# Sardine: a Modular Python Live Coding Environment

Raphaël Forment
Université Jean Monnet (Saint Étienne, ECLLA)
raphael.forment@gmail.com

Jack Armitage
Iceland University of the Arts (Reykjavik), Intelligent Instruments Lab
jack@lhi.is

**ABSTRACT**

**Sardine** is a live coding environment and library for Python 3.10+ focusing on the modularity and extensibility of several base components (clocks, parser, *handlers*). Sardine has been designed to be easily integrated with existing live coding environments as both a tool for experimentation, and a demonstration of various live coding techniques: temporal recursion, patterning, integration in various hardware and software setups. Although the tool is still in active early development, it has already been used in multiple public performances and algoraves, partly enabled by its support for MIDI IN/Out, OSC IN/Out and *SuperCollider*/*SuperDirt* one-way communication through OSC. This paper is dedicated to the introduction of the **Sardine** system, and the explanation of the main guidelines currently followed by contributors to the project. It will also present the preliminary results of our work through practical realisations that served as experimental validation during the early stages of development.

## 1 Introduction

Sardine is a live coding library based on Python 3.10+ focusing on modularity and extensibility of several base components. Despite still being in early alpha stage, Sardine is extensively documented on a dedicated website providing installation guides, tutorials and media examples. Sardine provides three main features linked together by the `FishBowl` – an environment handling synchronisation and communication between them:

- A *scheduling system* based on asynchronous and recursive function calls, inspired by the concept of temporal recursion (Sorensen 2013). Calls can be scheduled in musical time either on an `InternalClock` or a `LinkClock` based on the Link Protocol (Goltz 2018).
- A *modular handler* system allowing the addition and/or removal of various inputs/outputs (e.g. OSC, MIDI) or base components through a central `dispatch` environment named the `FishBowl`. This allows the customisation of IO logic, without the need to rewrite or refactor low-level system behaviour.
- A *concise number-based pattern programming language* with support for basic generative and musical syntax (MIDI notes, polyphony, etc), time-based patterns (clock and absolute time) and handling of symbols.

Sardine, by design, is in the direct lineage of previously released Python based libraries such as FoxDot (Kirkbride 2016), Isobar (Jones, n.d.) or the very recent TidalVortex (McLean et al. 2022). Initially conceived as a demonstration tool, Sardine partially emulates some selected features from the previously mentioned libraries or from the dominant live-coding dialects such as the TidalCycles pattern mini-notation (McLean 2014) or the Sonic Pi imperative scheduling syntax (Aaron 2016). Sardine is designed as an agnostic framework for approaching live coding using Python. Thus, the library aims to support different writing paradigms and different approaches to live performance based on the manipulation of source code. The reliance on regular Python asynchronous functions for scheduling and music writing means that Sardine is particularly suited to let each developer-musician follow their own personal coding style, providing a blank slate for experimental interface building. Furthermore, Sardine's design has been strongly influenced by McPherson and Tahiroğlu's concerns about the idiomatic patterns (McPherson and Tahıroğlu 2020) of usage enforced by computer music software, pushing users to repeat and strictly follow prefered patterns of usage. Sardine focuses on laying out the base infrastructure needed to support live coding in Python and wishes to encourage users to imagine diverse patterning idioms and live coding targets, mini-notations or user-facing scheduling mechanisms and syntax.


Figure 1: *Sardine first algorave in Lorient (France), 2022, October 13th. Photography: Guillaume Kerjean.*

The system's modular architecture is a first step towards the inclusion of more targets and custom input and output handling.

The version hereby presented – `v0.2.0` at time of writing – offers a first look into the complete intended design for the library, and is a near complete rewrite over the `0.1.0` version previously used by members of the French live coding scene and by the first global Sardine users. It features two different clock implementations, multiple handlers for IO (MIDI, OSC, SuperDirt), a robust asynchronous temporal recursive scheduling system, and a reimagining of the Player system previously introduced by FoxDot (Kirkbride 2016). Sardine's originality lies in its temporal model, strongly anchored in Python's default mechanisms for asynchronous programming. Sardine also features a modular overall architecture allowing it to be integrated into any live coding tooling and setup, capable of handling most Python-based scheduling duties or to be integrated into a larger mixed platform setup. It has been developed collectively with the help of John Phan, based on user requests and feedback gathered during a first period of experimentation that saw Sardine being used or integrated by musicians for several algoraves, network-based jams and musical performances.

Sardine has been developed exclusively using the Python programming language with few libraries depending on C++ code through bindings to external libraries. Despite the known shortcomings of Python for interpreted real-time programming (incomplete support of dynamic programming, slowness relative to other interpreted languages), we believe that this language is suitable for the implementation of a live coding library. The large collection of available libraries and modules and the popularity of the language ensures the affordance of good tooling and rich customization and integration options for different text editors, running environments, and more. Sardine already takes advantage of a thorough ecosystem of libraries focused on data input/output, network communication and text manipulation. Moreover, thanks to its lightweight and clear syntax, Python can be read by programmers coming from various backgrounds, making it a convenient platform for collaboration and experimentation with bespoke features needed by performers.

In the present article, we will introduce Sardine by detailing its goals (1) and base implementation centered on the scheduling mechanism (2), the environment/handler system (3) and the mininotation support (4). By doing so, we hope to highlight the basic principles of its inner workings, while providing some context on the current direction taken by the project and by its users.

