BbLearn: Sculpting Behavioral Clones in Modular Blackboard-Driven Music Systems

Fabian Ostermann Marco Pleines Günter Rudolph
Department of Computer Science
TU Dortmund University, Dortmund, Germany
{fabian.ostermann;marco.pleines;guenter.rudolph}@tu-dortmund.de

Abstract

Machine learning components in interactive music systems are often monolithic and opaque, offering little room for controlled modification or artist-guided steering. We present BbLearn, an extension to BbMuse, a blackboard-driven framework for real-time interactive music systems. BbLearn enables modular integration of learned behavior while preserving explainability at system level. Its workflow consists of three stages: (1) 'listen' records data exchanged through BbMuse's explicitly typed module interfaces. (2) 'clone' trains neural models to replicate hand-crafted modules via behavioral cloning; explicit typing means data collection and deployment require no additional configuration. (3) 'sculpt' then fine-tunes cloned models to evolve toward novel behavior using PPO against human-shaped rewards, with the clone serving simultaneously as warm start and as an anchor bounding reward-driven drift; the framework is designed to naturally extend toward reinforcement learning from human feedback. BbLearn is not a single new system but a platform: any module in any BbMuse project can be targeted for learned replacement. We present the design rationale, our open-source implementation, and a proof-of-concept on a Jazz walking bass generator that demonstrates the warm-start benefit and identifies multi-objective reward balancing as the central practical challenge.

1 INTRODUCTION

Recent advances in machine learning have produced impressive generative music systems, but the dominant designs are monolithic. They encapsulate compositional decisions, stylistic conditioning, and output generation inside a single large model whose internal behavior resists inspection, partial modification, or controlled steering. These properties sit poorly with artist-guided workflows that depend on transparent, targeted intervention, which is exactly the kind of intervention that traditional, hand-engineered interactive music systems have always supported. Briot et al. (2020) identified several challenges for modern AI-based music generation. Among them are explainability, interactivity, adaptability, and control, which are direct consequences of monolithic design: when the entire system is one black box, there is nothing smaller than the whole to explain.

Modular music systems, however, address these properties by construction. The blackboard-driven framework B♭Muse (BlackBoard MUSic Engine) (Ostermann, 2026) decomposes musical decision-making into independent modules that communicate through explicit, typed representations on a shared workspace (the so-called *blackboard*), so that each module's role is identifiable, its interface is inspectable, and any module can in principle be replaced without re-engineering the rest of the system. However, such frameworks traditionally rely on hand-crafted modules built with techniques like rule sets, small-corpus-based Markov models, or heuristic decision trees. Although nothing prevents an individual module from internally using a neural network, there is currently no principled, framework-supported way to introduce learned components that respects the typing discipline, exploits the data already flowing through the blackboard, and preserves system-level transparency. Where learning appears in such systems today, it is often designed ad hoc.

To close this gap, we present B♭Learn as a dedicated extension to B♭Muse. B♭Learn treats the blackboard as a seam: because every module's input and output are already explicit, typed, and observed, the framework can record any module's behavior, train a neural model to imitate it, and replace the original hand-crafted module without additional manual configuration. The proposed workflow proceeds in three stages: *listen*, *clone*, and *sculpt*. (1) *Listening* records the content of the typed representations that a hand-crafted module reads and writes on the blackboard. (2) *Cloning* fits a neural model to those recordings via behavioral cloning (BC), producing a drop-in replacement that already speaks the right interface and reproduces the original behavior in context. (3) Finally, we use the metaphor of *sculpting* for step-wise fine-tuning the cloned model away from imitation toward novel behavior. We apply *reinforcement learning* (RL) against human-shaped properties and musical constraints that can be formalized (e.g., staying in key, voice-leading smoothness, or range bounds) combined with human feedback that allows to incorporate subjective aesthetic intent that normally resists formalization. However, the latter is work in progress. This work will focus on the conceptualization of marrying RL and blackboard design. It further presents a proof-of-concept study that demonstrates its usefulness in creating highly original models.

It is important to note that B♭Learn is not a single system but a platform: existing B♭Muse projects become candidates for module-wise behavior replacement, while the system around it remains modular, typed, and inspectable. Explainability is preserved at system level through explicitly typed representations that are shared between opaque and transparent modules.

To make the workflow concrete, consider a B♭Muse project containing a walking-bass module driven by a small set of Jazz idiom rules: chord-tone targeting on strong beats,

predominantly stepwise motion in between, and a fixed register. B♭Learn can record this module's decisions during a live session, train a neural clone that reproduces them, and then sculpt the clone to discourage unwanted artifacts, say, excessive octave leaps, via RL, while a musician interactively steers its overall stylistic direction toward, for instance, denser chromaticism through human feedback. This exact example will be applied to the proof-of-concept study in Section 5.

The main contributions of this paper are:

- We identify a gap between monolithic learned systems and modular hand-crafted ones, and propose module-level learning as a bridge to **preserve explainability** at system level.
- We describe the *listen-clone-sculpt* pipeline as a **generic methodology** for introducing learned behavior into typed, blackboard-driven systems, including the use of the behavioral clone as both a warm start and an explicit anchor during sculpting.
- We release an **open-source implementation** of our RL extension to B♭Muse, that was designed with a focus on accessibility for the AI-music community.
- We **demonstrate feasibility** using a walking bass generator project as an example, showcasing the RL features that are currently implemented in B♭Learn.

B♭Learn is released[1] alongside this paper as a PyPI package (MIT license). It is bundled with `bbmuse` and depends on PyTorch only. The walking bass generator example is also publicly available (see Section 5) and offers an accessible entry point for experimenting with B♭Learn's features and pipeline stages.

2 BACKGROUND: B♭MUSE

B♭Muse (Ostermann, 2026) is a blackboard-driven Python framework for real-time interactive music systems. The blackboard metaphor, originally developed in classical AI for expert systems (Erman et al., 1980; Nii, 1986), imagines a group of specialists collaborating in a room whose only shared medium is a wall-mounted blackboard: each specialist watches the blackboard, contributes a partial result when its expertise applies, and the global solution emerges incrementally from these local contributions. The classical formulation has three components: the blackboard itself, a set of knowledge sources, and a controller that decides which knowledge source to activate. B♭Muse adopts a modern variant of this pattern, originally developed for real-time robotics in the context of soccer-playing robots (Röfer et al., 2007), in which the opportunistic, state-triggered activation of knowledge sources is replaced by an explicit flow of data with deterministic, automatically scheduled execution.

System state in B♭Muse is partitioned into named *representations*: typed, structured data objects that live on the global blackboard. *Modules* take the role of knowledge sources and declare, as part of their definition, which representations they `REQUIRE` (read) and which they `PROVIDE` (write). A third relation, `USES`, permits read access without imposing an update dependency and is the mechanism by which cycles in the module graph are avoided. The framework enforces that each representation has exactly one providing module. Together with `USES` for cycle-breaking reads, this allow the module graph to become a directed acyclic graph and admits a valid execution order via topological sorting (Kahn, 1962). This single design decision of declarative `REQUIRES`/`PROVIDES` over typed representations is what B♭Learn fundamentally relies on: every value flowing in and out of a module is explicit, typed, and observable, which means the data needed to learn a replacement for a module is, in principle, already available.

