

# PUMP UP THE JAMS: V0.2 AND BEYOND

Brian McFee<sup>1,2,\*</sup>, Eric J. Humphrey<sup>4</sup>, Oriol Nieto<sup>5</sup>, Justin Salamon<sup>1,3</sup>,  
Rachel Bittner<sup>1</sup>, Jon Forsyth<sup>1</sup>, and Juan P. Bello<sup>1</sup>

<sup>1</sup>Music and Audio Research Laboratory, New York University

<sup>2</sup>Center for Data Science, New York University

<sup>3</sup>Center for Urban Science and Progress, New York University

<sup>4</sup>MuseAmi, Inc.

<sup>5</sup>Pandora, Inc.

## ABSTRACT

This document describes the changes to the JSON Annotated Music Specification (JAMS) format and implementation between v0.1 and v0.2.

## 1. INTRODUCTION

The JSON Annotated Music Specification (JAMS) format was proposed by Humphrey *et al.* [3] as a mechanism to serialize structured annotations for musical content. Since the initial publication of the JAMS specification, we (the developers) have learned several lessons in building music information retrieval infrastructure on top of the existing framework. Consequently, we have revised the specification and implementation in various ways to better support a modern and extensible workflow. The purpose of this document is to explain the changes in JAMS following the first publication, describe their underlying motivation, and demonstrate how to effectively apply the current (v0.2.0) implementation.

Throughout this document, the previous specification of JAMS as described by Humphrey *et al.* [3] will be referred to as *JAMS-0.1*, while the current specification will be referred to as *JAMS-0.2*.

## 2. JAMS SPECIFICATION

In this section, we highlight the changes to the JAMS schema definition(s). Since these changes apply to

the file structure definition itself, they are independent of the software implementation used to parse or generate JAMS files. The software implementation of JAMS is available at <https://github.com/marl/jams>.

### 2.1 Unified observation types

In JAMS-0.1, there are four basic data types:

- *observation*,
- *event*,
- *range*, and
- *time series*.

The *observation* type is used to encode a fixed observed quantity, such as a chord label or semantic tag, as well as a quantitative measure of confidence in the value.<sup>1</sup> The latter three types define different ways of encoding the time index of an observation. *Event* is used for observations with no temporal duration (such as beats or onsets); *range* is used for observations that span a fixed portion of time (such as chords or segments); and *time series* is used to encode temporally continuous observations, such as melodic contours.

These three distinct views lead to efficient, compact representations, but can be difficult to work with in practice. Different tasks generally use different time index types, so the practitioner must both be aware of which index is used for any given task, and write code to handle it accordingly. Moreover, it becomes non-trivial to temporally align annotations across different tasks, since they must first be mapped into a common representation.

JAMS-0.2 simplifies this by reducing all observation types to a single format: regardless of task, each *observation* consists of a 4-tuple (*time*, *duration*, *value*, *confidence*). The *time* and *duration*

<sup>1</sup> A secondary value field is also provided, but we ignore it here for expository purposes.

\*Please direct correspondence to [brian.mcfee@nyu.edu](mailto:brian.mcfee@nyu.edu)



fields are constrained by the schema to be non-negative numbers.<sup>2</sup> By default, this simulates the *range* type of JAMS-0.1, but taking *duration*=0 recovers the *event* type as well, with a small amount of redundancy.

The *time series* type of JAMS-0.1 can be viewed as an efficiently coded, dense sequence of *range* observations with implicit durations. Recall that in JAMS-0.1, each *Annotation* object contains a list of observations in its *data* field. For high-frequency observations — such as melodic contours, sampled at 10Hz or greater — encoding a 4-tuple for each sample would be inefficient, due to redundantly listing the keys *time*, *value*, *duration*, *confidence*. JAMS-0.2 circumvents this by allowing a distinction between *sparse* and *dense* observation lists. Note that having standardized the observation format, the *Annotation*'s data field can be interpreted as an  $n \times 4$  table, which may be encoded in either a row-major (*sparse*) or column-major (*dense*) format. While the column-major format is generally more spatially efficient, the row-major format is more human-legible, and for most tasks, the difference in efficiency is negligible.

