AbletonBuddy: A Domain-Aware Natural Language Interface for Autonomous Control of Ableton Live via OSC

Pratham Vadhulas

Course: CS6235
Term: Fall 2025

November 27, 2025

Contents

1 Introduction

This project presents a full-stack application designed to bridge the gap between natural language intent and digital music production. The core objective is to develop a chat interface that translates specific user requests into Open Sound Control (OSC) messages to control Ableton Live in real-time. Unlike standard mapping tools, this system utilizes a Large Language Model (LLM) backend to interpret semantic intent (e.g., "Make the drums punchier") and execute the corresponding technical commands.

2 Motivation

The intersection of AI and creative tools is often hindered by the "translation gap" between human intent and software parameters.

The Efficiency Problem: Current state-of-the-art systems often rely heavily on generic protocols (such as MCP) or massive context injection to manage tool use. This approach is computationally inefficient, resulting in high latency and excessive token consumption, both of which are unacceptable in a real-time creative environment like music production.

The Workflow Solution: There is a critical need for a system that does not rely solely on brute-force context. Instead, we need a dynamic solution that leverages domain-specific heuristics, encoding the actual steps music producers take. By mimicking the workflow of a producer rather than just the API of the software, we can achieve a higher degree of autonomy with significantly lower computational overhead.

3 Related Work

The application of artificial intelligence in music production has seen significant growth, shifting from simple information retrieval and symbolic generation (MIDI) toward integrated, intelligent assistance within Digital Audio Workstations (DAWs) [**?**, **?**]. Recent efforts have focused on transforming AI from a generator into a co-creative partner capable of navigating complex software environments [**?**, **?**].

Current Landscape of Intelligent Music Assistants

Several prominent systems have emerged that attempt to bridge natural language and DAW control, each with distinct methodologies:

DAWZY: An open-source, voice- and text-driven assistant for the REAPER DAW [**?**]. It primarily utilizes code generation to execute low-level tasks, functioning effectively as a "scripting assistant" but lacking higher-level production nuance.

FilmComposer: Utilizes a multi-agent system operating a DAW to arrange and mix music specifically for silent films [**?**]. While innovative in its use of visual and rhythmic

inputs, its scope is limited to autonomous arrangement rather than granular control of existing production elements.

MixAssist: Introduces an audio-language dataset for training models to offer expert mixing advice [**?**]. It represents a step toward "expert systems" but focuses on providing advice rather than executing real-time control within the session.

Ableton Copilot & Loop Copilot: These systems represent the current state-of-the-art in session management [**?**, **?**]. Ableton Copilot is built on the Model Context Protocol (MCP) [**?**] to manage granular MIDI operations and device control. Similarly, Loop Copilot integrates LLMs with a Global Attribute Table (GAT) to maintain continuity during iterative editing.

Limitations of Current Approaches (The "Context" Gap)

While the systems above demonstrate the viability of AI in DAWs, they generally rely on the LLM's ability to translate natural language directly into actionable commands via the Model Context Protocol (MCP) or massive context injection [**?**, **?**].

This approach presents two critical inefficiencies:

Computational Inefficiency: Relying on MCP requires the system to constantly "read" and maintain the entire session state to maintain coherence. This "brute force" context management burns tokens rapidly and introduces latency, which is detrimental to the real-time flow of music production [**?**].

The "Theory vs. Production" Gap: Many existing models are trained on symbolic representations (music theory/MIDI) rather than audio engineering physics [**?**, **?**]. They struggle to bridge the gap between high-level creative intent (e.g., "Make it punchy") and the nuanced signal processing required to achieve it (e.g., specific compression ratios and EQ curves) [**?**].

Our Contribution: Domain-Specific Knowledge Priors

Our system addresses these limitations by moving away from generic MCP-based state recording. Instead, we propose a workflow that incorporates explicit, structured music producer knowledge priors.