B♭Muse follows the inversion-of-control paradigm: modules and representations are plain Python files, loaded dynamically via `importlib`, and the framework drives them by invoking *hook functions*: `_init()`, `_update(bb)`, `_close()` on modules. Their leading underscore marks them as engine-called rather than user-called. B♭Learn adopts that pattern: rather than introducing a separate configuration layer, it adds new optional hooks to representations and modules that let users declare how representations are encoded as tensors and override default loss functions (details are given in Appendix A.3). As a result, B♭Learn extends B♭Muse without modifying its core: a B♭Muse project remains a valid B♭Muse project whether or not B♭Learn is installed, and the engine continues to function as a standalone framework when no learning is involved, even if B♭Learn-specific hook functions are in place (technical details in Appendix A.1).

3 RELATED WORK

Foundational work on blackboard architectures and on multi-agent music systems is reviewed in the original B♭Muse paper (Ostermann, 2026), thus not repeated here. This section focuses on prior work concerned with introducing learned components into music systems and controlling them.

The closest algorithmic precedent for B♭Learn's sculpt stage is the RL Tuner of Jaques et al. (2017), in which an LSTM model for monophonic melodies trained by maximum likelihood is fine-tuned via deep Q-learning under a reward that combines hand-written music-theory rules with a KL-style anchor toward the original model. RL-Duet (Jiang et al., 2020) applies variants of this pattern to monolithic sequence models with model-based rewards. MusicRL (Cideron et al., 2024) extends the thread to *reinforcement learning from human feedback* (RLHF), fine-tuning a large pre-trained text-to-music model using RL on rewards derived from large-scale human preference data, following the broader RLHF paradigm of Christiano et al. (2017) and InstructGPT (Ouyang et al., 2022). B♭Learn's sculpt stage shares the basic pattern of these works (supervised cloning, then fine-tune against rewards while anchoring toward the imitated behavior) but inverts the scope. Where these systems train one fixed-purpose model end-to-end, B♭Learn treats fine-tuning as a per-module operation inside a larger modular runtime, uses the host B♭Muse project itself as the RL environment, and combines formalizable constraint rewards with preference-based rewards. Our *cloning* stage that warm-starts fine-tuning builds on the standard supervised-imitation paradigm originating with ALVINN (Pomerleau, 1989) and formalized by Bain and Sammut (1995).

For real-time musical applications, the Wekinator (Fiebrink et al., 2009; Fiebrink and Cook, 2010) introduced interac-

[1] B♭Muse and B♭Learn are available at: https://github.com/fabianostermann/bbmuse. The exact version described in this paper is archived at: https://doi.org/10.5281/zenodo.21633022.

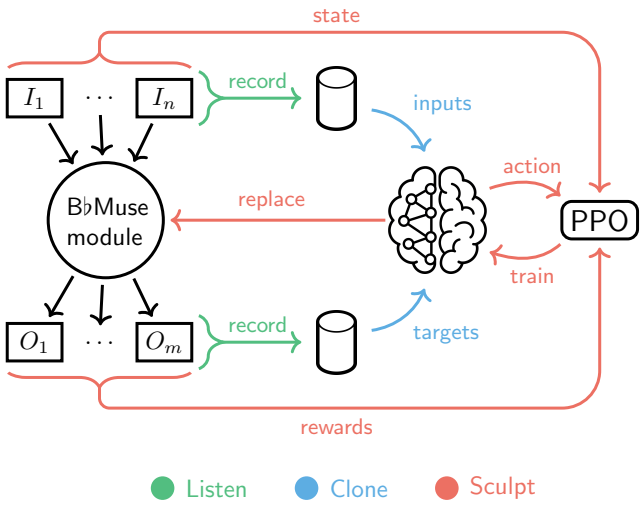


Figure 1: The three main stages of the B♭Learn pipeline

tive machine learning as a means for performers to train mappings from arbitrary input streams to control outputs by demonstration, without writing code. B♭Learn addresses a different layer of the stack: Wekinator targets end-user training of standalone input-to-control mappings, whereas B♭Learn targets developers extending the internal modules of a typed, blackboard-driven runtime with learned components that integrate seamlessly with existing code.

ComfyUI (Comfy Org, 2026) popularized node-based composition of pre-trained generative components for image, video, and audio workflows, but its modularity orchestrates fixed models and does not support training new ones. Recent LLM-based multi-agent systems for music composition (Deng et al., 2024; Xing et al., 2025) share the spirit of decomposing musical decision-making across specialized agents, but rely on natural-language messaging and frozen pre-trained LLMs rather than typed representations and trainable modules. To our knowledge, no existing framework provides a principled pathway from a hand-crafted module in a real-time interactive music system to a learned, sculpted replacement that preserves the original module's interface.

4 THE PIPELINE

B♭Learn organizes the path from a hand-crafted module to a sculpted neural replacement as three sequential stages, illustrated in Figure 1. *Listen* is passive: it records the typed information flowing in and out of one or more target modules, without altering system behavior. *Clone* trains a neural model on those recordings to reproduce the target module's input-output mapping via BC, producing a drop-in replacement that already speaks the correct interface and behaves indistinguishably from the original within the contexts the recordings covered. *Sculpt* then fine-tunes this clone away from imitation toward novel behavior using *proximal policy optimization* (PPO) (Schulman et al., 2017), with the cloned model serving as warm start and the original module serving as an explicit anchor that bounds how far sculpting may drift. In a practical application scenario, *apply & restore* complete the workflow as a fourth deployment-oriented stage that can be used to wrap a trained model as a native B♭Muse module (details in Appendix A.4). All stages share a single state machine with a hidden working directory at the original project's root, so that recordings, model checkpoints, and metadata persist across sessions and remain associated with the B♭Muse project they were derived from (technical details in Appendix A.2).

4.1 Listen

The *listen* stage produces the dataset on which subsequent stages depend. Given an existing B♭Muse project containing one or more hand-crafted modules whose behavior is to be learned, this stage records each target module's typed inputs I and outputs O as the project runs normally, yielding a temporally ordered episode of (input, output) pairs suitable for supervised BC. The stage is deliberately passive: B♭Learn does not modify the project, the blackboard, or the execution order, and the recorded session is functionally indistinguishable from an unrecorded one. This non-invasiveness is essential because the data we want to capture is precisely the data the project naturally produces. Recording must not perturb what is recorded.

When a listening session is started on a set of requested modules, B♭Learn attaches a *listener* to each, runs the underlying B♭Muse project, and on completion writes one episode per listener to disk. Subsequent listening sessions append further episodes to the same per-module storage, so that recordings from different musical contexts, different human collaborators, or different runs of an improvisational piece accumulate naturally.

