future internet

MDPI

Article

Modular Microservices Architecture for Generative Music Integration in Digital Audio Workstations via VST Plugin

Adriano N. Raposo [1,2,3,*,†] and Vasco N. G. J. Soares [1,2,†]

[1] Instituto de Telecomunicações, Rua Marquês d'Ávila e Bolama, 6201-001 Covilhã, Portugal; vasco.g.soares@ipcb.pt
[2] School of Technology, Polytechnic University of Castelo Branco, Av. Pedro Álvares Cabral n° 12, 6000-084 Castelo Branco, Portugal
[3] CAC-UBI Center for Applied Computing, University of Beira Interior, 6201-001 Covilhã, Portugal
* Correspondence: anraposo@ubi.pt
† These authors contributed equally to this work.

Abstract

This paper presents the design and implementation of a modular cloud-based architecture that enables generative music capabilities in Digital Audio Workstations through a MIDI microservices backend and a user-friendly VST plugin frontend. The system comprises a generative harmony engine deployed as a standalone service, a microservice layer that orchestrates communication and exposes an API, and a VST plugin that interacts with the backend to retrieve harmonic sequences and MIDI data. Among the microservices is a dedicated component that converts textual chord sequences into MIDI files. The VST plugin allows the user to drag and drop the generated chord progressions directly into a DAW's MIDI track timeline. This architecture prioritizes modularity, cloud scalability, and seamless integration into existing music production workflows, while abstracting away technical complexity from end users. The proposed system demonstrates how microservice-based design and cross-platform plugin development can be effectively combined to support generative music workflows, offering both researchers and practitioners a replicable and extensible framework.

Keywords: cloud computing; microservices; creative workflows; generative music; music production; virtual studio technology

Academic Editors: Dimitrios Dechouniotis and Ioannis Dimolitsas

Received: 13 September 2025
Revised: 7 October 2025
Accepted: 10 October 2025
Published: 12 October 2025

Citation: Raposo, A.N.; Soares, V.N.G.J. Modular Microservices Architecture for Generative Music Integration in Digital Audio Workstations via VST Plugin. *Future Internet* **2025**, *17*, 469. https://doi.org/10.3390/fi17100469

1. Introduction

DAWs (Digital Audio Workstations) are essential tools in modern music production, allowing composers, producers, and artists to manipulate audio and MIDI (Musical Instrument Digital Interface) data in increasingly complex workflows. Within these environments, VST (Virtual Studio Technology) plugins extend DAWs by providing virtual instruments, audio effects, and MIDI utilities. In parallel, advances in generative music algorithms—ranging from rule-based systems to deep learning—have enabled the automatic creation of harmonic and melodic structures, offering creative assistance or even full musical compositions [1].

Despite these developments, the integration of generative music into DAW workflows remains limited. Many generative systems operate as standalone applications, requiring users to manually export and import MIDI files [2,3]. This fragmented workflow can hinder experimentation and discourage adoption by musicians with limited technical expertise.

To address these challenges, microservices architecture offers a promising foundation for integrating generative music systems into broader production environments. Microservices are a design paradigm in which applications are decomposed into loosely coupled independently deployable services. Each service is responsible for a specific function and communicates with others through lightweight protocols such as HTTP (Hypertext Transfer Protocol) or messaging queues. This modular approach enhances scalability, facilitates technology heterogeneity, and simplifies deployment and maintenance. In the context of generative music, microservices enable components like machine learning models, music theory engines, user interfaces, and DAW integration bridges to evolve independently while collaborating within a cohesive system. Such an architecture is particularly well-suited for cloud-based or hybrid solutions that must balance performance, modularity, and user accessibility. This paper's contributions include the following:

- A Node.js orchestration layer that exposes a unified HTTP API (Application Programming Interface) and bridges the frontend with backend services (Section 4.2.1).
- A Java-based generative harmony engine encapsulated as an independent microservice (Section 4.2.3).
- An MIDI generation microservice leveraging the Python music21 library to convert chord sequences into structured MIDI files (Section 4.2.4).
- A JUCE-based VST plugin that acts as the user interface and DAW integrator, allowing musicians to trigger generation, retrieve results, and drag MIDI files directly onto the DAW timeline (Section 4.3.1).
- An evaluation of system performance and usability in a real-world music production context (Section 5.1).

The remainder of this paper is organized as follows: Section 2 reviews the relevant literature on generative music systems and VST integration. Section 3 presents the system's research framework, including the research questions and hypotheses. Section 4 describes the system architecture and implementation details of the backend microservices and frontend plugin. Section 5 illustrates the results obtained from using the system, including generated outputs and integration scenarios. Section 6 provides a discussion of the system's strengths, limitations, and directions for future work. Finally, Section 7 summarizes the main findings and contributions of this study.

2. Literature Review

The field of music technology has evolved rapidly at the intersection of audio engineering, software design, and artificial intelligence. This section reviews key areas relevant to the present work, beginning with the evolution of Digital Audio Workstations (DAWs) and their underlying software architectures. It then examines advances in intelligent and algorithmic music production, explores the landscape of generative music systems and chord modeling, and concludes with an overview of web and microservices technologies applied to music tools. Together, these perspectives provide the conceptual and technical foundation upon which the proposed system is built.

2.1. Digital Audio Workstations and Software Architectures

Digital Audio Workstations (DAWs) have long served as the central hub for modern music production. Traditional DAWs, such as those discussed by Leider [4], emphasize local monolithic architectures with comprehensive functionality. More recent research has explored the evolution of DAWs toward more specialized or distributed designs. Bianchi et al. [5] propose a GPU-oriented application programming interface for DAWs to optimize audio rendering in heterogeneous computing environments. Meanwhile, the WAM-studio project illustrates a web-based DAW framework [6,7], reflecting a shift

towards browser-accessible music production tools and supporting plugin standardization in the Web Audio Modules ecosystem.

2.2. Intelligent and Algorithmic Music Production

The integration of artificial intelligence (AI) into music production workflows is an active area of research. Moffat and Sandler [8] survey various intelligent music production approaches, spanning mixing assistance, audio analysis, and adaptive systems. Xu [9] and Wang [10] explore the application of computer music production software in new media environments and creative practices, respectively, while Ye [11] presents case studies illustrating its practical use in music creation. Kviecien [12] offers a holistic perspective that includes technical, musical, and legal considerations for AI-aided algorithmic composition systems.

In the context of plugin ecosystems, Silva et al. [13] investigate recommender systems tailored for audio plugin selection, helping producers navigate extensive plugin libraries based on contextual and collaborative filtering techniques.

2.3. Generative Music Systems and Chord Modeling

Generative music has gained significant traction through frameworks like Magenta [14], MuseNet [15], and AIVA [16], which utilize techniques such as Markov models, neural networks, and grammar-based systems. Conklin et al. [17] developed methods for chord sequence generation specifically targeting electronic dance music. Raposo and Soares [18] introduced generative models for jazz chords, enriching stylistic expressiveness. Zhao et al. [19] provided a comprehensive review of text-to-music systems, while Mon [20] evaluated LSTM-based architectures for symbolic music generation.

Despite these advancements, few works examine the encapsulation of generative music functionalities into modular microservices that can be embedded directly into DAWs or other production environments.

2.4. Web and Microservices Technologies for Music Tools

On the software infrastructure side, web technologies and microservices are increasingly leveraged to support modular scalable music production environments. Lei et al. [21] and Huang [22] evaluated web development frameworks such as Node.js for real-time asynchronous backend tasks relevant to music software.

From a systems engineering perspective, research into microservices architectures has outlined best practices for modularization, deployment, and orchestration [23,24]. Vlček [25] applied these concepts to synchronized music streaming, while Lin et al. [26] introduced MusicTalk, a microservice-based framework for musical instrument recognition. However, there remains a gap in applying these architectural paradigms to real-time generative music services and their integration into modern DAWs.

3. Research Framework

The following section outlines the research framework guiding this study. It defines the underlying motivation, identifies the key technical and usability challenges, and establishes the research questions and hypotheses that drive the design, implementation, and evaluation of the proposed generative music system. By clearly articulating these elements, we provide a structured foundation that connects the identified gaps in the field with the methodological choices presented in subsequent sections.

3.1. Research Gap and Motivation

There exists a gap between advanced generative music techniques and their seamless integration into everyday music production environments. Current solutions are either

overly technical, requiring scripting or coding expertise, or they impose a rigid workflow detached from standard DAW usage. There is a need for architectures that abstract technical complexity while remaining extensible and developer-friendly.

This research is motivated by the opportunity to democratize access to generative music tools by embedding them into common creative workflows. By decoupling the generative logic into scalable backend microservices and exposing a user-friendly frontend through a VST plugin, we aim to lower the barrier to entry and encourage adoption among musicians and producers.

3.2. Challenge

Designing such a system presents several technical and usability challenges:

- Modularization: how to decouple the generative, conversion, and control logic into isolated, maintainable components.
- Communication: how to establish reliable and low-latency communication between the plugin frontend and the cloud-based backend services.
- User Interaction: how to provide a minimal intuitive plugin interface that integrates with DAWs and allows drag-and-drop functionality for generated MIDI files.
- Compatibility: how to ensure cross-platform support for both the backend services (e.g., Java, Node.js, Python) and the JUCE-based VST plugin.

