ChatGPT

Apple Embedding Atlas – Comprehensive Documentation

Overview

Embedding Atlas is an open-source tool by Apple for interactive visualization of large-scale embeddings [1]. It allows you to **visualize**, **cross-filter**, and **search** through high-dimensional data (embeddings) alongside rich metadata in a responsive, browser-based interface [2] [1]. All computation is performed locally (either in-browser or on your machine), ensuring your data never leaves your environment [2]. Key features include:

- **Interactive Exploration:** Visualize and navigate through millions of data points with smooth zooming and panning [3]. Embedding Atlas automatically clusters data and can draw density contours to highlight dense regions and outliers [4].
- **Cross-Filtering & Linked Views:** Multiple coordinated views (embedding scatter plot, data table, and charts) allow you to filter and brush data across different metadata fields simultaneously [5].
- **Search & Nearest Neighbors:** Built-in full-text search and vector similarity search help find points similar to a query or to each other in real time [6].
- **Local Embedding Computation:** Supports computing embeddings from raw text or images using default or custom models (via Hugging Face Transformers). By default it uses **SentenceTransformers** (`all-MiniLM-L6-v2` for text) and **ViT** (`google/vit-base-patch16-384` for images) to generate embeddings [7]. Dimensionality reduction to 2D is done with **UMAP** for visualization [8]. All of this can happen in-browser via WebAssembly or on the local Python backend, ensuring privacy.
- **Multiple Integration Options:** Embedding Atlas is available as both a **Python package** (with a CLI tool, Jupyter widget, and Streamlit component) and as a **TypeScript/JavaScript library** (an npm package exposing reusable UI components) [9]. This makes it easy to integrate into Python workflows or front-end applications.

Under the hood, Embedding Atlas uses a combination of Python and Web technologies. The Python backend (packaged as `embedding-atlas` on PyPI) handles data loading, optional embedding generation (via PyTorch/SentenceTransformers), and can launch a local web server for the interface [7] [10]. The front-end (packaged as `embedding-atlas` on npm) is a WebGL/WebGPU-based visualization that can also perform heavy computations (UMAP, approximate nearest neighbors, clustering) via WebAssembly for scalability [11] [12]. A DuckDB database is used under the hood to manage the dataset and enable fast filtering and SQL queries across the data [13] [14]. The front-end and back-end communicate through a coordinator (from the Mosaic library) to keep the views in sync.

Installation and Setup

Embedding Atlas is distributed separately for Python and Node/TypeScript environments:

- **Python Installation:** Ensure you have Python 3.7+ and pip available. Install the PyPI package by running:

```
pip install embedding-atlas
```

This will install the `embedding_atlas` Python package, which includes the CLI tool, Jupyter widget, and Streamlit component [15] [16]. On Windows, you may use it via WSL if you encounter issues. If you plan to use a GPU for embedding computation, ensure you have a compatible PyTorch installed (with CUDA support) [17]. *Tip:* The developers recommend the ***uv*** package manager for one-command installation (e.g. `uvx embedding-atlas`) to avoid dependency issues [18], but this is optional.

- **Node/TypeScript Installation:** Ensure Node.js (>= 16) and npm are installed. Install the npm package into your project:

```
npm install embedding-atlas
```

This will fetch the Embedding Atlas library for use in web applications. The package bundles all necessary components (the embedding viewer, table, algorithms, etc.) [19]. For development or customization, note that the source is a monorepo with multiple sub-packages (components, viewer app, WASM algorithms, etc.) consolidated into this single npm package [20]. No additional build steps are required for basic usage, though a modern browser environment with WebGL2/WebGPU support is needed to fully leverage it [21].

Using Embedding Atlas in Python

The Python package provides three primary interfaces to use Embedding Atlas: a **Command-Line Interface (CLI)**, a **Jupyter Notebook widget**, and a **Streamlit UI component**. All three ultimately display the same interactive UI in a browser or embedded frame.

1. Command-Line Utility (CLI)

Once installed, the `embedding-atlas` command-line tool allows quick visualization of local or Hugging Face Hub datasets without writing any code [22] [23]. This is useful for exploring datasets directly from the terminal.

Launching the UI: Use the command `embedding-atlas [OPTIONS] <INPUT>` to start the tool [24]. The `<INPUT>` can be: - A path to a local dataset file in **Parquet** format (ideal for larger datasets) [25]. The Parquet should represent a table where each row is a data point, columns include metadata and possibly embeddings. - A **Hugging Face dataset identifier** (in the form `user/dataset`) to load a dataset from the Hugging Face Hub [26]. You can optionally specify dataset splits with `--split` if needed (e.g. `--split train --split test`) [27] [28]. Authentication for private datasets can be handled via `huggingface-hub` (e.g. `huggingface-cli login`) [29].

For example, to visualize the IMDB reviews dataset from Hugging Face:

```
embedding-atlas stanfordnlp/imdb --text "text" --sample 5000
```

This will download the dataset, use the `"text"` column for embedding, sample 5000 rows for speed, and launch the UI [30] [31].

After processing, the CLI prints a local URL (default **http://localhost:5055/**) – open this in your web browser to interact with the Embedding Atlas UI [10].

Embedding Computation: By default, if your dataset has a text column or image column, Embedding Atlas will automatically compute embedding vectors for those using a default model: *all-MiniLM-L6-v2* for text, or *google/vit-base-patch16-384* for images [7]. If the data already contains embedding vectors or you want to use a custom model, you have options: - **Specify Column Type:** Use `--text <col>` or `--image <col>` to explicitly choose the text or image column to embed [8] [32]. Use `--vector <col>` if you already have a column with pre-computed embedding vectors [33]. (If you provide none of these flags, the tool will try to guess the text column). - **Custom Models:** Use `--model <huggingface-model-name>` to specify a different embedding model [34]. Any model from Hugging Face Hub compatible with SentenceTransformers (for text) or image feature extraction can be used. If the model requires custom code (not a standard transformer), add `--trust-remote-code` to allow downloading and executing its code [35]. - **Batch Size:** Use `--batch-size <N>` to adjust the batch size for embedding computation (defaults are 32 for text, 16 for images) [36]. Larger batches use more memory but can be faster.

