Proceedings of the 7th International Conference on Live Coding (ICLC 2023)
April 19th - 23rd – Utrecht, Netherlands

LambDAW: Towards a Generative Audio Workstation

Ian Clester
Georgia Institute of Technology
ijc@gatech.edu

Jason Freeman
Georgia Institute of Technology
jason.freeman@gatech.edu

ABSTRACT

We present LambDAW, a novel system for integrating computation and composition that brings code directly into the digital audio workstation (DAW). It allows the composer to freely mix static and dynamic materials by embedding short expressions of code in the DAW timeline that generate audio and MIDI on demand. LambDAW moves code out of the text editor and computation out of the effects chain, bringing both into the timeline where they can refer to and transform other items. We propose that this move makes code more tangible and enables the composer to easily bring generativity into their existing practices. Additionally, we discuss LambDAW's affordances and implications for live coding. LambDAW takes the form of an open-source REAPER extension that executes Python code embedded in projects, enabling the user to benefit from both the existing REAPER and Python ecosystems.

1 Introduction

Digital Audio Workstations (DAWs) allow users to compose by arranging and manipulating musical materials on a timeline. Meanwhile, computer music environments such as Max/MSP, Pure Data, and SuperCollider allow users to compose by patching together virtual objects or writing code, and live coding languages further encourage experimentation and improvisation via compositional abstractions and concise notation. We aim to create a hybrid environment, a kind of Generative Audio Workstation that marries the generative and expressive possibilities of code with the direct manipulation, familiar interface, and rich features of the DAW.

To that end, we present LambDAW,[1] an extension for REAPER that enables the user to embed Python snippets directly into the DAW timeline. In effect, LambDAW introduces a new set of items into the DAW environment: in addition to the static audio and MIDI items that REAPER already supports, the user now has access to *dynamic*[2] audio and MIDI items generated on-demand from code expressions. What's more, these items can refer to *other* items in the timeline, enabling them to be defined as transformations of static items (or other dynamic items) and encoding these relationships directly in the DAW project.

DAWs have a long history of enabling custom computation, which we detail further in the next section. From the DAW perspective, we dislodge computation from its established place in the DAW (plugins in the effects chain) and drop it into the timeline, where it is exposed to the user alongside all the other musical materials. From the live coding perspective, we lure code out of the text editor or IDE and into the DAW, where it is directly manipulable like any other form of media, its output is visible, and it can refer to other items (or even itself).

We are inspired in our efforts by the humble spreadsheet, whose virtues are expounded eloquently by Lindenbaum, Kaliski, and Horowitz (2022) (emphasis in original):

> When software allows adding dynamic behavior as part of *gradual enrichment*, it supports *programming in the moment*. Spreadsheets support dynamic behavior with live, reactive, declarative formulas you can place anywhere you would otherwise have data. They treat data and formulas equally: where they appear in the interface, how they are edited, and how they reference each other. This interchangeability supports

[1] LambDAW is free software, and it can be found at https://github.com/ijc8/lambdaw. A demo video is available at https://youtu.be/5VU-ora7Wx0.

[2] We describe expression items as dynamic because their contents are generated by code at runtime; expressions can depend on random values, external state, or other items in the DAW, all of which may result in different output upon reevaluation.

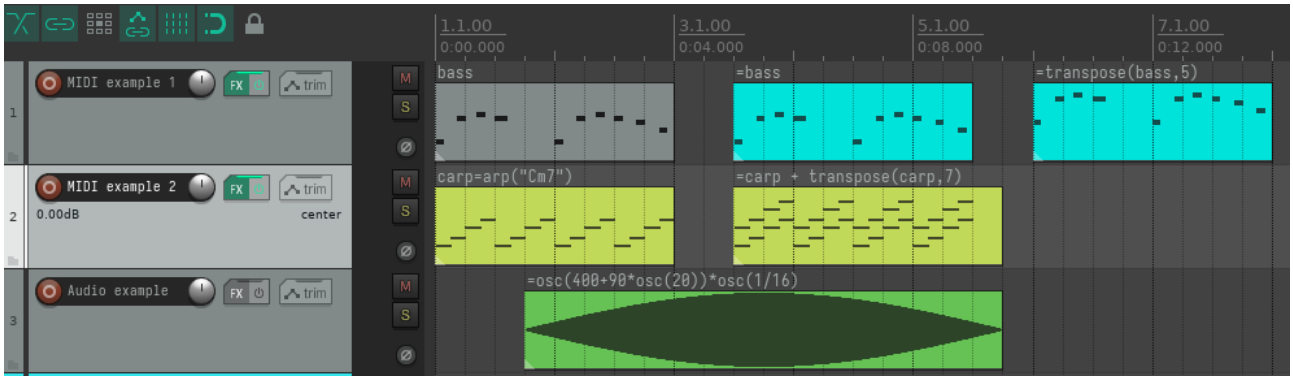

Figure 1: *LambDAW enables the user to create "expression items" that, like static media items, can be arranged on the timeline. Both the code and its generated output (audio or MIDI) are visible in the DAW, and the code may refer directly to other named items in the DAW, whether static or dynamic.*

> *programming in the moment* by allowing you to keep working with the same tools, in the same mindset—thinking only about the specific thing you are modeling—even though you have switched to programming. They don't force you to stop and switch to your *programmer hat*.

We aspire to capture this same feeling in the DAW, by making code (Python expressions) interchangeable with data (audio and MIDI items) in the timeline and in how they are referenced and manipulated by the user.

Of course, bringing code into the DAW and striving to enable *programming in the moment* raises the question of live coding potential. LambDAW borrows concepts from existing systems (such as cycles and deferred playback) in order to support live coding, and it includes features that are unusual among live coding environments, particularly regarding the visualization and recording of performances *as code*. We discuss the implications of these features, and the existence of code in the timeline, for live coding. And although LambDAW does not mandate a particular live coding notation or abstractions beyond standard Python, we demonstrate some integrations with existing Python libraries that are useful for composing and live coding with LambDAW.