Rather than relying on an LLM to "figure out" the session state from scratch every time, our system utilizes a domain-aware pipeline:

Intent Parsing: Decoupling the user's creative goal from the technical implementation.

Priors Consultation: referencing a static knowledge base of expert signal chains (e.g., standard vocal processing chains) rather than dynamically generating them via token-heavy reasoning.

Heuristic Execution: Using lightweight Open Sound Control (OSC) messages [**?**] mapped to these expert priors.

This approach allows the system to provide expert-level guidance that aligns with professional mixing philosophies while significantly reducing the token overhead and latency associated with standard MCP agents.

4 System Architecture & Methodology

4.1 High-Level Overview

The system is designed as a modular, event-driven application that decouples the user's natural language intent from the low-level technical execution. The architecture follows a Human-in-the-Loop (HITL) design, ensuring that generative AI suggestions are validated against structured "Knowledge Priors" before interacting with the live production environment.

4.2 The Tech Stack

Component	Technology	Role
Frontend	React.js	Provides the chat interface and real-time state visualization for the user.
Backend API	Flask (Python)	Orchestrates the application logic and manages websocket connections.
AI Orchestration	Marvin + Pydantic	Handles intent classification, entity extraction, and agentic workflow management using structured outputs.
Control Layer	AbletonOSC	The translation layer that converts Python commands into Open Sound Control (OSC) messages compatible with Ableton Live.

Table 1: Technology Stack Components

4.3 The "Agentic Swarm" Pipeline

Unlike standard "Single Shot" LLM calls, our backend utilizes a multi-step reasoning pipeline to ensure safety and accuracy in the creative workflow.

4.3.1 Step 1: Input & Disambiguation

Raw user input (e.g., "The drums sound weak") is first passed to a Disambiguation Module. This module checks for vagueness. If the request lacks context (e.g., "Fix it"), the system halts and requests clarification. If clear, it passes to the classifier.

4.3.2 Step 2: Classification & Task Extraction (The Router)

We utilize Marvin to classify the intent into specific domains.

Input: "Create a hiphop melody."

Classification: Composition_Task (vs. Song_Task or Track_Task).

Extraction: The system extracts structured entities using Pydantic models: {Task: "Create", Genre: "Hiphop", Element: "Melody"}.

4.3.3 Step 3: Agent Assignment & Knowledge Retrieval

Once classified, the task is assigned to a specialized Domain Agent (e.g., The Composition-Agent).

The Critical Difference: Instead of hallucinating a solution, the Agent retrieves Knowledge Priors—pre-validated heuristics (e.g., "Hiphop Melody = Minor Pentatonic Scale + 808 Bass + Syncopated Rhythm").

4.3.4 Step 4: Execution & Verification

The Agent attempts to execute the task via the OSC Layer.

Success: The state change is confirmed by Ableton (via an OSC reply).

Failure: If the track doesn't exist or the device is missing, an error flag is raised.

4.3.5 Step 5: The Summary Agent

Regardless of success or failure, the Summary Agent aggregates the logs. It translates the technical outcome back into natural language for the user (e.g., "I created a hiphop melody on track 1 using a minor pentatonic scale in C minor with syncopated rhythms." or "I couldn't create the melody—no MIDI track was available. Should I create a new track?").

4.4 Visual Architecture Diagram

Figure 1 illustrates the system architecture. The diagram shows the flow from the React frontend through the Flask/Marvin core, which includes the Disambiguator Node and Router that splits into specialized agents (Mixing Agent, Arrangement Agent). These agents consult the Knowledge Priors database before executing commands via the OSC protocol to Ableton Live. The feedback loop completes through OSC responses processed by the Summary Agent, which returns natural language feedback to the React UI.

Figure 1: System Architecture Diagram

5 Implementation Steps

The development of the system followed an iterative, incremental methodology. We began with a minimal viable command-line architecture to validate the core logic before progressively expanding into a full-stack application with a multi-agent swarm. This process was divided into five distinct phases.