Projection (Dimensionality Reduction): Unless disabled, the tool will reduce embeddings to 2D for visualization using UMAP [37]. You can fine-tune UMAP with: - `--umap-n-neighbors <int>`: number of neighbors for UMAP (default 15) [38]. - `--umap-min-dist <float>`: minimum distance parameter (controls clustering tightness) [39]. - `--umap-metric <metric>`: distance metric (default "cosine") [40]. - `--umap-random-state <int>`: random seed for UMAP (set for reproducibility) [41].

You can also skip computing projections by adding `--disable-projection` (especially if you only want to use the table/charts, or have precomputed coordinates) [42]. If you disable projection and do not supply precomputed coordinates, the embedding scatter plot will not be shown [42].

Using Precomputed Data: For full reproducibility or if you already have results, you may supply precomputed 2D coordinates and neighbors: - `--x <col>` and `--y <col>`: column names for precomputed X and Y coordinates of each point [43]. If provided, Embedding Atlas will use these for the embedding view instead of computing UMAP. - `--neighbors <col>`: a column with precomputed nearest neighbor info [44]. Each entry should be a JSON/dict with format `{"ids": [id1,id2,...], "distances": [d1,d2,...]}` – where *ids* are the indices (or row identifiers) of the nearest neighbors and *distances* are the corresponding distances [44]. If this is given, selecting a point in the UI will show its neighbors from this list [45]. (If not provided, the UI can perform approximate **vector search** on the fly via its own index, as described later.)

Launching Options: Additional CLI options: - `--sample <N>`: randomly sample N rows from the dataset (useful for very large datasets to reduce load) [46]. - `--host` and `--port`: specify the host (default `localhost`) and port (default `5055`) for the web server [47]. If the port is busy, use `--auto-port` to find an open port automatically [48]. - `--duckdb <mode>`: choose how the DuckDB database is used [14]. Options are `wasm` (run queries in the browser via DuckDB-WASM), `server` (run DuckDB queries on the Python server backend), or a `ws://<host>:<port>` URI to use a DuckDB server. By default, the CLI uses DuckDB in **browser (WASM)** mode for a fully local experience [14]. - `--stop-words <file>`: path to a file (CSV/Parquet) containing a column `word` listing stop words to omit in automatic label generation [49] (e.g. common words to ignore when picking cluster keywords). - `--labels <file>`: path to a file (CSV/Parquet) containing custom labels to display on the embedding plot [50]. The file should have columns `x`, `y`, `text` (and optionally `level` and `priority`) for each label [50]. Labels allow annotating clusters; if not provided, automatic labeling can be enabled (see widget options below). - `--export-application <zip-path>`: instead of launching a server, this will **export a standalone web application** (HTML/JS and data) as a ZIP file [51]. You can host or share this zip; users can open `index.html` inside it to view the embedding without any Python runtime. - `--version`: shows the version and exits [52].

After running the CLI with your data and options, open the provided local URL in a browser. You will see the full Embedding Atlas interface: an embedding scatter plot (if embeddings/projection provided), a data table, and possibly histogram/bar chart panels for metadata exploration. You can search text in the top bar, select points (drag a lasso or click), and filter via brushing on charts. When done, press **Ctrl+C** in the terminal to stop the server.

Reproducibility Tip: For consistent results over time, it's recommended to precompute your embeddings and UMAP projections and pass them to the tool (using `--vector`, `--x`, `--y` flags) [53]. This avoids run-to-run differences since the default models might change in future versions, and UMAP has some randomness [53]. The Python API provides utilities for this – see `compute_text_projection` below.

2. Jupyter Notebook Widget

Embedding Atlas provides a Jupyter notebook widget for interactive use in Python notebooks, including JupyterLab, Google Colab, VSCode, and other environments that support **AnyWidget** [54]. This lets you embed the full Embedding Atlas UI inline in a notebook cell.

Installation: Ensure the package is installed (`pip install embedding-atlas`) and that you have a notebook environment that supports ipywidgets/anywidget [54]. (In JupyterLab, the widget should work out-of-the-box; in classic Jupyter Notebook you may need to enable widgets extensions.)

Basic Usage: Use the `EmbeddingAtlasWidget` class from `embedding_atlas.widget` to create the widget. Pass in a pandas **DataFrame** (or PyArrow Table) containing your data:

```python
from embedding_atlas.widget import EmbeddingAtlasWidget

# df is a pandas DataFrame containing your data (columns can include text, etc.)
widget = EmbeddingAtlasWidget(df)
widget  # display the widget in the notebook
```

If you provide the DataFrame as-is (without specifying projection), the widget will start in "table + charts" mode only (since it has no 2D coordinates) [55] [56]. You can still use the table view and histograms on metadata, and even perform full-text search if you set the `text` parameter (to indicate which column contains text for searching).

Typically, you will want to include an embedding scatter plot. To do so, you need to compute embedding vectors and 2D projections beforehand (to avoid long computation blocking the notebook UI). The `embedding_atlas.projection` module provides convenience functions for this. For example, for a text column:

```
from embedding_atlas.projection import compute_text_projection

# Compute embeddings and 2D projection in-place in the DataFrame
compute_text_projection(df, text="description",
                        x="projection_x", y="projection_y",
neighbors="neighbors")
```

The above will: 1. Use a text embedding model to generate embeddings for the `"description"` column, 2. Project them to 2D with UMAP (adding columns `"projection_x"`, `"projection_y"`), 3. Compute nearest neighbors for each point (adding a `"neighbors"` column in the required format) [57].