Each listener observes one module by intercepting its update step. Immediately before the module executes, the listener encodes the current values of the module's required (and used) representations as tensors. Immediately after, it encodes the values of the module's provided representations. Encoding relies on the `_pack()` hook attached to every B♭Learn-compatible representation, which is the contract that makes typed blackboard values usable as model inputs and targets (implementation details in Appendix A.3). The triple of pre-update inputs, post-update outputs, and the implicit timestep index forms one frame of the episode. Concatenating frames over the duration of a session yields tensors of shape $(T, \cdot)$ per representation, where T is the number of times the module was scheduled. Listeners enforce shape consistency across frames so that episodes are immediately usable as training data without further preprocessing, and the resulting per-module episode files are the sole output of this stage and the sole input to the next.

4.2 Clone

The *clone* stage consumes one or more episodes recorded for a target module and produces a neural model that imitates the module's input-output behavior. The setup is classic BC (Pomerleau, 1989): the recorded module is treated as an expert policy, each frame as a state-action pair, and a parameterized model is fit by supervised regression onto the recorded outputs. Two properties of B♭Muse make this stage essentially configuration-free: first, every required, used, and provided representation already has a packed tensor form, so no additional preprocessing is required between recording and training. Second, the target module's `REQUIRES`, `USES`, and `PROVIDES` declarations specify the model's I/O structure explicitly, so the cloned model can be instantiated directly from the module's interface. The immediate result after cloning is a drop-in replacement that reproduces the original module's behavior in the contexts

the recording covered while exposing the exact same typed interface as the original module before.

Let $\mathcal{I} = \mathcal{R} \cup \mathcal{U}$ denote the set of input representations of the target module as the union of its required and used representations. Let $\mathcal{O}$ denote the set of provided (output) representations. For each $r \in \mathcal{I} \cup \mathcal{O}$, write d_r for the packed shape of r as fixed during the listen stage. The clone f_θ then decomposes into three parts that mirror this typed interface:

$$\mathbf{z}_r = E_r(\mathbf{x}_r) \in \mathbb{R}^{e_r} \qquad \text{for each } r \in \mathcal{I} \tag{1}$$

$$\mathbf{h} = B(\text{concat}\{\mathbf{z}_r : r \in \mathcal{I}\}) \in \mathbb{R}^{d_h} \tag{2}$$

$$\hat{\mathbf{y}}_p = H_p(\mathbf{h}) \in \mathbb{R}^{d_p} \qquad \text{for each } p \in \mathcal{O} \tag{3}$$

One *input encoder* E_r is instantiated per input representation r, one *output head* H_p per output representation p, and a single *backbone* B is shared across all heads so that cross-input correlations can be learned once. This per-representation factorization at the input and output is what makes the architecture configuration-free in the typical case: the module's interface declarations together with the packed shapes from the recorded episode fully determine the model's I/O structure, while the backbone alone carries free hyperparameters. The default backbone is a small MLP suitable for simple stateless behaviors. Users can integrate custom backbones using a specific backbone interface contract. An analogous extension point for custom encoders is planned (cf. Appendix A.3). Output heads produce raw logits without activation, deferring any nonlinear transformation to the per-representation loss and to the corresponding `_unpack()` hook used at deployment, so that classification, regression, and mixed targets can coexist within a single clone without architectural special-casing on the framework side.

Given a recorded episode of length T, with tensors $\mathbf{y}_p^{(t)}$ for each $p \in \mathcal{O}$ indexed by timestep t, the cloning objective is

$$\mathcal{L}^{\text{BC}}(\theta) \;=\; \frac{1}{T}\sum_{t=1}^{T} \frac{1}{|\mathcal{O}|} \sum_{p \in \mathcal{O}} \mathcal{L}_p\Big(\hat{\mathbf{y}}_p^{(t)}, \mathbf{y}_p^{(t)}\Big), \tag{4}$$

where each per-representation loss $\mathcal{L}_p$ is determined by the providing representation itself: if the representation defines a `_loss()` hook (formalized in Appendix A.3), that hook supplies $\mathcal{L}_p$ along with any transformation needed to align the head's logits with the recorded target. Otherwise, $\mathcal{L}_p$ falls back to MSE (design rationale in Appendix A.3). This factoring keeps the loss in step with the typed interface. Each provided representation knows how it should be compared, so heterogeneous outputs of a single module compose cleanly under one objective. Optimization itself is unremarkable: $\mathcal{L}^{\text{BC}}$ is minimized by gradient descent over θ using standard supervised learning. When T exceeds available memory, $\mathcal{L}^{\text{BC}}$ is approximated by its empirical estimate over mini-batches of frames sampled from the episode.

At the current state of development, we treat each frame independently rather than introducing sequence: this matches the per-update semantics of the underlying B♭Muse module and keeps the clone a stateless function from blackboard inputs to outputs. While RNNs are on the list of planned features, a memory mechanism is naturally available by requesting the last state of a self-provided representation with `USES`, which is a common pattern also in hand-crafted modules and can, if applied in the original module, even be automatically exploited.

4.3 Sculpt

The *sculpt* stage fine-tunes the model produced in Section 4.2 to move beyond imitation toward novel behavior shaped by musical objectives that the original hand-crafted module did not include. We frame this as an RL problem in which the cloned model is treated as a stochastic policy and trained against a reward signal expressing musical preference. The cloned weights serve a dual role: they are the policy's *warm start* (fine-tuning begins from a model that produces sensible output in context) and an *anchor* toward which the policy is explicitly regularized, bounding how far reward-driven optimization is allowed to drift from imitated behavior. Both roles matter because musical reward landscapes tend to be sparse and heavily under-specified: any tractable reward signal will leave most of action space unconstrained, and without an anchor the policy is free to exploit those gaps with degenerate but high-reward outputs (Agarwal et al., 2019).

A defining property of the B♭Learn sculpting process is that the B♭Muse project itself serves as the RL environment: the same execution loop that drives a live performance also drives policy training. This is realized by an *active listener*, an extension of the passive listener of Section 4.1. Similarly, it intercepts the target module's update step via monkey patching `_update(bb)` on the original module and encodes the inputs via `_pack()`. Then it samples an action from the current policy model and writes that action back to the blackboard via `_unpack()`, the inverse of `_pack()`. The original module is not removed during sculpting. Its outputs are still produced and recorded, but the policy's sampled action overwrites them on the blackboard, so the rest of the system observes the policy. After the action takes effect, a reward function `_reward(bb)` returns a scalar evaluating the just-produced output. This decentralized, dense, per-step reward design follows the standard reward-shaping logic that domain knowledge yields better gradient signal than sparse episodic feedback (Ng et al., 1999), and is well matched to formalizable musical constraints (e.g., key adherence, voice leading, range, rhythmic regularity), that can be encoded directly on the representations they constrain.