2 Related Work

DAWs already support some mechanisms for integrated computation. They can serve as hosts for plugins in various formats (VST2/3, LV2, AU, CLAP, etc.) that process audio and/or MIDI and live in the effects chain. Typically, they are pre-built by a plugin developer, and the user's control is limited to adjusting parameters and routing signals through different plugins. However, some plugins support live development within the plugin, such as Open Music Kontroller's *JIT*[3] (which supports live coding in C and Rust). Camomile is a DAW plugin that loads Pure Data patches, which can also be patched live (Guillot 2018). Products like Native Instruments's Reaktor or Andrew Belt's VCV Rack Pro similarly allow live patching within a DAW plugin. Some DAWs also have built-in support for custom plugin development, such as Ableton's Max for Live, Reaper's JSFX, and Bitwig's Grid. Neil Cosgrove's *LNXStudio*,[4] a DAW written in SuperCollider, similarly allows users to modify the source of instruments and effects from within the DAW. Crucially, the plugin paradigm of integrated computation largely ignores the timeline: plugins exist out-of-time in the effects chain. The timeline can serve as a source of input for plugins when the transport is running, but the timeline is not exposed to the plugin interface, and thus plugins cannot interact directly with its contents. For plugins, this is generally a good thing: by acting as audio or MIDI processors that don't know (or need to know) much about the DAW, plugins are more general and can be useful in more situations, and in more kinds of plugin hosts (beyond DAWs). By the same token, however, their integration with the DAW as a compositional environment is limited.

Gibberwocky also fits into this category and merits special mention, as Roberts and Wakefield (2016) explicitly set out to bring live coding into the DAW. As they put it, "it is a much smaller leap to add generative capabilities to existing musical practice than to start over with built-from-scratch sounds as well as a brand new method of production." We agree and likewise propose integrating with the DAW. But instead of "exploit[ing the DAW] as a processing engine," as in their work with Ableton Live, we embrace the DAW as a full music-making environment for composition and performance. To that end, we not only make use of the DAW's routing and effects-processing capabilities, but also integrate closely with the timeline.

[3] https://open-music-kontrollers.ch/lv2/jit/
[4] https://lnxstudio.sourceforge.io/

A few DAWs also allow for another kind of integrated computation, which is considerably rarer than plugin hosting: scripting. For instance, REAPER supports scripting in EEL2, Lua, and Python. Ardour supports scripting in Lua. Zrythm supports scripting in Scheme and JavaScript via Guile. Scripts are tightly integrated with the DAW and are commonly used to create custom commands and automate tasks. LambDAW takes advantage of this integration and is built on the scripting facilities of REAPER. Unlike scripts, LambDAW expressions appear directly in the timeline and (rather than just performing side effects in the DAW) generate associated content that appears with them, such as audio or MIDI.

Our work is also related to projects which try to marry the DAW and the IDE. For example, *Blue* (Yi 2007) is an "Integrated Music Environment" which allows the composer to make music with Csound in a visual environment. *Ossia Score* is an "intermedia sequencer" which can be scripted and used to orchestra other musical systems. Our work has some goals in common with these projects. But instead of throwing out the DAW and building a new, hybrid environment from the ground up, we extend a proven, performance-ready DAW (REAPER) with generative and live coding capabilities. We get all the features of the DAW "for free," including multitrack recording, audio plugin hosting, flexible routing, robust performance, and all the rest, and we offer a gentle introduction to generativity to the DAW's existing userbase. The tradeoff, of course, is that we must work within existing framework and interface of the DAW, but given that our goal is close integration, this is not so much a drawback as a necessary challenge.

Also in this category is *Manhattan* by Nash (2014), which brings code fragments into a music tracker. Our work is well-aligned with Nash's, who is likewise inspired by the spreadsheet paradigm of end-user programming and seeks to develop a "unified environment" supporting a mix of compositional practices. But we opt to build on the free-form environment of the DAW, which supports both audio and MIDI items anywhere on a timeline, rather than the grid-based (and more spreadsheet-like) interface of the tracker. And we focus on the use of functional expressions to generate the contents of audio/MIDI items, rather than imperative constructs to modify control flow in the timeline itself.

A final example in this category is EarSketch (Mahadevan et al. 2015), an educational platform that allows students to make music with code.[5] The EarSketch interface prominently features both a DAW and an IDE side-by-side. However, EarSketch separates the DAW and IDE for pedagogical purposes, omitting features for directly manipulating DAW contents so that students learn to manipulate it with code. Also, each project has a complete program associated with it, which generates the entire contents of the DAW every time it is executed (though this has not prevented its use in live coding, as in Freeman and Magerko (2016)). In contrast, LambDAW integrates code into the timeline, retaining all of the direct manipulation features of the DAW and eroding the boundary between manipulation and computation. The expressions associated with items in the timeline may be evaluated independently, so that the user can work on individual snippets without re-generating the whole project.

There are a few products that aim to explicitly fill the role of "Generative DAW." These include FMOD Studio (for game audio) and Intermorphic's Wotja (for generative music, especially ambient). The main difference is that instead of rendering to a static audio file at the end, audio processing can be deferred to the time of playback. This is an interesting distinction that we may explore further in the future, but from an interface perspective, the computational aspects remain outside the timeline, as in conventional DAWs.