After this, create the widget with those column names:

```
widget = EmbeddingAtlasWidget(df, text="description",
                              x="projection_x", y="projection_y",
neighbors="neighbors")
widget
```

This will display the full Embedding Atlas UI inside the notebook, with the points plotted according to your precomputed projection [58]. You can interact with it (zoom, select points, search, etc.) just as in the standalone web UI.

Selecting Data: One powerful feature of the widget is the ability to get the current selection back into Python. After a user interacts and selects a subset of points (for example, via lasso or filtering), you can call:

```
selected_df = widget.selection()
```

This returns a pandas DataFrame of the currently selected points (or `None` if no selection) [56]. This allows for iterative analysis – you could, for instance, run further computations on the selected subset or plot them in another way.

Widget Parameters: The `EmbeddingAtlasWidget` constructor accepts several optional arguments to configure its behavior [59] [60]:

- **data_frame:** The pandas DataFrame or pyarrow Table to visualize. (This is the first positional argument.)
- **row_id:** Optional string for the column to use as a unique row identifier [61]. If not provided, an index column will be added automatically. Use this if your data has a natural ID column (like an index or primary key).
- **x, y:** Column names for the 2D projection coordinates [62]. Provide these if you have precomputed embedding coordinates (e.g., via UMAP). If not provided, the embedding scatter plot will be disabled (unless `enable_projection` was True and data contains text/image to compute, but in the widget context you typically precompute to avoid blocking).
- **text:** Column name containing textual data [60]. If specified, this text will be used for tooltips on hover and for the search feature (full-text search queries this column) [63]. It also enables automatic label generation (if `labels` not provided).
- **neighbors:** Column name for precomputed nearest neighbors in the format described earlier (IDs and distances) [64]. Providing this enables the **Nearest Neighbors** panel in the UI for any selected point [65].
- **labels:** Controls cluster label overlays on the embedding view [66]. You can set `labels="automatic"` to let Embedding Atlas automatically generate descriptive labels for clusters (using density-based clustering + TF-IDF on the text data) [67]. Set `labels="disabled"` to turn off all labels. Or pass a **list of label objects** (each an object with `x`, `y`, `text`, and optional `level` and `priority` fields) to use custom labels [66].
- **stop_words:** If using automatic labels, you can pass a list of stop words to ignore (or leave None to use a default English stop word list) [68].
- **point_size:** Override the point size used in the scatter plot [69]. By default, point size is auto-computed based on density (so that dense clusters render points smaller).
- **show_table, show_charts, show_embedding:** Booleans (all default True) to control which views are visible when the widget is first rendered [70]. For example, you might set `show_embedding=False` if you only want to show the table and charts initially, or hide charts etc. Users can still toggle these views from the UI controls later.
- **connection:** (Advanced) A `DuckDBPyConnection` object [71]. By default the widget uses an in-memory DuckDB connection (`duckdb.connect()`) to hold the data. If you want to supply your own connection (for example, to an existing DuckDB database), you can pass it here. This is useful to share the database between multiple widgets or to persist queries.

Usage Note: The EmbeddingAtlasWidget uses the **AnyWidget** framework, which means the front-end UI is implemented in JavaScript and rendered in an iframe inside the notebook. Ensure your environment supports iframes and you have internet access to load anywidget (or have it installed) if needed. In offline or air-gapped environments, consult anywidget docs to serve the widget assets locally [54].

3. Streamlit Component

For users of **Streamlit**, Embedding Atlas offers a convenient component to embed the visualization in a Streamlit app. This allows you to create an interactive dashboard with Embedding Atlas alongside other Streamlit widgets.

Installation: Install via pip as usual (`pip install embedding-atlas`). Ensure you have Streamlit installed and running.

Usage: Use the `embedding_atlas` function from `embedding_atlas.streamlit` to render the component in your app. This function behaves similarly to the Jupyter widget but returns a value (the selection predicate) instead of a DataFrame directly. Example:

```python
import streamlit as st
from embedding_atlas.streamlit import embedding_atlas
from embedding_atlas.projection import compute_text_projection

# Assume df is a pandas DataFrame loaded with data
compute_text_projection(df, text="description",
                        x="projection_x", y="projection_y",
neighbors="neighbors")

# Embed the Embedding Atlas UI in the Streamlit app
selection_dict = embedding_atlas(df, text="description",
                                 x="projection_x", y="projection_y",
neighbors="neighbors",
                                 show_table=True)
```

This will render the interactive Embedding Atlas UI in the Streamlit app. The return value `selection_dict` is a Python dictionary containing at least one key: `"predicate"` [72] [73]. The predicate is a SQL WHERE clause (string) that represents the current selection/filter in the UI. You can use this predicate to filter your DataFrame using DuckDB or pandas. For example:

```python
import duckdb
predicate = selection_dict.get("predicate")
if predicate:
    # Use DuckDB to query the dataframe with the predicate
    result_table = duckdb.query_df(df, "df", f"SELECT * FROM df WHERE
{predicate}").to_df()
    st.dataframe(result_table)  # display the filtered table
```

In the predicate string, column names and values are formatted for SQL. This approach allows you to retrieve the selected subset of data after the user interacts, and then use it in other Streamlit elements (e.g., show detailed info, charts, etc.) [74] [75].

Just like the widget, you can call `embedding_atlas(df)` **without** specifying `x`/`y` to run in table-and-charts only mode [76] (useful if you only want to let users filter by metadata and search, without a scatter plot).

Function Arguments: `embedding_atlas(data_frame, **options)` accepts the following parameters (very similar to the widget) [77] [78]:

- **data_frame:** the DataFrame to visualize (required).
- **x, y:** names of columns for 2D projection coordinates [79].
- **text:** name of the text column (for tooltips/search) [80].
- **neighbors:** name of column with precomputed neighbors (for similarity search) [78].
- **labels:** either `"automatic"`, `"disabled"`, or a list of label objects to use for cluster labeling on the plot [81] (same semantics as widget).
- **stop_words:** list of stop words for automatic labeling [82].
- **point_size:** override point display size [83].
- **show_table, show_charts, show_embedding:** booleans to control which sub-views are shown initially [84].
- **key:** a unique string key for the Streamlit component instance (optional, used if you have multiple embedding_atlas components in one app to maintain state separately) [85].

