# WILDFX: A DAW-POWERED PIPELINE FOR IN-THE-WILD AUDIO FX GRAPH MODELING

*Qihui Yang  Taylor Berg-Kirkpatrick  Julian McAuley  Zachary Novack*

University of California San Diego
La Jolla, USA
qiy009@ucsd.edu, tberg@ucsd.edu, jmcauley@ucsd.edu, znovack@ucsd.edu

## ABSTRACT

Despite rapid progress in end-to-end AI music generation, AI-driven modeling of professional Digital Signal Processing (DSP) workflows remains challenging. In particular, while there is growing interest in neural black-box modeling of audio effect graphs (e.g., reverb, compression, and equalization), existing research pipelines often lack direct access to the plugin chains, routing structures, and metadata used in professional production workflows. We introduce **WildFX**, a Docker-containerized pipeline for generating multitrack audio mixing datasets with rich effect graphs, powered by a professional Digital Audio Workstation (DAW) backend. **WildFX** supports DAW-hosted commercial and open-source plugins in VST/VST3/LV2/CLAP formats, together with routing structures such as sidechains and crossovers and a parallelized rendering workflow. A minimalist metadata interface simplifies project and plugin configuration. Experiments provide an initial validation through blind estimation of mixing graphs, plugin parameters, and gain values, illustrating that WildFX can produce graph-labeled DAW-rendered datasets suitable for downstream learning tasks. The code is available at: https://github.com/IsaacYQH/WildFX

## 1. INTRODUCTION

Recent advances in large-scale music generation have produced capable end-to-end systems [1, 2, 3, 4, 5, 6], spanning sample-level generation [7, 8] through complete songs. However, end-to-end generation does not directly address professional production workflows, which rely on Digital Audio Workstations (DAWs), heterogeneous Digital Signal Processing (DSP) plugins, and flexible signal routing.

Learning-based audio-effect research includes plugin identification [9], parameter estimation [10, 11], neural effect modeling [12, 13, 14], style transfer [15], AI-assisted mixing [16], and full audio processing graph estimation [17]. Many approaches use differentiable or Python-native processors because they facilitate controlled experiments and gradient-based optimization. RTNeural and Neutone support deployment of neural audio models [18, 19], while products from Neural DSP indicate commercial interest [20]. Nevertheless, these systems do not generally provide automated dataset-generation access to heterogeneous DAW-hosted plugin chains and routing structures.

This creates a dataset-generation gap: many learning tasks require both rendered audio and labels describing the effects, parameters, gains, routing, and graph structure that produced it. A practical pipeline should therefore let users specify plugin collections, parameter spaces, and graph-generation settings, then render paired audio and annotations with minimal manual DAW operation.

We introduce **WildFX**, a Dockerized pipeline for generating multitrack audio datasets with graph annotations from a professional DAW backend. A lightweight metadata schema represents input stems, plugins, parameters, routing edges, gains, sidechain inputs, and output paths. WildFX controls `REAPER` on Linux and supports VST, VST3, LV2, and CLAP plugins, including bridged Windows plugins, as well as routing structures such as sends, sidechains, splitters, and multiband processing.

WildFX provides three practical capabilities: DAW-rendered generation from user-specified plugins and routing structures, machine-learning-ready graph and audio export, and parallelized batch rendering. We validate the generated data through blind estimation of mixing graphs, plugin parameters, and gains. These experiments assess whether WildFX produces structured examples suitable for downstream graph-inference tasks rather than proposing a new estimation model.

## 2. RELATED WORK

### 2.1. Learning-Based Audio Effect Analysis and Modeling

Learning-based audio-effect research includes both neural modeling of audio effects and related analysis or inference tasks. Neural audio effect modeling, in the strict sense, focuses on learning models that simulate the input-output behavior of audio effects. Adjacent tasks include identifying effect types from processed audio, estimating effect parameters, recognizing the order of effects in a chain, and inferring full audio processing graphs.

Early work in this domain focused on basic classification tasks such as identifying guitar amplifier and pedal types from processed audio [9]. A particularly common subfield is *parameter estimation*, where gradient-based methods are used to infer the parameters of an audio effect module, such as a low-frequency oscillator [10] or compressor [11].

More recently, researchers have studied order-aware audio effect chain recognition, where the goal is to identify both the effect types and their order in a processing chain [21]. Related work has further addressed full audio processing graph estimation, where the model must infer both the graph topology and the parameters of the effects in the processing graph [17, 15].

Beyond these tasks, substantial effort has been directed toward black-box neural modeling of audio effects [12, 22, 13, 14]. Rather than replacing existing DSP processors entirely, a complementary line of work studies how learning systems can identify, control, or estimate the behavior of existing processors and processing chains [23, 24]. This motivates hybrid or grey-box settings in which

**Table 1:** Linux VST3 plugin compatibility.

| Tool | Status | Count | % |
|---|---|---|---|
| `pedalboard` | Successfully loaded | 56 | 58.3 |
| | Failed to load | 40 | 41.7 |
| DawDreamer | Fully working | 74 | 77.9 |
| | Loaded but silent | 3 | 3.2 |
| | Failed to load | 17 | 17.9 |
| | Crashed | 1 | 1.1 |
| WildFX | Successfully loaded | 96 | 100.0 |

neural models reason about plugin parameters, effect identities, and graph structure while retaining the underlying DSP processor.

Complementary work controls or estimates existing processors rather than replacing them, enabling style transfer and hybrid or grey-box systems that retain the underlying DSP implementation [23, 24, 15].