Standardizing the observation format both simplifies upstream code to interact with JAMS objects, and generalizes the previous definitions. (For instance, time series now have explicit durations/sampling rates, and gaps in observations are now permitted.) The one thing that we lose in this process is the notion of time-independent annotations, such as *tag*, *genre*, and *mood* in JAMS-0.1. This is because all observations in JAMS-0.2 are required to have a time and (potentially 0) duration. However, we argue that this is an advantage for three reasons. First, it is possible to have full-track observations by setting *time*=0 and *duration* to the full track duration, so no functionality is lost. Second, in reality, every observation type may vary over time, so the schema should support this explicitly. Finally, it forces the annotator to be explicit about the valid timing of an observation, and facilitates partial annotation (see section 4).

## 2.2 Task- vs. Annotation-major layout

As illustrated in fig. 1, JAMS-0.1 took a *task-major* approach to structuring annotations. A collection of supported tasks was defined within the JAMS-0.1 schema, such as *tag*, *genre*, *chord*, *key*, *melody*, *etc.* Annotations of a particular task would then be accessed by indexing the array of annotations corresponding to that task, e.g.:

<sup>2</sup> The *value* and *confidence* fields are left unconstrained at this point, but are defined subsequently depending on the *namespace* as defined in section 2.2.

```
{
  "file_metadata": {
    "artist": "The Beatles",
    "duration": { "value": 260.627 },
    "jams_version": "0.0.1",
    "title": "01_-_Come_Together"
  },
  "beat": [
    {
      "annotation_metadata": { ... },
      "data": [
        {
          "label": { "value": 1 }, "time": { "value": 1.196 }
        },
        {
          "label": { "value": 2 }, "time": { "value": 1.904 }
        },
        ...
      ]
    }
  ],
  "chord": [
    {
      "annotation_metadata": { ... },
      "data": [
        {
          "start": { "value": 0.0 },
          "end": { "value": 1.172266 },
          "label": { "value": "N" }
        },
        {
          "start": { "value": 1.172266 },
          "end": { "value": 12.585238 },
          "label": { "value": "D:min" }
        },
        ...
      ]
    }
  ],
  "key": [
    {
      "annotation_metadata": { ... },
      "data": [
        {
          "start": { "value": 0.0 },
          "end": { "value": 1.01 },
          "label": { "value": "Silence" }
        },
        {
          "start": { "value": 1.01 },
          "end": { "value": 70.673 },
          "label": { "value": "Key\\tD:minor" }
        },
        ...
      ]
    }
  ],
  "segment": [
    {
      "annotation_metadata": { ... },
      "data": [
        {
          "start": { "value": 0.0 },
          "end": { "value": 1.0 },
          "label": { "value": "silence" }
        },
        {
          "start": { "value": 1.0 },
          "end": { "value": 35.861 },
          "label": { "value": "intro/verse" }
        },
        ...
      ]
    }
  ]
}
```

**Figure 1.** Example of a JAMS-0.1 object. The example has been abridged to highlight schematic changes in JAMS-0.2.

```

{
  "file_metadata": {
    "artist": "The Beatles",
    "duration": 260.627,
    "identifiers": {},
    "jams_version": "0.2.0",
    "release": "",
    "title": "01_-_Come_Together"
  },
  "sandbox": {},
  "annotations": [
    {
      "namespace": "beat",
      "annotation_metadata": { ... },
      "sandbox": {},
      "data": [
        {
          "time": 1.196, "duration": 0.0,
          "value": 1.0, "confidence": 1.0
        },
        {
          "time": 1.904, "duration": 0.0,
          "value": 2.0, "confidence": 1.0
        },
        ...
      ]
    },
    {
      "namespace": "chord",
      "annotation_metadata": { ... },
      "sandbox": {},
      "data": [
        {
          "time": 0.0, "duration": 1.172266,
          "value": "N", "confidence": 1.0
        },
        {
          "time": 1.172266, "duration": 11.412972,
          "value": "D:min", "confidence": 1.0
        },
        ...
      ]
    },
    {
      "namespace": "key_mode",
      "annotation_metadata": { ... },
      "sandbox": {},
      "data": [
        {
          "time": 1.01, "duration": 69.663,
          "value": "D:minor", "confidence": 1.0
        },
        ...
      ]
    },
    {
      "namespace": "segment_open",
      "annotation_metadata": { ... },
      "sandbox": {},
      "data": [
        {
          "time": 0.0, "duration": 1.0,
          "value": "silence", "confidence": 1.0
        },
        {
          "time": 1.0, "duration": 34.861,
          "value": "intro/verse", "confidence": 1.0
        },
        ...
      ]
    }
  ]
}