## 2 Methodology and objectives: a framework for exploring live-coding in Python

Sardine is born out of a curiosity for the implementation of similarly featured Python-based live-coding libraries such as FoxDot, Isobar or the very recent TidalVortex (McLean et al. 2022). At its inception, the Sardine project was thought as an attempt to provide a functional but barebones live coding library for demonstration purposes in a dissertation manuscript; a library capable enough for showing the impact of design and implementation choices on the possibilities of musical expression and on the expressiveness offered by a live coding environment. Therefore, a particular attention has been given to reproducing or at least paving the way for the reproduction of different coding styles and representation of timed musical information. Initial work for the `0.1.0` has been based upon an older personal attempt at writing

a live coding library, then named *ComputerTalk*[1]. ComputerTalk's base design was not suitable with Sardine's goal, and so the base has quickly evolved after the first initial public tests, aiming for increased modularity of the system in order to maximise IO options.

The development of Sardine began initially in a period of frantic collaborations and joint performances with the Parisian *Cookie Collective* ("Cookie Collective" 2016) and the digital audio community from Lyon, in particular *th4* and *rAlt144MI*. Stemming from the demo and shader-coding scenes, Cookie are known for complex multimedia performances, with each member relying on bespoke hybrid audio-visual setups, ranging from low end computing devices to complex synthesizers and circuit-bent video mixers. Cookie are also known for working in an improvised manner, customising setups for each venue depending on audience needs and expectations. This need for customisation gave rise to the idea of a modular interface that could be used and mastered by every member of the collective, while allowing for jam-ready synchronisation with other musicians and live-coders. The recent splitting of FoxDot's development into several scattered branches reinforced the need for a central, customisable and easily editable Python interface, the language being used particularly in the French live coding community. Due to an open-ended development process, Sardine has been gradually shifting towards its current modular architecture, allowing each performer to refine the system, from simple MIDI note output to more convoluted SysEx and OSC protocols. The invaluable help and expertise from John Phan has allowed for a rewrite of every base mechanism. The finalisation of Sardine's new framework allows focus to turn towards introducing new features and improving existing ones, with users being encouraged users to propose ideas and contribute code and documentation.

## 3 Sardine implementation and installation

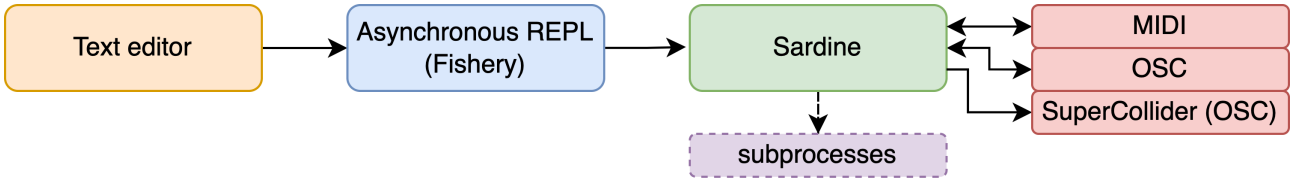


**Figure 1**: *Software layers of the Sardine system stack.*

Sardine is implemented and distributed as two complementary Python modules: `sardine` and `fishery`. These modules typically work hand-in-hand to provide an interface to the user. `fishery` provides amendments to the default Python asynchronous REPL[2] and constitutes the entry point for the Sardine system, accessible by typing `python -m fishery` or simply `fishery` after install. `fishery` is a modified version of the base Python asynchronous REPL, and importing it also imports `sardine` and will by default start a new live session. To help new users, a terminal based configuration client (`sardine-config`) is also provided and can be used to setup various options before starting `fishery`. Configuration files are stored in a single folder inside a OS-specific default standard location (`e.g. .local/share` on UNIX systems), and include a general JSON file, a blank `.py` script for loading user-specifc Python code, and files needed to properly configure a SuperCollider or SuperDirt session. Despite its initial complexity, this approach makes Sardine more accessible to novice users who may not be familiar with the command line and Python development tools. Although modularity of configuration is greatly encouraged, many of the input and output components are disabled by default, making the installation of other programs like SuperCollider entirely optional.

Sardine does not require its own editor or dedicated plugin; an editor only needs to support spawning an asynchronous Python REPL and piping code from a text buffer into it, which is generally supported by most standard Python plug-ins[3]. With this approach, Sardine has been successfully tested with Atom, VSCode, Emacs, Vim/Neovim and Jupyter Notebooks. Sardine considers the Python interpreter as a code receiver and a monitoring tool for displaying useful information to the user, such as the state of a SuperCollider sub-process, or of an event loop.

Reliance on an audio backend requires booting another application. So far, SuperCollider and SuperDirt are natively supported to do this by their own Sardine components. Even though the installation of these backends is still necessary for users willing to use them, integration is done in such a way that there is no need – later on – to actively take care and monitor any of these dependencies. A basic API is offered through the `SC.send()` function, allowing to run arbitrary `sclang` code in the subprocess session. The addition of more managed subprocesses is planned and will be explored in the coming months, including deeper SuperCollider integration, a CSound backend, and more. Combinations of

[1] Some videos of this older system can be found on Raphaël's YouTube channel: https:/youtube.com/watch?v=MHGYtlKibUo
[2] *Read, Eval, Print, Loop*: mechanism used by most interpreted languages to quickly process user input from the command line.
[3] The process for setting up various interfaces is extensively detailed on Sardine's Website.

provided functions already allow some amount of customisation for patterning hardware and software synthesizers through MIDI and OSC.