3.3. Research Questions and Hypotheses

Based on the identified gaps and challenges, this study is guided by the following research questions:

- Can a modular microservices architecture efficiently integrate generative music capabilities into a VST plugin while maintaining low-latency performance suitable for standard DAW workflows?
- How does the proposed cloud-based and local-processing hybrid approach impact the computation time, scalability, and system maintainability?
- Can the architecture support secure and scalable access for potential commercial deployment without compromising usability?

From these questions, we formulate the corresponding hypotheses:

- The modular architecture will provide real-time or near real-time generative output (average response times under 1 s) suitable for interactive music composition.
- Decoupling the generative engine, MIDI conversion, and control logic into microservices will enhance maintainability and scalability compared to monolithic systems.
- API key-based authentication and integration with a database microservice will enable secure and flexible access control, supporting commercial deployment without negatively affecting the workflow efficiency.

These research questions and hypotheses establish the framework for evaluating the proposed system in terms of performance, usability, and practical deployment potential, providing clear criteria for assessing the success of the implementation described in the following section.

4. Architecture and Implementation

This section describes the design and implementation of the proposed system. The architecture consists of a frontend VST plugin enabling user interaction and a backend composed of modular microservices responsible for chord generation, MIDI conversion, and data management. The implemented methods include the generative chord engine, MIDI transformation processes, API orchestration, and plugin-based user interactions.

System performance and usability are evaluated within this framework, with response times, request handling, and user interactions systematically logged and analyzed through the database and gateway microservices to assess the effectiveness and efficiency of the overall system.

4.1. The Architecture

As illustrated in Figure 1, the system is divided into a *frontend*, composed of the VST plugin that acts as the user-facing entry point within the DAW, and a *backend*, composed of microservices and related applications that handle generative processing, data conversion processing, and data management. This separation of concerns ensures scalability, flexibility, and ease of maintenance.

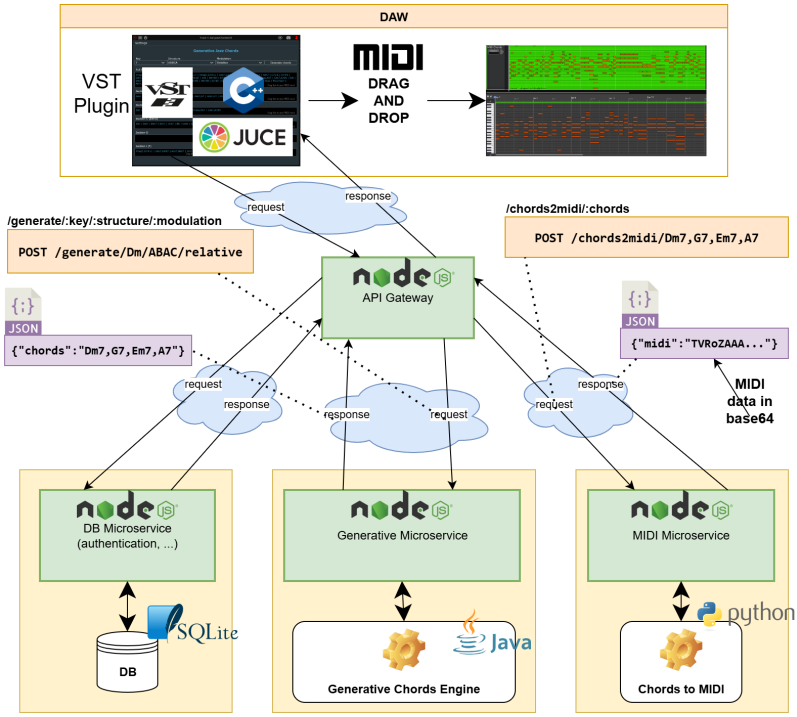

Figure 1. Global solution architecture.

4.2. The Backend

The backend constitutes the core of the proposed architecture, encapsulating all computational and data management tasks behind the VST plugin interface. It is composed of a set of modular microservices, each designed to perform a specialized function while remaining loosely coupled to ensure scalability, maintainability, and extensibility. As shown in Figure 1, these services include the *Generative Music Microservice*, which is responsible for producing textual chord sequences using algorithmic and machine learning techniques; the *MIDI Microservice*, which converts the textual chord sequences to the MIDI format; the *Database Microservice*, which provides persistent storage and retrieval of user API keys and session logs; and the *Gateway Microservice*, which coordinates communication between the frontend and backend while enforcing security and access control. Together, these microservices form a distributed system. In our implementation, we used Node.js as the web services framework, but the same architecture presented in Figure 1 can be implemented using any other web framework for REST (Representational State Transfer) services.

4.2.1. Gateway Microservice

The API Gateway microservice serves as a central access point for clients, proxying requests to underlying microservices while handling authentication, logging, and data

conversion. In our prototype, it is implemented using Express.js [27] and uses `Axios` for inter-service HTTP communication. Configuration is loaded via `dotenv`, and the service listens on a configurable port.

All protected routes require an API key passed in the `x-api-key` header, which the gateway validates by querying the database microservice. Requests with invalid or missing keys receive `401` or `403` responses.

The gateway exposes the following endpoints:

- /api/keys—retrieves available musical keys.
- /api/structures—retrieves chord structure templates.
- /api/modulations—retrieves available modulations.
- /api/generate/:key/:structure/:modulation—generates chord sequences for a given key, structure, and modulation, where requests are validated via API key, proxied to the generation microservice, and logged in the database.
- /api/chords2midi/:chords—converts a chord sequence string into a base64 MIDI file using the MIDI microservice.
- /api/history/:apikey—retrieves a user's request history, and only the owner of the API key can access their history.

All protected endpoints log requests and responses to the database microservice, ensuring traceability and monitoring of client activity.

All endpoints simply route client requests to the corresponding microservice. Detailed explanations of each endpoint's functionality are provided in the following subsections dedicated to the respective microservices.

4.2.2. Database Microservice

The Database Microservice provides persistent storage for API keys and user request logs, supporting authentication and traceability across the system. In our implementation, the service is built using `Node.js` with the `Express.js` framework, and it leverages `SQLite` as the storage backend for convenience; however, any other relational database management system (RDBMS) could be used in place of SQLite. Cross-origin requests are allowed via the `CORS` middleware, and JSON (JavaScript Object Notation) request bodies are automatically parsed using `express.json()`.

Upon startup, the service initializes a SQLite database located at `./db/database.db`, ensuring that the following tables exist:

1. api_keys: Stores user API keys with an `active` flag to enable or disable access. The `key` column is unique and serves as the primary key.
2. generation_history: Stores logs of user requests and responses, including a timestamp, the associated API key, and the request/response data. The `api_key` column is a foreign key referencing `api_keys`.

The service exposes the following REST endpoints:

- /api/validate/:key—Verifies whether an API key exists and is active, returning a boolean `valid` field. The API key identifies the user and can be an arbitrary string, used for authentication purposes. It may also serve additional purposes, such as billing or usage control in commercial applications.
- /api/log—Logs a generation request and response, validating that the API key exists and is active before insertion. Each log entry includes a timestamp.
- /api/history/:key—Retrieves the generation history for a given API key, returning all requests and responses in reverse chronological order. The `response` field is automatically parsed from JSON.

All endpoints simply route the client requests to the corresponding database operations. Details on how these endpoints are used are explained in the relevant sections for the API Gateway and other microservices. The service listens on a configurable port, defaulting to 3001, and runs on all network interfaces.

4.2.3. Generative Music Microservice

The Generative Music Microservice is implemented as an REST API using Node.js. Upon receiving a client request from the gateway service, the microservice delegates the generative task to a Java Generative Engine, which implements the method recently proposed by Raposo and Soares in [18], which models jazz harmony using a corpus of 1382 standards. The approach integrates key information, song structure, and chord sequences to generate harmonically coherent progressions that balance stylistic authenticity with creative exploration. Unlike traditional first-order Markov models, the framework introduces a novel combination of section chord patterns, harmonic transition probabilities, and stylistic constraints, enabling the synthesis of both conventional and innovative chord sequences. Evaluation of the generated dataset demonstrated strong statistical alignment with the original corpus across multiple analytical parameters, confirming that the method preserves the essence of jazz harmony while allowing for the discovery of novel harmonic pathways. Algorithm 1 presents the core algorithm for generating the chord sequences. Due to its length, a complete description of the algorithm is out of the scope of the present paper, and the reader is referred to [18] for a full explanation of the algorithm steps.

Algorithm 1 Core algorithm for generating the chord progressions

Require: A key k and a structure S
Ensure: An array of chords C
if k is null **then**
  $k \leftarrow randomKey()$
end if
if S is null **then**
  $S \leftarrow randomStructure()$
end if
for $i = 1$ to $sizeOf(S)$ **do**
  $section \leftarrow S[i]$
  $P \leftarrow randomPattern(section)$
  $originChord \leftarrow randomFirstChord(k)$
  Add $originChord$ to C
  for $j \leftarrow 1$ to $sizeOf(P)$ **do**
    $n \leftarrow P[j]$
    for $k = 1$ to n **do**
      $destinyChord \leftarrow randomDestinyChord(k, originChord)$
      Add $destinyChord$ to C
      $originChord \leftarrow destinyChord$
    end for
  end for
  $k \leftarrow keyModulation()$
end for
return C

It is important to notice that the proposed architecture is modular, allowing the integration of any other generative chord engine, which enables flexibility in experimenting with alternative harmonic generation methods or future models, and can be readily expanded to accommodate other musical genres beyond jazz.