### 2.2. Python Packages for Neural Audio Effect Processing

The growing interest in learning-based audio-effect research has led to several Python packages aimed at facilitating research and development in this domain. These packages can be broadly grouped into differentiable audio-processing implementations and interfaces to existing audio plugins or rendering engines. Among differentiable packages, DDSP (Differentiable DSP) [25] provides differentiable implementations of common audio processing operations such as filters and oscillators. Other differentiable implementations include DiffMoog [26], GRAFX [27], PyNeuralFx [28], and NablAFx [29]. In terms of the other kind, the `pedalboard` library offers a lightweight Python interface for applying built-in effects and selected VST3 plugins, and Steinmetz et al. [23] developed custom interfaces for specific commercial plugins. DawDreamer [30] is closely related to WildFX, as it provides a programmable rendering engine for plugin-based audio graphs. These tools are valuable for research workflows that require programmatic access to audio effects.

To assess VST3 compatibility, we tested publicly available Linux plugins using loading tests for `pedalboard` and WildFX/REAPER, and one-second renders at 44.1 kHz with a 512-sample buffer in isolated subprocesses for DawDreamer. The tests cover 96 plugins for `pedalboard` and WildFX and 95 for DawDreamer; other supported formats and bridged Windows plugins were not evaluated in this study.

## 3. DATASET & GENERATION PIPELINE

Here, we describe our methodology for building WildFX. As a data generation and processing pipeline, we first detail WildFX's dockerized deployment environment, followed by its interface protocol with the DAW `REAPER`. We then discuss the core data structure in WildFX, and how such objects are used for efficient data generation.

### 3.1. Deployment Environment

We utilize a Docker container to encapsulate all dependencies and software components required for end-to-end execution. To conserve storage, both the dataset directory and the Linux plugin folder are mounted from outside the container. Our provided `Dockerfile` sets up three key tools:

1. (1) `REAPER`, a professional DAW;
2. (2) `Wine`, a compatibility layer for running Windows applications on Linux;
3. (3) `yabridge`, a compatibility layer for mounting Windows audio plugins.

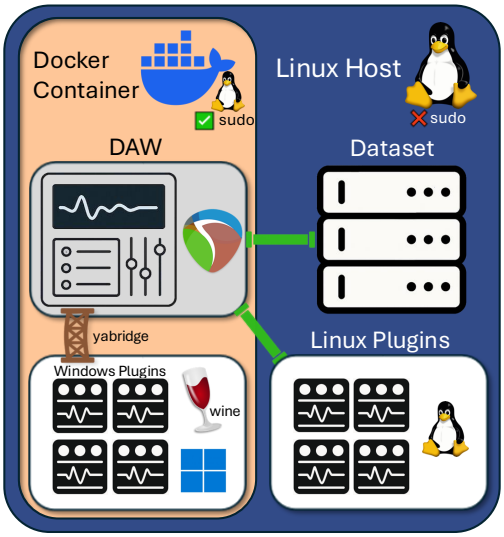


**Figure 1:** Deployment Environment of **WildFX**

Together, `Wine` and `yabridge` allow REAPER to host `.exe`-format plugins designed for Windows on Linux servers. REAPER then renders audio based on project metadata, as detailed in Section 3.4. All generated data are written to mounted directories and accessible outside the container.

Docker is a platform for developing, shipping, and running applications in lightweight, isolated containers. Using the provided `Dockerfile`, users can easily build a self-contained environment for the WildFX data generation pipeline, eliminating the need for manual dependency management—a critical feature for reproducible open-source research. To further simplify setup, we include a `devcontainer.json` configuration compatible with Visual Studio Code's Dev Container extension. This approach is preferable to manual `docker run` workflows, as VS Code automates some file system mounting and permission handling, and offers a more transparent development experience by exposing the source code directly within the editor.

For many users, particularly in institutional or shared computing environments, `sudo` or root access is restricted. However, tools like `REAPER` and many commercial audio plugins require system-level dependencies, such as the `jackd` audio server, which is essential for proper DAW operation. WildFX addresses this challenge by encapsulating all necessary services within the container, reducing the need for users to manually install system-level audio dependencies on the host, though Docker itself may require appropriate permissions depending on the environment. This design ensures accessibility across diverse user environments. The complete deployment architecture is illustrated in Figure 1.

### 3.2. DAW Control

We use the Python library `reapy` to interface with `REAPER` through its scripting API, `ReaScript`. While `ReaScript`

allows programmatic control of `REAPER` via Python, it typically relies on a graphical user interface. In contrast, `reapy` can be imported into any Python environment and establishes a connection to a running `REAPER` instance. Through its `Project` class, users can control essential DAW operations, including track creation, selection, and routing (e.g., adding sends).

However, some global project-level functions, such as rendering, are not fully exposed in `reapy`. These can be accessed via the lower-level API in `reapy.reascript_api`, including through the general-purpose command interface `Main_OnCommand`, which triggers REAPER actions by their command IDs.

Since `REAPER` is designed for interactive GUI use, operating it in headless mode via APIs provides no built-in error reporting or inspection tools. This limitation poses challenges for reliable automation. To ensure correctness, we validated every step of the pipeline manually on a Linux system with GUI support before deploying the system in headless environments.