Sardine is packaged as a regular Python module, making use of the `pyproject.toml` module configuration and packaging format defined by PEP 660, meaning only a base Python 3.10+ runtime is required for installation. Sardine has recently been packaged and released on `PyPi`, allowing any Python user to install it using the `pip install sardine-system` command. All C++ dependencies and `wheels` – binary distribution of compiled packages – are available for every major platform. This also means Sardine is compatible with Python 3.11.

## 3.1 Event loop and scheduling system

### 3.1.1 Event loop

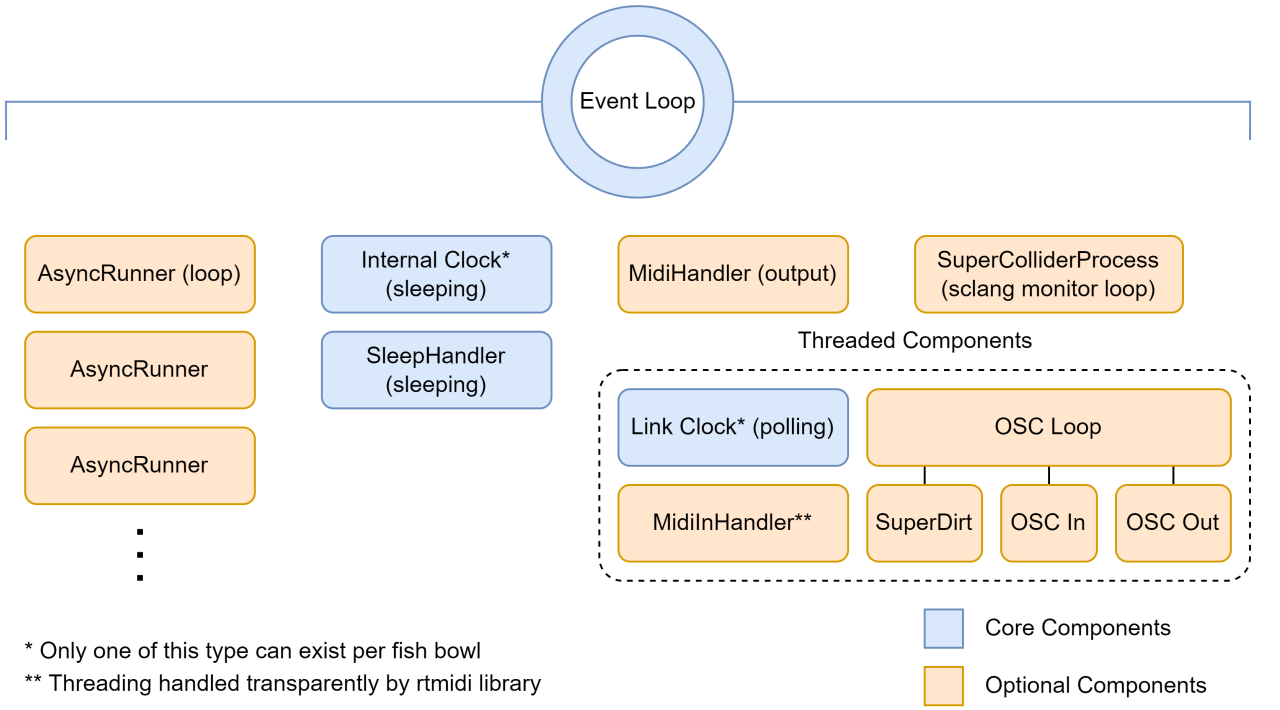


**Figure 2:** *Architecture diagram of the customised asynchronous event loop.*

Sardine makes use of Python's asynchronous programming features, specifically the `asyncio` REPL prototype introduced by Python 3.8 (Selivanov 2019). The UVLoop (Selivanov 2016) drop-in replacement event loop is also used in order to speed up asynchronous call scheduling (except on Windows). Several hot-patches to the asynchronous loop have been introduced by John Phan (*thegamecracks*) to make its behaviour consistent on every major OS platform. Sardine is laid out as a series of abstractions built on the base loop, making it aware of tempo and timing. Sardine's clock (either `internal` or `link`) automatically starts whenever the system is imported, but pure asynchronous calls can still be handled even if the clock is stopped. The `LinkClock` allows Sardine users to connect their session to a global tempo on a local network, enabling networked synchronisation of several Sardine instances and/or other Link-enabled devices. All Sardine clocks provides the same interface, allowing the system to retrieve the current bar, beat and position in musical time. Time drift compensation and time shifting needed by some features is handled by the low-level event loop system.

Asynchronous clock consistency is covered by tests (in the `tests/` folder) and favourably compares with alternatives offered by similar, more widely used threaded clocks. Development of such a feature has proven to be a difficult technical challenge due to the specificity of the task, and of the relatively obscure and scarcely documented inner workings of each OS's internal schedulers. Threaded components are still used for various IO operations to lighten the load of the event loop and to alleviate the temporal cost of message processing. Many Sardine components are optional and can be activated on demand by the user, apart from the `clock`, `AsyncRunners` and `SleepHandler` core abstractions needed for `asyncio` loops. Basing Sardine's custom event loop on Python's asynchronous interpreter allows for the evaluation of any top-level asynchronous `await` instructions that would be forbidden by the main interpreter. However, Python

asyncio ultimately behaves differently on every major OS due to its binding with system-level mechanisms, and more empirical testing is needed to document OS-specific limitations of this approach.