Looking beyond the DAW, we draw inspiration from other systems that allow mixing direct input with computation. The canonical example is spreadsheet software, which enable users to fill cells with formulas—which compute values and may depend on other parts of the spreadsheet—in addition to literal values. We are also inspired by the more recent examples of *Potluck* and *Inkbase* from Ink & Switch, freeform editors (for text documents and sketches, respectively) that also allows the user to gradually introduce live computations that can refer to other parts of the document (Litt et al. 2022; Lindenbaum, Kaliski, and Horowitz 2022). These have particularly inspired LambDAW's support for referring to other items in the timeline from expressions. We note a connection between our work and the work of Martinez (2021) on *Computational Counterpoint Marks*, extending Common Western Music Notation for algorithmic and interactive music via code expressions (in a custom language) embedded in the score. This interface parallels our embedding of Python expressions in the DAW timeline. Similarly, the music notation program *NoteAbility Pro* supports embedding control messages directly in the score for controlling Max/MSP or Pure Data for interactive music (Hamel 2006).

Within the realm of live coding, our work can be viewed in the lens of hybrid live coding systems. Hybrid practices such as code jockeying (CJing) involve mixing code with other means of expression, such as graphical widgets and hardware controllers, as in Visor (Purvis, Anslow, and Noble 2020), or other musical interfaces, such as the Ableton-esque "scene launcher" in DeadCode (Beverley 2020). LambDAW similarly hybridizes code with an alternate interface: if DeadCode was conceived as "the impropable combination of TidalCycles live coding language and the Ableton Live DAW," then LambDAW began as cross between a Python notebook and REAPER, with a corresponding emphasis on the timeline instead of the scene launcher. We consider how LambDAW fits into discussions of liveness in sec. 4.2.

Finally, we draw on the live coding community's work on terse, composable notation, which pairs well with LambDAW's short expressions in the timeline. We use pattern notation from FoxDot by Kirkbride (2016) in some of our examples below, and we have started experimenting with Tidal Cycles notation (McLean and Wiggins 2010) via recent work on

[5]Coincidentally, EarSketch was also originally built on REAPER's scripting facilities before becoming a web application.

Vortex (McLean 2021). Unusually, LambDAW supports capturing live coding performances *as code*, for future analysis, composition, or remixing. We note that kilobeat (Clester 2021) and SuperCollider (via its History class) also support recording live coding performances as code, but LambDAW further preserves the association between the actual output from a performance and the code that generated it by recording both into the timeline.

3 Design

3.1 Premise

LambDAW extends REAPER by introducing an additional, generative category of media into the DAW environment. To explain how this works, we first establish some terminology: in REAPER, the *timeline* contains *tracks*, which can hold any kind of material—unlike some DAWs, they are not separated into audio and MIDI tracks. Tracks can contain media *items*, which occur at a particular time in the timeline and have a duration. Like tracks, items are untyped. Finally, items contain *takes*, which have user-defined names and contain media of a particular type: audio, MIDI, or video. Items may contain multiple takes of different types, although this is somewhat unusual. Although items are technically untyped, we use terms such as "audio item" and "MIDI item" to refer to items containing only takes of one type for convenience.

LambDAW introduces the concept of an *expression take* (or *expression item*, when an item consists of only expression takes). Like other items, expression items exist on the timeline and can be moved around, copied, and otherwise manipulated in familiar ways. And like other takes, expression takes may contain audio or MIDI. But each expression take also has an associated code snippet, the *expression*, which generates the take's the audio or MIDI content. What's more, these expressions can refer to other items in the DAW; for example, the user can record a synth riff on a MIDI keyboard, then create an expression item that is equal to that riff transposed up a fourth, reversed, played at half speed, etc. If they later edit the original riff, it's easy to reevaluate the expression item so that it stays in sync. This enables the user to combine static and dynamic musical materials in a direct way, without having to switch to another space to edit code. It also allows the user to express relationships between parts in the timeline, encoding these relationships in the project itself and avoiding the tedious work of updating stale copies and transformations.

3.2 Expressions

Expression items have names prefixed by =, like spreadsheet formulas, and they can contain arbitrary Python expressions.[6] LambDAW evaluates these expressions when they are created, when they change, and when the user forces evaluation. It then inspects the results of the expressions, which can represent MIDI events or audio samples, and converts them into the appropriate type of take in the expression item.

This conversion also goes in the other direction: LambDAW creates objects in the global namespace corresponding to items in the DAW timeline. This allows expressions to refer to other items in the timeline by name. For example, the user can record or compose a MIDI item named `my_cool_riff`, and then refer back to it from an expression item with the name `=my_cool_riff`, which (upon evaluation) will be filled with the same content as the original. This provides an easy way to bring back musical material without just copying and pasting it, which can leave stale copies if the original material is later tweaked. Furthermore, expressions can use other items like any other variable, so we can have another item named `=transpose(my_cool_riff, 5)` which *transforms* `my_cool_riff` and will sync up after reevaluation.

This presents an important design question regarding when evaluation should occur. In principle, we could determine which items an expression depends on (either through static analysis or dynamic observation of which items are accessed), and then automatically reevaluate the expression whenever any of its dependencies change. This behavior would follow the spreadsheet model, where formulas automatically reflect changes in referenced cells. However, this approach has some drawbacks when applied to a musical composition environment. It means that local edits can have immediate, non-local effects, and it takes some control away from the user, leading to potentially surprising modifications. If the user has nondeterministic expressions, for example, then reevaluating them automatically might throw out something that the user wanted, such as a particular outcome or variation that they cannot readily retrieve.[7] Additionally, the cost of evaluating an expression may be high enough such that constant reevaluation is prohibitively expensive (as in the case of non-trivial audio synthesis, or, say, MIDI generated by a large neural net). Thus, we instead opt to reevaluate expressions on demand (and when created or edited). This is a more notebook-like model of execution, as in the cells of a Jupyter notebook, which similarly do not track inter-cell dependencies but instead are executed on demand. Since expressions do not execute at the same time, this approach also avoids potentially tricky interface issues

[6] Expression items can have a name before the = so that they can also be referenced in expressions. The general form is thus `name=some python expression`, where `name` is optional.