### 3.3. Data Structure

The WildFX pipeline defines a robust and extensible data structure tailored for modeling professional audio effect graphs. To ensure compatibility with Digital Audio Workstations (DAWs) and commercial plugins, the schema is both hierarchical and modular. YAML files encode the high-level project structure, including audio input sources, their routing through effect chains, and the directed connections between nodes. JSON files specify plugin-level details such as parameter constraints, valid ranges, and preset configurations. This separation between structural and plugin-specific metadata improves clarity, reusability, and maintainability. Additionally, we provide utilities to convert the `Project` class into a `networkx` graph object, as discussed in Section 3.3.2.

#### *3.3.1. Core Components*

The WildFX data schema comprises several interrelated Python classes, each representing a critical component of the audio effect graph:

- `FXSetting`: Represents individual audio effects within an effect chain. Each `FXSetting` instance encapsulates the plugin's name (`fx_name`), type (`fx_type`), optional preset index (`preset_index`), parameters (`params`), input/output channel counts (`n_inputs`, `n_outputs`), and optional sidechain configuration (`sidechain_input`). Parameter validation is performed using constraints defined in the corresponding JSON preset files, ensuring DAW-compatible and realistic parameter settings. In Section 3.4.2, we demonstrate how this structured representation facilitates efficient parallel processing.
- `ChainDefinition`: Encapsulates sequences of FXSetting objects into effect chains, defining how audio signals sequentially flow through multiple effects. Each chain specifies outgoing connections with gain information (`next_chains`) to subsequent effect chains, enabling complex multi-path routing (e.g., parallel processing and sidechains). Empty chains are allowed as pass-through stages in equivalent layered representations.
- `InputAudio`: Clearly defines audio input files, types and their entry points (input_FxChain) into the graph, thus establishing explicit start points within the effect graph structure.
- `Project`: Represents the complete specification of an audio effect graph for a given mixing project. A Project instance aggregates multiple ChainDefinition instances, a set of InputAudio objects, and the final output audio path. Robust validation within the Project class ensures graph integrity, enforcing critical constraints such as acyclicity, valid indexing, correct sidechain routing, and logical consistency of audio inputs and outputs.

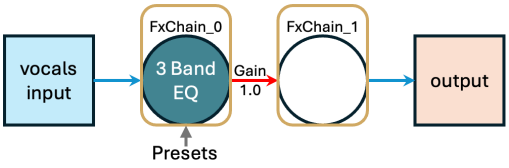


**Figure 2:** Mixing Graph with the Provided Sample

Figure 3 shows a compact example of the project-level YAML metadata and plugin-level JSON preset metadata used by WildFX.

In Figure 3, the `params` field is empty because the plugin uses preset index 2 from its corresponding JSON preset file. All plugin parameters are sampled from the discretized set defined in `valid_params`. A `null` value in the JSON file indicates use of the plugin's default setting. Additionally, the values in `next_chains` represent gain in linear amplitude and are not rescaled to perceptual units (e.g., dB). Figure 2 illustrates the corresponding graph structure.

#### *3.3.2. Graph Features Designation*

The heterogeneous graph data are stored as Python pickled `networkx` graph objects for efficient loading. For audio input nodes, it has three features: `type='audio'`, `label` and `instance`, where `instance` marked the exact source of the audio and `label` could be defined for specific tasks. AFx plugin nodes, whose `type` should be `'fx'`, have an additional feature `params` stored as a Python dictionary holding all parameters specific to each AFx plugin instance.

The `type` and `label` features also exist in edges along with a `gain` feature. All available values are: `type={'send_signal', 'split_signal'}`, `label={'main', 'control'}`. For example, the feature of a splitter's outgoing edge would be `{type='split_signal', label='main'}`, and the feature of the incoming edge for sidechain/control signal would be `{type='send_signal', label='control'}`. `gain` are all in linear scale.

#### *3.3.3. Data Structure Restrictions*

For the ease of batch rendering in a headless client, we impose the following restrictions to the data structure metadata files.

1. **Graph Acyclicity:** The audio effect graph must be a Directed Acyclic Graph (DAG).
2. **Valid Chain Indices:** All references to chain indices (e.g., in `next_chains`, `input_FxChain`, and `sidechain_input`) must correspond to valid chain indices.
3. **Defined Inputs:** Chains designated as input entry points via `input_audios` must have no predecessors.
4. **I/O:** There must be exactly one final output chain with no outgoing connections and at least one input source.

**YAML Project Metadata Example**

```yaml
FxChains:
  - FxChain:
      - fx_name: "VST3: 3 Band EQ"
        fx_type: "eq"
        preset_index: 2
        params: []
        sidechain_input: null
    next_chains:
      1: 1
  - FxChain: []

input_audios:
  - audio_path: "vocals.wav"
    audio_type: "vocal"
    input_FxChain: 0

output_audio: "mixed_output.wav"
customized: true
```

**JSON Plugin Preset Example**

```json
{
  "fx_name": "VST3: 3 Band EQ",
  "fx_type": "eq",
  "n_inputs": 2,
  "n_outputs": 2,
  "valid_params": {
    "Low": [0.0, 0.01, ..., 1.0],
    "Mid": [0.0, 0.01, ..., 1.0],
    "High": [0.0, 0.01, ..., 1.0]
  },
  "presets": [
    [null, null, null, 0.12, 0.69, 0.21],
    [null, null, null, 0.72, 0.63, 0.09],
    [null, null, null, 0.05, 0.00, 0.28]
  ]
}
```