### 3.1.2 Scheduling

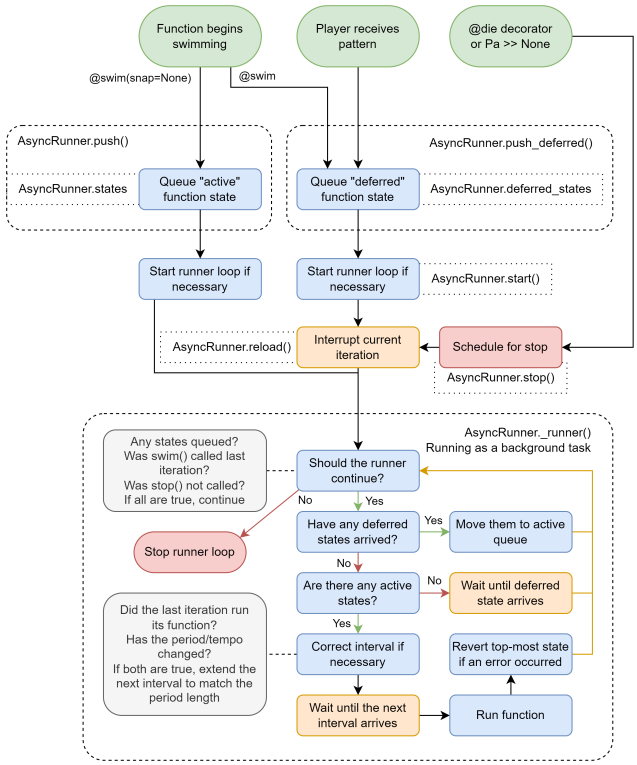


**Figure 3:** *Lifetime of an asynchronous 'swimming function'.*

Python does not feature native tail-call recursion support (Rossum 2009a, 2009b), making the infinite recursion of a function, as Sardine requires, a delicate task. Support for this central feature is based on John Phan's `AsyncRunners`, which is the basis for every repetitive operation – such as a repeating pattern – scheduled within Sardine. In the spirit of the nautical theme of Sardine, a temporal recursive function is called a *swimming* function and is labelled in code as an `AsyncRunner`. Swimming functions can be started using the `@swim` decorator[4], stopped using the `@die` decorator, and can receive updates throughout its lifetime on the scheduler (see Figure 3).

Decorated asynchronous Python functions are passed to the scheduler, making them repeat every `p` (for `period`), a time measured in beats relative to the clock currently in use. The content of a given function will be re-evaluated for every recursion cycle and state can be preserved either by passing arguments to a subsequent call or by relying on global state. Swimming functions are a powerful construct for building abstractions dealing with time, code re-evaluation and dynamic lifetime management of code components. Iterators, for example, can be built by incrementing a variable passed as an argument. Random generators can be built by calling a simple native random function whose result will be dynamically updated for each recursion.

Swimming functions will automatically start on-the-beat. The start of a function can target a specific point in musical time by specifying a special `snap` argument that is intepreted as an offset in beats from the beginning of the next bar. The `period` argument of a given function is the only required argument for a function to be considered as a valid swimming function. All Sardine components are based on the assumption that their evaluation context will be as a swimming function. They can receive any arbitrary Python code and/or call the various players defined by the Sardine system to handle IO operations. Thus, the prototype of a basic musical function using the previously defined model looks like the following:

```
@swim                                    # swimming decorator (swim or die)
def swimming_function(p=0.5, i=0):       # p: (period), i (custom iterator)
    print('I am swimming in time.')      # genuine function call
    D('bd, hh, cp, hh', i=i)             # call to the 'Dirt' SuperDirt interface.
    ...                                  # other user specified code
    again(swimming_function, p=0.5, i=i+1) # recursion callback with argument passing
    # remove the call to again() to stop the recursion from happening, stopping the runner.
```

**Figure 4:** *A commented complete example of a 'swimming' recursive function.*

Multiple abstractions can be built on top of the basic swimming function mechanism, allowing for a terse user-facing syntax. Building abstractions on top of `@swim` is helpful to allow newcomers to grasp the system's temporal model. The FoxDot-inspired *surfboard* mechanism is the first available abstraction demonstrating this principle. It automatically handles its own scheduling logic and provides its own iterators to the default parser. As demonstrated by the following

[4]Decorators in Python are used to add a behaviour to an object such as a function without modifying the base object itself.

example, it also provides customised musical logic, thus adding a completely new flavour of patterning and scheduling. Following this model, Sardine future versions are likely to include user-based playing modes built upon the basic abstractions provided by the library.

```
Pa >> d('bd, hh, cp, hh', p=0.5) # Terser version of the above swimming function.
Pb >> d('voodoo', span=2)        # 'span' extends the inner-hidden swimming
                                 # function duration to span over twice the
                                 # duration of Pa.
Pc >> d('voodoo, tabla', legato=0.1, span=2, p='1,2,3,4')
                                 # duration values of Pc will be fitted to the
                                 # duration of the given timespan.
Pc >> None                       # alternative to @die
```