[7] See, for example, the behavior of spreadsheet cells that invoke `RAND()`. The contents of such cells change frequently and perhaps unexpectedly, including when unrelated cells are modified and when the spreadsheet is reloaded.

like how to deal with circular references.[8] The drawback of this model is that outputs may be stale; we mitigate this by making it easy to reevaluate expressions (as described in sec. 5.1) and automatically reevaluating expressions on edit.

The association of expressions with particular items on the timeline encourages the use of self-contained expressions that simply generate the contents of their take. But expressions can also read global state and have side effects. Reading global state is necessary for referencing other items on the timeline, but it can also be used to get at information like the host item's position on the timeline or the current position of the playhead. Thus the user can write an expression that sounds different based on when it occurs, and LambDAW can provide the current cycle number to expressions when live coding (see sec. 4.3 below).

The fact that expressions can have side effects allows them to do things like advance the state of the PRNG in Python's `random` module (important for expressions involving chance), print things to REAPER's console, and further manipulate the timeline. For example, an expression could both generate audio for its item and also insert automation points below the item, add markers to the timeline, change the project tempo, etc. To avoid unexpected surprises, it is best if expressions keep modifications local—preferably to their own contents, or at least to things in their vicinity. We encourage this usage through the interface and default behavior of LambDAW, but we do not attempt to restrict the capabilities of expressions, so the full power (and danger) of DAW scripting is available.

3.3 Project Modules

As noted, LambDAW expressions are arbitrary Python expressions. Given that these expressions appear directly in the timeline alongside their items, brevity is paramount. Abstractions matter, and the functions and types available for use—which, along with Python syntax, determine the effective *notation*—in snippets determine the range of ideas that can be reasonably expressed in a tight space.

LambDAW aims for generality and thus mostly leaves the question open. We anticipate that different users will want to express musical ideas in different ways, and that this will even vary with the kind of music that an individual user wants to make in different projects. Therefore, LambDAW loads a *project module* which is imported before evaluating any expressions. This project module can be customized by the user to import whatever modules or define whatever functions and classes that they want to work with. It lives with the DAW project, so the user can have a different set of definitions for each project. Reusable definitions can be pulled out into an another module and imported by multiple project modules. The project module is also a convenient place to load resources that may be used across multiple snippets (such as audio files, data for sonification, or trained models).

In analogy, Jupyter pairs well with libraries such as `numpy`, `matplotlib`, `scipy`, and `pandas`, among others, but it does not load any of them by default or mandate their use. Instead, it leaves it up to the user to decide what is relevant for what they are trying to do. LambDAW's position is much the same: it presents an interface to compute within the DAW, but it does not mandate the use of any particular library, musical framework, pattern language, etc. That said, we are very interested in finding integrations that work well with LambDAW, and we present some of them here.

For example, `math` is a built-in module that often comes in handy. By putting `from math import *` into the project module, we can then write audio expressions like so:[9]

```
=sin(2*pi*440*t/sr) for t in range(5*sr)                    # 440 Hz sine wave
=sin(2*pi*440*t/sr) * sin(2*pi*110*t/sr) for t in range(5*sr)  # Ring modulation
=sin(2*pi*440*t/sr + sin(2*pi*110*t/sr)) for t in range(5*sr)  # Phase modulation
```

And by importing `random`, we can create some simple MIDI expressions:

```
# Four random pitches in the range [C4,C5)
=[{"pitch": randrange(60,72), "start": i, "end": i+1} for i in range(4)]
```

If we import a third-party library such as Aleatora (Clester and Freeman 2021), we can write expressions of comparable and greater complexity much more concisely, as in these audio expressions:

```
=osc(440)                          # 440 Hz sine wave
=osc(440) * osc(110)               # Ring modulation
=osc(440 + 90*osc(110))            # Frequency modulation
=osc(400 + 90*osc(20)) * osc(1/16) # FM & AM
=bpf(rand, 500*(1 + osc(0.1)))     # Bandpass-filtered noise, with LFO-controlled filter frequency
```

[8] In a spreadsheet, circular references generate a `#REF!` error that appears in every cell involved in the cycle.

[9] For convenience, an extra pair of parentheses is inserted around the expression before it is evaluated. This allows generator expressions, like `(x*2 for x in range(4))`, to be specified without the extra layer of parentheses, as in `x*2 for x in range(4)`.

Note that these expressions return infinite streams. LambDAW only pulls samples from iterables until it reaches the end of the expression item, which prevents these from rendering forever.

We can also generate MIDI and organize audio samples via Aleatora's support for FoxDot (Kirkbride 2016) patterns:

```
=midi.events_to_notes(tune([0,1,2,P*(3,4)]))  # Simple melody (MIDI)
=beat("x-{ob}[--]")                           # Simple drum pattern (audio)
```

Beyond introducing useful imports and definitions, the user project module also provides a place to customize how musical material is serialized and deserialized, giving the user greater control over the conversion. By default, LambDAW expects expressions to return iterables of either notes (as dictionaries) or floats (audio samples). By defining a custom conversion function, we can modify its behavior and make it understand other types. For example, the following project module tells LambDAW how to handle Aleatora event streams, and it allows us to remove the `midi.events_to_note` call in our "Simple melody" expression:

```
import lambdaw
from aleatora import *

# Here, we override LambDAW's default behavior to
# control how it converts Python values into DAW items.
def custom_convert_output(output, *args):
    # `output` may be a (potentially endless) iterable.
    # So, we peek at the first item in order to determine the type.
    first, output = lambdaw.peek(output)
    if isinstance(first, tuple):
        # Aleatora event stream: convert it to a stream of notes.
        output = midi.events_to_notes(output)
    # Then fallback to the default converter.
    return lambdaw.convert_output(output, *args)

lambdaw.register_converters(output=custom_convert_output)
```

Now we can write our melody more concisely as `=tune([0,1,2,P*(3,4)])`, without the surrounding function call. In addition to aiding brevity, overriding converters enables the user to add new meaning to expression types. For example, the user could setup a converter that interprets a string as text to be synthesized as speech, or one that interprets a 2D numpy array as a background image for the current item in the timeline. These possibilities are left open by giving the user the opportunity to override the default behavior.