In our implementation, the microservice is developed with the `Express` framework in Node.js and serves as a bridge to the Java-based generative engine. Communication

with the Java program is handled via the `child_process.spawn` function, which executes the `GenJazzChords.jar` program with different arguments depending on the requested functionality. The service listens on port 3002 (or on other arbitrary ports as needed) and exposes multiple REST endpoints, each returning results in JSON format to ensure interoperability with other system components.

The available endpoints are as follows:

- /api/keys: Returns a list of supported tonal centers (keys) for chord sequence generation. This information is used by the VST plugin to populate a selection menu, allowing the user to choose the desired key.
- /api/structures: Provides predefined song structures (e.g., AABA, ABAC) that define the formal organization of the generated progression. This information is also used by the VST plugin to populate a selection menu for choosing the desired structure.
- /api/modulations: Lists possible harmonic modulations that can be incorporated into the generated sequences. Similar to the previous endpoints, this information is used by the VST plugin to populate a selection menu for selecting the modulation type.
- /api/generate/:key/:structure/:modulation: The main endpoint of the service, responsible for generating a chord progression based on a specified key, song structure, and modulation option. The response includes the main key, the selected structure, and a chord sequence for each section, as illustrated in the example JSON structure below.

An example of a call to the service using the URL shown in Listing 1 is:
http://ipaddress:3002/api/generate/C/ABAC/Dominant

In this URL, `ipaddress` should be replaced with the actual IP address or domain name of the server where the generative microservice is hosted and running.

Listing 1. JSON response returned by the generative music microservice for key `C`, structure `ABAC`, and modulation `Dominant`.

```
{
  "key": "C",
  "structure": "ABAC",
  "sections": [
    {
      "chords": "C6,Eb07|Dm7,G7|G7,Cmaj7|Dbmaj7,Gm7|C7|C7|Fmaj7|
          Fmaj7,Bb7,Em7,A7",
      "label": "A",
      "key": "C"
    },
    {
      "chords": "Cm7|Cm7|Cm7|F7|Bm7|E7#9|Am7|Bm7",
      "label": "B",
      "key": "G"
    },
    {
      "chords": "Gmaj7|B7,Bbmaj7|B07,Cm7|F7,Bm7|E7|Am7|D7,Gmaj7|
          Cmaj7,Bm7b5",
      "label": "C",
      "key": "G"
    }
  ]
}
```

Each endpoint triggers the execution of the Java program with the corresponding argument (e.g., `keys`, `structures`, or `modulations`). The standard output from the Java

process is captured, parsed as JSON, and returned to the client. In case of errors, such as invalid output or a non-zero exit code, the microservice responds with a structured JSON error message to facilitate client-side handling.

The microservice returns results in JSON format, encapsulating essential musical information such as the main key, the overall song structure, and the chord progression assigned to each section. This representation ensures both human readability and machine interpretability, supporting further processing by other backend components (e.g., a MIDI service) or visualization in the frontend.

This design allows the microservice to remain lightweight while delegating the computationally intensive task of statistical chord sequence generation to the Java backend. It also provides flexibility to extend the API with new endpoints without modifying the underlying generative model. For a comprehensive presentation of the generative method, the reader is referred to Raposo and Soares [18].

4.2.4. MIDI Microservice

The MIDI Microservice is responsible for converting textual chord progressions into the MIDI format, enabling integration of the generated harmonic structures into digital audio workstations (DAWs) or other MIDI-compatible tools. In our implementation, the service is built using `Node.js` with the `Express.js` framework, and it delegates the actual chord-to-MIDI conversion to a dedicated Python script. This design illustrates the flexibility of the architecture, allowing microservices to be implemented in different programming languages when appropriate.

The service exposes a single REST endpoint:

- /api/chords2midi/:chordprogression—Receives a textual chord progression as parameter and invokes the Python script `chords2midi.py`, passing the progression as an argument. The script generates an MIDI representation of the progression and returns the result as JSON. The microservice validates that the script executes successfully and ensures that the output can be parsed into valid JSON before returning it to the client.

Error handling is included to capture and report issues such as Python execution errors, malformed output, or invalid input. Standard error output from the Python process is logged for debugging purposes, but does not necessarily prevent a successful response.

The service listens on port 3003 by default and demonstrates how the proposed architecture accommodates hybrid environments, where Node.js manages HTTP requests, while Python performs specialized music-related processing.

The `chords2midi.py` script, implemented in Python, performs the core task of transforming symbolic chord progressions into MIDI data. It uses the `music21` [28] library to model chord symbols, temporal structure, and MIDI translation. The script follows these steps:

1. Parsing the progression: the input string is parsed into bars and chords (e.g., `C,Am7|F,G`) by splitting on the bar separator "`|`" and comma separators for chords within each bar.
2. Normalizing accidentals: the helper function `fix_flats()` ensures that flats are correctly interpreted by replacing textual representations (e.g., "`Bb`") with symbols understood by `music21`.
3. Generating a music21 stream: a `Stream` is created, with a tempo mark inserted at the beginning. Each bar is assumed to last four beats, and chords within the bar share the duration equally. For instance, a bar with two chords allocates two beats to each.
4. MIDI conversion: the chord stream is translated into a `MidiFile` object, written to an in-memory buffer, and encoded into a Base64 string to ensure safe transport over JSON.

5. Output: The script prints a JSON object containing the Base64-encoded MIDI, which the Node.js microservice then returns to the client.

As an example, a call to the following command:
http://localhost:3003/api/chords2midi/Dm7|G7|Cmaj7|Cmaj7
produces the following JSON response presented in Listing 2:

Listing 2. JSON response returned by the MIDI microservice for chord progression `Dm7|G7|Cmaj7|Cmaj7`. The `midi` string contains a base64 encoding of the chord sequence in the MIDI format.


```
{
  "midi": "TVRoZAAAAAYAAQACJ2BNVHJrAAAAFAD/UQMHoSAA/1gEBAIYCM5g/
y8ATVRyawAAAJUA/wMAAOAAQACQMloAkDVaAJA5WgCQPFqCuwCAMgAAgDU
AAIA5AACAPAAAkCtaAJAvWgCQMloAkDVagrsAgCsAAIAvAACAMgAAgDUAAJAw
WgCQNFoAkDdaAJA7WoK7AIAwAACANAAAgDcAAIA7AACQMFoAkDRaAJA3Wg
CQO1qCuwCAMAAAgDQAAIA3AACAOwDOYP8vAA=="
}
```


This modular approach separates HTTP request handling (Node.js) from domain-specific symbolic music processing (Python), providing extensibility and maintainability. Furthermore, as with the Node.js microservices code, the Python implementation will be made publicly available, ensuring reproducibility and transparency.

4.3. The Frontend

It is well known that the frontend of a system constitutes the user-facing component, serving as the interface through which users interact with backend services and data. This section presents the implementation of the frontend of our architecture, which is essentially realized as a user-friendly GUI-based (Graphical User Interface) VST plugin integrated into a Digital Audio Workstation (DAW).

4.3.1. The VST Plugin

Virtual Studio Technology (VST) is a widely adopted software interface standard that enables the integration of virtual instruments and audio effects into Digital Audio Workstations (DAWs). A VST plugin extends the functionality of a DAW by providing additional sound synthesis, processing, or control capabilities that seamlessly interact with the host environment. In practice, VST plugins are most commonly developed using the JUCE framework [29] (and its accompanying Projucer project management tool) in combination with the C++ programming language, which together offer a robust cross-platform foundation for audio software development. In line with this practice, our VST plugin was also implemented using JUCE and C++. As is standard in JUCE-based plugins, our implementation is internally divided into two main components, a separation enforced by the JUCE plugin model itself: the processor, which connects to the backend through the Gateway microservice and handles all processing tasks, and the editor (GUI), which contains the visual components and manages all interactions with the user.

Before starting to use the plugin, the user must configure it by entering their API key and the URL of the Gateway service. To enable this, our plugin includes a simple configuration window, shown in Figure 2 (available in the Settings menu). Both the API key and the backend URL are stored permanently; so, the user only needs to reconfigure the plugin if either of these values changes.

After this initial configuration, the user can largely ignore the backend, as all interactions can be handled directly through the plugin's GUI, shown in Figure 3. The interface is organized into several sections: at the top, users can select the key, song structure, and modulation type, and trigger chord generation via a dedicated button "Generate

chords". Below this, the "Full Song" section displays the entire generated progression, while individual sections (A, B, C, D, i, v) show the corresponding chords for each part of the song. Each chord sequence can be dragged directly to a MIDI track in the DAW. The layout is clear and hierarchical, allowing users to navigate and manipulate both complete progressions and specific sections with ease.

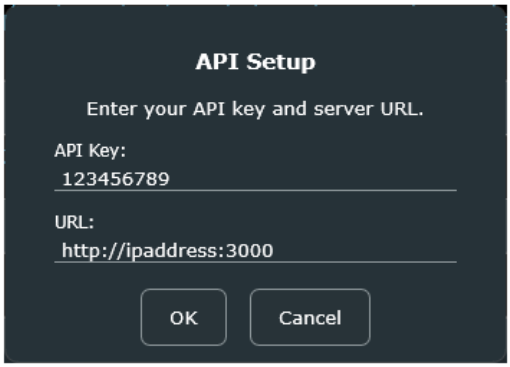

Figure 2. The plugin's configuration window.

Notice that each box containing chord sequences can be directly dragged and dropped into the DAW. Additionally, users may manually edit the chord progression before exporting it if desired. A further useful feature is that, if needed, the user can bypass the generative capabilities altogether and compose an entire progression manually, still taking advantage of the system's chords-to-MIDI functionality to drag it into the DAW. This feature leverages the MIDI microservice presented in Section 4.2.4, ensuring seamless integration between chord editing and MIDI export.