**Figure 5**: *A 'surfboard', custom FoxDot-like emulation adding a new playing mode to Sardine.*

Worthy of note is Sardine's `sleep()` method, which has been overridden from the default Python `time.sleep()` function that would, if used, block the event loop. This function defers the execution of any statement or expression defined thereafter to $x$ beats in the future, even if these events take place after the next recursive call, a phenomenon known as oversleeping. Unlike `time.sleep()`, Sardine's `sleep()` does not block the function from running to its end, instead it temporarily affects the value of `clock.time` and extends the perceived time of methods using that property. This mechanism mimics the `sleep()` statement found in other live coding tools such as Sonic Pi (Aaron, Orchard, and Blackwell 2014).

```
@swim
def sonorous_cake(p=2, i=0):
    D('jvbass!4, jvbass:4', midinote='C,Eb,G,D', i=i) # SuperDirt calling sample playback
    sleep(1)                                          # Deferring further operations to next beat
    D('jvbass:4!4', midinote="C', C''!3", i=i)        # Other sample playback
    again(sonorous_cake, p=2, i=i+1)                  # Recursive call
```

**Figure 6**: *Usage of the 'sleep' method to defer execution, mimicking Sonic Pi.*

## 3.2 Environment, dispatch and handlers

### 3.2.1 The `FishBowl`

While scheduling takes an important role in the overall modular design of the Sardine library, its logic wouldn't function without a central piece of the system called the `FishBowl`. The `FishBowl` is an environment for software components, and handles synchronisation and coordination between all the different pieces composing a Sardine system. It is designed so that every component of the system can talk and instantly access data held by any other component. The `FishBowl` allows components to subscribe to it through `hooks`, and to react to messages through a `dispatch` system. The `bowl.dispatch('stop')` message is an example of such an event which stops the clock and requests for the collaboration and immediate response of multiple components. Naturally, some components are more important than others and can thus be considered as hard dependencies. Other soft dependencies, mainly the various IO handlers available, can be added and removed from the environment or session at any point in time. The `clock` and the `parser` are two hard dependencies that cannot be completely removed, but can be swapped. They provide the basic mechanisms needed by other modular components to properly function. The fluidity of the `FishBowl` mechanism allows for the addition and removal of modular logic to any Sardine system capable of answering to any message currently being dispatched to other components. One can switch from the `internal` to the `link` clock on-the-fly if the need arises to synchronise with other players, or add a new OSC receiver. Even though the current version of Sardine does not feature multiple parsers, the parser can also be switched.

```
bowl = FishBowl(clock=clock(tempo=config.bpm, bpb=config.beats)) # declaring the bowl
...
midi = MidiHandler(port_name=str(config.midi))                   # instance of new component
bowl.add_handler(midi)                                           # adding to the environment
M = midi.send                                                    # aliasing for playability
```

**Figure 6**: *Excerpt from Sardine boot process, addition of a MIDI Output.*

### 3.2.2 Case-study of a component: the MIDI sender

In the previous code example, a MIDI handler was added to the `FishBowl`, giving access to a new MIDI output. *Senders* are one type of Sardine modular components which requires the collaboration of multiple parts of the system to function properly. The `M` (`midi.send`) function serves as the central output and user interface for this component. It is the only function that the user plays with during a session. To operate efficiently, it requires an access to the `parser` for patterning and composing a valid message, to the `clock` for sending its message in musical time, and to the `SleepHandler` to precisely time calls between a Note On and Note Off message. By declaring itself to the environment, it gains access to these required features, that will be accessed transparently without having to deal with lower-level logic. Consequently, user interaction can be implemented through one minimal function only, letting the system handle the hard and slightly convoluted asynchronous scheduling calls taking place in the background.

```
# basic MIDI note scheduling (duration handled by bowl.SleepHandler)
M(note=60, velocity=100, channel=0, dur=0.25)
# patterning a similar call with added component-specific logic (strings parsed by bowl.parser)
M(note='C@penta, C.., G3', velocity='80~100', channel='[0:10]', i=i, r=2)
```

**Figure 7**: *Sending MIDI using multiple components of the bowl.*

Similar senders or handlers can be implemented for various operations requiring collaboration between multiple parts of the system. Given that each of these adhere to the `BaseHandler` abstract base class, adding a component to Sardine does not require refactor the base system. Most of the internal critical components of Sardine work by taking advantage of interconnection of every component, allowing one component to affect the behavior of the whole environment, if needed.

## 3.3 Sardine pattern language

### 3.3.1 Sardine default pattern language

Initially for demonstration and usability purposes, a small domain-specific language (DSL) for musical patterns has been developed for Sardine using Lark (Shinan n.d.) for LALR parsing. The DSL's source code is directly included in the sardine module, in the sequences/ directory. The DSL's source code is directly included in the `sardine` module, in the `sequences/` directory. A DSL was needed to deal with the limited support provided by Python for syntactic macros (à la Lisp) and operator overloading. The use of the hard `parser` dependency is limited to parsing string arguments provided to any handler's `send` method. These `send` functions, common to any sender, act as its principal interface for patterning and output alike, enabling the creation of complex data and music patterns, evolving over time in the context of a swimming function.