4 Live Coding with LambDAW

In addition to the features described so far, which apply in the context of composing or producing in the DAW, LambDAW supports live coding, augmenting the DAW's support for live performance. We discuss the considerations involved in designing LambDAW's live coding support, and the implications of using LambDAW for live coding, particularly for the capture, visualization, and instrumentation of live coding performances. We also touch on how LambDAW relates to ongoing discussions of liveness.

4.1 Design

One might imagine that LambDAW could be used for live coding as described so far, without any special support, as it already emphasizes "programming in the moment." For instance, one could create some expression items, start looping, and then edit the expressions live, causing the items to automatically re-generate. The problem with this approach is that items are regenerated immediately, which means that the contents of an audio or MIDI item may be replaced midway through the loop, causing discontinuities as the DAW abruptly switches from the old output to the new. To resolve this, we can generate the new output immediately but defer its activation until the next loop. We experimented with several variations on this approach, including waiting until the end of the loop to replace the take contents, and employing a double-buffering strategy where the contents are immediately placed on a new take, and the item's active take is switched at the end of the loop. All of these strategies, however, were dependent on the timing of the deferred loop in ReaScript (see sec. 5.2) and could result in glitchy behavior in certain cases.

Ultimately, we attained reliable behavior by taking advantage of the DAW's native, robust scheduling mechanism, the transport. Rather than using the DAW's loops and trying to modify the timeline at just the right time, we instead defer the playback of newly-generated items by simply placing them in the future, further along the timeline. We borrow from existing live coding systems the notion of the *cycle* and continually fill the *upcoming* cycle—the one just after the one that the playhead is currently in—with new expression items and generated output.[10] This ensures that, as long as the item output can be generated in time, there is no discontinuity within a cycle, because switching between takes only happens at the intended boundary between cycles. Scheduling items for the next cycle in advance is analogous to the way many other live coding systems schedule events ahead-of-time for the SuperCollider server or Web Audio API.

Because each bit of code is used to generate items for every upcoming cycle per-track, it makes more sense to associate the expression with the *track* rather than with a particular item.[11] Thus, while live coding, the user edits expressions in the track control panel on the left rather than in items on the timeline. Then, for each upcoming cycle, the track title is copied to the titles of the pending items. If the user changes the track title before the next cycle starts, the changed is propagated to the pending item, causing it to automatically reevaluate the updated expression before the cycle begins.

4.2 Implications

After we settled on this model for live coding, we realized that explicitly looping in the DAW was no longer necessary: the mechanism of cycles generated from track titles naturally creates a kind of *unrolled loop* in the timeline that automatically repeats expressions until the live coder updates them. This unrolled loop both keeps patterns going live and also serves as a record of the entire live coding performance, including both code and its output. Because this record contains code (in the form of expression items) *in addition to* its output, it can be further analyzed, composed, and remixed later—at the level of code rather than recorded audio/MIDI.

Because LambDAW operates within the interface of a DAW, both code and the MIDI/audio it generates is visible to the performer and audience. The performer benefits from seeing pending output in the next cycle and can make any necessary adjustments before it plays. And, by projecting the DAW interface, they can show the audience both code and output, getting a convenient and direct form of visualization for free. (Though, as discussed in sec. 6, we are also interested in using LambDAW for video.)

Live coding in the DAW also suggests the possibility of hybrid performances: mixing live instrumental performance with live coding. We imagine scenarios in which a live coder/instrumentalist might establish a jam via live coding, then play a riff on MIDI keyboard or take a solo on sax, and then reference and transform what they just played through code. Or a live coder might play in a band with instrumentalists, routing their signals through the DAW to riff on their bandmates' ideas. Given that LambDAW already supports mixing and matching static and dynamic materials when composing in the timeline, we are keenly interested in the possibility of doing the same in live performance, and this is a possibility we are just beginning to explore.

Looking beyond its dedicated live coding mode, we can consider how LambDAW (in "composition mode") relates to live coding and related practices. The closest relative to LambDAW within live coding is probably DeadCode (Beverley 2020), mentioned earlier in sec. 2. As Beverley notes: "By deliberately encouraging the preparation and reuse of code, CJing seems to deviate from the form of improvisation, liveness, and ephemerality discussed earlier." The same could be said of LambDAW; outside of live coding mode, it is principally an environment for writing and refining persistent "dead" code in the studio rather than ephemeral "live" code on the stage. Like Dead, LambDAW aims to support "both static composition and improvised exploration" through its composition and live coding modes; the user can go back and forth between composition and improvisation as easily as starting/stopping recording. Any composition can serve as the start of a live coding session, and any live coding session can serve as the start of a composition.

4.3 Usage Example

In this section, we make things more concrete by describing a simple live coding performance with LambDAW (a video is available at https://youtu.be/5VU-ora7Wx0&t=284s). As mentioned in sec. 3.3, LambDAW does not mandate any particular abstractions or notations for expressions, but we use Aleatora and FoxDot patterns here for the sake of example.

First, the user starts up the DAW, and LambDAW starts with it. Then the user creates a new project, and LambDAW automatically creates a corresponding project module using the user's provided template.[12] If desired, the user can go

[10] By default, a cycle is one measure long (based on the DAW project's tempo and time signature), but this can be configured by the user.

[11] If we used, say, the latest item on the track as the source of the expression for the next cycle, the performer would have to keep switching items to update the code for the next cycle, and would have to switch repeatedly if they took more than one cycle to update the expression. See fig. 2 for clarification.