```

**Figure 2.** The contents of fig. 1 in JAMS-0.2 format.

This structure is conceptually simple and easy to work with, but it poses several practical limitations. First, it requires that all tasks be specified *a priori* within the JAMS schema. Consequently, each time a new task is introduced in the future, the core JAMS schema must be modified to accommodate it. This is clearly undesirable, as it could lead to fragmentation of the JAMS specification if (when) different groups decide to extend the task definitions in one direction or another.

Second, it provides no means of distinguishing between different variations of a task. As a simple example, take the case of *tags*. Different data sets are annotated using different vocabularies, which may be closed (e.g., GTZAN [?] or CAL500 [?]) or open (e.g., last.fm). This implies that the validity of a tag annotation depends upon the target vocabulary, which is not explicitly coded within the schema. (Indeed, an exhaustive coding of *all* tag vocabularies within a fixed schema is impossible.) As a more nuanced example, *chord* annotations can be drawn from different vocabularies (e.g., including or suppressing extensions), or even radically different annotation styles, such as the pop-style annotations of Isophonics [1] compared to the roman numeral annotations of the Rock Corpus [2]. In these cases, it is hardly sensible to group these variations together under a single task, since their annotations are not directly comparable.

To resolve these issues, JAMS-0.2 adopts an annotation-major (rather than task-major) structure. Instead of grouping annotations by task at the top-level, a JAMS-0.2 object contains a single list of *Annotations*. Figure 2 illustrates how the contents of the example in fig. 1 are encoded in JAMS-0.2.

The annotation-major structure allows for the same core schema to be retained as new tasks are introduced, since there is no explicit dependence on the task definitions. However, since all annotations are collected in a single, anonymous data structure, we will need a new way to distinguish between annotations for different tasks. This leads us to the task *namespace* abstraction.

### 2.3 Task namespaces

Each annotation object declares its task through a string-valued field called *namespace*. A *namespace* in JAMS-0.2 is simply a partial schema declaration which defines the following properties:

- an identifier, e.g., “beat” or “tag\_cal500”;
- schema declarations for the *value* and *confidence* fields;

- whether data should be encoded in *dense* or *sparse* form; and
- a brief plain-text description of the task.

The identifier is included within *Annotation* objects to specify which namespace they should be validated against. The schema declarations for *value* and *confidence* are both optional, but can be used to impose constraints on the permissible contents of an observation.<sup>3</sup>

This abstraction allows for both a more general set of supported tasks, in that there may be many *tag* namespaces, and more precise task definitions for each specific namespace. For instance, a valid *tag\_cal500* annotation must have a value drawn from the correct vocabulary, whereas a *tag\_open* annotation may contain any string in its *value* field; however, in both cases, the value must be a string, and this constraint was not possible in the JAMS-0.1 schema.

With the namespace abstraction, it is possible for observations to have arbitrarily structured value and confidence fields. Figure 3 provides a complete example namespace definition for *mood\_thayer* annotations, in which each observed value is an ordered pair of numbers encoding *valence* and *arousal* in the Thayer mood model [?].

For convenience, namespaces are grouped into high-level task categories by their identifiers. We stress that this grouping is merely cosmetic, and there is no strict underlying hierarchy of tasks. Table 1 lists the namespaces supported in JAMS v0.2.0.

Finally, namespaces are defined externally to the core schema, and new namespaces can be imported dynamically with no modifications to the JAMS implementation itself. This makes it possible to develop and share custom annotation specifications.

### 3. IMPLEMENTATION

To support the schema changes described in the previous section, the JAMS python implementation was dramatically revised in 0.2.