Return Value: A `dict` with at least `{"predicate": <SQL WHERE clause or None>}` representing the selection [86]. (Future versions might include additional keys in this dict for more complex interactions.)

Note: The first time you use the component, Streamlit may print a message about a new custom component being installed or ask permission (since it's loading the frontend code). Allow it to proceed. Also, ensure the `embedding_atlas` call is within your `streamlit_app.py` and not inside an st.cache, as it needs to run live to render the UI.

Using Embedding Atlas in TypeScript / JavaScript

The Embedding Atlas npm package allows front-end developers to integrate the tool's visualization components into web applications. This is ideal if you want to build a custom UI or add embedding visualization to an existing web app. The library provides React and Svelte component wrappers for convenience, as well as plain JavaScript/TypeScript APIs.

At a high level, the Embedding Atlas front-end consists of a **coordinated set of components**: - **EmbeddingAtlas**: a high-level composite component that includes the full UI (embedding view, table, charts, search bar, etc.) – essentially the entire tool interface as seen on the demo site [87]. - **EmbeddingViewMosaic**: the 2D embedding scatter plot view that connects to a data source via a **Mosaic** coordinator (enables linking with other views) [88]. - **Table**: a tabular view for the dataset (with support for sorting, resizing columns, etc.) [89]. - **EmbeddingView**: a lower-level embedding scatter plot view that can be used standalone with raw coordinate arrays (does not require Mosaic/DuckDB) [90]. - **Algorithms**: utility functions (UMAP, KNN, clustering) provided for computational needs in the browser [11].

Typically, if you want an out-of-the-box solution, you would use **EmbeddingAtlas** component directly, which includes the other views internally. For custom integrations, you might use **EmbeddingViewMosaic** and **Table** separately.

1. Data Setup and Coordinator

Embedding Atlas components rely on a data source that is managed via **Mosaic** (a coordination layer for data visualization components). Under the hood, Mosaic uses **DuckDB** to store and query data, whether in-browser (DuckDB-WASM) or via a server. When using Embedding Atlas in a web app, you need to load your dataset into a DuckDB table and set up a Mosaic **Coordinator** to link the components.

Loading Data: The easiest way to get data in is to use the same Parquet or Arrow table that you would use in Python. For example, you can host a Parquet file and fetch it in the browser, then use DuckDB-WASM to query it. Alternatively, if your web app has a back-end, you can have an API that supplies data. The details of data loading are beyond this documentation, but the Embedding Atlas components assume that the data is accessible via a DuckDB SQL query interface.

Coordinator: Mosaic provides a global coordinator (via `coordinator()` function) that synchronizes selection and filtering state across components. In many cases, you can simply use the default coordinator. If you have multiple independent embeddings on one page, you might use multiple coordinators (not common).

For example, if using Mosaic's default coordinator:

```
import { coordinator } from "mosaic-framework";  // hypothetical import, Mosaic
might be integrated already
const coord = coordinator();  // get the default coordinator
```

(Note: The actual Mosaic API is referenced in Embedding Atlas docs as `idl.uw.edu` *project. The Embedding Atlas package might re-export a* `Coordinator` *type or you import from a Mosaic library. Check the repository for Mosaic integration details.)*

2. Using the EmbeddingAtlas Component (Full UI)

The `EmbeddingAtlas` component provided by the library renders the complete UI and manages the internal state. You simply need to pass it the data source (table name and relevant column names) and a coordinator.

Installation: Already covered (via `npm install embedding-atlas`). Ensure you have a bundler or environment that supports importing ES modules.

React Usage: If you are using React:

```
import { EmbeddingAtlas } from "embedding-atlas/react";

// Inside a React component:
<EmbeddingAtlas
  coordinator={coord}  // a Coordinator instance (required)
  data={{
    table: "my_data_table",
    id: "id_column",
    projection: { x: "x_column", y: "y_column" },  // optional, if embedding
coords available
    text: "text_column"  // optional, for tooltips/search
  }}
  colorScheme="light"
  embeddingViewConfig={{ pointSize: 2.5 }}
  onStateChange={(state) => console.log("New state:", state)}
/>
```

In the above: - `coordinator` is the Mosaic coordinator instance to use (required) [91] . - The `data` prop specifies the data source: you must give at least a **table** name (the DuckDB table where your data resides) and an **id** column name (unique row identifier) [92] [93] . If you have projection coordinates, provide them as `projection: {x: "...", y: "..."}`; if not provided and no projection is computed, the embedding scatterplot will be inactive [94] . If you have a text column, provide it as `text: "..."` to enable tooltip content and text search [63] . You can also specify `neighbors: "neighbors_col"` in `data` if you have a precomputed neighbors column [95] . - `colorScheme` can be "light" or "dark" to set the UI theme (or `null` to auto) [96] . - `embeddingViewConfig` can pass configuration to the embedding plot. For example, you can set `pointSize` here (similar to CLI `--point-size`) [97] [98] , or toggle density vs point mode, etc. (See **EmbeddingView Config** below for details.) - `onStateChange` is a callback that fires whenever the viewer state changes (e.g., selection, filters, layout changes) [99] . The state is an object you can serialize (it contains the current charts, layout, selection predicate, etc.) [100] [101] . You could store this to restore the UI later via the `initialState` prop if desired. - Other notable props shown in docs include `embeddingViewLabels` (to provide custom labels objects for the plot) [102] , `searcher` (to provide a custom search interface; e.g., a vector search index; if not provided, default full-text search on the text column is used) [103] , and `onExportSelection` / `onExportApplication` callbacks to override the behavior when a user clicks "Export Selection" or "Export Application" in the UI [104] . By default, the UI will attempt to download a JSON/CSV of the selection or a ZIP of the app, but you can intercept those events.