[12] The project module is akin to startup files in other live coding systems: Tidal's `BootTidal.hs`, FoxDot's `startup.py`, Sonic Pi's `init.rb`, etc.

ahead and create some tracks with virtual instruments and effects, or they might have created the project using a saved project template with tracks ready to go. In any case, when the user is ready to begin live coding, they start recording.[13]

Now, the user can enter expressions as track titles. Let's say the user's project module contains the custom converter from sec. 3.3, plus the following definitions:

```
from random import *
import reapy

Scale.default = "dorian"
project = reapy.Project()

def p(pattern, *args, **kwargs):
    return beat(P+(pattern), dur=project.time_signature[1], *args, **kwargs)

def t(pattern, *args, **kwargs):
    return midi.events_to_notes(tune(P+(pattern), dur=project.time_signature[1], *args, **kwargs))

def transpose(notes, offset):
    return [{**note, "pitch": note["pitch"] + offset} for note in notes]
```

The definitions of `p` and `t` use Aleatora's support for FoxDot pattern strings (mentioned in sec. 3.3) to provide a concise way to write patterns. Then, the user names a track `=p("x")` to start a kick drum going. They decide to switch to a different sample and rename it: `=p("|x2|")`. Then they name another track `=p("----")` to get some hi-hats. Then they change the the title introduce more rhythmic subdivisions and random accents: `=p("-[--]-[---]", amp=PRand(1,2))`. On another track, they put a snare with a randomized sample on the offbeat: `=p(" o", sample=PRand(2))`.

Satisfied with samples for the moment, they code up a bassline on a track with a VSTi: `bass=t([0,7,6,4])`. On another track, the put an intermittent horn line up a fifth, referring back to the bassline in case they change it later: `=choice([transpose(bass, 7), ()])`. Then they do a little subtractive synthesis, filtering some noise at varying frequencies with `=bpf(rand, randrange(200, 2000))`. At this point, they might pause to tweak some synth or effects parameters in the DAW or to adjust the mix, leaving their code to keep running in the meantime. If they have the automation mode set to touch, latch, or write, then their parameter changes will be captured along with the rest of the performance. Finally, they change the bassline expression to `bass=transpose(bass, -1)`, using self-reference and temporal recursion to have the bassline modify itself from cycle to cycle (with the intermittent horn line following along). They end their performance by manipulating the DAW's faders directly to fade out.

This brief example might not win a prize for best live coding set, but it does begin to demonstrate how LambDAW can be used for live coding in practice. At the end of the performance, our live coder can stop recording, and later go back and review their performance and their code, and perhaps bring some ideas they like into the studio for future work.

5 Implementation

5.1 Interface

LambDAW takes the form of a REAPER extension written in Python. In creating it, we repurposed many of the existing elements in the DAW interface to serve additional roles for generative music. Generally, this process involved identifying good matches in REAPER and then tweaking them to fit.

For example, we decided early on to repurpose take names for the display of code. REAPER already displays take names right above the item containing the take. Both the name and content is visible. This is ideal for LambDAW expressions, as we want the code to be associated with a specific place on the timeline, and for both the code and its output to be visible. What's more, it's fairly convenient to edit a take name: pressing F2 with the item selected opens up the properties window, with the take name field already focused and selected, and pressing Enter will save the changes and close the properties. The only tweak required here was adjusting the font: using a monospace typeface (we like Iosevka[14] for this purpose, but the end user can choose whatever suits them) and adjusting the size.

[13] LambDAW only generates new items for upcoming cycles when the DAW is recording. This conceptually matches what is happening (the live coding performance is being recorded), avoids unexpectedly editing the timeline when the user is just playing it back, and prevents possible confusion about the interaction of cycles and seeking (as the DAW does not allow seeking while recording).

[14] https://typeof.net/Iosevka/

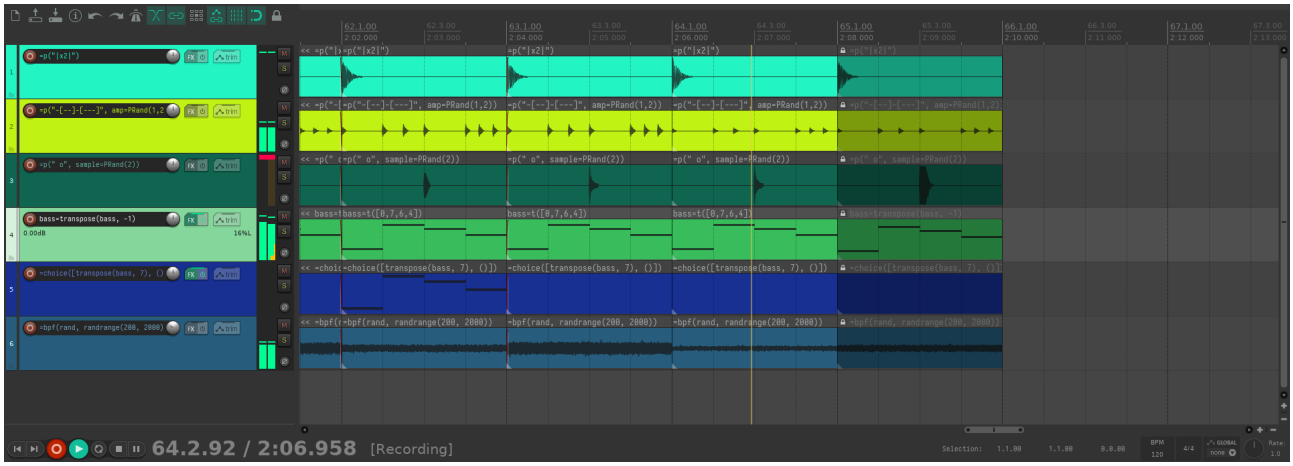

Figure 2: *Live coding with LambDAW. The performer edits expressions in the track control pane on the left, which are used to generate items in each subsequent cycle in the timeline. Pending items in the next cycle are indicated by REAPER's item lock, which makes them appear in a dimmer color (to indicate that they are not yet "final") and prevents the user from accidentally moving them.*

Similarly, when used for live coding, we wanted to find a way to indicate that a generated item in the next cycle was pending rather than finalized. We first explored the option of changing the item color. Then we realized REAPER already has a feature that works nicely for this: item locks. REAPER already shows locked items in a slightly dimmed color (based on the track color by default), and it also prevents moving locked items. These qualities are ideal for indicating items in a pending cycle that have not been finalized on the timeline yet: they are visually distinct, and they can't be manipulated, which makes sense because they aren't really "there" yet. As noted previously, we repurpose track names to contain expressions when live coding in the same way that we reuse take names normally.