The deterministic clone f_θ is lifted to a stochastic policy by treating each output head's prediction as the mean of a Gaussian with a learnable, state-independent log standard deviation:

$$\pi_\theta(a^p \mid s) = \mathcal{N}\big(a^p;\ \mu_\theta^p(s),\ \exp(\log \sigma^p)\big),\ p \in \mathcal{O}, \tag{5}$$

where the state s is the module's required and used representations ($\mathcal{I}$ from Section 4.2) at the current timestep, $\mu_\theta^p(s)$ is the mean produced by the architecture of Section 4.2 under current parameters, and one $\log \sigma^p$ parameter is added per provided representation. The per-head distributions allow for closed-form sampling, log-probabilities, and entropies. We treat each head's action as conditionally independent given s, so log-probabilities decompose over p. For discrete output heads, we treat the logits as a continuous space as a first approximation, deferring proper categorical treatment to future work.

Optimization uses PPO (Schulman et al., 2017) with the standard clipped surrogate, decomposed across heads. Letting $\rho_t^p(\theta) = \pi_\theta(a_t^p \mid s_t)/\pi_{\theta_{\text{old}}}(a_t^p \mid s_t)$ denote the per-head importance ratio between current and rollout-time parameters, the PPO loss is

$$\mathcal{L}^{\mathrm{PPO}}(\theta) = -\,\mathbb{E}_t\Bigg[\sum_{p\in\mathcal{O}} \min\Big(\rho_t^p(\theta)\,A_t,\; \mathrm{clip}\big(\rho_t^p(\theta),\,1-\epsilon,\,1+\epsilon\big)\,A_t\Big)\Bigg], \tag{6}$$

with the global advantage A_t. Our current implementation provides A_t^u as the discounted return $\sum_{k\geq 0}\gamma^k\, r_{t+k}^u$, with r_t^u the u-th user-defined reward emitted at timestep t. Standard PPO integrates a learned value baseline (critic) that uses generalized advantage estimation (GAE) (Schulman et al., 2016), which we decided not to include in the list of B♭Learn features yet, as it was recently shown to be less important in RLHF contexts (Ahmadian et al., 2024). However, we applied z-score normalization to all the returns r^u across the episode timesteps (zero mean, unit variance) before averaging to the global advantage A_t.

Further, we make use of the average policy entropy

$$\mathcal{H}(\theta) \;=\; \mathbb{E}_t\left[\sum_{p\in\mathcal{O}} H\big[\pi_\theta(\cdot \mid s_t)^p\big]\right], \tag{7}$$

which we maximize to discourage premature collapse of the policy onto a single deterministic mode.

Finally, we add a BC anchor that regularizes the policy mean toward the frozen clone $\mu_{\theta_0}^p$ (θ_0 are the network parameters at the start of sculpt) using the same per-representation losses that drove cloning,

$$\mathcal{L}^{\mathrm{BC}}(\theta) \;=\; \mathbb{E}_t\left[\sum_{p\in\mathcal{O}} \mathcal{L}_p\big(\mu_\theta^p(s_t),\; \mu_{\theta_0}^p(s_t)\big)\right]. \tag{8}$$

RLHF practice often uses a KL penalty against a frozen reference policy (Christiano et al., 2017; Ouyang et al., 2022) to the same end. We anchor on the per-representation loss instead because it reuses the same loss defined for cloning. With KL, we would need to commit to a distributional form per head, which conflicts with the framework's typed per-representation contract, because the cloned content lives in the means rather than in the variances of the constructed Gaussian policy. However, a special procedure specific to B♭Learn is possible for the anchor loss: we can directly anchor to the original module behavior, which is available because we can observe its ground truth output during PPO's exploration phase with no additional effort. When deciding to do so, the predicted output $\mu_{\theta_0}^p(s_t)$ becomes the oracle output $\mathbf{y}_p^{(t)}$ from Equation (4), that may even surface further unseen but wanted behavior not learned by π_{θ_0}, and, on top of that, also saves memory as well as computational costs for sampling the old model.

Finally, the complete sculpt objective is

$$\mathcal{L}^{\mathrm{sculpt}}(\theta) \;=\; \mathcal{L}^{\mathrm{PPO}}(\theta) \;-\; a\,\mathcal{H}(\theta) \;+\; b\,\mathcal{L}^{\mathrm{BC}}(\theta), \tag{9}$$

in which $a, b \geq 0$ trade exploration and imitation fidelity against reward-driven adaptation. A study of the empirical effects of the individual loss components is presented in Section 5.

B♭Learn currently sculpts one module per session. Multi-agent RL across several simultaneously sculpted modules is a natural extension and a major item on the project's roadmap. A second practical consideration concerns the cost of trajectory collection. Because the sculpting environment is the B♭Muse project itself, rollout speed is bounded by the project's execution speed, which varies considerably across use cases. Purely symbolic projects can often be run faster than real-time and parallelized across processes, making data collection cheap. Projects driving external MIDI devices suffer a bottleneck from needing one device per process. And in B♭Learn's most interesting use case, which clearly is interactive collaborative performance with a human in the loop, the environment is real-time by definition, which puts a hard floor under sample efficiency and sharpens the appeal of starting from an already-competent clone.

5 PROOF OF CONCEPT

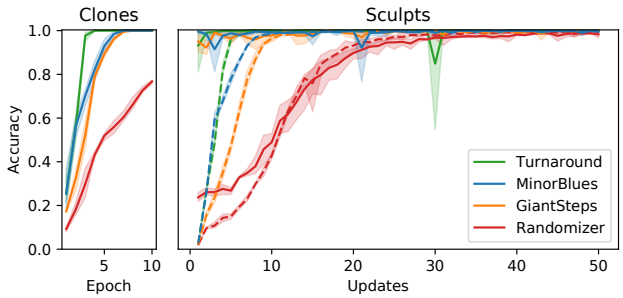
We evaluate B♭Learn on a small B♭Muse project that generates a Jazz walking bass.[2] A walking bass is one of the foundational textures of mainstream Jazz: a monophonic accompaniment in which the bassist plays one quarter note per beat while moving through the harmony of a repeating chord progression (Levine, 1995). The fixed rhythmic surface removes timing from the modeling problem entirely. What remains is rule-based pitch selection: outline each chord, walk smoothly rather than leap, target chord roots on downbeats, stay in range.

We study three hand-crafted bassist modules sharing the same typed interface. They require `ChordContext` (current and following chord), `CurrentBeat` (in 4/4 time signature), and `NotesMemory` (last played notes), and provide the `NextNote`. First, a simple *baseline* (a) is a deterministic one-liner that walks the chord shape directly. Second, a sophisticated version (c) based on *stochastic* decision-making with role-based pitch selection, stochastic chord-tone choices, directional approach toward the next chord, stochastic range selection, and repetition constraints. The third version is fully *deterministic* (b) and is derived from the stochastic one by pinning all stochastic decisions to fixed outcomes, exercising the same rule structure without the noise, and therefore sitting between the other two in perceived quality and variety.