**Figure 3:** Example WildFX metadata: YAML project structure (left) and JSON plugin preset specification (right).

5. **FX Type Compliance:** Every effect type (`fx_type`) must be within the predefined allowed set.
6. **Splitter Rules:** (1) Splitters must be positioned as the last effect in a chain. (2) Chains with multiple outgoing connections (`next_chains`) must have a splitter as the last effect. (3) The final output chain cannot end with a splitter.
7. **Sidechain Constraints:** (1) Only one sidechain-enabled plugin per chain is allowed. (2) Sidechain source chains must not originate from splitter outputs. (3) Sidechain routing can only happen within the chains inside the same project and same layer.
8. **Parameter Validity:** Effect parameters must match allowed values and ranges defined in their corresponding JSON preset files.
9. **Input and Output Channels:** `n_inputs` and `n_outputs` must match values specified in the corresponding preset JSON.
10. **Non-Negative Gains:** Connection weights (gain values) defined in `next_chains` must be non-negative floats.

The same-layer sidechain constraint should not be interpreted as allowing circular sidechain dependencies. If chain A uses chain B as a sidechain input while chain B uses chain A as a sidechain input, neither control signal can be rendered first. Such cycles are therefore rejected by the graph acyclicity constraint. These restrictions ensure that each `FxChain` rendering task is executable in one pass. Section 3.5 discusses the resulting representational coverage and equivalent layered decompositions.

### 3.4. Generation Pipeline

The typical workflow of the WildFX pipeline is illustrated in Figure 4. After installing the desired audio effect plugins and launching `REAPER`, a bundled `.lua` script retrieves an inventory of the plugins recognized by the DAW. From this inventory, a subset can be selected for preset generation using `gen_presets.py`, which creates structured JSON files defining valid parameter spaces and example presets. Once these files are available, `gen_projects.py` generates project-specific YAML metadata encoding the routing topology and plugin configurations. The topology is generated layer-by-layer using the batch-processing algorithm in Figure 5. Finally, the rendering engine processes this metadata while respecting plugin order, signal flow, and sidechain routing.

#### *3.4.1. Pipeline Interface*

To support large-scale preset generation from real-world audio plugins, our system provides a modular command-line interface with key functionalities outlined below:

In the preset generation part:

- **Plugin Selection.** Users can specify plugins via (1) `-plugin-name {NAME TYPE}` to target specific plugins, (2) `-plugin-list` to load a batch from a CSV, and (3) `-use-reduced-set` or `-use-full-set` to quickly sample canonical plugin sets.
- **Parameter Sampling.** The system discretizes the range of each parameter of interest using a distribution suited to that parameter, reducing sampling heterogeneity across diverse parameter types.
- **Cluster-Based Validation.** Cluster-based validation can be disabled with `-no-cluster-validation`. When enabled, we sample candidate parameter configurations from each plugin's valid ranges and render them on a reference audio input. We retain $\min(2^{N_p}, 1024)$ presets, where $N_p$ is the number of adjustable parameters, and render at least $5\times$ as many candidates, with a minimum of 100. The rendered outputs are peak-normalized, converted to mono if needed, and represented by MFCC features. KMeans is then used to select representative parameter settings, reducing redundant samples while covering perceptually distinct plugin behaviors.

To generate structured multitrack mixing projects with diverse audio effect graphs, our system provides a configurable interface that supports the following key functionalities:

- **Dataset-Aware Stem Extraction.** Users can define custom logic to extract stems and instrument labels (e.g., `Bass`, `Guitar`) via `-dataset-name` and project-folder parsing. The

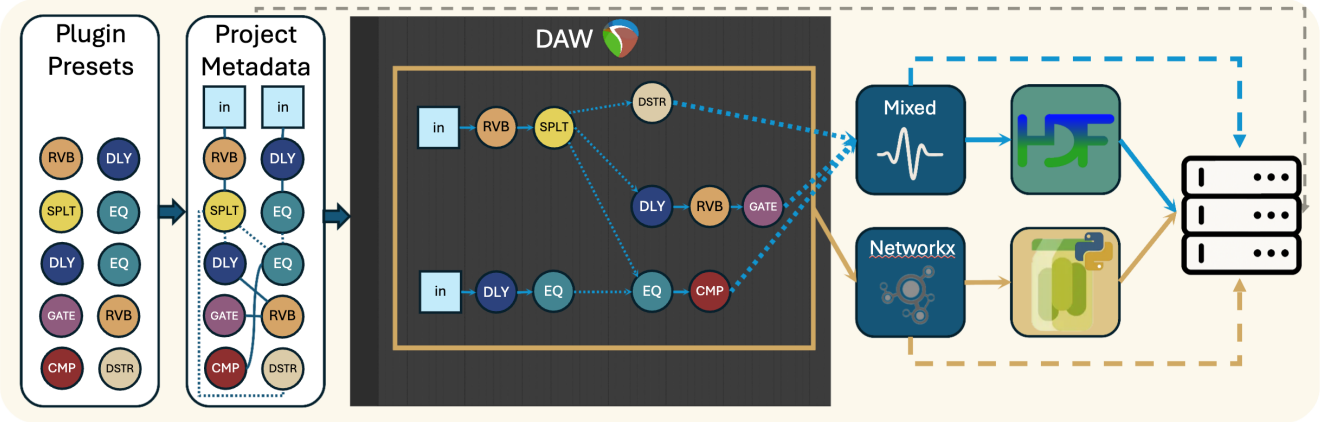


**Figure 4:** Overview of the **WildFX** workflow. Users specify plugins to generate presets, which are then used to synthesize project metadata defining audio effect graphs. The metadata is rendered using `REAPER` in a headless environment and exported as waveforms, compressed HDF5 data, YAML graph specifications, or pickled `NetworkX` objects. External dashed lines indicate optional data storage.

provided interface supports directory structures similar to Slakh2100 [31].