In addition to repurposing existing UI elements, we extend the DAW with several commands to support LambDAW. These are summarized here; we include our keybindings as an example (which avoids conflict with default shortcuts), but these may be chosen by the user:

Table 1: LambDAW commands and possible shortcuts, chosen to avoid conflict with the DAW's default keybindings.

Command	Shortcut
Evaluate selected expressions	Ctrl+W
Evaluate all expressions	Ctrl+Shift+W
Toggle selected expressions	Ctrl+Shift+A
Edit project module	Ctrl+Shift+E
Reload project module	Ctrl+Shift+R

In addition the commands listed above, there is also a command to start the LambDAW session, which we do not have bound to any key but instead run automatically on startup. We also bind the built-in REAPER command "Track: Rename last touched track" to Ctrl+Enter for ease of editing when live coding. (The built-in keybindings Ctrl+Alt+Up/Down, which navigate through tracks, are also useful for live coding.)

The evaluation commands force re-evaluation of the selected expressions or all expressions, respectively. The command "Toggle selected expressions" is used to enable or disable expressions in items (or, in live coding mode, tracks). Since only names of the form `name=expression` are evaluated by LambDAW, the command simply replaces = with ≠ in the names of selected items or tracks; this prevents the expression from being reevaluated without throwing away or hiding the code. The command "Edit project module" launches a text editor to edit the project module for the current project; after editing, the user can invoke "Reload project module" to reload the module and refresh the bindings.

Since LambDAW is a DAW extension, we gain in base features and integration but trade off some flexibility. For instance, although take names work well for our purposes, it would be even better if they supported formatting so that we could have syntax highlighting. That said, REAPER is a remarkably extensible DAW, and we are far from exhausting that extensibility. We may explore options like using Christian Fillion's ReaImGui[15] extension to create custom UI components for LambDAW or writing a REAPER extension in C++ to gain greater control in the future.

[15] https://github.com/cfillion/reaimgui

5.2 Engineering & Limitations

In contrast to some of the interface decisions described above, using Python was an obvious choice: REAPER supports it as a scripting language, it has a rich ecosystem of third-party libraries, and it has features (operator overloading, generator expressions, list comprehensions, etc.) that lend themselves well to concise yet powerful expressions. On top of the ReaScript Python API, we also use `reapy`, a Python library by Roméo Després that provides a more ergonomic interface for much of the REAPER API.[16]

However, implementing LambDAW as a set of Python ReaScripts presented some challenges. REAPER creates a new embedded interpreter whenever it executes a ReaScript, which poses problems if we want to have a persistent runtime for evaluating expressions. Reloading modules every time is slow, especially if the user's project module loads assets from disk. More critically, some extension modules—including ones very relevant to this application, such as `numpy`—crash the host application if they are loaded more than once.[17]

To avoid constant reloading and crashes involving extension modules, we implement LambDAW as a long-running REAPER script using `defer()`, which enqueues a callback in REAPER's event loop to be executed in the near future (akin to `setTimeout()` in JavaScript).

The commands *Evaluate selected expressions* and *Evaluate all expressions* don't directly evaluate expressions and update items, but instead store a pending action in REAPER's extension state. Then, whenever the `defer()`d callback fires (every 30ms or so), LambDAW checks the extension state for pending actions and carries them out. It also scans the DAW for new or modified expressions for automatic reevaluation whenever there is a pending action, or once every four `defer()`s. The polling approach is hardly ideal, but it works.

Another issue is that LambDAW evaluates expressions directly in ReaScripts, which run in the GUI thread. A slow expression (say, `=time.sleep(5)`) causes the UI to hang. Due to Python's global interpreter lock (GIL), threading will not help if the expression is CPU-bound (say, `=x/1e8 for x in range(100000000)`). And there is currently no way to interrupt evaluations, so if the user accidentally triggers an infinite loop, their only recourse is to restart the DAW.

6 Conclusion & Future Work

In this paper, we described LambDAW, a system for end-user programming directly in the DAW timeline. We believe that bringing code directly into the timeline, where it can be manipulated like other kinds of media, presents an novel and powerful alternative to its traditional place in the effects chain. Furthermore, expression items' ability to refer to other items in the timeline opens up possibilities for mixing and matching static and dynamic material, and for encoding intent more directly in the project timeline such that the system itself is aware of the relationship between musical materials. We also presented LambDAW's support for live coding and potential implications for live coding capture, visualization, and performance.

Looking ahead, there are several possible avenues for future work. In the near-term, we plan to equip LambDAW to deal with additional kinds of items in REAPER: video, automation, and subproject. Support for video items would allow for multimedia coding, while support for automation items would round out LambDAW's integration into the DAW. REAPER's subprojects suggest a way to allow expressions to generate and place multiple items in the DAW (along the lines of the Functional Console[18] ReaScript) while still having the generated content in a container (the subproject) with an associated expression.

More work remains in exploring and building out integrations. As LambDAW evaluates expressions written in Python, the user can bring in anything from the vast Python ecosystem: libraries for working with networking, DSP, ML, and WebAssembly are just a few of the possibilities that could pair with LambDAW in interesting ways. Already, LambDAW integrates nicely with Aleatora and FoxDot patterns, and we have started experimenting with bringing in Tidal patterns via Vortex (McLean 2021). Also, we would like to make useful integrations easy to access via a default library (perhaps distributed with LambDAW and imported in the default project module template) in order to reduce LambDAW's barrier to entry while maintaining its generality and customizability.