Patterns play an important role in the workflow of audio/visual live coders, allowing them to define rich evolving structures spanning over time (Magnusson and McLean 2018). A generic interface `Pat()` is also available to increase the patternability of Python code or function calls done in the context of recursive swimming calls. This basic pattern language is best defined as a rich and terse interface dealing with lists of arbitrarily typed elements ranging from numbers to MIDI notes, samples or synthesizer names, or even OSC addresses. Pattern strings given for each keyword argument provided to a `send` method are resolved by the parser as arbitrarily-nested lists, which are in turn used for composing an output message. Quite possibly, multiple connections to different parsers will be supported in future versions as a way to vary the idioms available for composing patterns. For now one parser is supported in each instance of the `FishBowl`, and its use is optional.

```
Pat('1, 2, 3, 2~40, 5!4, C@fifths', i=i) # standalone call to the parser, yielding one value
D('amencutup:[1,10]', legato='0.1~0.8', room='0.5', dry='[0.1,0.5:0.05]', i=i) # several calls nested
                                        # inside a more densely composed call to SuperDirt.
```

**Figure 8**: *Several calls to the parser spread in different Sardine methods*

Extensive support has been dedicated to list-based operations for the composition of sequences. Binary arithmetic operators such as `+`, `-`, `*` and `%` can work either on single tokens or on lists (on both sides). Lists can be arbitrarily nested. List slicing and value extraction has been re-implemented in a fashion similar to that of Python. Unary operators such as `abs()`, `sin()` or similar scientific calculation functions work in a similar way, with the function being mapped to

each element of the list if needed. Custom operators have also been defined such as `x~y` (choosing a number in range), `x|y|z` (choosing between *x* elements). Other operators have been borrowed to similar pattern languages such as Ziffers (Alonen n.d.) and TidalCycles: `'` for octave up, `.` for octave down, `:` for sample choice, `!` for repetition, among many others. Basic music notation is handled through the conversion of specific tokens to single MIDI notes (`C#4` or `D#4` parsed as `61`), silences (a dot `.` or elipses `...`), list objects (`C@penta` parsed as `[60,62,64,67,69]`) with support for transposition, chord and structure inversion. A complete list of all supported operations is provided to the user through Sardine's documentation. Support for random and generative structures, albeit basic, has been implemented. Again, the implementation of this feature has been facilitated by the definition of the parser as a component of the `FishBowl`. This allows the parser to query the environment and `bowl.clock` in search of semi-random number generators, such as the measure number (`m`), or current execution time given as clock time (`$`).

```
# Middle-C MIDI Note with default velocity and channel (M, alias for midi.send)
M(note=60)
# C major natural seventh chord with velocity in between 80 and 120, channel either 0, 1 or 2
M(note='<C@maj7>', velocity='80~120', channel='0|1|2', i=i)
# SuperDirt call, picking samples '0' to '20' in order in the 'drum' folder. Speed parameter
# ramping from 1 to 10 in increments of 2, shape is the sin function of current time divided by 2.
D("drum:[0:20]", speed='[1:10,2]', shape='sin($)/2', i=i)
```

**Figure 9**: *Various commented examples of Sardine patterns.*

Querying values in the possible multiple patterns per sender is done by providing a single pattern-wide iterator (labelled `i`) as an argument to each `send` function. Indexing errors are taken care of by making this index cyclical over the length of each pattern. The design of the iterator is a key creative choice for the user. Hence, the preferred method for browsing through the reduced list patterns can be chosen depending on context: sequentially, in reverse, or using a random number generator. More arguments, namely `rate` and `div`, can help in specifying how the iterator will be applied to the gathered patterns, adding another layer of patterning:

- `rate` (alias `r`): compress or extend the number of iterations needed to move from a list index to the preceding/next. Used as an equivalent to slowing down or speeding up the iteration over patterns.
- `div` (alias `d`): a modulo operation between the iteration count and `div` that will determine if the pattern will be played. Used to generate interesting rhythms by confronting sender calls with different `div`s.

Based on our experience, the iterator-based pattern system is well suited to a system based on temporal recursion, partly as recursive operations are often used as iteration tools in functional approaches to programming. Multiple iterators can be used in the same pattern by playing around with the `Pat()` mechanism previously described. This allows for the creation of arbitrarily complex patterns composed of multiple values assigned to any parameter accessible through a given sender. Even though the list of features provided by the pattern language is dense, its architecture is not complex and allows for quick customisation and feature addition. This parser also provides basic patterning functionality, while new parsers are being added to future versions of Sardine.

### 3.3.2 Planned extensions of the parser mechanism

The basic Sardine parser is already useful for increasing the playability of the system, however multiple extensions are planned for future versions. These additions will come in two categories, the first focusing on additions and improvements to the basic parser, and the second on adding new parsers mini-notations. Support for the Ziffers numerical notation created by Miika Alonen, defined as a PEG parser, is in progress. To be supported, Ziffers will require the inclusion of a new type of `Player` similar to the previously mentioned surfboards, relying on Sardine's low-level scheduling mechanisms. Supporting Ziffers will test the ease of integrating new paradigms into Sardine, while opening up new ways to write melodies and harmonic content. Simultaneously, the basic parser is currently undergoing refactoring. Although the pattern language currently supports the definition of custom operators and function calls with arbitrary numbers of arguments, the language is not advanced enough to support all desired operations. Adding high-order functions and easier function calls will be prioritised, as well as better matching for basic tokens. The basic parser aims to support a large range of functions typically found in most functional programming languages, paving the way for a more functional pattern writing style typically found in other live coding systems.