Svelte and Vanilla JS: Similarly, Svelte users can import from `"embedding-atlas/svelte"` and use `<EmbeddingAtlas ... />` in a Svelte component [105] . For direct JS (no framework), you can instantiate the component class:

```
import { EmbeddingAtlas } from "embedding-atlas";

const targetElement = document.getElementById("container");
const props = {
  coordinator: coord,
  data: { table: "my_data_table", id: "id_column", projection: {x: "x", y:
"y"}, text: "text" },
  // ...other props...
};
const atlas = new EmbeddingAtlas(targetElement, props);
// Later, to update props:
```

```
atlas.update(newProps);
// To destroy when done:
atlas.destroy();
```

This low-level usage is supported by the library for integration into any environment (similar to Svelte's compiled output usage) [106] [107].

EmbeddingAtlas Props: Here is a summary of major properties you can pass to `<EmbeddingAtlas>` (in addition to `data` and `coordinator` discussed above):

- **initialState:** an `EmbeddingAtlasState` object to restore a previously saved UI state [108]. This can include prior chart configurations, layout mode, selection predicate, etc., and is useful for preserving user sessions [109] [101].
- **embeddingViewConfig:** an object to configure the embedding plot's behavior [110]. For example:
- `mode`: "points" vs "density" mode (point cloud or density heatmap) [97],
- `pointSize`: number to set point radius manually [98],
- `autoLabelEnabled`: boolean to turn on/off automatic cluster labeling [111],
- `autoLabelOptions` such as `autoLabelDensityThreshold` and `autoLabelStopWords` to tweak label generation [112].
- **embeddingViewLabels:** an array of label objects (each with `x, y, text, level?, priority?`) to display fixed labels on the plot [102]. If this prop is not set and `autoLabelEnabled` is true (default when a text column is present), the system will generate labels automatically [113].
- **searcher:** an object to override search functionality [114]. By default, the component uses full-text search on the `text` column and its own approximate KNN for vector search. If you need custom search (for example, using an external vector index or a different text search logic), you can provide a `Searcher` object here. The `Searcher` interface can have methods like `query(queryString)` for full-text and `nearestNeighbors(id)` or `vectorSearch(vector)` for similarity. If you set `searcher` to `null`, all search functionality in the UI will be disabled [115].
- **tableCellRenderers:** allows custom rendering for cells in the data table [116]. You can specify a mapping from column name to either a custom component or `"markdown"` to render that column's text as Markdown [116]. This is useful to, say, render images in a column or format text in a special way.
- **onExportSelection:** a callback `function(predicate, format) -> Promise` triggered when the user clicks the "Export Selection" button [117]. The `predicate` is a SQL where clause for the selection (or `null` if selecting all), and `format` is one of `"json" | "jsonl" | "csv" | "parquet"`. You can use this to handle data export on your terms (e.g., send the predicate to your server to generate a file). If not provided, the default behavior will try to use the browser to download the data (which requires all data to be in browser memory).
- **onExportApplication:** a callback `function() -> Promise` for when the user clicks "Export Application" (to download the entire app as a zip) [99]. If you implement this, you likely want to call the library's export under the hood or provide your own packaging.
- **onStateChange:** as described earlier, called whenever any significant state changes [118] (selection, filters, layout, etc.). The state object includes things like current filter predicate, chart states, layout info, etc., which you can persist or use to sync with other parts of your application [101].
- **cache:** a `Cache` object to speed up initialization [119]. This is an advanced feature; the library can cache intermediate results (like precomputed layouts or search indexes). If you reuse the component

frequently with the same data, providing a cache can eliminate repeated expensive computations on load.

For additional customization (e.g., theming beyond light/dark or embedding custom overlays), you might need to use lower-level components or the config/theme options described below.

3. Using Lower-Level Components (EmbeddingViewMosaic, Table, etc.)

If you prefer to assemble your own UI or integrate only parts of Embedding Atlas, the library exposes the core components:

- **EmbeddingViewMosaic:** This is the interactive embedding scatter plot that connects to a DuckDB table via Mosaic (as used inside EmbeddingAtlas). You might use this if you want the embedding view in one part of your app and custom controls elsewhere. Its props include all those needed to render the scatter plot: `table`, `x`, `y` columns (required) [120], an optional `category` column for coloring points by category [121], `text` column for tooltips [122], `identifier` (id column) [123], plus many of the view and interaction props similar to EmbeddingAtlas (selection, tooltip, width/height, labels, etc.) [124] [125]. You also pass a `coordinator` (or it will use a default one) to sync with other Mosaic components [126]. The EmbeddingViewMosaic supports callbacks like `onTooltip`, `onSelection`, `onRangeSelection` for lasso, etc., and customization such as `customTooltip` and `customOverlay` to draw custom elements on the canvas [127] [128] [129] [130]. (See the official docs for more on custom overlays/tooltips if needed.)

Example (React):

```
import { EmbeddingViewMosaic } from "embedding-atlas/react";
<EmbeddingViewMosaic
    table="my_data_table" x="x" y="y"
    category="cluster_id" text="description" identifier="id"
    filter={someSelection}  // to link filtering with a selection object
    onSelection={(pts) => console.log("Selected points:", pts)}
    labels={myLabelsList}
/>
```

This would render just the scatter plot. You might pair it with the Table component (below) and some custom charts.