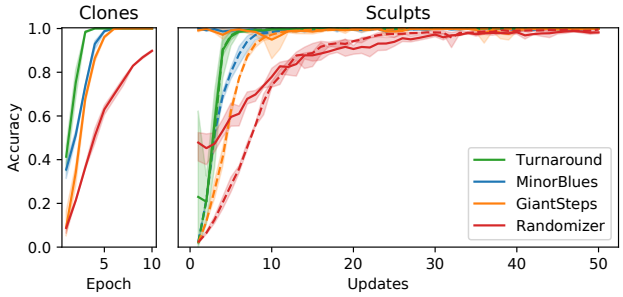
In a first experiment, we test whether the clone stage produces faithful imitations of the three target modules and how cloning fidelity depends on (1) the behavioral complexity of the target, (2) the difficulty of the chord progression presented to it, and (3) the presence of stochastic decision-making. Figure 2 reports behavior accuracy (measured as fraction of timesteps where the cloned model's output matches the target module's output) on four progressions of contrasting harmonic character: a classic four-bar *Turnaround*, a *Minor Blues*, the harmonically dense *Giant Steps*, and a *Randomizer* that emits a uniformly random chord at every new bar. We applied a basic set of manually determined hyperparameters with no automated tuning: 10 epochs, learning rate $\eta = 0.001$, batches of size 256, discount factor $\gamma = 0.99$, and 5 runs for each configuration.

The left panels of Figure 2 confirm the hypotheses (1) and (2) on the deterministic targets (a) and (b): cloning is fast and reaches near-perfect accuracy within ten epochs on the three musically meaningful progressions, with the

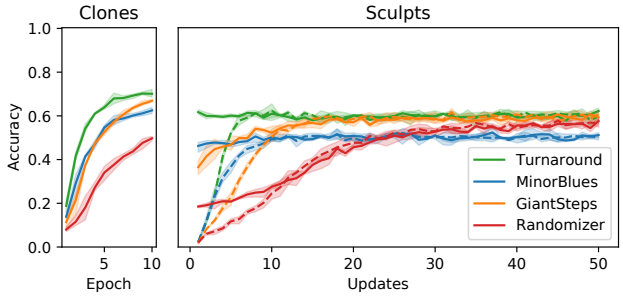
[2]The source code of the Walking Bass Generator is provided here: https://doi.org/10.5281/zenodo.21959400. It makes a good starting point and test-bed for first steps with B♭Learn. An overview of the system's construction can be found in Appendix B, Figure 4.



(a) Simple baseline



(b) Deterministic Bassist



(c) Stochastic Bassist

Figure 2: Accuracy during cloning (left) and sculpting (right) against the target module on four chord progressions increasing in difficulty. The dashed curves show runs without pre-training. Curves show the mean with shading indicating the min/max range.

simpler turnaround converging fastest, and *Randomizer* learning more slowly because the uniformly random chord stream provides no exploitable structure beyond the immediate `ChordContext`. The stochastic target (c) tells a different story: accuracy plateaus at 0.5–0.7 across all progressions and never approaches 1.0. This is expected and not a failure of cloning: when the target module makes irreducibly stochastic decisions at certain beat positions, no point predictor can match it on those decisions, and the plateau height reflects the proportion of decisions that are deterministic given the inputs (e.g., approach note from below with probability 0.6, above otherwise). Modeling distributions rather than points (e.g. via mixture density heads (Bishop, 1994)) would lift this ceiling. As stochastic decision-making is a very common approach to automatic composition (Xenakis, 1992), this highlights a key opportunity to extend B♭Learn's capabilities.

The right panels of Figure 2 address the warm-start question directly: the dashed curves, which begin sculpting from a freshly initialized network with no prior cloning, eventually reach the same accuracy as the solid curves starting from a cloned model, confirming that PPO-driven sculpting works on its own, but the warm-started runs converge to that accuracy substantially earlier on every progression, demonstrating the operational value of the cloned warm start.

A note on units: each *update* step in the sculpt panels comprises two seconds of B♭Muse environment runtime, during which trajectories are collected, followed by ten epochs of cheap policy optimization on the collected samples. We measure environment time in seconds rather than scheduler steps, because when B♭Muse's group-based parallel execution feature is used, there is no global step counter (Ostermann, 2026), and because human-in-the-loop deployment, which is B♭Learn's primary target, makes wall-clock the operationally meaningful budget. Cloning, by contrast, is lightning fast: a full ten-epoch run in the left panels of Figure 2 completes in under a second, whereas the 50 sculpt updates in the right panels take 169 s. The asymmetry is structural: cloning is supervised over a fixed buffer while sculpting must collect rollouts in the live environment (we ran at 200 000 bpm, which obviously is also not possible for every scenario). That is the biggest argument for the warm start: the clone delivers most of the basic imitation competence at negligible cost, freeing the more expensive sculpt stage for the constraint and preference objectives that imitation alone cannot reach.

In a second experiment, we examine how human-shaped rewards can influence a BC policy. We started using the simple baseline version of the bassist and defined three musical rules that are not fully aligned with its original behavior: Rule (1) rewards when successive note pitches are close using the variance of differences between successive notes; Rule (2) penalizes tone repetitions by counting occurrence, and Rule (3) rewards notes with higher pitches. (1) and (2) are intentionally constructed as slightly contradictory, while (3) is not musically meaningful to the original behavior (bassists play low), presenting an intentionally adversarial test case. Figure 3 shows three distinct optimization configurations: one with rewards only, a second with BC anchor at $b = 0.01$, and a third with the same BC anchor but also 10 epochs of pre-training using the clone stage. We introduced an entropy bonus of $a = 0.001$ and performed 5 runs.

Rule (1) is fulfilled by all configurations. With pre-training, the warm-start benefit can clearly be identified (green curve). Rule (2) keeps the tone repetitions at a minimum. Pre-training, however, breaks down at the start but recovers over the course of the 200 updates. Rule (3) obviously gets sacrificed when the BC anchor is enabled. While the reward-only run easily exploits the reward, the BC anchor to the original behavior, which always plays low, hinders the fulfillment of this rule. However, a key point of motivation is that this situation is fully observable and therefore explainable, because we explicitly know the current reward. To emphasize the rule's target behavior we can easily decide to either increase its reward weight or lower the BC anchor coefficient b. Development plans include an interactive feature for B♭Learn, where the user can react by inspecting and dynamically weighting coefficients and reward functions via a specifically designed GUI.