#### 3.1 Search

As described in 2.3, all annotation objects are now collected in a single list at the top level. This presents a difficulty for users: in the presence of multiple annotations spanning various tasks, how can one efficiently select a specific annotation? A common use-case might be selecting only the annotations

<sup>3</sup> The term *namespace* was chosen to connote that the *value* and *confidence* fields keep the same *name* in different tasks, but their interpretation varies according to *namespace*. This is analogous to the notion of namespace encapsulations in software engineering.

**Table 1.** Namespaces supported in JAMS v0.2.0.

Task group	Namespace
Beat	beat
	beat_position
Chord	chord
	chord_harte
	chord_roman
Key	key_mode
Lyrics	lyrics
Mood	mood_thayer
Onset	onset
Pattern	pattern_jku
Pitch	pitch_class
	pitch_hz
	pitch_midi
Segment	segment_open
	segment_salami_function
	segment_salami_upper
	segment_salami_lower
	segment_salami_tut
Tag	tag_cal10k
	tag_cal500
	tag_gtzan
	tag_medleydb_instruments
	tag_open
Tempo	tempo

mood\_thayer.json

```
{ "mood_thayer":
  {
    "value": {
      "type": "array",
      "items": {"type": "number"},
      "minItems": 2,
      "maxItems": 2
    },
    "dense": false,
    "description": "Time-varying
      emotional measurements as
      ordered pairs of (valence,
      arousal)"
  }
}
```

**Figure 3.** An example namespace definition file for *mood\_thayer*. Each observed value is an array of exactly two numbers, and observations are packed sparsely. No constraints are placed upon the *confidence* field.

matching a given task, *e.g.*, finding all beat annotations. More advanced examples are also possible, such as filtering by annotator, curator, or arbitrary sandbox entries.

To address this problem, we introduced the `JAMS.search()` method. This method acts as a filter over the list of annotations, and performs a recursive descent over the object hierarchy to find matching fields. For example, to find all the beat annotations, one simply needs to execute the following:

```
>>> jam = jams.load('filename.jams')
>>> anns = jam.search(namespace='beat')
```

The resulting `anns` object is a (possibly empty) collection of annotation objects matching the query. Multiple simultaneous query conditions are possible, and are interpreted disjunctively. The following example finds all annotations that have either beat as a namespace, or *isophonics* as a corpus:

```
>>> anns = jam.search(namespace='beat',
...                   corpus='isophonics')
```

In fact, search results are provided as a list-like object that again implements `search()`, so that conjunctions are supported by successive queries. To find annotations that match both the namespace and corpus fields, one could execute the following:

```
>>> anns = jam.search(namespace='beat')\
...                 .search(corpus='isophonics')
```

### 3.2 Data frames

The JAMS-0.1 implementation provided a direct object mapping between the JSON representation and its instantiation in Python. Consequently, the code to access the elements of a JAMS object is a simple traversal of the data structure, *e.g.*:

```
>>> ann = jam.beat[0]
>>> first_beat = ann.data[0].time
```

One advantage of this approach is that it would yield nearly identical code in any other language (such as JavaScript).

However, in practice, working with data in this format can be somewhat cumbersome. For example, evaluation scripts (such as `mir_eval` [5]) typically expect data in an array format. This can be accomplished with some minor contortion by iterating over the observations:

```
>>> all_beats = [o.time for o in ann.data]
```

More generally, certain common operations like thresholding or label manipulation are simply easier with natively array-oriented representations.

Since JAMS-0.2 encodes all annotation data in a table-friendly format, we instead opted to provide a table-interface in the Python implementation. This is accomplished by translating annotation data fields into a pandas data frame object [4] upon construction. The choice of using a data frame (rather than a numpy array) carries several advantages:

- labeled fields;
- heterogeneous data types;
- advanced query operations (join, merge, etc);
- missing value support; and
- temporal indexing.

Accessing individual observations in JAMS-0.2 looks nearly identical to JAMS-0.1 (once an annotation object has been selected):

```
>>> ann = jam.search(namespace='beat')[0]
>>> first_beat = ann.data.time[0]
```

*# This would also work:*

```
>>> first_beat = ann.data.loc[0].time
```

```
>>> all_beats = ann.data.time
```

However, the `ann.data` object itself can now be operated upon as an array or data frame.