- **Topology Synthesis.** The pipeline builds DAG topologies with configurable complexity (`-complexity`), number of FX chains (`-min-chains`, `-max-chains`), number of input stems (`-min-stems`, `-max-stems`), and controlled probability of sidechains and splitters.
- **Plugin Assignment.** Available FX plugins are loaded from a preset directory and sampled per chain with optional sidechain insertion. Chains needing splitters automatically insert plugins with matching band counts.
- **Parametric Control.** Each chain's depth is sampled from a user-defined categorical distribution (e.g., `-chain-depth 0.1,0.6,0.3`) to model varying processing complexity across chains.
- **Variability Support.** The `-variable-density` flag randomizes graph density, chain-depth distribution, and routing probabilities across projects, producing a broader distribution of configurations for training and evaluation.
- **Batch Project Generation.** The script supports generating large numbers of projects (`-num-projects`), with automatic resampling over existing folders to meet the target count. Output is serialized in validated YAML format using the shared data schema.

The main function can save human-readable `.wav` and `.yaml` files or training-ready HDF5 audio and pickled `NetworkX` graph objects.

#### *3.4.2. Layer-based Multi-Project Batch Processing*

The minimal processing unit in DAWs is a single path of AFx. Therefore, we need to divide the whole processing graph into chunks only containing paths. To ensure efficient and dependency-aware rendering of complex audio effect graphs, WildFX employs a layer-based execution strategy inspired by Kahn's algorithm [32] for topological sorting. In this scheme, each project is treated as a directed acyclic graph (DAG), where:

- Each node corresponds to an `FXChain` (in the graph data we provide, each node represents a plugin), representing a sequence of AFx plugins applied.
- Directed edges encode audio signal flow, including main connections and sidechain routings, between chains.

This motivates the `FxChain` data structure. WildFX performs mixing in Python before effect processing at each layer, which enables independent paths to be grouped for parallel rendering without requiring the full multitrack graph to be routed inside the DAW.

At runtime, the system processes nodes in dependency-respecting layers: (1) Nodes with zero unresolved dependencies across all projects (i.e., in-degree zero) form the current layer, (2) These nodes are batch-processed in parallel, including audio mixing, plugin parameterization, and rendering via `REAPER`. (3) Upon successful execution, their outputs are stored, and in-degrees of successor nodes are decremented. (4) Nodes whose in-degree reaches zero become eligible for the next layer.

The approach in Figure 5 provides deterministic, deadlock-free scheduling while supporting **(1) splitter-aware routing**, which handles multi-output plugins that branch signals into parallel chains; **(2) sidechain synchronization**, which coordinates sidechain sources and consumers; and **(3) batch-aware grouping** of independent rendering tasks.

We do not quantify speedup relative to simpler project-level parallelism in this work; such an ablation remains future work. The present evaluation establishes dependency-correct execution for the supported acyclic audio-effect graphs.

### 3.5. Mixing Graph Representational Coverage

WildFX represents a project as a DAG with one sink node of out-degree zero and at least one source node of in-degree zero. Multiple outputs can be represented as separate output graphs, duplicating shared AFx settings along the corresponding output paths when necessary.

The constraints in Section 3.3.3 permit at most one sidechain-enabled plugin and one terminal splitter per `FxChain`. The project generator samples graphs that satisfy these constraints and vali-

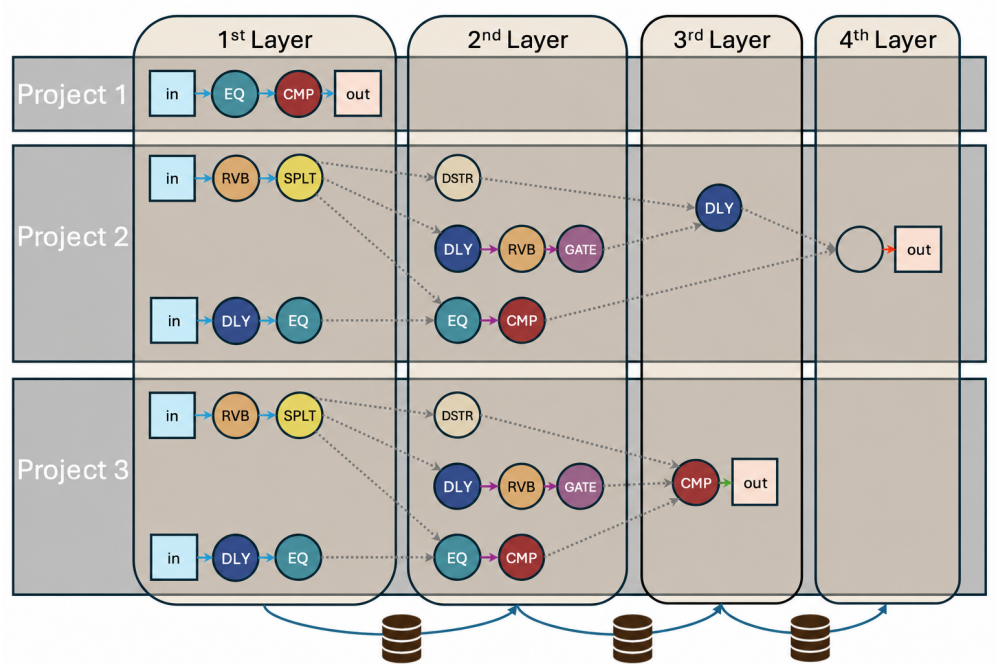


**Figure 5:** Layer-based multi-project batch processing. Four layers across three projects are determined using Kahn's topological sorting algorithm [32], treating AFx processing paths as hyper-nodes. Paths within a layer share a color, and dependencies across layers are shown as gray dashed arrows. The DAW batch-processes the paths in each layer and stores outputs required by subsequent layers.