- **Table:** This component renders a tabular view of a DuckDB table (or a filtered view) and is also linked via Mosaic. Use it to display metadata columns in a grid with sorting, etc. Key props include `table` (table name), `columns` (array of column names to display) [131], `rowKey` (unique id column) [132], and optional `filter` (a Mosaic Selection to filter which rows are shown, e.g., based on selection in EmbeddingView) [133]. You can also control initial column widths, hide certain columns via `columnConfigs`, and get a callback when a column is resized or shown/hidden (`onColumnConfigsChange`) [134] [135]. The Table component supports theming (fonts, colors) via a `theme` prop [136] [137] and can highlight rows or scroll to a particular row on command (using `highlightedRows` and `scrollTo` props) [138] [139]. It also provides an `onRowClick` callback to react to user clicks [140].

Example (React):

```
import { Table } from "embedding-atlas/react";
<Table
  table="my_data_table"
  columns={["id","name","score","category"]}
  rowKey="id"
  showRowNumber={true}
  filter={someSelection}
  onRowClick={(row) => console.log("Clicked row:", row)}
/>
```

This will show a table of the specified columns, only for rows matching `someSelection` (if provided). The table and embedding view can share the same coordinator so that selections and filters propagate between them.

- **EmbeddingView:** This is a variant of the embedding plot that does *not* rely on DuckDB or Mosaic. Instead of pointing to a table, you pass the actual coordinate arrays (and optional category array) directly as props [141] [142]. This is useful for simpler use cases or if you already have the data in JS and don't need cross-filtering. It has similar props for selections and callbacks, but since it isn't tied to Mosaic, linking it with a Table requires manual handling of selection events. In most cases, if you plan to integrate with other views, **EmbeddingViewMosaic** is preferred. But if you just want to drop a standalone scatter plot of points (with pan/zoom and selection) into a page, EmbeddingView is appropriate.

For instance:

```
import { EmbeddingView } from "embedding-atlas";
const xCoords = new Float32Array([...]);
const yCoords = new Float32Array([...]);
const categories = new Uint8Array([...]);  // optional

new EmbeddingView(document.getElementById("plot"), {
  data: { x: xCoords, y: yCoords, category: categories },
  onSelection: (pts) => { ... },
  labels: myLabels
});
```

This directly renders an interactive scatter given coordinate arrays [143] [144]. You manage any interactions via the callbacks.

All these components use WebGL under the hood for rendering millions of points efficiently, and they share common features (like the ability to generate cluster labels if given a text or via `queryClusterLabels` callback, etc.). The **theme** and **config** customization options for EmbeddingView/EmbeddingViewMosaic allow you to adjust visual aspects (colors, font, whether to show a status bar, etc.) [145] [146]. For example, you can provide a `theme` object with `light` and `dark` sub-objects to override default colors for things like **clusterLabelColor**, **background colors**, **hover highlights**, etc. [147] [145]. This is advanced usage but can be helpful for blending the plot's look into your application's theme.

4. In-Browser Algorithms API

The Embedding Atlas front-end package also exposes certain algorithms as standalone functions, in case you need to compute embeddings or nearest neighbors directly in the browser (for instance, to update the visualization or for other interactive analysis). These are **WebAssembly-accelerated** implementations, allowing heavy computations without server calls [11]:

- **UMAP Dimensionality Reduction:** The library provides `createUMAP()` to perform UMAP in JS/WASM. It is based on the `umappp` C++ implementation [148]. Usage example:

```
import { createUMAP } from "embedding-atlas";

const count = 2000, inputDim = 100, outputDim = 2;
const data = new Float32Array(count * inputDim);
// ... fill `data` with your high-dimensional embedding vectors ...

const umap = await createUMAP(count, inputDim, outputDim, data, { metric:
"cosine" });
// Now you can iteratively optimize or run to completion:
umap.run();             // run until convergence (default epochs)
// or for animation or incremental update:
// for (let epoch = 0; epoch < 100; epoch++) { umap.run(epoch); }

const embedding2D = umap.embedding();  // Float32Array of length
count*outputDim
umap.destroy();  // free memory when done
```

  This allows you to compute 2D (or 3D etc., by setting outputDim) projections client-side [12] [149]. You could use this to recompute UMAP when new data arrives, or to do an animated UMAP projection over time for effect [150]. The `metric` option can be changed (e.g. "euclidean", "cosine", etc.), and other UMAP parameters may be supported (check documentation or function signature in code).

- **Approximate Nearest Neighbors (KNN) Search:** The function `createKNN()` allows building an approximate KNN index (via HNSW or NN-Descent algorithms from the `knncolle` library) in the browser [151]. Example:

```
import { createKNN } from "embedding-atlas";

const count = 2000, dim = 100;
const data = new Float32Array(count * dim);
// ... fill data with your vectors ...

const knn = await createKNN(count, dim, data, { metric: "cosine" });
// Query the index:
const queryVec = new Float32Array(dim);
// ... fill queryVec ...
const k = 10;
const result = knn.queryByVector(queryVec, k);
// result will be an object containing indices and distances of the k
nearest neighbors
knn.destroy();
```

If you already have an embedding matrix in the browser (e.g., output of a model or loaded from file), you can use this to support interactive vector search. The Embedding Atlas UI uses a similar approach for its vector search bar when you don't precompute neighbors – it builds an index on the fly (likely with NN-Descent by default) [152] [153] .

- **Density-Based Clustering:** To support automatic label generation, the library can cluster the 2D point density. The `findClusters()` function takes a density grid (such as one produced by a kernel density estimate over the embedding points) and identifies clusters in it [154] . Usage:

```
import { findClusters } from "embedding-atlas";

// Assume densityMap is a Float32Array of size width*height representing
point density
const clusters = await findClusters(densityMap, width, height);
```

This returns an array of cluster objects, each with properties describing the cluster [155] :

- `identifier` : cluster ID [156] ,
- `sumDensity` : total density sum in cluster,
- `meanX, meanY` : centroid location (density-weighted) [157] ,
- `maxDensity` and `maxDensityLocation` : peak density value and its location [158] ,
- `pixelCount` : number of pixels in the cluster region [159] ,
- `boundary` : an array of polygons (each polygon is an array of `[x,y]` points) outlining the cluster area [160] ,
- `boundaryRectApproximation` : an array of bounding boxes approximating the cluster shape (each `[x1,y1,x2,y2]`) [161] .