This proof-of-concept evaluation shows that the major challenge of sculpting is to balance between the BC anchor and the reward landscape. Each reward is valid on its own, but combining them is not a trivial procedure. Instead, we face all the challenges that multi-objective optimization (Roijers et al., 2013) brings. Special attention on that topic will be given in future work that targets thorough evalu-

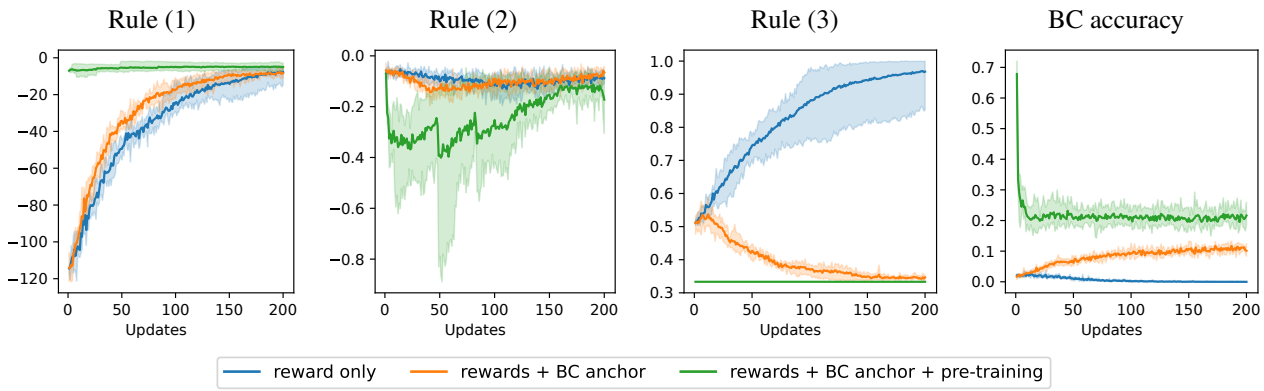


Figure 3: Rule-based returns and BC accuracy over PPO updates. Curves show the mean with shading indicating the min/max range.

ation of applying different sculpting strategies to diverse scenarios. To obtain our results, we had to stabilize the gradient descent by applying the same z-score normalization as for the discounted returns (cf. Section 4.3) to each reward signal independently, before averaging them into a combined reward.

The final open experiment is to integrate human feedback. This would be straightforward on the one hand, as it is just another reward function to add to the advantage calculation, but on the other hand, it is much sparser than human-shaped rules and will therefore surely benefit from a dedicated reward model. Further, collecting human feedback will need some (graphical) UI. Therefore, both are planned features of B♭Learn.

6 DISCUSSION

B♭Learn's contribution to explainability is best understood as preserving it where prior modular systems already had it, rather than adding new explainability machinery to the learned components themselves. A learned module produced by clone or sculpt is internally as opaque as any neural network. What B♭Learn preserves is the surrounding scaffolding: the module still declares the same `REQUIRES`, `USES`, and `PROVIDES` relations as the hand-crafted version it replaced. Every value flowing into and out of it remains a typed, named representation on the blackboard. The role the module plays in the larger compositional flow is unchanged. A user who could trace a faulty harmonic decision back to a specific module before learning was applied can do exactly the same trace afterward. This is a meaningful but bounded claim: system-level explainability is preserved even where component-level explainability is lost. We believe it is the right claim for the kind of interactive music systems B♭Muse targets, where understanding emerges from how decisions interact across modules at least as much as from any single decision.

Even though introduced in the context of modular systems, B♭Learn and its sculpt mechanism need not be used only inside complex multi-module systems. A small B♭Muse project consisting of a single trainable module bracketed by simple input- and output-shaping modules (e.g., one that converts the project's intended inputs into a typed representation, and one that consumes the module's typed output and emits whatever external signal is desired) also gives a useful warm start for highly original monolithic models. A rule-based first version of the intended behavior can be coded as the target module, the clone stage transfers it into a neural model with the right interface, and the sculpt stage then evolves it under constraint and preference rewards. We did not foreground this use case in the rest of the paper because we see its main contribution in larger systems, but it is a natural side-door for projects whose primary goal is shaping a single model rather than coordinating many.

Despite the fact that we have developed and validated our *listen-clone-sculpt* approach in the domain of real-time interactive music systems, neither the methodology nor the implementation is restricted to music. B♭Muse is rooted in properties (typed representations on a shared workspace, explicit `REQUIRES`/`PROVIDES` interfaces, automatic dependency-driven scheduling) that it inherits from real-time robotics and that apply equally to any domain that benefits from modular real-time control. Visual generative art systems, game AI, procedural content generation, or multimodal interactive installations are natural targets, and a learned module replacing a hand-crafted one would, in any of these settings, inherit the same system-level explainability that B♭Learn preserves.

Several limitations bound the scope of this paper. The evaluation is a feasibility study: the walking bass generator example shows that the pipeline runs end-to-end and that the principal design decisions behave as intended, but they do not establish quantitative claims about musical quality, generalization across project types, or comparative advantage at scale. We have not run a user study. This is an important item of remaining work. Finally, sculpting currently targets one module per session. Exploring multi-agent learning methods for bringing multi-module BC and RLHF to B♭Muse projects also is exciting future work.

7 CONCLUSIONS

We have presented B♭Learn, an extension to the blackboard-driven framework B♭Muse that turns any hand-crafted module in any B♭Muse project into a candidate for learned, artist-steerable replacement. The *listen-clone-sculpt* pipeline exploits the typed interface that B♭Muse already enforces: *listening* records a module's typed inputs and outputs without perturbing the system, *cloning* fits a neural drop-in replacement via behavioral cloning, and *sculpting* fine-tunes that clone away from imitation through PPO with human-shaped rewards and a behavioral-cloning anchor that bounds drift. A proof-of-concept on a walking bass generator demonstrates that all three stages work end-to-end, that cloned warm starts substantially accelerate sculpt convergence, and that the multi-objective balance

between BC anchor and reward landscape poses the central challenge.

One major goal of the B♭Learn project clearly is to make it easier for artists to apply multi-agent RLHF to generative music. However, our framework at this point has more advantages for experts, who can now more easily develop complex neural-based music systems from scratch. For intermediate programmers, however, surely a lot of user-definition and manual code writing must be streamlined with predefined configurations to choose from or even automatic action space and loss instantiation, which is a reasonable goal for future improvement. Therefore, we are now working on a platform-independent GUI editor for B♭Muse based on NiceGUI, which will also have features for GUI-started training runs, where the user is guided through the B♭Learn stages and loss components, and scalarization weights in training sessions can be visualized and interactively controlled.

B♭Learn can also be used as a platform for exploring and comparing RLHF methods and variants (similar to the study in Section 5). We placed great emphasis on a clean code base that allows us and others to extend B♭Learn by several interchangeable RL components in the future.

REFERENCES

Agarwal, R., Liang, C., Schuurmans, D., and Norouzi, M. (2019). Learning to generalize from sparse and underspecified rewards. In *Proceedings of the 36th International Conference on Machine Learning (PMLR)*, pages 130–140. PMLR.

Ahmadian, A., Cremer, C., Gallé, M., Fadaee, M., Kreutzer, J., Pietquin, O., Üstün, A., and Hooker, S. (2024). Back to basics: Revisiting REINFORCE-style optimization for learning from human feedback in LLMs. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12248–12267. Association for Computational Linguistics.

Bain, M. and Sammut, C. (1995). A framework for behavioural cloning. In Furukawa, K., Michie, D., and Muggleton, S., editors, *Machine Intelligence 15: Intelligent Agents*, pages 103–129. Oxford University Press, Oxford. Workshop held at St. Catherine's College, Oxford, July 1995; volume published 1999.

Bishop, C. M. (1994). Mixture density networks. Technical Report NCRG/94/004, Aston University.