Another area of future work concerns LambDAW's architecture and implementation. As discussed earlier, LambDAW includes several workarounds for issues with REAPER's Python support, and it can cause the DAW to hang due to evaluating snippets in the UI thread. Both of these issues could be solved by performing evaluation in a separate process, which would allow evaluation to proceed in parallel with DAW operation in an independent interpreter. This separate process would serve much the same role as a Jupyter kernel in the context of notebooks. Exploiting this similarity and

[16] https://github.com/RomeoDespres/reapy

[17] For more information about this problem, see https://github.com/numpy/numpy/issues/8097 and https://github.com/python/cpython/issues/78490, both of which are open issues as of this writing.

[18] https://forum.cockos.com/showthread.php?t=263580

using `jupyter-client` to communicate with actual Jupyter kernels may provide a path towards supporting many other languages[19] in LambDAW. It may also simplify the process of porting LambDAW to other extensible DAWs, such as Ardour or Zrythm.

Finally, much work remains towards the broader goal of realizing a truly Generative Audio Workstation. In LambDAW, the output generated by expression items is effectively static until the user regenerates it. And the result of rendering a piece composed using LambDAW is still a static audio file. These are non-issues when using LambDAW for live coding, as new items are continually generated and the "final product" is the live performance, in the moment, rather than the recording. But we aim to support not only rendered music and live music, but also that strange beast in between—generative music. To that end, we intend to explore ways to make LambDAW more generative, such as automatically regenerating some expression items during playback in the DAW, enabling dynamic control flow by associating expressions with timeline regions, or exporting the entire DAW project (with expressions) into a "generative bundle" suitable for use with distribution systems like Alternator (Clester and Freeman 2022) or Streaaaam (Hollerweger 2021).

References

Beverley, Jamie. 2020. "Liveness, Code, and DeadCode in Code Jockeying Practice." In *Proceedings of the 2020 International Conference on Live Coding (ICLC2020)*, 117–31. Limerick, Ireland: University of Limerick.

Clester, Ian. 2021. "Kilobeat: Low-Level Collaborative Livecoding." In *Proceedings of the International Web Audio Conference*. virtual/Barcelona, Spain: UPF.

Clester, Ian, and Jason Freeman. 2021. "Composing the Network with Streams." In *Proceedings of the 16th International Audio Mostly Conference*, 196–99. virtual/Trento, Italy: ACM.

———. 2022. "Alternator: A General-Purpose Generative Music Player." In *Proceedings of the International Web Audio Conference*. Cannes, France: UCA.

Freeman, Jason, and Brian Magerko. 2016. "Iterative Composition, Coding and Pedagogy: A Case Study in Live Coding with EarSketch." *Journal of Music, Technology & Education* 9 (1): 57–74.

Guillot, Pierre. 2018. "Camomile: Creating Audio Plugins with Pure Data." In *Proceedings of the Linux Audio Conference*. Berlin, Germany.

Hamel, Keith. 2006. "Integrated Interactive Music Performance Environment." In *Proceedings of the 2006 Conference on New Interfaces for Musical Expression*, 380–83. NIME '06. Paris, France: IRCAM — Centre Pompidou.

Hollerweger, Florian. 2021. "Streaaam: A Fully Automated Experimental Audio Streaming Server." In *Audio Mostly 2021*, 16–168. virtual/Trento, Italy: ACM.

Kirkbride, Ryan. 2016. "FoxDot: Live Coding with Python and SuperCollider." In *Proceedings of the 3rd International Conference on Live Interfaces*, 193–98. Brighton, UK: University of Sussex.

Lindenbaum, James, Szymon Kaliski, and Joshua Horowitz. 2022. "Inkbase: Programmable Ink," November. https://www.inkandswitch.com/inkbase/.

Litt, Geoffrey, Max Schoening, Paul Shen, and Paul Sonnentag. 2022. "Potluck: Dynamic Documents as Personal Software," October. https://www.inkandswitch.com/potluck/.

Mahadevan, Anand, Jason Freeman, Brian Magerko, and Juan Carlos Martinez. 2015. "EarSketch: Teaching Computational Music Remixing in an Online Web Audio Based Learning Environment." In *Proceedings of the International Web Audio Conference*. Paris, France: IRCAM.

Martinez, Juan Carlos. 2021. "Extending Music Notation as a Programming Language for Interactive Music." In *ACM International Conference on Interactive Media Experiences*, 28–36. Virtual Event, USA: ACM.

McLean, Alex. 2021. "Alternate Timelines for TidalCycles." In *Proceedings of the 6th International Conference on Live Coding*. virtual/Valdivia, Chile.

McLean, Alex, and Geraint Wiggins. 2010. "Tidal–Pattern Language for the Live Coding of Music." In *Proceedings of the 7th Sound and Music Computing Conference*, 331–34. Barcelona, Spain: UPF.

Nash, Chris. 2014. "Manhattan: End-User Programming for Music." In *Proceedings of the International Conference on New Interfaces for Musical Expression*, 221–26. London, United Kingdom: Goldsmiths, University of London.

[19] https://github.com/jupyter/jupyter/wiki/Jupyter-kernels

Purvis, Jack, Craig Anslow, and James Noble. 2020. "Visor in Practice: Live Performance and Evaluation." In *Proceedings of the 2020 International Conference on Live Coding (ICLC2020)*, 163–76. Limerick, Ireland: University of Limerick.

Roberts, Charles, and Graham Wakefield. 2016. "Live Coding the Digital Audio Workstation." In *Proceedings of the 2nd International Conference on Live Coding*. Hamilton, Ontario, Canada: McMaster University.

Yi, Steven. 2007. "Blue: A Music Composition Environment for Csound." In *Proceedings of the 5th International Linux Audio Conference (LAC07)*, 129. Berlin, Germany: TU-Berlin.