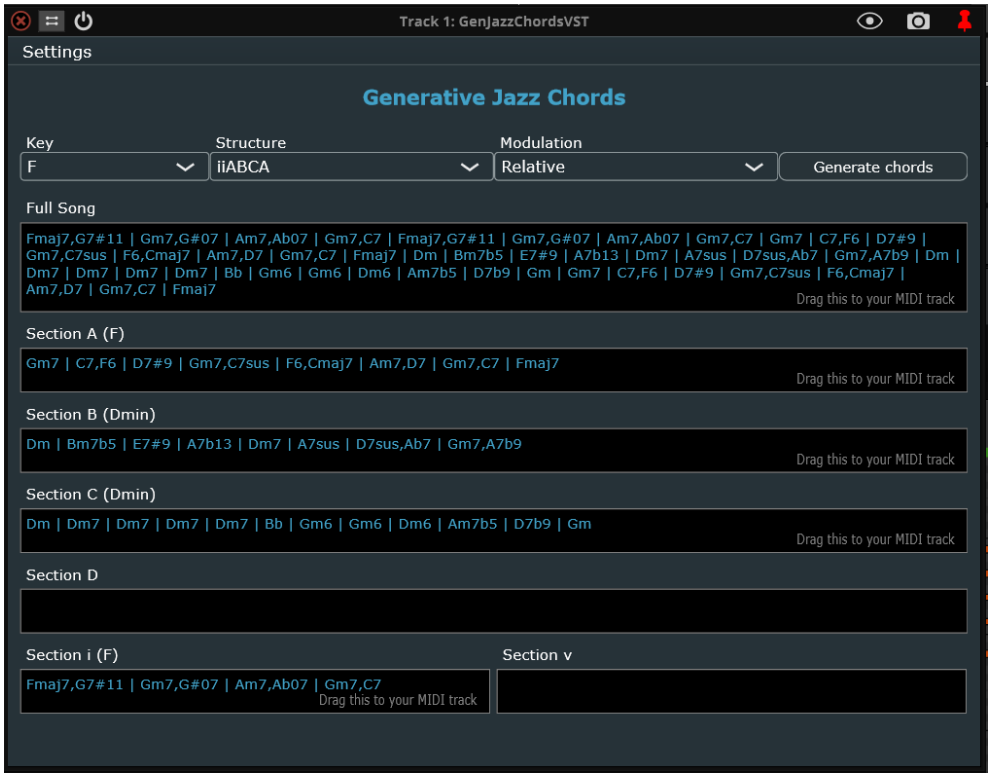

Figure 3. The VST plugin GUI.

The design of the GUI presented in Figure 3 directly follows the structure and parameters proposed in the generative method for jazz chord progressions by Raposo and

Soares [18], ensuring consistency between the theoretical framework and its practical implementation. Specifically, the inclusion of key selection, song structure definition, and modulation type reflects the configurable parameters highlighted in their work, while the hierarchical display of the full song and its subsections (A, B, C, D, i, v) mirrors the organization of the generated progressions described in the paper. This close alignment guarantees that the plugin faithfully translates the proposed method into an accessible interface. Nevertheless, the architecture here presented is sufficiently flexible to be adopted and adapted for other generative approaches, since its modular design naturally accommodates different sets of parameters and progression models while preserving an intuitive workflow for end users.

4.4. Hardware, Networking, and Environment

The proposed architecture is designed to remain independent of specific hardware configurations, networking equipment, or execution environments. As illustrated in Figure 1, the system follows a modular microservices approach in which each service operates autonomously and communicates with others through lightweight JSON-based requests and responses. This design principle ensures that the individual components—the VST plugin, the API Gateway, and the supporting microservices—can be deployed on heterogeneous hardware without requiring specialized devices or dedicated infrastructure.

The reliance on widely adopted, free, and cross-platform frameworks and libraries further strengthens the portability of the system, making it compatible with multiple operating systems and hardware environments. By employing standard HTTP protocols for inter-service communication, the architecture avoids dependencies on particular networking technologies, thus enabling flexible deployment in local setups, institutional infrastructures, or cloud-based environments.

This independence from hardware and networking constraints not only facilitates reproducibility and broader accessibility but also provides scalability, as additional instances of the microservices can be seamlessly deployed to accommodate higher computational demands or distributed workloads.

4.5. Tools, Libraries, and Frameworks

The system was developed using a combination of programming languages, frameworks, and libraries that provide support for both the frontend VST plugin and the backend microservices architecture, as summarized in Table 1 and illustrated in Figure 1. The selection was driven not only by performance and functionality requirements but also by the availability of free cross-platform tools and libraries, enabling deployment across diverse operating systems and hardware environments.

Table 1. Tools, libraries, and frameworks used in the system.

Technology/Library	Purpose/Description	Components
Node.js (v22.14.0)	Runtime for asynchronous requests and JSON communication.	Gateway Microservice, DB Microservice, Generative Microservice, MIDI Microservice
Express.js (4.21.2)	Routing HTTP requests in API Gateway and microservices.	Gateway Microservice, DB Microservice, Generative Microservice, MIDI Microservice
SQLite (3.46.1)	Lightweight relational database for persistent storage.	Database Microservice
sqlite3 (5.1.7)	Database interface for Node.js microservice.	Database Microservice
Java (17.0.12)	Implements generative engine for music creation.	Generative Microservice, Custom Generative Engine

Table 1. *Cont.*

Technology/Library	Purpose/Description	Components
Python (3.13.0)	Converts chord data into MIDI sequences.	Chords to MIDI Engine
music21 (9.7.1)	Generates MIDI files or streams from chord data.	Chords to MIDI Engine
JSON	Standard format for all requests/responses between frontend, API, and microservices.	All Components
C++ (17)	Core language for high-performance, real-time audio processing.	VST Plugin
JUCE (v8.0.5)	Framework for cross-platform audio applications and VST/AU plugins.	VST Plugin

4.6. System Workflow

This subsection briefly describes the overall system workflow, outlining how user interactions through the VST plugin GUI are translated into backend processes and coordinated with the supporting microservices. The sequence diagram in Figure 4 presents the complete workflow of the VST plugin system for chord generation and MIDI export. The process starts when the user requests a chord progression from the plugin frontend. The frontend sends this request along with the API key to the gateway microservice, which validates the key by querying the DB microservice. Once validated, the chord request is forwarded to the generative microservice, which invokes the generative engine to compute the chord sequence and returns the result to the frontend via the gateway. The user can then view the generated chord sequence in the plugin. If the user decides to drag and drop the chords into the DAW, the sequence is sent to the MIDI microservice, which converts it to MIDI data and returns it back to the frontend, completing the workflow. All interactions that require a response before proceeding, such as key validation, chord generation, and MIDI conversion, are represented as synchronous messages in the diagram.

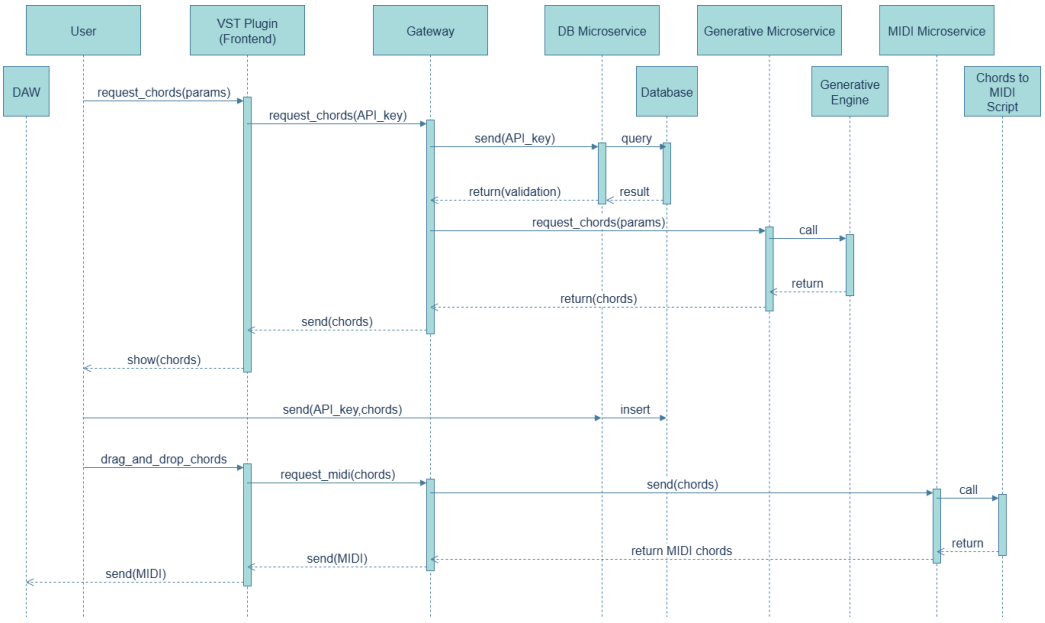

Figure 4. Sequence diagram illustrating the workflow of the VST plugin system, including interactions with the gateway and microservices.

Security Model for API Keys, Billing, and Subscription Functionalities

We propose a multi-layered security model to ensure safe access to the generative VST microservices while supporting API key management, billing, and subscription functionalities. The design integrates authentication, authorization, communication security, and auditing mechanisms, as described below.