Briot, J.-P., Hadjeres, G., and Pachet, F.-D. (2020). *Deep Learning Techniques for Music Generation*. Computational Synthesis and Creative Systems. Springer, Cham.

Christiano, P. F., Leike, J., Brown, T. B., Martic, M., Legg, S., and Amodei, D. (2017). Deep reinforcement learning from human preferences. In *Advances in Neural Information Processing Systems 30 (NeurIPS)*, pages 4299–4307.

Cideron, G., Girgin, S., Verzetti, M., Vincent, D., Kastelic, M., Borsos, Z., McWilliams, B., Ungureanu, V., Bachem, O., Pietquin, O., Geist, M., Hussenot, L., Zeghidour, N., and Agostinelli, A. (2024). MusicRL: Aligning music generation to human preferences. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, volume 235 of *Proceedings of Machine Learning Research*, pages 8968–8984. PMLR.

Comfy Org (2023–2026). ComfyUI: A modular diffusion model GUI, API and backend with a graph/nodes interface. https://github.com/Comfy-Org/ComfyUI. Open-source software, GPL-3.0 license. Initial public release January 2023.

Deng, Q., Yang, Q., Yuan, R., Huang, Y., Wang, Y., Liu, X., Tian, Z., Pan, J., Zhang, G., Lin, H., Li, Y., Ma, Y., Fu, J., Lin, C., Benetos, E., Wang, W., Xia, G., Xue, W., and Guo, Y. (2024). ComposerX: Multi-agent symbolic music composition with LLMs. In *Proceedings of the 25th International Society for Music Information Retrieval Conference (ISMIR)*, pages 669–679.

Erman, L. D., Hayes-Roth, F., Lesser, V. R., and Reddy, D. R. (1980). The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty. *ACM Computing Surveys*, 12(2):213–253.

Fiebrink, R. and Cook, P. R. (2010). The Wekinator: A system for real-time, interactive machine learning in music. In *Proceedings of the 11th International Society for Music Information Retrieval Conference (ISMIR), Late-Breaking Demo*, Utrecht, Netherlands.

Fiebrink, R., Trueman, D., and Cook, P. R. (2009). A meta-instrument for interactive, on-the-fly machine learning. In *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, pages 280–285, Pittsburgh, PA, USA.

Godot Engine Contributors (2014–2026). Godot engine. https://godotengine.org/. Open-source 2D and 3D game engine, MIT license. Initial public release 2014.

Jaques, N., Gu, S., Turner, R. E., and Eck, D. (2017). Tuning recurrent neural networks with reinforcement learning. In *5th International Conference on Learning Representations (ICLR), Workshop Track*.

Jiang, N., Jin, S., Duan, Z., and Zhang, C. (2020). RL-Duet: Online music accompaniment generation using deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 710–718.

Kahn, A. B. (1962). Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562.

Levine, M. (1995). *The Jazz Theory Book*. Sher Music, Petaluma, CA.

Ng, A. Y., Harada, D., and Russell, S. J. (1999). Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the 16th International Conference on Machine Learning (ICML)*, pages 278–287. Morgan Kaufmann.

Nii, H. P. (1986). The blackboard model of problem solving and the evolution of blackboard architectures. *AI Magazine*, 7(2):38–53.

Ostermann, F. (2026). BbMuse: A blackboard-driven framework for real-time interactive music. In *Proceedings of the International Computer Music Conference (ICMC)*.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P. F., Leike, J., and Lowe, R. (2022). Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems 35 (NeurIPS)*.

Pomerleau, D. A. (1989). ALVINN: An autonomous land vehicle in a neural network. In Touretzky, D. S., editor, *Advances in Neural Information Processing Systems 1 (NIPS 1988)*, pages 305–313. Morgan Kaufmann.

Röfer, T., Brose, J., Göhring, D., Jüngel, M., Laue, T., and Risler, M. (2007). GermanTeam 2007. In *RoboCup 2007: Robot Soccer World Cup XI Preproceedings*.

Roijers, D. M., Vamplew, P., Whiteson, S., and Dazeley, R. (2013). A survey of multi-objective sequential decision-making. *Journal of Artificial Intelligence Research*, 48:67–113.

Schulman, J., Moritz, P., Levine, S., Jordan, M. I., and Abbeel, P. (2016). High-dimensional continuous control using generalized advantage estimation. In *Proceedings of the 4th International Conference on Learning Representations (ICLR)*.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint 1707.06347*.

Wang, Z. and Wu, L. (2023). Theoretical analysis of inductive biases in deep convolutional networks. In *Proceedings of the 37th International Conference on Neural Information Processing Systems (NeurIPS 2023)*, pages 74289–74338.

Xenakis, I. (1992). *Formalized Music: Thought and Mathematics in Composition*. Number 6 in Harmonologia Series. Pendragon Press, Stuyvesant, NY, 2nd edition.

Xing, P., Plaat, A., and van Stein, N. (2025). CoComposer: LLM multi-agent collaborative music composition. *arXiv preprint arXiv:2509.00132*.

A SYSTEM DESIGN AND IMPLEMENTATION

A.1 Architecture and project layout

B♭Learn is implemented as an extension to the B♭Muse engine without modifying it. The implementation follows a strict dependency rule: code from B♭Learn imports from the B♭Muse core, but the reverse is fully avoided. As a consequence, the B♭Muse engine remains a standalone framework with no new dependencies introduced by B♭Learn. Therefore, a project developed without learning features has no dependencies on heavy deep learning packages such as PyTorch, and a project deployed after the apply step of Section A.4 runs without B♭Learn on the target machine, since the rewritten module loads its model individually through PyTorch using B♭Muse's native blackboard interface.

To keep the user-facing extension uniform with B♭Muse's own conventions, B♭Learn relies on the same concept of inverse control via *hook functions* (framework-called functions whose names begin with a leading underscore) that B♭Muse adopted from the Godot game engine (Godot Engine Contributors, 2026). The hooks introduced by B♭Learn (formalized in Section A.3) attach to representations rather than modules.

A.2 State machine and the hidden working directory

B♭Learn's pipeline is operationalized as a state machine, with state persisted across sessions in a hidden working directory `.bblearn/` located at the root of a B♭Muse project. The choice of a hidden directory under the project root is inspired by `.git/`: the state is local to the project, easy to ignore in version control, and never intermixed with the project's own source. Within `.bblearn/`, artifacts are organized per target module and per session. Each module gets its own subtree containing numbered episode files written by listening sessions, numbered clone runs (each with a checkpoint history and a final model), and numbered sculpt runs of the same form. Subsequent invocations of any pipeline stage append rather than overwrite, so multiple recordings, clones and policies can coexist for the same module and be selected for downstream stages by an integer identifier. For user-friendliness, B♭Learn also has a git-like status feature that summarizes information.

Once recordings or trained models exist for a module, the module's `REQUIRES`, `USES`, and `PROVIDES` declarations and the packed shapes of its representations should be treated as frozen for as long as those artifacts are intended to remain valid. Cloned and sculpted models embed the I/O dimensionality of their target module structurally, so changing the signature invalidates downstream stages. B♭Learn detects shape mismatches when loading episodes or models and aborts. Users who deliberately want to re-record under a changed signature can simply do so and begin a fresh numbered recording series alongside the old, or can manually delete the `.bblearn/` directory to reset the state machine.