# 4 Sardine usage


**Figure 10**: *Sardine v0.1.0 code during an algorave (Zorba, Paris), November 3rd, 2022*

Over its relatively short timespan, from September to December 2022, Sardine has already been used multiple times for public music performance. The thrill and danger arising from using an unstable and unpolished software has constituted an obstacle as well as an appealing perspective for most users (Roberts and Wakefield 2018). Jams, informal meetings and dialogue with the wider live coding community were also of the utmost importance to study the relevance and/or the inefficiency of integrating Sardine, in often already pre-established musical contexts. Here we mention a few specific examples of Sardine usage, and show how being able to live code in Python provides unique advantages (and also potential disadvantages). Numerous specialised devices or handlers have been created to facilitate Sardine interacting with other systems. Hybridations between multiple live coding environments, such as *Sardine/TidalCycles* and *Sardine/Sonic Pi* have been tried successfully. Initial failures mostly arised from improper or careless implementations of the synchronisation mechanism. Documentation pertaining to the implementation of live-coding oriented musical clocks and synchronisation mechanism is scarce and most of it had to be inferred from the inner workings of similar open-source libraries. In order to help with the collective effort of documenting live coding practices, we hope to upload a very thorough explanation of the system on the Sardine website to help future developers.

A proto-sender specialised in SysEx communication with a Yamaha TX7 synthesizer unit has been designed by Raphaël in order to enhance melodic and timbral capabilities of *Ralt144MI* (Rémi Georges) current live coding setup, previously mostly based on TidalCycles, MIDI controllers and audio-video hardware. Using a simple dictionary of `lambda` functions, this mechanism uses the previously described `Pat()` function to provide a general patterning interface for each individual parameter defined in the MIDI specification of the unit. Subsequent performances led to the inclusion of more bespoke mechanisms that ultimately made their way into the main codebase, having been proved useful in the context of live performances (`span`, `snap`, support for polyphony, etc...). This experience also proved the usefulness of adding better support for the definition of custom-fit *senders* and output/patterning interfaces.

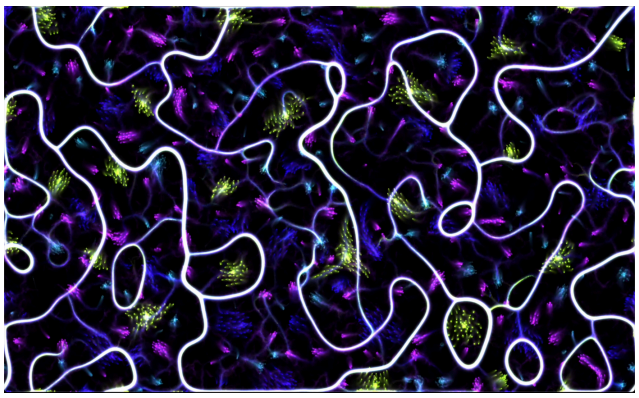

**Figure 11**: *Tölvera artificial life simulation run inside a Sardine `@swim` function.*

At the Intelligent Instruments Lab in Reykjavík, Jack Armitage has been using Sardine to live code the Magnetic Resonator Piano (MRP) (McPherson and Kim 2010). In this case, Sardine enabled developing a Python API client for the MRP much more interactively, due to being able to redefine `@swim` functions on the fly. Another attractive feature of Sardine being written in Python in this case, is that the entire Python ecosystem can be explored for machine learning based approaches to instrumental control and interaction[5]. This also positions Sardine as a bridging platform for researchers and practitioners interested in hybrid live coding interfaces, as in this case

[5]Python codebase for the Magnetic Resonator Piano: https://github.com/Intelligent-Instruments-Lab/iil-mrp-tools/

where live coding and instrumental practices are blurred. Jack Armitage has also been live coding artificial life simulations using Sardine, in the *Tölvera* library as part of the *Agential Scores* project (Armitage 2022). This is one of the first examples of Sardine being used for visual output. In this case, the Taichi numerical simulation library was able to run its animation frame loop as a `@swim` function, with Sardine's recursion occurring fast enough to give 60fps graphics[6] (see Figure 11). These early examples showed the diversity of approaches that are possible in the Python ecosystem, and they also usefully highlighted usability pain points and performance bottlenecks with Sardine. As a result, substantial rewriting efforts have been taking place to accommodate these needs. Motivation for the rewrite process was found in the perspective to support more of these creative endeavours that would be hard to undertake in more closed or less configurable live coding systems.

# 5 Conclusion

## 5.1 Current issues and shortcomings

Sardine is both a new software/environment and an architecture model for a Python based live coding library. Developed only by a small dedicated team of developers with few enthusiastic users, some features are already hard to keep a track of, for they are not often used. Much needed updates are currently delayed to address more urgent concerns. Though tests have recently been introduced, only time-critical parts of the system are currently extensively covered. The `v.0.2.0` rewrite focused on improvements to the temporal model and component handling, and although it's promising, it remains fragile and requires careful, time-consuming testing to identify regressions and new issues brought by additions to the model. As such, regular updates are being released to increase the robustness and playability of the system.