API Key Management

Strong randomly created API keys are generated, with each key scoped to specific endpoints or actions, such as read-only or tier-limited access. API keys are stored securely using cryptographic hashing to prevent exposure in the event of a database compromise. Rate limits per key are enforced to prevent abuse and ensure fair usage according to the subscription tier.

Authentication and Authorization

API keys can be used in combination with bearer tokens, to carry subscription-level information and expiration data. Role-based access control differentiates users by subscription tier (e.g., free, standard, premium) and enforces endpoint-specific permissions. Automatic expiration and revocation mechanisms limit the lifetime of API keys or tokens and mitigate the risks associated with compromised credentials.

Billing and Subscription Security

Subscription status is verified on each API request to ensure that the access level aligns with the user's plan. Metadata related to subscriptions, usage, or billing is stored in encrypted form to ensure confidentiality and integrity.

Communication Security

All communications between clients and backend services must be encrypted.

Auditing and Logging

All access events, including API usage, failed access attempts, and billing-related requests, are logged to enable anomaly detection and traceability. Alerts can be generated for suspicious behaviors, such as repeated invalid API keys or unusual payment activity.

The proposed security model combines API key authentication, subscription-aware authorization, secure billing integration, encrypted communications, and auditing mechanisms. Together, these components provide a robust and flexible framework for secure access to generative VST microservices while accommodating different subscription tiers and billing models.

5. Evaluation

This section presents the performance evaluation of the two core microservices: the Generative Microservice and the Chords-to-MIDI Microservice. We first report the results of the local performance tests, which capture the raw computation times measured without network effects. These results establish the baseline performance of each service. Subsequently, we provide an Internet performance estimate for both microservices, in which the locally measured computation times are combined with estimated network overheads to approximate the end-to-end response times that would be experienced in a real deployment scenario.

All local tests were conducted on a consumer laptop equipped with a 3rd Gen Intel(R) Core(TM) i7-1355U CPU running at 1.70 GHz, 16 GB of installed RAM, and a 64-bit operating system (Microsoft Windows 11). All processing was performed on the CPU, with no use

of GPU acceleration. The software environment included Java version 17.0.12 (LTS, build 17.0.12+8-LTS-286, Java HotSpot 64-Bit Server VM), Python 3.13.0, and Node.js v22.14.0.

5.1. Generative and Chords-to-MIDI Performance

The runtime scenario for the local performance evaluation consisted of three Node.js-based microservices: the Gateway Microservice, the Generative Microservice, and the Chords-to-MIDI Microservice. These microservices are described in detail in Sections 4.2.1, 4.2.3 and 4.2.4. During the experiments, all three services were executed simultaneously on the test machine, each listening on a separate port. The Gateway Microservice was responsible for measuring the processing time (ms) and the size of the response data (bytes), which were logged to two separate CSV files: one for the generative processing measurements and another for the MIDI conversion measurements.

The performance of the Generative and MIDI conversion microservices was evaluated separately, because the generation of chord sequences does not always trigger their conversion to MIDI. Conversion occurs only on demand, when the user explicitly drags and drops a sequence into the DAW, or when the user chooses to convert explicitly written (not generated) chord sequences (see Figure 4). This separation ensures that each microservice's performance is measured independently under realistic usage scenarios.

The tests were automated using a Python script that invoked the gateway's generation endpoint 100 times for each of the following song structures: A, AB, ABC, ABCD, iABCD, and ivABCD. These structures were selected to cover all possible section types available in the generative engine. Additionally, since the generated JSON data do not include the full chord sequence for optimization purposes—only the structure and the chords associated with each section—the ivABCD structure represents, in theory, the largest possible data output.

For each target structure, we computed the minimum, maximum, mean, and standard deviation (SD) of both the JSON response sizes and the processing times. These descriptive statistics provide a concise characterization of the local performance of the generative process. The results are summarized in Table 2 and illustrated in Figure 5. The mean and standard deviation (SD) values are highlighted in bold to emphasize their statistical significance, as they represent the central tendency and variability of the system's performance across structures. As shown in Table 2, the mean response size increases progressively with the number of musical sections—from structure A to ivABCD—reflecting the expected growth in data output as the harmonic complexity rises. However, despite this increase in response size, the computation time remains relatively steady across structures, with only moderate variation in both the mean and SD. This stability indicates that the generative process scales efficiently with the structural complexity, maintaining responsiveness suitable for real-time composition and DAW integration. By highlighting these statistical measures, the table provides a clear understanding of the balance between the generative complexity and performance consistency.

Table 2. Generative performance: descriptive statistics by structure.

Structure	Response Size (Bytes)				Computation Time (ms)			
	Min	Max	Mean	SD	Min	Max	Mean	SD
A	89	235	**135.42**	**24.80**	726	1090	**801.99**	**68.38**
AB	35	331	**221.29**	**44.53**	661	1102	**869.42**	**93.07**
ABC	35	500	**308.83**	**87.31**	652	1111	**874.88**	**113.70**
ABCD	35	535	**378.76**	**116.86**	789	1082	**915.77**	**85.04**
iABCD	35	608	**456.74**	**125.47**	763	1048	**857.66**	**69.65**
ivABCD	35	767	**545.43**	**217.52**	779	1146	**867.44**	**68.68**

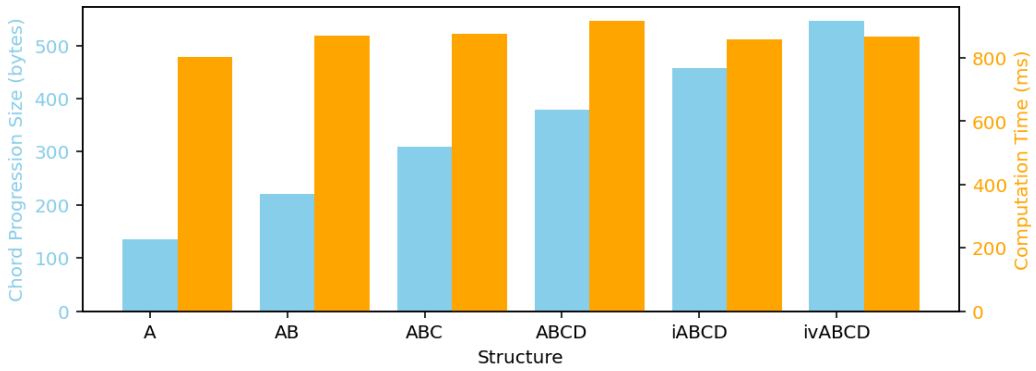

Figure 5. Generative microservice performance by structure. The coordinates correspond to response size (bytes) and computation time (ms) for each structure. Sample sizes are as reported in Table 2.

Table 3 summarizes the overall descriptive statistics for the JSON response sizes produced by the Generative Microservice and their corresponding processing times. On average, the generative process requires slightly over 800 ms to complete, i.e., less than one second.

Table 3. Generative performance: overall descriptive statistics.

Statistic	Response Size (Bytes)	Computation Time (ms)
Min	35	652
Max	767	1146
Mean	341.08	864.53
SD	182.87	90.77

For each target structure, we then computed the minimum, maximum, mean (average), and standard deviation of both the MIDI response sizes and the processing times. These descriptive statistics provide a concise characterization of the local performance for the MIDI conversion process. The results are summarized in Table 4 and further illustrated in Figure 6.

Table 5 summarizes the overall descriptive statistics for the MIDI response sizes produced by the Chords-to-MIDI Microservice and their corresponding processing times. On average, the conversion process requires slightly over 1200 ms to complete, i.e., a little bit more than one second.

Table 4. Chords-to-MIDI performance: descriptive statistics by structure.

Structure	Size (Bytes MIDI)				Time (ms)			
	Min	Max	Mean	SD	Min	Max	Mean	SD
A	35	1828	628.08	308.70	965	1797	1226.07	96.15
AB	35	2268	939.33	575.28	1161	1548	1274.97	64.91
ABC	35	2880	1312.96	890.36	1137	1485	1298.19	69.81
ABCD	35	3696	1664.24	1101.05	1128	1820	1328.12	95.83
iABCD	35	4000	1719.51	1341.83	1083	1580	1331.49	88.45
ivABCD	35	5020	1784.28	1859.69	1116	1786	1339.55	127.14

Table 5. Chords-to-MIDI performance: overall descriptive statistics.

Statistic	Size (Bytes MIDI)	Time (ms)
Min	35	965
Max	5020	1820
Mean	1322.24	1297.99
SD	1187.31	99.97

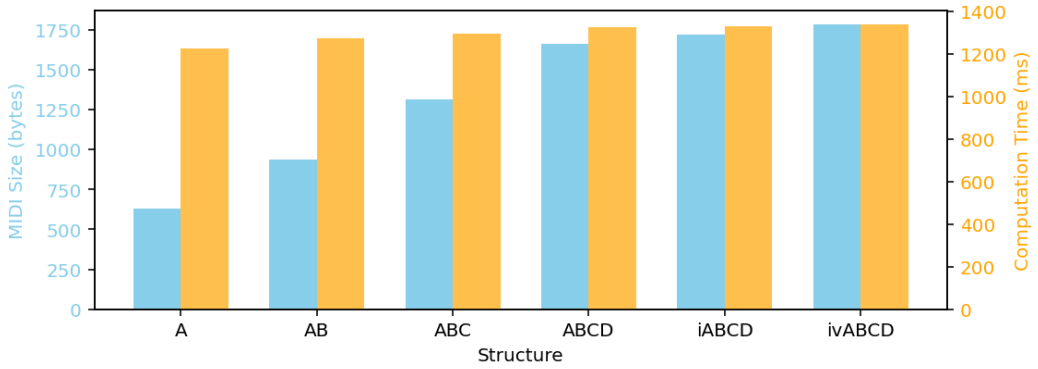

Figure 6. Chords-to-MIDI microservice performance by structure. The coordinates correspond to MIDI file size (bytes) and computation time (ms) for each structure. Sample sizes are as reported in Table 4.

