugin - module remote control with TouchOSC or the Electra One MIDI Controller - VCV Community, accessed April 26, 2025,

https://community.vcvrack.com/t/beta-rsbatechmodules-plugin-module-remote-control-with-touchosc-or-the-electra-one-midi-controller/23599
40. Controlling patch-cables connections through MIDI - Development - VCV Community, accessed April 26, 2025, https://community.vcvrack.com/t/controlling-patch-cables-connections-through-midi/7668
41. Is there a way to access a right click menu item with a knob or CV? - VCV Rack, accessed April 26, 2025, https://community.vcvrack.com/t/is-there-a-way-to-access-a-right-click-menu-item-with-a-knob-or-cv/19616
42. Is there a module that lets you control parameters? - VCV Community, accessed April 26, 2025, https://community.vcvrack.com/t/is-there-a-module-that-lets-you-control-parameters/20939
43. Interface module to control several modules at once - VCV Community, accessed April 26, 2025, https://community.vcvrack.com/t/interface-module-to-control-several-modules-at-once/13232
44. Share your patch selections (.vcvs) - Page 3 - Plugins & Modules - VCV Community, accessed April 26, 2025, https://community.vcvrack.com/t/share-your-patch-selections-vcvs/21818?page=3
45. Using MindMeld MixMaster and EQMaster - Plugins & Modules - VCV Community, accessed April 26, 2025, https://community.vcvrack.com/t/using-mindmeld-mixmaster-and-eqmaster/21477
46. Shared variables across modules - Development - VCV Community, accessed April 26, 2025, https://community.vcvrack.com/t/shared-variables-across-modules/6922
47. Expander - Thread safe? - Development - VCV Community, accessed April 26, 2025, https://community.vcvrack.com/t/expander-thread-safe/11029
48. Sharing memory between modules - c++ - Stack Overflow, accessed April 26, 2025, https://stackoverflow.com/questions/4616148/sharing-memory-between-modules
49. Any VCV native event emitter? - Development, accessed April 26, 2025, https://community.vcvrack.com/t/any-vcv-native-event-emitter/4476
50. MarcBoule/ImpromptuModular: Virtual Eurorack Modules for VCV Rack - GitHub, accessed April 26, 2025, https://github.com/MarcBoule/ImpromptuModular
51. mhetrick/nonlinearcircuits: VCV Rack ports of Nonlinear Circuits modules - GitHub, accessed April 26, 2025, https://github.com/mhetrick/nonlinearcircuits
52. wapiflapi/admiral: Modules for VCV Rack - GitHub, accessed April 26, 2025, https://github.com/wapiflapi/admiral
53. Monome modules dev log - Announcements - VCV Community, accessed April 26, 2025, https://community.vcvrack.com/t/monome-modules-dev-log/3683
54. Reaktor 6 Blocks are like getting a modular in your laptop for $199 ..., accessed