The Embedding Atlas UI uses this to draw contour outlines and label clusters with representative keywords. You could use the cluster info yourself for custom analytics or visualizations.

These algorithmic interfaces enable advanced use-cases like dynamic re-computation of embeddings or custom search UIs entirely in the browser. They are optimized for performance (via C++ and Rust implementations compiled to WebAssembly) and can handle thousands to millions of points depending on memory and compute power available in the client [11].

Data Format and Usage Notes

To effectively use Embedding Atlas, you should prepare your data in the expected format: - **Tabular Data:** The core of Embedding Atlas is a table of data points (rows). Each row represents an item (e.g., a sentence, an image, a product) and can have many columns of metadata (e.g., text, category labels, numeric scores). For large datasets, store this table in a Parquet file (recommended) or Arrow for efficient loading. In Python, the data can be a pandas DataFrame. - **Embeddings:** If you want to visualize high-dimensional embeddings, you can either have the raw high-dimensional vectors in one column (and let Embedding Atlas compute a projection) or precompute a 2D projection: - *Raw vectors:* put them as an array-like column (e.g., each cell contains a list/vector of floats) and use `--vector` flag (CLI) or pass that column name to the widget (though the widget currently doesn't auto-project – you'd have to project manually or rely on the CLI's `enable_projection`). This approach is less common; typically you either have text/image to embed or you pre-project. - *Precomputed 2D coords:* have two float columns, e.g. `emb_x` and `emb_y`, for the 2D coordinates. Then pass `--x emb_x --y emb_y` (CLI) or `x="emb_x", y="emb_y"` in Python. This is most straightforward for consistent results. - **Text data:** If your data has a text field that describes each item, provide its column name (CLI `--text` or widget `text=`). This text will be used in the UI for: - Tooltips when hovering points (shows the text content), - Full-text search queries, - Automatic cluster labeling (the tool picks frequent words from the text in each cluster, excluding stop words) [66]. - **Image data:** If you have image data, currently the CLI can handle it via `--image <column>` [162] if the column contains image file paths or PIL image objects (in a DataFrame). The default model for images will be applied (ViT). In a web context, image handling would require you to embed image features first (the front-end doesn't directly encode images). - **Neighbors:** If available, precompute nearest neighbors for each point (perhaps using a more precise offline method) and store them in a column as JSON strings or Python dicts (as described earlier). This greatly speeds up the "similar items" functionality and can improve label generation (since labels use neighboring points to pick keywords). - **Unique ID:** Ensure there is a unique identifier for each row (the tool can add one as an `id` if not provided). This is important for linking selection and for referencing neighbors. - **DuckDB limitations:** If using DuckDB-WASM (default in browser), very large datasets (millions of rows) can be memory-intensive in the browser. For extremely large data, consider sampling or using the server mode (where Python/duckdb serves queries). The `--sample` option is handy to reduce size for initial exploration [46]. - **Saving State:** If you want to save the state of an interactive session (selected filters, layout, etc.), you can use the state management callbacks (in JS `onStateChange` / `initialState`, or in Python perhaps use the returned predicate). This can help in generating reproducible reports or resuming analysis.

Architecture and Design Insights

Embedding Atlas is architected as a **bimodal tool**: a Python back-end for data/ML tasks and a Web front-end for visualization: - The **Python package** (`embedding_atlas`) contains the CLI and integration points. It relies on libraries like **Hugging Face Transformers/SentenceTransformers** for embedding computation and **UMAP** (via `umap-learn`) for projection [37]. It also uses **DuckDB** in Python to prepare the data and possibly serve it (especially in server mode or for the Streamlit integration) [14]. When you launch the tool via CLI or widget, the Python side will handle heavy computations (embedding model inference can use CPU or GPU, and UMAP projection) and then either start a local web server or, in notebooks, emit a front-end bundle that runs in the output. - The **Front-end (JavaScript)** is responsible for rendering and interactivity. It is built with WebGL2/WebGPU for rendering millions of points efficiently [21]. The front-end performs tasks like dynamic querying of data (via DuckDB WASM) and approximate nearest neighbor search for quick similarity lookups [151]. Notably, the front-end includes **WebAssembly** modules: one for UMAP (`umappp` via Emscripten) and one for clustering (`density-clustering` in Rust) [163]. This means the browser can compute new projections or cluster densities on the fly without server round-trips, enabling a fully offline experience. - **Mosaic Coordination:** The use of Mosaic (a framework from UW IDL) indicates a design for cross-linked visualizations. The coordinator synchronizes selections: for example, brushing a range in a histogram filter will translate to a selection predicate that filters the table and highlights points in the scatter plot, and vice versa. The Embedding Atlas front-end components are essentially specialized Mosaic views for embeddings and data tables. - **Extensibility:** The system allows extension in both environments. In Python, you could swap in any embedding model (text or image) accessible via HuggingFace by specifying `--model`, or even plug in your own embeddings by providing them as data. In JavaScript, the exposed components and algorithm functions mean you can build custom UI around the core visualization (for instance, adding custom panels, or using the EmbeddingView in an entirely different context). The ability to export the entire application as a static bundle (`--export-application`) is also a design choice to facilitate sharing results: you can generate an interactive visualization and host it as a simple static site or send it to colleagues [51]. - **Performance:** Through WebGPU and WASM, Embedding Atlas can handle *millions* of points in the browser with interactive frame rates [21]. Order-independent transparency and LOD techniques are used for rendering dense point clouds clearly [164]. The point rendering adjusts size by density so that even overlapping points remain discernible [97] [98]. DuckDB provides fast querying for cross-filters, and heavy computations (UMAP, KNN) are either done ahead in Python or asynchronously in the browser. This design ensures that even large embedding datasets (e.g., 200k points as noted in examples) can be explored smoothly.