5.1 Phase 1: Core Architecture & Agent Framework

The initial phase focused on establishing the foundational logic without the overhead of web protocols. We developed the core orchestration framework, which included the bi-directional Open Sound Control (OSC) layer for communicating with Ableton Live and the memory management system.

During this stage, the primary multi-stage pipeline—*Disambiguation* → *Classification* → *Extraction* → *Execution* → *Summarization*—was implemented and validated using a single "Song Agent." This isolated environment allowed us to perfect the conversation management (creation, resumption, and deletion) and task execution logic before introducing the complexity of a full agent swarm.

5.2 Phase 2: API Exposure

Once the CLI workflow was stable, we encapsulated the core logic within a Flask-based RESTful API. This phase required mapping the internal Python agent methods to exposed HTTP endpoints, establishing a standard for request handling, error propagation, and status codes. Rigorous integration testing was conducted using Postman to ensure that the API layer correctly translated external HTTP requests into the underlying agent operations, effectively preparing the system for client-side integration.

5.3 Phase 3: Frontend Integration

To bridge the gap between the backend logic and the end-user, we developed a React.js frontend. This phase focused on User Experience (UX), specifically the creation of a chat interface capable of capturing natural language input and rendering the system's real-time state. The frontend was tightly coupled with the Flask endpoints, requiring end-to-end validation to ensure that asynchronous agent responses were accurately visualized in the browser, thereby completing the full-stack loop.

5.4 Phase 4: Domain Expansion (The Agent Swarm)

With the full-stack architecture in place, the system was systematically scaled from a single agent to a comprehensive suite of eight specialized domain agents. We adopted a standardized expansion pattern for each new domain: defining the specific context, implementing the agent instructions, developing the OSC tools, and finally performing integration testing.

The resulting swarm covers the full breadth of DAW operations:

- **Structural Management:** The *Track*, *Scene*, *View*, and *Application Agents* manage the session layout and global settings.
- **Sonic Manipulation:** The *Device*, *Clip*, *Clip Slot*, and *Song Agents* handle the direct modification of audio and MIDI data.

By adhering to a modular implementation strategy, we ensured that adding complex capabilities (such as device control) did not disrupt the stability of the existing architecture.

5.5 Phase 5: Multi-Agent Collaboration & Optimization

The introduction of multiple agents necessitated a dedicated optimization phase. Initial stress testing with complex, multi-step prompts (e.g., "Create a new track and add a reverb") revealed bottlenecks in state synchronization and error propagation across agent boundaries.

To address this, we refined the coordination mechanisms to ensure memory consistency. This involved optimizing the task execution pipeline to handle sequential dependencies between agents and implementing robust recovery strategies for partial failures. This phase was critical in transforming a collection of individual tools into a cohesive, collaborative system.

6 Experimental Setup & Results

To validate the efficacy of the system in a way that is reproducible and directly tied to semantic control, we construct an automatic, objective evaluation based on a suite of natural language prompts. Concretely, we define a corpus of test cases spanning the major Ableton domains covered by our agents, with prompts stored in category-specific files such as `TRACK.md`, `DEVICE.md`, `CLIP.md`, and `SONG.md`. The corresponding agents expose eight high-level classes: *Application*, *ClipSlot*, *Clip*, *Device*, *Scene*, *Song*, *Track*, and *View*—that together span the controllable parameter space of Ableton Live for this work. Each line-level test case specifies: (i) a natural language instruction, (ii) any required initial preconditions on the Live set, and (iii) a canonical target post-condition on the world-state (e.g., which tracks/devices should exist, which parameters should be set to which values, how clips and scenes should be arranged). Higher-level notions of "composition" or artistic quality would require subjective, qualitative assessment and are therefore considered out of scope for this project.

6.1 Prompt Corpus & Split

The prompt corpus is used both during development (for iterative debugging of agents and tool wiring) and for held-out evaluation. To obtain an unbiased estimate of performance, we reserve 20% of the prompts in each domain as an evaluation set that is never seen during

development. At evaluation time, we sample from this held-out subset and execute the full agent stack end-to-end for each prompt, starting from a standardized initial Live set configuration.