5.2. Cloud Network Simulation for Generative and Chords-to-MIDI Microservices

The end-to-end time required to process a chord progression request from the VST plugin client to the cloud-hosted microservices can be expressed as

$$T_{\text{total}} = T_{\text{upload}} + T_{\text{compute}} + T_{\text{download}} + T_{\text{RTT}}, \tag{1}$$

where T_{upload} is the request transfer time, T_{compute} the microservice computation time, T_{download} the response transfer time, and T_{RTT} the round-trip latency between client and server. Since the request payloads are negligible compared to the response sizes, T_{upload} can be ignored. The main contributors are the computation time, round-trip latency, and response transfer.

The response transfer time is given by

$$T_{\text{download}}\ (\text{ms}) = \frac{S \times 8}{B \times 10^6} \times 1000, \tag{2}$$

where S is the response size in bytes, and B is the available bandwidth in Mbps.

5.2.1. Generative Microservice Performance

From Table 3, the mean response size is $\overline{S} = 341$ bytes, and the mean computation time is $\overline{T}_{\text{compute}} = 865$ ms. Considering the bandwidth scenarios in Table 6, the simulated end-to-end times are reported in Table 7.

As observed, the network transfer time is negligible (sub-millisecond) even under low bandwidth. The computation latency (652–1146 ms) is the dominant factor, with geographic round-trip times (10–200 ms) being the only additional relevant overhead.

Table 6. Representative cloud bandwidth scenarios.

Scenario	Bandwidth (Mbps)
Mobile (low-end)	3
Average broadband/4G	10
Good home internet	50
High-speed broadband	100
Data center/fiber	1000

Table 7. Simulated end-to-end latency for generative service (mean response, no RTT).

Bandwidth (Mbps)	T_{download} (ms)	T_{total} (ms)
3	0.91	865.9
10	0.27	864.8
50	0.05	864.6
100	0.03	864.5
1000	0.003	864.5

5.2.2. Chords-to-MIDI Microservice Performance

From Table 5, the mean response size is $\overline{S} = 1322$ bytes, and the mean computation time is $\overline{T}_{\text{compute}} = 1298$ ms. The simulated end-to-end latencies are reported in Table 8.

Table 8. Simulated end-to-end latency for Chords-to-MIDI service (mean response, no RTT).

Bandwidth (Mbps)	T_{download} (ms)	T_{total} (ms)
3	3.53	1301.5
10	1.06	1299.1
50	0.21	1298.2
100	0.11	1298.1
1000	0.011	1298.0

Here, the impact of the payload transfer is slightly more visible at very low bandwidths (3 Mbps adds $\approx$ 3.5 ms), but it remains negligible compared to the computation time (965–1820 ms).

To illustrate the relative contribution of the transfer time to the overall latency, we generated line plots for both microservices across representative bandwidth scenarios (3, 10, 50, 100, 1000 Mbps). Each plot shows the following:

- Processing time (dashed line): the mean microservice computation time, constant across bandwidths.
- Total time (solid line): the sum of the processing time and transfer time for each bandwidth.
- Annotated labels: the numerical difference between the processing and total times (i.e., the transfer time) is displayed above each point.

As shown in Figure 7, the total time line essentially overlaps with the processing time line because the transfer time is extremely small (less than 1 ms at 3 Mbps). The annotations above each point show the precise transfer time, confirming its negligible effect on overall latency.

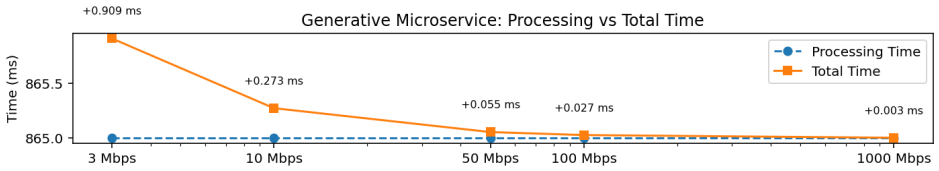

Figure 7. Generative microservice: line plot showing the processing time (dashed) and total time (solid) across bandwidth scenarios. The coordinates correspond to time in milliseconds (ms). Small annotations indicate the transfer time contribution, which is negligible compared to the processing time.

Figure 8 shows the Chords-to-MIDI microservice, which has a larger output size and slightly higher transfer times. Even at 3 Mbps, the transfer contributes only around

3.5 ms to the total time (mean computation time 1298 ms), confirming that the service is computation-bound.

Overall, these plots clearly demonstrate that, for both microservices, the transfer time is negligible compared to the processing time across all realistic bandwidth scenarios.

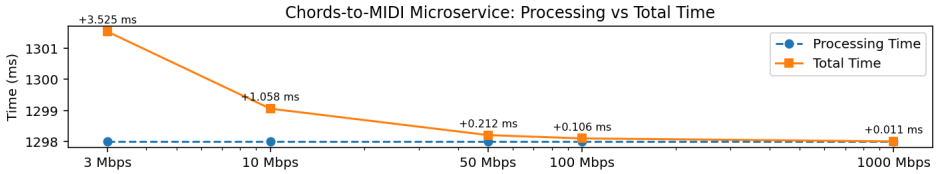

Figure 8. Chords-to-MIDI microservice: line plot showing processing time (dashed) and total time (solid) across bandwidth scenarios. The coordinates correspond to time in milliseconds (ms). Annotated labels indicate transfer time, which remains minor compared to computation time.

5.3. VST Plugin Integration and Usability

The VST plugin was designed to seamlessly integrate into a standard DAW workflow, allowing users to generate chord progressions and immediately utilize them as MIDI data. As shown in Figure 9, the complete chord sequence generated by the plugin can be dragged and dropped directly onto a MIDI track. Notably, the chords displayed in Figure 9 correspond exactly to the chord sequence presented in the plugin interface (cf. Figure 3), ensuring a direct and transparent mapping from the GUI to the DAW.

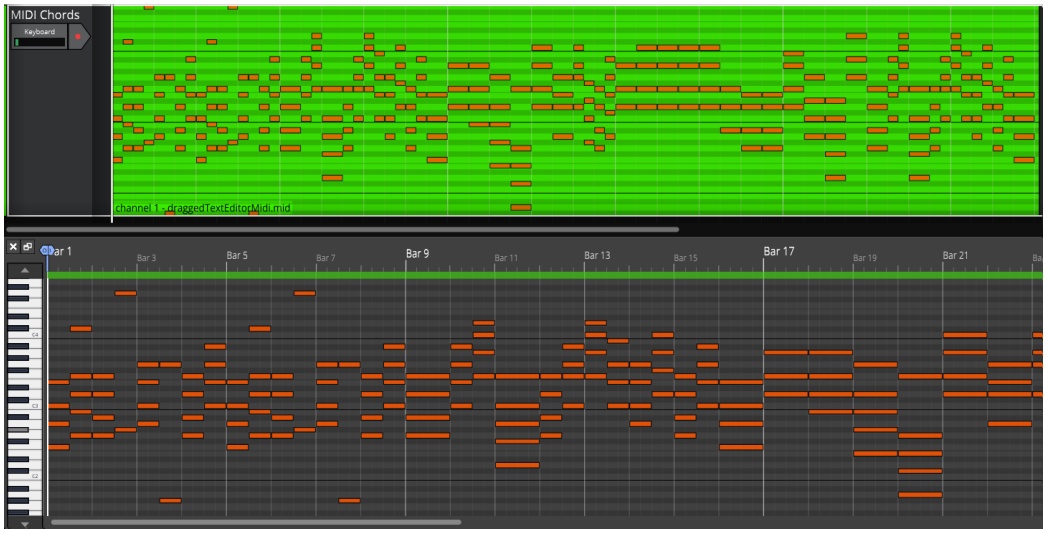

Figure 9. Generated chords dragged directly from the VST plugin to a MIDI track in a DAW.

This functionality was implemented to provide an intuitive workflow, enabling composers and producers to experiment with generated chord sequences in real time, directly within their preferred DAW environment. Users can thus quickly audition, modify, and arrange the generated progressions without leaving the plugin interface, enhancing both efficiency and creative flexibility.

6. Discussion

Generative VST plugins can operate either through cloud-based services or entirely on local machines, each approach offering distinct benefits and trade-offs. Cloud-based generative VSTs leverage powerful remote servers to handle complex computations, enabling highly sophisticated generative models without taxing the user's CPU or memory. They are ideal for collaborative workflows and quick access to updates and new features. However, they rely on stable internet connections, can introduce latency, and may raise

privacy concerns regarding user projects. Local-processing generative VSTs, in contrast, run entirely on the user's machine, providing low-latency performance and full data privacy, even offline. They are constrained, however, by the hardware's computational limits, which may restrict model complexity or real-time capabilities, and updates require manual installation. Table 9 presents a comparison between the two approaches.

Table 9. Comparison of cloud-based vs. local-processing generative VST plugins.