Sardine is in need of documentation focused on new and inexperienced users. The installation process still requires configuring a text editor and the installation of a Python runtime. Existing text-based documentation provides a thorough tour and exploration of the system, but lacks friendly videos, tutorials and content that could boost Sardine's adoption curve. As a temporary solution to this, we show code used by performers in a special `Demo` section of the website. Automatic codebase documentation is planned, to liberate time to focus on development. The source code is extensively documented for contributors, but currently invisible to end-users. Sardine's modularity can be considered as both its major strength and weakness. Initial configuration of Sardine can be quite intimidating for newcomers that may not know what they wish to accomplish with it, especially regarding external IO (MIDI, OSC, SuperDirt, etc.). Most options are disabled by default and must be added manually to configure a session.

## 5.2 Learning, contributing, testing

The Sardine project is freely usable and modifiable by its users. It is currently hosted on GitHub under the GNU General Public License v.3.0. We warmly encourage anyone interested to try this experimental system and help report and triage bugs, and collectively build this new live coding system. A Discord server and a TOPLAP `#sardine` channel are used for communication around the project. Sardine is currently by no means as complete as other existing propositions used by the community. We are still in the process of catching up with known alternatives, thinking as Sardine as a way to extend or collaborate with other musicians and visualists using different live coding environments.

# 6 Acknowledgments

Raphaël Forment: I warmly thank my thesis supervisors Laurent Pottier and Alain Bonardi for their support and advice in the creation of this tool. I thank the doctoral school *3LA* from the University of Lyon for the funding it provided to this research. I extend my thanks to the musicians and friends who allowed me to take **Sardine** on stage and to present it to a wider audience these few last months, in particular the Cookie Collective, Rémi Georges, Yassin Siouda and many more from the online Sardine chat channels.

Jack Armitage: my work is supported by the Intelligent Instruments project (INTENT), which is funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant agreement No. 101001848). I contributed use case examples and testing feedback on Sardine's performance, and assisted with proof reading and final editing of the paper.

[6]Code from this experiment can be found on the *Tölvera* GitHub repository: https://github.com/Intelligent-Instruments-Lab/iil-python-tools

## References


10 Aaron, Samuel. 2016. "Sonic Pi, Performance in Education, Technology and Art." *International Journal of Performance Arts and Digital Media* 12 (2): 171–78.

Aaron, Samuel, Dominic Orchard, and Alan F Blackwell. 2014. "Temporal Semantics for a Live Coding Language." In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modeling & Design*, 37–47.

Alonen, Miika. n.d. "Ziffers: Numbered Notation for Composing Algorithmic and Generative Music." *GitHub Repository*. https://github.com/amiika/ziffers; GitHub. Accessed December 15, 2022.

Armitage, Jack. 2022. "Agential Scores." *Intelligent Instruments Lab*. https://iil.is/research/agential-scores (Last Accessed: 2022-12-14).

"Cookie Collective." 2016. *Collective Website*. https://cookie.paris/all/ (Last Accessed: 2022-12-14).

Goltz, Florian. 2018. "Ableton Link–a Technology to Synchronize Music Software." In *Proceedings of the Linux Audio Conference*, 39–42.

Jones, Daniel John. n.d. "Isobar." *GitHub Repository*. https://github.com/ideoforms/isobar.

Kirkbride, Ryan. 2016. "Foxdot: Live Coding with Python and Supercollider." In *Proceedings of the International Conference on Live Interfaces*, 194–98.

Magnusson, Thor, and Alex McLean. 2018. "Performing with Patterns of Time." In *The Oxford Handbook of Algorithmic Music*. Oxford University Press. https://doi.org/10.1093/oxfordhb/9780190226992.013.21.

McLean, Alex. 2014. "Making Programming Languages to Dance to: Live Coding with Tidal." In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modeling & Design*, 63–70.

McLean, Alex, Damian Silvani, Raphaël Forment, and Sylvain Le Beux. 2022. *TidalVortex Zero*. Zenodo. https://doi.org/10.5281/zenodo.6456380.

McPherson, Andrew, and Youngmoo Kim. 2010. "Augmenting the Acoustic Piano with Electromagnetic String Actuation and Continuous Key Position Sensing." In *NIME*, 217–22.

McPherson, Andrew, and Koray Tahıroğlu. 2020. "Idiomatic Patterns and Aesthetic Influence in Computer Music Languages." *Organised Sound* 25 (1): 53–63.

Roberts, Charlie, and Graham Wakefield. 2018. "Tensions and Techniques in Live Coding Performance." In *The Oxford Handbook of Algorithmic Music*. Oxford University Press. https://doi.org/10.1093/oxfordhb/9780190226992.013.20.

Rossum, Guido Van. 2009a. "NeoPythonic, Tail Recursion Elimination." *Personal Blog*. https://neopythonic.blogspot.com/2009/04/tail-recursion-elimination.html (Last Accessed : 2022-12-14).

———. 2009b. "NeoPythonic, Final Word on Tail Calls." *Personal Blog*. http://neopythonic.blogspot.com/2009/04/final-words-on-tail-calls.html (Last Accessed: 2022-12-14).

Selivanov, Yury. 2016. "UVLoop." *GitHub Repository*. https://github.com/MagicStack/uvloop; GitHub.

———. 2019. "Implement Asyncio REPL." https://github.com/python/cpython/issues/81209.

Shinan, Erez. n.d. "Lark." *GitHub Repository*. https://github.com/lark-parser/lark; GitHub. Accessed December 15, 2022.

Sorensen, Andrew. 2013. "The Many Faces of a Temporal Recursion." http://extempore.moso.com.au/temporal_recursion.html (Last Accessed: 2022-12-14).