6.2 Execution & Ground Truth Checking

For every evaluation prompt, the system is given only the natural language instruction; it must decide which agent(s) to invoke, which low-level Live API calls to make, and in what order. After execution, we snapshot the resulting Ableton world-state and compare it against the canonical target for that prompt. This comparison is purely symbolic and objective: a test is marked as *successful* if and only if all required predicates hold (e.g., a track with a given name exists, a specified device is present on a particular track, a parameter lies within a tolerance band), and as *failed* otherwise.

6.3 Aggregate Metrics

From these per-prompt binary outcomes, we compute several aggregate metrics: overall success rate across the entire evaluation set; per-domain success rates (e.g., Track, Device, Clip, Scene, Song, Application, View); and error breakdowns by failure mode (e.g., misparsed intent, wrong agent, correct agent but incorrect parameters). Because the evaluation harness logs all model calls and tool invocations, we additionally measure token usage and end-to-end latency for each prompt, validating that the "No-MCP" architecture remains cost-effective and responsive even under multi-step semantic requests.

6.4 Results

To directly evaluate the semantic interface in a task-oriented, objective manner, we construct a benchmark of natural language "test prompts" that mirror the three task classes above (structural, parameter-routing, and abstract semantic requests). This benchmark is drawn from the same pool of prompts used during system development, but we hold out 20% as a disjoint evaluation set. For each prompt, we specify a canonical target world-state in Ableton (e.g., the presence of tracks/devices, parameter values, or clip routing), and we execute the system end-to-end to determine whether the final Live set matches this target. This yields binary success/failure labels for each prompt, from which we compute aggregate metrics such as overall success rate, per-task-type success rate, and error breakdowns by agent (e.g., Track vs. Device vs. Translation Layer), as well as token and latency statistics for each successful completion.

In addition to this automatic evaluation, we envision a complementary human study following the within-subjects protocol described above: participants would complete the three task families both manually and with the semantic interface while we record TTC, interaction steps, and NASA-TLX scores. Human raters would also perform blind A/B listening on the resulting mixes for Task C to validate that automatically "successful" states correspond to

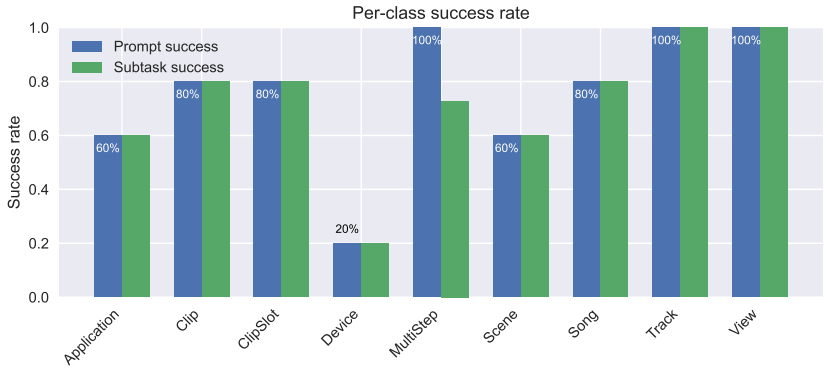

Figure 2: Per-class prompt- and subtask-level success rates on the held-out natural language test prompts. Bars show prompt-level success for each agent class, with overlaid subtask-level success for multi-step prompts, as computed by the analysis script.

perceptually preferable outcomes. Together, the objective prompt-level benchmark and the subjective human study provide a comprehensive picture of how well the system translates natural language intent into concrete production outcomes.