Feature/Aspect	Cloud-Based Generative VSTs	Local-Processing Generative VSTs
Processing Location	Remote servers (cloud)	User's local machine
Computational Power	High-leverages server-grade hardware	Limited by local CPU/GPU
Latency	Potential latency due to network connection	Low latency, suitable for real-time performance
Internet Requirement	Required	Not required
Data Privacy	Project data sent to cloud-potential privacy concerns	Full control over data, secure offline usage
Collaboration	Easier to share and collaborate remotely	Limited collaboration features
Updates/Model Improvements	Instant deployment of new features and models	Manual updates needed
Cost	Often subscription-based	Usually one-time purchase, hardware dependent
Scalability	Easy to scale for complex models	Limited by local hardware

The results presented in the previous sections highlight both the technical feasibility and the practical implications of adopting a cloud-based modular microservices architecture for generative music integration within a VST plugin. By decoupling core functionalities—such as chord progression generation and MIDI conversion—into independent cloud and local services, the system demonstrates scalability, flexibility, and ease of maintenance compared to monolithic plugin designs. At the same time, the performance evaluations revealed trade-offs between computational efficiency and network transfer overhead, raising important considerations for real-world deployment in creative workflows. In this section, we critically examine these findings, discussing the implications for latency-sensitive DAW environments, usability for musicians and producers, and the potential of this approach to inform future developments in generative audio technologies.

6.1. Modular Architecture and Maintainability

The adoption of a modular microservices architecture provides several distinct advantages over a monolithic design, particularly in terms of maintainability, scalability, and system resilience. By decoupling the chord generation engine from the chord-to-MIDI conversion microservice, each component can evolve independently, allowing targeted optimizations, bug fixes, or feature extensions without introducing regressions into other parts of the system. Furthermore, the introduction of a dedicated gateway microservice introduces an additional layer of abstraction between the client and the backend services. This gateway not only unifies access through a stable API but also enables seamless redirection of requests to alternative or updated service instances. Such a design simplifies maintenance, facilitates incremental updates, and allows controlled experimentation with improved algorithms or service versions without disrupting user experience. As a result, the architecture ensures both long-term sustainability and adaptability to future developments in generative music technologies.

We employed two different programming languages across the backend services: Java for the generative engine and Python for textual chords-to-MIDI translation. This choice was made to demonstrate that the proposed architecture is language-agnostic, thus enhancing modularity and flexibility. In principle, any other programming language could have been adopted for the services without impacting the overall architecture. Moreover, we leveraged an existing generative engine developed in Java from our previous work [18], which further motivated this design decision. Python was selected in part to take advantage of the rich ecosystem of music-related libraries, in particular `music21` [28], which greatly facilitated the chord-to-MIDI translation task. While using multiple languages can introduce challenges, such as managing dependencies, ensuring interoperability, and maintaining consistent deployment pipelines, modern containerization and service orchestration technologies mitigate most of these issues, allowing heterogeneous components to coexist seamlessly within the same architecture.

While the current generative engine focuses on jazz harmony, it is important to note that the proposed microservices-based architecture is not limited to this genre. The same system can be adapted to other musical styles with minimal modifications, such as adjusting the API endpoints and replacing the generative engine with one tailored to the desired genre. This flexibility highlights the generality of the architecture and its potential to support multi-style generative music systems.

6.2. Commercial Integration Potential

In addition to the modular organization of the core microservices, the adoption of an API key–based authentication mechanism combined with a dedicated database microservice provides a solid foundation for deploying the proposed architecture in commercial contexts. An API key system enables secure and controlled access to the generative services by uniquely identifying and validating client applications, thereby preventing unauthorized usage and ensuring accountability. When integrated with a database microservice, this approach facilitates the implementation of billing models and subscription-based access control. The database can efficiently track client activity, usage statistics, and subscription status, allowing the system to enforce restrictions based on plan type, quota, or payment validity. Such a design not only supports flexible business models—ranging from freemium to tiered subscriptions—but also enhances scalability by decoupling authentication and account management from the generative and conversion services. Consequently, the architecture is not only technically robust but also commercially viable, ensuring that service provision aligns with contractual and financial considerations while maintaining seamless integration with client applications.

6.3. Server-Side Processing and Client Independence

Another important advantage of the proposed architecture lies in the server-side execution of both the generative processing and the chord-to-MIDI conversion tasks. By delegating these computationally intensive processes to cloud-hosted microservices, the overall performance and reliability of the system become independent of the client hardware or local computing capabilities. This design choice ensures that users operating on less powerful devices—such as entry-level laptops or even mobile platforms—can benefit from the same high-quality and efficient generative output as those with high-performance workstations. Furthermore, centralizing the processing on the server side facilitates consistent performance across heterogeneous client environments, eliminating discrepancies caused by variations in hardware, operating system configurations, or background workloads. This not only broadens the accessibility of the system to a wider user base but also enhances maintainability and scalability, as performance optimizations and updates need only be

applied once on the server infrastructure rather than distributed across all client machines. In turn, this approach aligns with contemporary trends in cloud-based music production tools, enabling professional-grade performance without imposing hardware constraints on the end user.

6.4. Performance, Real-Time Usability, and Network Efficiency

Based on the performance evaluation presented in the previous section, the results provide strong evidence that the proposed system achieves real-time usability within a digital audio workstation environment. The generative microservice demonstrated the ability to produce complete song harmonies—i.e., entire chord sequences—in an average time of approximately 800 ms, which falls within an acceptable range for real-time interaction during composition. The chord-to-MIDI microservice exhibited an average execution time of less than 1.5 s. Although this duration is perceptible to the user, it remains sufficiently fast for the associated workflow, since the conversion and dragging of complete chord progressions into a DAW track are not latency-critical tasks. While the observed latencies are acceptable in the context of music composition and offline workflows, it is important to emphasize that they remain unsuitable for live performance scenarios, where real-time responsiveness is critical.

Furthermore, the results indicate that the overall system performance does not strongly depend on network bandwidth or communication speed, since the adopted data structures were optimized to minimize transfer times, making the architecture resilient across heterogeneous client environments. Both microservices are dominated by computation rather than data transfer. The generative microservice produces very small responses (tens to hundreds of bytes), making network transfer virtually irrelevant. The chords-to-MIDI microservice produces larger outputs (hundreds to thousands of bytes), leading to slightly higher network costs, but still minor compared to server-side computation. In realistic deployments, round-trip times (RTT) of 10–200 ms are more significant than bandwidth overheads.

To illustrate these aspects in practice, five demonstration videos (Videos S1–S5) have been submitted as Supplementary Material to this manuscript, showcasing the real-time usability of the system within a DAW environment and its seamless integration into music production workflows.

Regarding the API gateway, we do not anticipate significant performance issues, since the gateway itself does not perform any computational processing but only routes client requests to the appropriate services and returns their responses. The main potential concern lies in data transfer scalability, particularly under high volumes of concurrent requests. Nevertheless, this is a well-understood challenge in distributed architectures, and established solutions such as load balancing and horizontal scaling of the gateway can effectively mitigate such risks, ensuring reliable and efficient operation.

These findings empirically validate the core thesis of this work: a modular microservices-based architecture can successfully integrate advanced generative music capabilities into DAWs while preserving usability and workflow continuity. In practical terms, the measured performance ensures that the system supports creative tasks without interrupting the user's compositional process, thereby demonstrating both technical feasibility and musical practicality. Ultimately, this confirms that the proposed solution is not only technically sound but also user-friendly, making it suitable for real-world adoption in music production contexts.

6.5. Limitations and Future Work

A potential limitation of the proposed architecture concerns the handling of concurrent requests, which are very likely to occur in realistic real-time usage scenarios. For instance, simultaneous invocations of the generative microservice and the chord-to-MIDI conversion microservice by multiple clients could lead to performance degradation if not properly managed. Nevertheless, this limitation can be effectively mitigated with contemporary cloud computing infrastructures, which support elastic service provisioning and automatic scaling. Such capabilities allow computing resources to be dynamically allocated on demand, thereby ensuring that performance remains consistent even under high concurrency conditions. As future work, we plan to explore scalability using containerization, deploying multiple instances of each service concurrently. This setup would allow the system to efficiently handle increased user demand while maintaining responsiveness and reliability, providing empirical validation of its scalability.

Another limitation relates to latency sensitivity for live performance scenarios. While the system demonstrates acceptable responsiveness for composition and DAW integration, the measured generation time of approximately 800 ms for full chord sequences and the 1.5 s required for chord-to-MIDI conversion may be perceptible in interactive real-time performance contexts. Future work could explore incremental or streaming generation approaches that produce chord sequences progressively, thereby reducing the perceptual latency for live use.

A further limitation concerns the generalization capabilities of the generative model. The system may produce chord progressions biased toward the training dataset, limiting stylistic diversity or adaptability to unusual harmonic contexts. Future research could investigate style-conditioning mechanisms, adaptive learning from user feedback, or integration of multiple models to enhance versatility and creative control.

As a direction for future work, a systematic study of the architecture's concurrent resilience in a cloud-based implementation would be valuable. This would provide further insights into the scalability of the system and its robustness in supporting professional, large-scale, or commercial deployments.

7. Conclusions

This work demonstrates the feasibility and advantages of a modular microservices-based architecture for integrating generative music capabilities into VST plugins. By separating core functionalities—such as chord progression generation and chord-to-MIDI conversion—into independent services, the system achieves flexibility, maintainability, and scalability that are difficult to attain in monolithic designs. The architecture supports heterogeneous implementations, allowing different programming languages and tools to coexist while remaining interoperable through well-defined APIs.