A.3 The hook contract

User-facing extension of B♭Muse for learning is concentrated in five hooks that attach to representation files: `_pack()`, `_unpack()`, `_loss()`, `_reward()`, and `_encoder()`. Together they constitute the contract that bridges B♭Muse's typed blackboard values with the machine learning world of tensors.

`_pack()` encodes a representation's current state into a tensor of fixed shape, which is the form in which the representation enters the recording buffers, the input encoders of a clone, and the rollout buffers of sculpting. The shape must be consistent across all calls within a session. B♭Learn captures it on the first call and asserts on subsequent calls. `_pack()` is required on every representation that participates in a learned module, whether as input or as output.

`_unpack(p)` is the inverse: it accepts a tensor `p` of the same shape that `_pack()` produces and writes the corresponding state back into the representation. It is required on every representation that is an output of a learned module, so that the model's predictions can be applied to the blackboard during sculpt rollouts (cf. Section 4.3) and during native execution after the apply step (cf. Section A.4).

`_loss(p, t)` specifies the per-representation supervised loss used by cloning (Eq. (4)) and the BC anchor in sculpting (Eq. (8)) for that representation. Output heads (Section 4.2) always emit raw logits with no nonlinearity applied. In `_loss()`, the user may apply a transformation T (e.g., softmax, sigmoid, tanh, or some custom mapping) that is appropriate for the representation, computing $\mathcal{L}(T(p),\, x)$ internally. Usually, the exact same T should be applied inside `_unpack()`, so that the representation produced at inference time matches the space in which the loss was computed. Currently, this pairing is the user's responsibility, and it is also where they retain full control over the output semantics of any learned module, without the need of to tie the framework to assumptions about activation functions or output distributions.

The loss hook is optional. When not provided, B♭Learn falls back to mean square error (MSE). For regression targets, it works reasonably fine. For one-hot categorical targets, cross-entropy would of course be preferable, but it is unsafe to apply by default since B♭Learn cannot know whether a head's output is intended to live in logit, probability, or some other space. Streamlining loss selection for common representation types is on the roadmap, and may follow the example of Stable-Baselines3 with its explicit `Box` and `Discrete` constructors. For now, the method of choice is a custom `_loss()`. However, this design decision is likely to change with the next version of B♭Learn.

`_reward(bb)` is used for the heuristic rule-based musical constraints that the user can design. Those rewards have the benefit of being naturally denser than human feedback. Reward functions can be created in the default directory `rewards/`. Any file ending on `*.py` will be dynamically loaded and the function `_reward(bb)` is run in the PPO data collection process. To allow for comparisons of values on different representations, every reward function is provided a copy of the blackboard `bb`. If multiple reward hooks are used, the framework will track them by name (filepath stem), so individual inspection is possible.

`_encoder()` is a feature for the near future worth mentioning to discuss the limitation of the default input encoder. Currently, multi-dimensional inputs are flattened before feeding them into the shared backbone, which is adequate when a B♭Muse project only uses low-dimensional symbolic representations. However, structure is lost for representations such as audio spectrograms or piano rolls where convolutional inductive bias (Wang and Wu, 2023) would help. The `_encoder()` hook will let users override the default flattening with a custom encoder module, mirroring at the per-input level what the optional backbone path of Section 4.2 already provides for the shared trunk.

A.4 Apply & Restore

The *apply* feature deploys a sculpted model back into the B♭Muse project as a native module. B♭Learn edits the target module's source file in place: the original hook bodies (`_init`, `_update`, and where present `_close`) are commented out rather than deleted, and replacement hooks are inserted that import PyTorch, load the sculpted checkpoint into memory in `_init()`, and perform a forward pass in `_update()`. Inputs to the model are obtained by calling `_pack()` on the module's required and used representations. Outputs are written back to the blackboard by calling `_unpack()` on each provided representation. The result is a self-contained B♭Muse module that runs the learned behavior natively, with no runtime dependency on B♭Learn. Therefore, the project remains executable even if B♭Learn is not installed. The complementary *restore* feature reverses this rewrite, uncommenting the original hooks and removing the inserted ones, so that the original hand-crafted module can be reinstated at any time. Because both edits leave the file syntactically valid and target only the hook region, apply and restore are also fast enough to be invoked at runtime, even allowing for hot-swapping between original and learned versions during live sessions.

B WALKING BASS GENERATOR

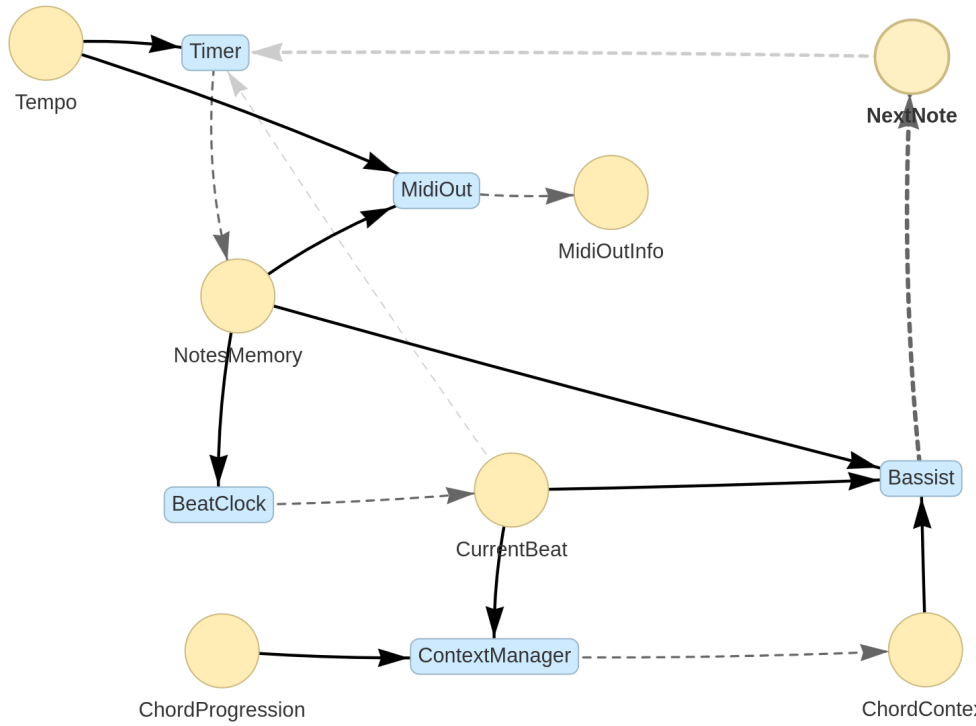


Figure 4: The system architecture and dependency graph of the *Walking bass generator* B♭Muse project.

The source code of the *Walking bass generator* B♭Muse project is available here: https://doi.org/10.5281/zenodo.21959400