dates them before rendering; it does not claim to execute every unrestricted routing graph directly. More complex chains can often be expressed by splitting them into multiple `FxChain` objects, each containing one sidechain-enabled plugin or terminal splitter. Empty chains are available to preserve signal dependencies when an equivalent layered representation requires a pass-through stage.

For example, direct routing from a splitter output into a sidechain consumer is excluded because it can create nested send configurations that are difficult to batch safely. An equivalent supported representation separates the sidechain-consuming portion into the next layer and inserts an empty chain on the control path. This decomposition clarifies the coverage provided by the constrained data structure without implying unrestricted graph support.

## 4. EXPERIMENTS

### 4.1. Dataset

Using the **WildFX** pipeline, we generated two datasets—referred to as the *shallow* and *deep* datasets—derived from Slakh2100 [31], each configured with distinct structural and signal-processing parameters. Both datasets consist of 5,000 training projects and 270 validation projects, each representing a multitrack audio mixing session. The controlled five-plugin inventory in Table 2 is intended as a proof-of-concept evaluation rather than a demonstration of the pipeline's full plugin-format coverage. For both configurations, we enabled `-variable-density`, with a sidechain probability of 0.2 and a splitter probability of 0.1. The deep dataset samples instrument stems from `{piano, guitar, bass, drums}`, whereas the shallow dataset is restricted to `{guitar, bass}`. The generation parameters are detailed in Table 3.

**Table 2:** Plugin Inventory

| AFx Plugin | Format | Type |
|---|---|---|
| 3 Band EQ | VST3 | EQ |
| 3-Band Splitter | JS | Splitter |
| Samurai Delay | VST3 | Delay |
| Schroeder | VST3 | Reverb |
| ZamCompX2 | VST3 | Compressor |

**Table 3:** Dataset Configuration Parameters

| Parameter | Shallow | Deep |
|---|---|---|
| `-min/max chains` | 3, 5 | 3, 10 |
| `-min/max stems` | 1, 2 | 1, 4 |
| `-chain-depth distribution` | [.1, .7, .2] | [.1, .3, .4, .2] |

Using complete Slakh2100 tracks rather than fixed-length crops, average processing times per generated project were 12 and 15 seconds for the shallow and deep datasets under a dummy `jackd` server, compared to 3.5 and 5 seconds with an `alsa jackd` server. Since Slakh2100 tracks have variable durations, these timings are averaged per complete generated project.

### 4.2. Blind Estimation of Mixing Graphs

We evaluate whether WildFX can generate graph-labeled audio data suitable for structured learning by training a blind audio processing

**Table 4:** Comparison of performance metrics across dataset configuration and processing settings. **PT Loss**: Prototype Decoding Loss. **PR Loss**: Parameter Loss. Lower is better for all metrics.

| Setting | PT Loss | PR Loss | Gain Loss | Edge Error Rate | Node Error Rate |
|---|---|---|---|---|---|
| Shallow + Autoencoding | **2.335** | 2.121 | **1.101** | 0.087 | **0.568** |
| Shallow + Decoding | 2.511 | 2.096 | 1.120 | 0.093 | 0.625 |
| Deep + Autoencoding | 2.872 | **2.089** | 1.417 | **0.070** | 0.625 |
| Deep + Decoding | 2.927 | 2.448 | 1.510 | 0.094 | 0.690 |

graph estimator. We follow the framework of Lee et al. [17], where a reference audio signal $y$ is first encoded into a latent representation $z$, and the processing graph is reconstructed in two stages. The model first predicts a prototype graph $\hat{G}_0$, including processor types and graph connectivity, and then estimates the remaining continuous attributes $\hat{p}$, including plugin parameters and edge gains. We evaluate two model configurations from this framework: autoencoding (AE), where the prototype graph is available to the parameter estimator, and prototype decoding (PD), where the prototype graph is first inferred from the audio representation. Combined with the shallow and deep WildFX datasets, this yields four experimental settings.

*Training objective.* We train with the same two-stage objective as Lee et al. [17]. The total loss is

$$\mathcal{L} = \mathcal{L}_{\text{proto}} + \mathcal{L}_{\text{param}},$$

where $\mathcal{L}_{\text{proto}}$ measures errors in the discrete prototype graph and $\mathcal{L}_{\text{param}}$ measures errors in continuous plugin and gain parameters. The prototype loss is

$$\mathcal{L}_{\text{proto}} = \frac{1}{N}\sum_{n=1}^{N} \text{CE}(\hat{T}_n, T_n) + \frac{1}{2E}\sum_{e=1}^{E}\Big[\text{CE}(\hat{O}_e, O_e) + \text{CE}(\hat{I}_e, I_e)\Big],$$

where $N$ and $E$ are the numbers of nodes and edges, $\hat{T}_n$ denotes processor-type logits, and $\hat{O}_e$, $\hat{I}_e$ denote output- and input-channel logits for each edge. We use cross-entropy with label smoothing of 0.1 for these categorical terms. The parameter loss is

$$\mathcal{L}_{\text{param}} = 8 \cdot \frac{1}{|M_p|}\sum_{i \in M_p} |p_i - \hat{p}_i| + 0.4 \cdot \frac{1}{|M_g|}\sum_{j \in M_g} |g_j - \hat{g}_j|,$$

where $M_p$ and $M_g$ are valid masks for plugin parameters and edge gains, respectively. The coefficients 8 and 0.4 follow the implementation used in our reproduced graph-estimation model and balance the relative scale of parameter and gain errors.