The performance evaluations indicate that the system is suitable for composition workflows within DAWs, with latencies acceptable for creative production tasks. Server-side processing ensures consistent performance across client hardware, broadening accessibility for users with varied computing resources. Furthermore, the API gateway and database microservice provide a foundation for secure, commercially viable deployment, supporting subscription models and controlled access.

While current limitations include perceptible latency in live performance contexts and potential constraints under high concurrency, these challenges can be addressed through scalable cloud infrastructures, streaming generation techniques, and optimized resource provisioning. The architecture is also adaptable to multiple musical genres, demonstrating its generality and potential for broader applications in generative music.

Overall, the proposed system combines technical robustness, workflow usability, and commercial readiness, offering a practical approach for bringing advanced generative music tools to modern music production environments. Future work will focus on enhancing the scalability, reducing the latency for live performance, and improving the stylistic versatility of the generative models.

Supplementary Materials: The following supporting information can be downloaded at: https://www.mdpi.com/article/10.3390/fi17100469/s1, Video S1: VideoS1.mp4; Video S2: VideoS2.mp4; Video S3: VideoS3.mp4; Video S4: VideoS4.mp4; Video S5: VideoS5.mp4.

Author Contributions: Conceptualization, A.N.R. and V.N.G.J.S.; methodology, A.N.R. and V.N.G.J.S.; software, A.N.R.; validation, V.N.G.J.S.; formal analysis, A.N.R. and V.N.G.J.S.; data curation, A.N.R.; writing—original draft preparation, A.N.R. and V.N.G.J.S.; writing—review and editing, A.N.R. and V.N.G.J.S.; funding acquisition, V.N.G.J.S. All authors have read and agreed to the published version of the manuscript.

Funding: V.N.G.J.S. and A.N.R. acknowledge that this work was funded by FCT/MECI through national funds and, when applicable, co-funded EU funds under UID/50008: Instituto de Telecomunicações.

Data Availability Statement: The raw data supporting the conclusions of this article will be made available by the authors on request.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

API	Application Programming Interface
CSV	Comma-Separated Values
CPU	Central Processing Unit
DAW	Digital Audio Workstation
DB	Database
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
JUCE	Audio plugin development framework
MIDI	Musical Instrument Digital Interface
RAM	Random Access Memory
RDBMS	Relational Database Management System
REST	Representational State Transfer
SD	Standard Deviation
VST	Virtual Studio Technology

References

1. Fernández, J.D.; Vico, F. AI methods in algorithmic composition: A comprehensive survey. *J. Artif. Int. Res.* **2013**, *48*, 513–582. [CrossRef]
2. Hadjeres, G.; Crestel, L. The Piano Inpainting Application. *arXiv* **2021**, arXiv:2107.05944. [CrossRef]
3. Guo, R. Closing the Loop: Enabling User Feedback and Testing in Symbolic Music Generation through a Python Framework and Ableton Live Integration. In Proceedings of the AIMC 2023, Johor Bahru, Malaysia, 21–23 July 2023. Available online: https://aimc2023.pubpub.org/pub/9pokd9f8 (accessed on 13 September 2025).
4. Leider, C.N. *Digital Audio Workstation*, 1st ed.; McGraw-Hill, Inc.: Columbus, OH, USA, 2004.

5. Bianchi, D.; Avanzini, F.; Baratè, A.; Ludovico, L.A.; Presti, G. A GPU-Oriented Application Programming Interface for Digital Audio Workstations. *IEEE Trans. Parallel Distrib. Syst.* **2022**, *33*, 1924–1938. [CrossRef]
6. Buffa, M.; Vidal-Mazuy, A. WAM-studio, a Digital Audio Workstation (DAW) for the Web. In Proceedings of the WWW '23 Companion: Companion Proceedings of the ACM Web Conference 2023, New York, NY, USA, 30 April–4 May 2023; pp. 543–548. [CrossRef]
7. Buffa, M.; Demont, S. Can you DAW it Online?: The Challenges of a Web-Based Open Source Workstation. In Proceedings of the 2024 IEEE 5th International Symposium on the Internet of Sounds (IS2), Erlangen, Germany, 30 September–2 October 2024; pp. 1–8. [CrossRef]
8. Moffat, D.; Sandler, M.B. Approaches in Intelligent Music Production. *Arts* **2019**, *8*, 125. [CrossRef]
9. Xu, W. Research on the application of computer music production technology in new media environment. In Proceedings of the 2021 International Conference on Computer Information Science and Artificial Intelligence (CISAI), Kunming, China, 17–19 September 2021; pp. 824–827. [CrossRef]
10. Wang, Y. The Application of Computer Music Production Software in Music Creation. In Proceedings of the 2021 International Conference on Computer Technology and Media Convergence Design (CTMCD), Sanya, China, 23–25 April 2021; pp. 107–110. [CrossRef]
11. Ye, X. Music Creation with Computer Music Production Software. In Proceedings of the 2022 International Conference on Electronics and Devices, Computational Science (ICEDCS), Marseille, France, 20–22 September 2022; pp. 156–160. [CrossRef]
12. Kwiecień, J.; Skrzyński, P.; Chmiel, W.; Dąbrowski, A.; Szadkowski, B.; Pluta, M. Technical, Musical, and Legal Aspects of an AI-Aided Algorithmic Music Production System. *Appl. Sci.* **2024**, *14*, 3541. [CrossRef]
13. Moura da Silva, P.M.; Cavalcante Mattos, C.L.; de Souza Júnior, A.H. Audio Plugin Recommendation Systems for Music Production. In Proceedings of the 2019 8th Brazilian Conference on Intelligent Systems (BRACIS), Salvador, Brazil, 15–18 October 2019; pp. 854–859. [CrossRef]
14. Roberts, A.; Engel, J.; Raffel, C.; Hawthorne, C.; Eck, D. A Hierarchical Latent Vector Model for Learning Long-Term Structure in Music. In Proceedings of the 35th International Conference on Machine Learning (ICML), Stockholm Sweden, 10–15 July 2018; pp. 4364–4373. [CrossRef]
15. Payne, C. MuseNet. OpenAI Blog, 2019. Available online: https://openai.com/blog/musenet (accessed on 13 September 2025).
16. AIVA Technologies. AIVA: Artificial Intelligence Virtual Artist. Available online: https://www.aiva.ai/ (accessed on 13 September 2025).
17. Conklin, D.; Gasser, M.; Oertl, S. Creative Chord Sequence Generation for Electronic Dance Music. *Appl. Sci.* **2018**, *8*, 1704. [CrossRef]
18. Raposo, A.N.; Soares, V.N.G.J. Generative Jazz Chord Progressions: A Statistical Approach to Harmonic Creativity. *Information* **2025**, *16*, 504. [CrossRef]
19. Zhao, Y.; Yang, M.; Lin, Y.; Zhang, X.; Shi, F.; Wang, Z.; Ding, J.; Ning, H. AI-Enabled Text-to-Music Generation: A Comprehensive Review of Methods, Frameworks, and Future Directions. *Electronics* **2025**, *14*, 1197. [CrossRef]
20. Mon, Y.J. LSTM-Based Music Generation Technologies. *Computers* **2025**, *14*, 229. [CrossRef]
21. Lei, K.; Ma, Y.; Tan, Z. Performance Comparison and Evaluation of Web Development Technologies in PHP, Python, and Node.js. In Proceedings of the 2014 IEEE 17th International Conference on Computational Science and Engineering (CSE), Chengdu, China, 19–21 December 2014; pp. 661–668. [CrossRef]
22. Huang, X. Research and Application of Node.js Core Technology. In Proceedings of the 2020 International Conference on Intelligent Computing and Human-Computer Interaction (ICHCI), Sanya, China, 4–6 December 2020; pp. 1–4. [CrossRef]
23. Aderaldo, C.M.; Mendonça, N.C.; Pahl, C.; Jamshidi, P. Benchmark Requirements for Microservices Architecture Research. In Proceedings of the 2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE), Buenos Aires, Argentina, 22–22 May 2017; pp. 8–13. [CrossRef]
24. Fávero, L.F.; Almeida, N.R.d.; Affonso, F.J. A Systematic Mapping Study on the Modernization of Legacy Systems to Microservice Architecture. *Appl. Syst. Innov.* **2025**, *8*, 86. [CrossRef]
25. Vlček, J.; Bobák, M. Synchronized Music Streaming Driven by Microservice Architecture. In Proceedings of the 2023 IEEE 23rd International Symposium on Computational Intelligence and Informatics (CINTI), Budapest, Hungary, 20–22 November 2023; pp. 000093–000098. [CrossRef]
26. Lin, Y.B.; Cheng, C.C.; Chiu, S.C. MusicTalk: A Microservice Approach for Musical Instrument Recognition. *IEEE Open J. Comput. Soc.* **2024**, *5*, 612–623. [CrossRef]
27. Express.js—Fast, Unopinionated, Minimalist Web Framework for Node.js. Available online: https://expressjs.com/ (accessed on 12 September 2025).

28. Cuthbert, M.S.; Ariza, C. Music21: A Toolkit for Computer-Aided Musicology and Symbolic Music Data. In Proceedings of the 11th International Society for Music Information Retrieval Conference (ISMIR 2010), Utrecht, The Netherlands, 9–13 August 2010; Downie, J.S., Veltkamp, R.C., Eds.; pp. 637–642.
29. JUCE: A C++ Cross-Platform Audio and Plugin Framework. Official Website/GitHub Repository. 2025. Available online: https://github.com/juce-framework/JUCE (accessed on 13 September 2025).