Concretely, our held-out prompt benchmark yields an overall success rate of 75.6% across all classes (see Figure 2). Per-class prompt-level success rates reveal that the interface is highly reliable for structural operations on *Track*, *View*, and *MultiStep* prompts (all at 100% success), and performs strongly on *Clip*, *ClipSlot*, and *Song* (each at 80%), while *Application*, *Scene*, and especially *Device* prompts remain more challenging (with success rates of 60%, 60%, and 20%, respectively). To better capture behavior on multi-step instructions, we additionally compute a subtask-aware metric following the same logic as our analysis script: simple prompts are treated as a single subtask, whereas multi-step prompts contribute multiple subtasks based on their internal decomposition. Under this lens, we observe that multi-step prompts achieve a subtask success rate of 72.7%, indicating that, even when entire sequences occasionally fail, a substantial fraction of their individual tool calls and intermediary goals are nevertheless executed correctly.

7 Meta Skills

This project required moving beyond standard software engineering practices into the realm of experimental AI system design. The challenges encountered—specifically regarding latency, hallucinations, and protocol translation—necessitated the development of several key meta-skills.

7.1 Hybridizing Stochastic and Deterministic Logic

A primary learning outcome was realizing the limitations of "pure" Generative AI in real-time systems. Initially, the temptation was to allow the LLM to manage the entire state via

Class	# Prompts	Prompt success	Subtask success
Application	5	60.0%	60.0%
Clip	5	80.0%	80.0%
ClipSlot	5	80.0%	80.0%
Device	5	20.0%	20.0%
MultiStep	5	100.0%	72.7%
Scene	5	60.0%	60.0%
Song	5	80.0%	80.0%
Track	5	100.0%	100.0%
View	5	100.0%	100.0%

Table 2: Quantitative summary of held-out evaluation results by agent class. Prompt success is the fraction of prompts for which the final Ableton world-state exactly matches the canonical target; subtask success is the fraction of subtasks completed correctly, following the subtask accounting implemented in `analyze_results.py`.

the Model Context Protocol (MCP). However, we identified that this approach introduced unacceptable latency and token costs ("burning tokens") for a creative workflow.

We developed the skill of **Architectural Decoupling**: recognizing which parts of a system require creative reasoning (the "What") and which require rigid, hard-coded execution (the "How"). By implementing "Knowledge Priors" (deterministic look-up tables) alongside the LLM, we learned to optimize for computational efficiency without sacrificing the flexibility of natural language, effectively bridging the gap between static automation and dynamic AI agents.

7.2 Constraint Engineering in Probabilistic Systems

Working with Large Language Models introduces non-determinism—a critical risk when controlling live audio software where a hallucinated command could corrupt a session. This project shifted our focus from simple "Prompt Engineering" (trying to persuade the model) to **Constraint Engineering** (forcing the model to obey syntax).

By utilizing libraries like Marvin and Pydantic to enforce strict schema validation *before* any command reaches the OSC layer, we learned to treat the LLM not as a magic box, but as an untrusted component that requires rigorous sanitation. This meta-skill involves designing "guardrails" that allow for creative expression while mathematically guaranteeing system stability.

7.3 Semantic Mapping of Abstract Domains

A significant challenge was translating vague human artistic intent (e.g., "make it punchier") into precise numerical parameters (e.g., "set Compressor Ratio to 4:1, Attack to 10ms"). This required developing the skill of **Domain-Specific Abstraction**.

We had to codify the tacit knowledge of a music producer into a structured logic that a machine could parse. This went beyond coding; it required analyzing the "physics" of music production and creating a translation layer that maps qualitative adjectives to quantitative signal processing commands. This experience highlighted the difficulty and necessity of grounding AI reasoning in domain-specific reality rather than general text patterns.

8 Conclusion & Future Work

8.1 Refinement & Stabilization

The current development phase focuses on system hardening. While the functional feature set is complete, continuous refinement is applied to improve response latency and handle edge cases in user intent. This stage prioritizes the resolution of bugs discovered during real-world usage scenarios and the optimization of the OSC message throughput to ensure a seamless, real-time user experience.

9 Deliverables