JAMS data frame objects interpret all time and duration fields as `timedelta` types. In addition to facilitating semantic validation — time

and duration fields are enforced to contain non-negative values — this enables pandas to efficiently align and resample multiple annotations with non-uniform timings. Upon serialization, these values are converted back to raw floating point representations in units of seconds.

### 3.3 Dynamic namespaces

As described in section 2, JAMS-0.2 adopts an extensible task framework. In the Python implementation, this is supported by dynamic construction of the full schema at run-time. Each namespace is defined in a self-contained file (e.g., fig. 3), and when the JAMS library is imported, it searches for all namespace definitions within the distribution, adding each to a dictionary of available namespaces.

This dynamic namespace implementation carries two benefits. First, it decouples the namespace definitions from the core schema, allowing namespaces to evolve over time without changing the core structure. Second, it allows practitioners to define and import namespaces for their own tasks without modifying the JAMS library. Consequently, this should ameliorate the need to fork and modify the JAMS implementation, thus preventing fragmentation of the codebase.

New namespaces can be added at runtime by the following code fragment:

```
>>> import jams
>>> jams.schema.add_namespace(
...     '/path/to/my_namespace.json')
```

### 3.4 Validation

The JAMS-0.2 namespace framework also facilitates task-dependent data validation via JSON schema. This helps ensure that annotations are not only *syntactically* correct, but (at least partially) *semantically* correct. Whenever a JAMS object is serialized or deserialized, it is run through a schema validation which ensures that the data is well-formed, and each observation fits the specification of its containing namespace. We note that this was not possible in JAMS-0.1 because the relatively coarse observation types (e.g., tag) were too broad to support precise specification of allowable values.

Validation errors in JAMS-0.2 can be handled in either *strict* or *non-strict* mode. In strict mode, errors invoke an exception and interrupt the program. In non-strict mode, errors simply issue a warning and do not interrupt the program. Finally, because validation can be a relatively expensive operation, it can be bypassed entirely on load if the

practitioner is confident that the data has already been validated. These different validation modes are exemplified by the following code fragment:

```
# With strict validation (default)
>>> jam = jams.load('file.jams')

# With lax validation
>>> jam = jams.load('file.jams',
...                 strict=False)

# With no validation
>>> jam = jams.load('file.jams',
...                 validate=False)
```

### 3.5 mir\_eval integration

The `mir_eval` package provides reference implementations of common evaluation metrics for various tasks. Because `mir_eval` uses a variety of (well-defined) flat annotation formats for its input, JAMS-0.2 provides bindings which translate JAMS annotations into `mir_eval`-format, call the appropriate evaluation routine, and return the resulting dictionary of scores.

The evaluation bindings are contained in the `eval` submodule, which provides a simple, consistent interface to evaluators, e.g.:

```
# Get the first beat annotation from the
# reference and estimation objects
>>> ann_r = ref.search(namespace='beat')[0]
>>> ann_e = est.search(namespace='beat')[0]

# Call the evaluator
>>> scores = jams.eval.beat(ann_r, ann_e)
```

All evaluation bindings accept two annotations (reference and estimate), and additional keyword arguments which can be passed through to configure the evaluator. Each evaluation binding also verifies that the input annotations belong to the correct namespace(s) and pass validation.

### 3.6 Serialization IO

JAMS-0.1 was developed for serializing data to disk and back, and thus assumed that all serialization targets were filenames. JAMS-0.2 relaxes this assumption, and allows serialization to open file-like python objects as well. This is done transparently by simply passing a file-like object to the input-output routines instead of a filename:

```
>>> with open('input.jams') as fd:
...     jam = jams.load(fd)
```

This functionality can be useful in the context of a web server, where the destination is not a file on disk but an open HTTP connection to a browser.