In summary, Embedding Atlas offers an end-to-end solution for embedding visualization: from computing embeddings (if needed) to rich interactive exploration – all within a local environment. Whether you integrate it via Python (for data science workflows) or JavaScript (for web apps), it provides a consistent interface and powerful defaults, while also allowing deep customization for advanced use cases.

Usage Examples

To tie everything together, here are two quick, end-to-end usage examples – one in Python and one in TypeScript:

Python Example: Visualize a text dataset in a Jupyter notebook.

```python
# Install if not already
!pip install embedding-atlas datasets

from datasets import load_dataset
from embedding_atlas.widget import EmbeddingAtlasWidget
from embedding_atlas.projection import compute_text_projection
```

```
# Load a dataset from Hugging Face Hub
dataset = load_dataset("ag_news", split="train[:10000]")
df = dataset.to_pandas()

# Compute embeddings and 2D projections
compute_text_projection(df, text="text", x="umap_x", y="umap_y",
neighbors="neighbors")

# Launch the Embedding Atlas widget
widget = EmbeddingAtlasWidget(df, text="text", x="umap_x", y="umap_y",
neighbors="neighbors")
display(widget)

# After interacting, retrieve selected subset (if any)
selected_points = widget.selection()
```

In this example, we load 10k news articles, compute sentence embeddings and UMAP projection, and visualize them. We could then use `selected_points` to see, for instance, all articles in a particular cluster selected via lasso.

TypeScript Example: Integrate Embedding Atlas in a React app to visualize precomputed embeddings.

```
import React, { useEffect, useState } from 'react';
import { EmbeddingAtlas } from 'embedding-atlas/react';
import * as duckdb from '@duckdb/duckdb-wasm';  // assume duckdb-wasm to load
data

const App: React.FC = () => {
  const [coord] = useState(() => /* initialize Mosaic coordinator */ undefined);
  const [dataLoaded, setDataLoaded] = useState(false);

  useEffect(() => {
    async function loadData() {
      // Load data into DuckDB (this is pseudocode, adjust for actual duckdb
usage)
      const db = new duckdb.AsyncDuckDB();
      await db.instantiate(/* ... */);
      const conn = await db.connect();
      // Assume we have a Parquet file 'embeddings.parquet' hosted or bundled
      await conn.query(`CREATE TABLE my_data AS SELECT * FROM
'embeddings.parquet';`);
      setDataLoaded(true);
    }
    loadData();
  }, []);
  if (!dataLoaded) {
    return <div>Loading data...</div>;
  }

  return (
    <EmbeddingAtlas
      coordinator={coord}
      data={{ table: "my_data", id: "id", projection: {x: "umap_x", y:
"umap_y"}, text: "description" }}
      colorScheme="dark"
      onStateChange={(state) => console.log("Selection predicate:",
state.predicate)}
    />
  );
};

export default App;
```

Here, we load a DuckDB table from a Parquet (using duckdb-wasm), then render the EmbeddingAtlas. This would show the points (using `umap_x`, `umap_y` columns for positions) and enable full interaction. We log the SQL predicate of any selection to the console (as an example of reacting to user interaction). In a real app, we might use `state.predicate` with DuckDB to derive related insights or update other components.

Conclusion

Embedding Atlas provides a powerful set of tools to bridge machine learning embeddings with interactive visualization. Through its Python APIs, data scientists can quickly explore and present embedding spaces, and with the TypeScript library, developers can build rich, customized applications around these visualizations. This documentation covered the major components, APIs, and configurations to help you integrate Embedding Atlas into your workflow or product. For more details, refer to the [official GitHub repository】 [1] and the [online documentation site】 [165], and join the community discussions for tips and support. Happy embedding exploration!

Sources:

- Apple Embedding Atlas GitHub README and Documentation [1] [37] [55] [166] [12] (Apple Inc., 2025) – *Detailed usage guide for CLI, Python widget, Streamlit, and JS components.*

---

[1] [9] [165] Overview | Embedding Atlas
https://apple.github.io/embedding-atlas/overview.html

[2] [3] [28] [29] [30] [31] [34] [35] Embedding Atlas
https://huggingface.co/docs/hub/en/datasets-embedding-atlas

4 5 6 21 164 Embedding Atlas
https://apple.github.io/embedding-atlas/

7 8 10 14 15 17 18 22 23 24 25 26 27 32 33 36 37 38 39 40 41 42 43 44 46 47 48 49 50 51 52 53 162 Command Line Utility | Embedding Atlas
https://apple.github.io/embedding-atlas/tool.html

11 12 148 149 150 151 152 153 154 155 156 157 158 159 160 161 Algorithms | Embedding Atlas
https://apple.github.io/embedding-atlas/algorithms.html

13 54 55 56 57 58 59 60 61 62 64 65 66 67 68 69 70 71 Python Notebook Widget | Embedding Atlas
https://apple.github.io/embedding-atlas/widget.html

16 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 Streamlit Component | Embedding Atlas
https://apple.github.io/embedding-atlas/streamlit.html

19 45 63 87 91 92 93 94 95 96 99 100 101 102 103 104 105 106 107 108 109 110 114 115 116 117 118 119 166 EmbeddingAtlas | Embedding Atlas
https://apple.github.io/embedding-atlas/embedding-atlas.html

20 163 Development Instructions | Embedding Atlas
https://apple.github.io/embedding-atlas/develop.html

88 97 98 111 112 113 120 121 122 123 124 125 126 127 128 129 130 145 146 147 EmbeddingViewMosaic | Embedding Atlas
https://apple.github.io/embedding-atlas/embedding-view-mosaic.html

89 131 132 133 134 135 136 137 138 139 140 Table | Embedding Atlas
https://apple.github.io/embedding-atlas/table.html

90 141 142 143 144 EmbeddingView | Embedding Atlas
https://apple.github.io/embedding-atlas/embedding-view.html