*Optimization and validation.* We train each model using AdamW [33] with a peak learning rate of $3 \times 10^{-4}$, weight decay of 0.1, gradient clipping with maximum norm 1.0, and a batch size of 32. The learning rate uses a linear warmup for 1k steps followed by linear decay, and each model is trained for 30k steps. Validation is performed on held-out projects generated from an unseen source distribution, while the reported results use the final checkpoint after training. This budget is substantially shorter than the 200k-step training used in the original study [17], and each setting is trained once. The results should therefore be interpreted as an initial validation of the generated dataset and evaluation protocol rather than a statistically conclusive comparison between model variants.

*Metrics.* Table 4 reports five metrics. Prototype (PT) loss evaluates the discrete graph-decoding objective, including processor-type and edge-channel prediction. Parameter (PR) loss measures the error of predicted plugin parameters over valid parameter positions. Gain loss measures the error of predicted edge gains. Edge error rate measures the fraction of incorrectly predicted graph connections or edge attributes, while node error rate measures the fraction of incorrectly predicted processor nodes or node configurations. Together, these metrics evaluate both the structural and continuous components of the predicted audio processing graph.

*Results and discussion.* The results show that the generated WildFX datasets can be used to train and evaluate a structured graph-estimation model. The shallow dataset is easier than the deep dataset, as expected, since it contains fewer stems and shorter processing chains. AE generally performs better than PD, especially on prototype and gain-related metrics, which suggests that estimating continuous parameters is easier when the graph structure is known. In contrast, node error rates remain relatively high, indicating that processor-type recognition is a major failure mode in this more realistic plugin-rendered setting. These results highlight the difficulty of blind graph estimation on DAW-rendered plugin chains and motivate WildFX as a benchmark for future models, rather than as evidence that the reproduced estimator is fully optimized.

## 5. LIMITATIONS & DISCUSSIONS

While our experimental results fall short of those reported in the original study [17], we attribute this discrepancy to two primary factors. First, the original implementation relied on a limited set of simplified plugins with minimal parameter complexity, which reduces the difficulty of the estimation task. In contrast, our dataset incorporates real-world plugins with diverse and nontrivial behaviors. Second, our models were trained on datasets that are orders of magnitude smaller—approximately 100 times smaller—than those used in the original work. Despite these constraints, our reproduced model achieves reasonably strong performance, indicating that existing graph-estimation methods can be applied to WildFX data, while leaving substantial room for improvement.

These results highlight that there remains substantial room for improvement in blind estimation of audio mixing graphs. We believe that our dataset serves as an important milestone toward this goal, offering a more realistic and challenging benchmark for future research in this area.

## 6. REFERENCES


[1] A. Agostinelli, T. I. Denk, Z. Borsos, J. Engel, M. Verzetti, A. Caillon, Q. Huang, A. Jansen, A. Roberts, M. Tagliasacchi *et al.*, "MusicLM: Generating music from text," *arXiv:2301.11325*, 2023.

[2] C. Donahue, A. Caillon, A. Roberts, E. Manilow, P. Esling, A. Agostinelli, M. Verzetti *et al.*, "SingSong: Generating musical accompaniments from singing," *arXiv:2301.12662*, 2023.

[3] Z. Novack, G. Zhu, J. Casebeer, J. McAuley, T. Berg-Kirkpatrick, and N. J. Bryan, "Presto! Distilling steps and layers for accelerating music generation," in *ICLR*, 2025.

[4] S. Forsgren and H. Martiros, "Riffusion: Stable diffusion for real-time music generation," 2022. [Online]. Available: https://riffusion.com/about

[5] Z. Evans, J. D. Parker, C. Carr, Z. Zukowski, J. Taylor, and J. Pons, "Stable Audio Open," *arXiv:2407.14358*, 2024.

[6] R. Yuan, H. Lin, S. Guo, G. Zhang, J. Pan, Y. Zang, H. Liu, Y. Liang, W. Ma, X. Du *et al.*, "YuE: Scaling open foundation models for long-form music generation," *arXiv:2503.08638*, 2025.

[7] Z. Novack, Z. Evans, Z. Zukowski, J. Taylor, C. Carr, J. Parker, A. Al-Sinan, G. M. Iodice, J. McAuley, T. Berg-Kirkpatrick, and J. Pons, "Fast text-to-audio generation with adversarial post-training," *arXiv:2505.08175*, 2025.

[8] S. Nercessian and J. Imort, "InstrumentGen: Generating sample-based musical instruments from text," *arXiv preprint arXiv:2311.04339*, 2023.

[9] M. Comunità, D. Stowell, and J. D. Reiss, "Guitar effects recognition and parameter estimation with convolutional neural networks," *arXiv preprint arXiv:2012.03216*, 2020.