Moreover, the JAMS IO routines now support compressed file targets by specifying the `.jamz` (i.e., JAMs Zipped) extension to the file name:

```
>>> jam = jams.load('input.jamz')

# This would also work
>>> jam = jams.load('input.jams.gz',
...                 fmt='jamz')
```

The compressed JAMS format can significantly increase storage efficiency at the cost of direct human legibility. In many situations, this trade-off is acceptable.

#### 4. FUTURE DIRECTIONS

In this section, we describe the current work in progress and speculative features to come in future revisions.

##### 4.1 Namespace conversions

As shown in table 1, many of the existing namespaces are similar enough to share common representations and evaluation schemes, and can therefore be grouped into high-level categories. In some cases, it is even possible to construct explicit mappings between namespaces. This can be useful for simultaneously modeling or comparing data from different corpora.

As concrete examples, `chord_harte` is a strict subset of `chord`, and `tag_cal500` is a strict subset of `tag_open`. In these cases, the mapping is a trivial substitution of the *namespace* identifier in the annotation. A less trivial example can be found in the mapping between `pitch_hz` and `pitch_midi`, where the values must undergo a unit conversion. Finally, one may wish to convert a `chord_roman` annotation to `chord` format, which requires a substantial (and non-invertible) manipulation of the data.

Although complicated, implementing automatic namespace conversion — even if it is occasionally non-invertible — would be valuable for simplifying modeling and evaluation of tasks across different datasets.

##### 4.2 Local namespaces and unstructured data

JAMS-0.2 provides functionality for local extensions of the supported namespaces, but it can be tedious to add namespace definitions manually in each application. We therefore plan to introduce functionality to support a *local* namespace repository, in addition to the definitions which ship in the

**Table 2.** New namespaces planned for JAMS v0.2.1.

Task group	Namespace
Misc	blob vector
Segment	multi_segment

main distribution. This repository would be specified by an environment variable or configuration file, and reduce the amount of custom code needed to support local extensions to the namespaces.

In addition to expanded support for local modification, we plan to introduce three new namespaces in 0.2.1 as listed in table 2.

The `multi_segment` namespace is similar to the existing `segment` namespaces, except that it introduces an additional *level* field to the values which can be used to encode a multi-layer or hierarchical segmentations.

The `vector` namespace provides values which are arbitrary arrays of numbers. This can be useful for regression problems in which the annotation targets are vector-valued, such as collaborative filter prediction, or higher-dimensional extensions of the Thayer mood model. The `vector` namespace does not enforce that each observation’s array is of the same length, so great care must be taken in documenting annotations using this namespace.

The `blob` namespace can be used to store arbitrarily structured values which don’t otherwise fit in an existing schema. This namespace should be viewed as a last resort to storing within JAMS. Whenever possible, we recommend using the most specific namespace that characterizes the annotations of interest.

##### 4.3 Partial annotations

For a variety of practical reasons, annotations frequently do not span the entire duration of a track. Ideally, annotations should therefore define the time extent over which the annotation is valid. While JAMS-0.2 provided some functionality to encode this (via the *duration* fields or a *sandbox* entry) it was not standardized, and no provision exists to support partial annotations of zero-duration events.

Starting in JAMS-0.2.1, each annotation object will also contain optional time and duration field. By convention, if these fields are left null, then the annotation should be assumed to span the entire track.

## 5. REFERENCES

- [1] Reference annotations: The Beatles, 2009. <http://isophonics.net/content/reference-annotations-beatles>.
- [2] Trevor De Clercq and David Temperley. A corpus analysis of rock harmony. *Popular Music*, 30(01):47–70, 2011.
- [3] Eric J Humphrey, Justin Salamon, Oriol Nieto, Jon Forsyth, Rachel M Bittner, and Juan P Bello. JAMS: a JSON annotated music specification for reproducible MIR research. In *International Society for Music Information Retrieval Conference, ISMIR*, 2014.
- [4] Wes McKinney. Data structures for statistical computing in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010.
- [5] C. Raffel, B. McFee, E. Humphrey, J. Salamon, O. Nieto, D. Liang, and D.P.W. Ellis. mir\_eval: a transparent implementation of common MIR metrics. In *Proceedings of the 15th International Society for Music Information Retrieval Conference, ISMIR*, 2014.