April 26, 2025, https://cdm.link/reaktor-6-blocks-like-getting-unlimited-modular-laptop-199/
55. Reaktor 6 Blocks 1.2 Tutorial #1: Using Blocks Signals to Control Voltage | Kadenze, accessed April 26, 2025, https://www.youtube.com/watch?v=-15njXbyVJ4
56. Reaktor 6 Blocks 1.2 Tutorial #3: Using Open Sound Control to Control Voltage | Kadenze, accessed April 26, 2025, https://www.youtube.com/watch?v=irGJeXhX3-Q
57. Play. Patch. Build. - EuroReakt Blocks with Michael Hetrick | Native Instruments - YouTube, accessed April 26, 2025, https://www.youtube.com/watch?v=nL4TNaJm-3M
58. Building a Modular Synthesiser Part 1: Introduction and Planning, accessed April 26, 2025, https://www.rs-online.com/designspark/building-a-modular-synthesiser-part-1-introduction-and-plannin
59. MIDI to CV and back: Can your “regular” synth become semi-modular? - Loopop, accessed April 26, 2025, https://loopopmusic.com/midi-to-cv-and-back-can-your-regular-synth-become-semi-modular
60. Creative Control: MIDI to CV Converters - Perfect Circuit, accessed April 26, 2025, https://www.perfectcircuit.com/signal/midi-to-cv
61. I'm giving an hour long lecture on eurorack, CV, and patching in 2 months! What should I be sure to touch on? : r/modular - Reddit, accessed April 26, 2025, https://www.reddit.com/r/modular/comments/1aed5y1/im_giving_an_hour_long_lecture_on_eurorack_cv_and/
62. Modular Synthesis: The Secret to Mastering Digital Sound Design - Unison Audio, accessed April 26, 2025, https://unison.audio/modular-synthesis/
63. Thread-Safe leaky singleton : r/cpp - Reddit, accessed April 26, 2025, https://www.reddit.com/r/cpp/comments/7j3s46/threadsafe_leaky_singleton/
64. Trouble understanding options under "Engine / Threads" - VCV Community, accessed April 26, 2025, https://community.vcvrack.com/t/trouble-understanding-options-under-engine-threads/11846
65. FL%P-N - the NOAA Institutional Repository, accessed April 26, 2025, https://repository.library.noaa.gov/view/noaa/39551/noaa_39551_DS1.pdf
66. engine::Module Struct Reference - VCV Rack API, accessed April 26, 2025, https://vcvrack.com/docs-v2/structrack_1_1engine_1_1Module
67. Benchmarking Eight Serialization Formats in C and C++ (JSON, BSON, CBOR, flexbuffers, msgpack, TOML, XML, YAML) : r/cpp - Reddit, accessed April 26, 2025, https://www.reddit.com/r/cpp/comments/1drz3eg/benchmarking_eight_serialization_formats_in_c_and/
68. In your opinion, what's the value of MessagePack? I worked with it when writing - Hacker News, accessed April 26, 2025, https://news.ycombinator.com/item?id=20604597
69. Performance of Serialization Libraries in a High Performance Computing Environment, accessed April 26, 2025,

https://uh-ir.tdl.org/bitstreams/d10140e3-00d6-4774-9e06-8e7253c06039/download
70. FlatBuffers serialization timing · Issue #25 · Alois-xx/SerializerTests - GitHub, accessed April 26, 2025, https://github.com/Alois-xx/SerializerTests/issues/25
71. VCVRACK on Multicore processor, accessed April 26, 2025, https://community.vcvrack.com/t/vcvrack-on-multicore-processor/624
72. VCV Performance Issues : r/vcvrack - Reddit, accessed April 26, 2025, https://www.reddit.com/r/vcvrack/comments/1ilnh9x/vcv_performance_issues/
73. Details and comparison of miRack and VCV Rack multithreading implementation and performance, accessed April 26, 2025, https://mirack.app/2020/06/09/details-and-comparison-of-mirack-and-vcv-rack-multithreading-implementation-and-performance/
74. so let's compare systems via average cpu load in v2 - VCV Community, accessed April 26, 2025, https://community.vcvrack.com/t/so-lets-compare-systems-via-average-cpu-load-in-v2/15082
75. Objective way to measure VCV Rack performance? (acceptable on Mac), accessed April 26, 2025, https://community.vcvrack.com/t/objective-way-to-measure-vcv-rack-performance-acceptable-on-mac/13425
76. Performance Question - Windows 10 - VCV Rack, accessed April 26, 2025, https://community.vcvrack.com/t/performance-question-windows-10/4677
77. VCV Rack 2: Multithreaded CPU usage, accessed April 26, 2025, https://community.vcvrack.com/t/vcv-rack-2-multithreaded-cpu-usage/14925
78. VCV Community invitation · Issue #248 · VCVRack/library - GitHub, accessed April 26, 2025, https://github.com/VCVRack/library/issues/248