[10] C. Mitcheltree, C. J. Steinmetz, M. Comunità, and J. D. Reiss, "Modulation extraction for LFO-driven audio effects," *arXiv preprint arXiv:2305.13262*, 2023.

[11] S. H. Hawley, B. Colburn, and S. I. Mimilakis, "SignalTrain: Profiling audio compressors with deep neural networks," *arXiv preprint arXiv:1905.11928*, 2019.

[12] M. A. Martínez Ramírez, E. Benetos, and J. D. Reiss, "Deep learning for black-box modeling of audio effects," *Applied Sciences*, vol. 10, no. 2, p. 638, 2020.

[13] C.-Y. Yu and G. Fazekas, "Singing voice synthesis using differentiable LPC and glottal-flow-inspired wavetables," *arXiv preprint arXiv:2306.17252*, 2023.

[14] M. Comunità, C. J. Steinmetz, and J. D. Reiss, "Differentiable black-box and gray-box modeling of nonlinear audio effects," Feb. 2025, arXiv:2502.14405 [cs]. [Online]. Available: http://arxiv.org/abs/2502.14405

[15] C. J. Steinmetz, S. Singh, M. Comunità, I. Ibnyahya, S. Yuan, E. Benetos, and J. D. Reiss, "ST-ITO: Controlling audio effects for style transfer with inference-time optimization," Oct. 2024, arXiv:2410.21233 [cs]. [Online]. Available: http://arxiv.org/abs/2410.21233

[16] J. Koo, M. A. Martínez-Ramírez, W.-H. Liao, S. Uhlich, K. Lee, and Y. Mitsufuji, "Music mixing style transfer: A contrastive learning approach to disentangle audio effects," in *ICASSP 2023-2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2023, pp. 1–5.

[17] S. Lee, J. Park, S. Paik, and K. Lee, "Blind estimation of audio processing graph," in *ICASSP 2023 - 2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Jun. 2023, pp. 1–5, iSSN: 2379-190X. [Online]. Available: https://ieeexplore.ieee.org/document/10096581

[18] J. Chowdhury, "RTNeural: Fast neural inferencing for real-time systems," 2021. [Online]. Available: https://arxiv.org/abs/2106.03037

[19] C. Mitcheltree, B. Teleaga, A. Fyfe, N. Masuda, M. Schäfer, A. Bradic, and N. Tokui, "Neutone SDK: An open source framework for neural audio processing," 2025. [Online]. Available: https://arxiv.org/abs/2508.09126

[20] Neural DSP Technologies Oy, "Neural DSP plugins," https://neuraldsp.com/plugins, accessed: 2026-05-31.

[21] A. Wada, T. Nakamura, and H. Saruwatari, "Hyperbolic embeddings for order-aware classification of audio effect chains," in *Proceedings of the 28th International Conference on Digital Audio Effects (DAFx25)*, 2025.

[22] C. J. Steinmetz and J. D. Reiss, "Efficient neural networks for real-time analog audio effect modeling," *arXiv preprint arXiv:2102.06200*, 2021.

[23] C. J. Steinmetz, N. J. Bryan, and J. D. Reiss, "Style transfer of audio effects with differentiable signal processing," *Journal of the Audio Engineering Society (JAES)*, 2022.

[24] C. J. Steinmetz, T. Walther, and J. D. Reiss, "High-fidelity noise reduction with differentiable signal processing," *arXiv preprint arXiv:2310.11364*, 2023.

[25] J. Engel, L. Hantrakul, C. Gu, and A. Roberts, "DDSP: Differentiable digital signal processing," in *ICLR*, 2020.

[26] N. Uzrad, O. Barkan, A. Elharar, S. Shvartzman, M. Laufer, L. Wolf, and N. Koenigstein, "DiffMoog: A differentiable modular synthesizer for sound matching," *arXiv preprint arXiv:2401.12570*, 2024.

[27] S. Lee, M. Martínez-Ramírez, W.-H. Liao, S. Uhlich, G. Fabbro, K. Lee, and Y. Mitsufuji, "GRAFX: An open-source library for audio processing graphs in PyTorch," *arXiv preprint arXiv:2408.03204*, 2024.

[28] Y.-T. Yeh, W.-Y. Hsiao, and Y.-H. Yang, "PyNeuralFx: A Python package for neural audio effect modeling," *arXiv preprint arXiv:2408.06053*, 2024.

[29] M. Comunità, C. J. Steinmetz, and J. D. Reiss, "NablAFx: A framework for differentiable black-box and gray-box modeling of audio effects," Feb. 2025, arXiv:2502.11668 [cs]. [Online]. Available: http://arxiv.org/abs/2502.11668

[30] D. Braun, "DawDreamer: Bridging the gap between digital audio workstations and Python interfaces," 2021. [Online]. Available: https://arxiv.org/abs/2111.09931

[31] E. Manilow, G. Wichern, P. Seetharaman, and J. L. Roux, "Cutting music source separation some Slakh: A dataset to study the impact of training data quality and quantity," 2019. [Online]. Available: https://arxiv.org/abs/1909.08494

[32] A. B. Kahn, "Topological sorting of large networks," *Commun. ACM*, vol. 5, no. 11, p. 558–562, Nov. 1962. [Online]. Available: https://doi.org/10.1145/368996.369025

[33] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," 2019. [Online]. Available: https://arxiv.org/abs/1711.05101