ChatGPT

Semantik Embedding Atlas Guide

A. High-Level Overview of Embedding Atlas

What is Embedding Atlas? Embedding Atlas is an open-source toolkit from Apple for interactive visualization of high-dimensional data (embeddings) at scale. It addresses common pain points in exploring embeddings by making it easy to load data, project it to 2D, and interactively analyze patterns and clusters – all **entirely in the browser** with no external server needed [1]. This privacy-first design ensures that data never leaves the user's machine [2], yet it still enables smooth exploration of even very large embedding datasets (millions of points) in real time [3].

Problems it solves: Embedding Atlas reduces "friction" in embedding visualization by eliminating tedious setup (it computes projections and embeddings on the fly if needed), and by integrating metadata into the visualization [4]. Traditional tools often only show scatter plots and require separate steps or tools to inspect metadata; Embedding Atlas instead provides **linked views** (scatter plots, tables, charts) so users can filter and cross-reference data points with their metadata in one place [5]. It includes advanced techniques like density-based clustering with automated labeling of clusters [4] to help users immediately see groupings in the data without manual effort. In short, it's designed to make understanding an embedding space as intuitive as navigating a map.

Key capabilities relevant to Semantik:

- **Interactive 2D visualization:** Embedding Atlas renders high-dimensional embeddings in 2D (e.g. via UMAP) and lets users pan, zoom, and explore interactively. The scatter plot can handle large point counts using GPU acceleration (WebGPU with WebGL2 fallback) for smooth performance [6]. This means Semantik can visualize thousands to millions of document chunks as points, and the UI remains responsive.

- **Cross-filtering and linked views:** A hallmark of Atlas is the ability to link the embedding view with other views like data tables or histograms. Users can brush or filter by metadata (e.g. by document type or date range), and the scatter plot updates instantly to show only the filtered subset [5]. This is powerful for Semantik's use case of multi-document analysis – for example, filtering the embedding to show chunks from certain folders or time periods. (Even if Semantik doesn't use this immediately, Atlas's design makes it feasible in the future.)

- **Brushing, selection, and faceting:** Users can select points (individually or via lasso/brush) on the scatter plot, and those selections can drive other actions or views. Embedding Atlas supports multiple selection modes (click, shift-click for multi-select, lasso drag for area select) and highlights selected points. This aligns with Semantik's need for allowing users to select a set of chunks and then perform actions like viewing them or running searches.

- **Nearest-neighbor search and point queries:** Embedding Atlas makes it easy to find similar points. It can display the nearest neighbors of a selected point (either via precomputed data or on-the-fly search) [7] [8], and it even supports full-text search queries to highlight points matching a query [9]. This means in a Semantik integration, a user could click a chunk and quickly retrieve similar chunks, or type a keyword and see matching chunks highlighted, all through the Atlas interface. (Semantik's backend would handle the actual search, but Atlas provides the UI hooks for it.)

- **Cluster detection and labeling:** Embedding Atlas can automatically detect cluster structures in the 2D projection (using a density map) and label them with meaningful terms [4] [10]. In practice, this means it tries to find groups of points that form a cluster and, if text data is available, suggest a label (e.g. a keyword that represents that cluster). This is useful for Semantik because it provides at-a-glance understanding of what each region of the embedding represents (e.g. a cluster of points might get labeled "Project Plan" if many chunks there contain those words). We can also supply our own labels or categories, as Semantik often knows the grouping (like by document or folder).

- **Privacy and offline use:** Importantly, Atlas's design aligns with scenarios where data privacy is critical. All heavy computations – embedding model inference, dimensionality reduction, clustering – can run in the user's browser via WebAssembly and WebGPU [11]. For Semantik's enterprise clients, this means if Semantik exposes Atlas in the frontend, the sensitive document data isn't being sent to a third-party service for visualization; it stays within Semantik's environment or the user's browser. (In our specific integration, we might precompute embeddings/projections on the backend, but Atlas itself does not phone home or require an internet connection – it's an offline-friendly widget.)

In summary, Embedding Atlas gives Semantik a powerful interactive map of document embeddings. Users can **visually explore relationships** between documents and chunks, use **linked metadata filters** to slice and dice the data, and leverage **search and clustering** to dig deeper – all through a polished web component that handles the heavy lifting (rendering, UI interactions, etc.) seamlessly [6] [12].

B. Embedding Atlas Data Model vs. Semantik's

Embedding Atlas and Semantik both deal with datasets of embedding vectors plus metadata, but they conceptualize the data slightly differently. Understanding how Atlas expects data to be structured will help map Semantik's outputs into Atlas's inputs.

Atlas's expected data model: In Embedding Atlas, the core unit is a **table of data points** (think of it like a dataset or DataFrame). Each row in the table represents one data point (e.g. one document or one chunk) and can have multiple columns: some columns may be the high-dimensional embedding vector, others may be metadata (like text, category labels, IDs), and optionally precomputed projection coordinates. Atlas uses an in-browser database (DuckDB via WebAssembly) to manage this data table and run queries for filtering and search [13]. Key aspects of the Atlas data model:

- **Rows and columns:** Every data point is a row, with columns for various attributes. For instance, if you loaded a CSV or Parquet into Atlas, each row could have a `text` column (the content), some metadata columns (e.g. `author`, `date`), and perhaps an `id` column (unique identifier). Atlas doesn't enforce a strict schema; it's flexible (it even says there's "no required schema" beyond some known column names) [14]. However, there are conventions: if a column named `projection_x` and `projection_y` exist (or whichever columns you specify as X and Y), Atlas will use those as the 2D coordinates [15]. If not, it will compute a projection (UMAP) in the browser. Similarly, you can provide a `neighbors` column with precomputed nearest neighbor lists [7], or Atlas can compute approximate neighbors on the fly with its WASM algorithms.

- **DuckDB and Mosaic:** Under the hood, the full Atlas app loads your data into DuckDB (a SQL database) running either in-browser or on a local server, and it uses a coordination layer called Mosaic for linking views [16]. Mosaic manages shared state (like what filter is active, or what selection is made) across multiple components (embedding view, table, charts). DuckDB executes queries to filter the data based on selections (for example, “Country = US or France”) directly in the browser for speed [13]. This architecture allows Atlas to cross-filter large datasets efficiently without needing a backend server.

- **Data columns of interest:** For the embedding visualization, Atlas cares about certain columns: the **embedding vector** (if provided, or else it will generate one using a model for text/images), the **projection coordinates** (x, y in 2D, either given or computed via UMAP), a **unique identifier** for each row, an optional **text column** (used for tooltips and search), and optional **category columns** (for coloring). In the Atlas React API, when using the full `EmbeddingAtlas` component, you specify these via a data config, e.g. `{ table: "data_table", id: "id_column", projection: {x: "x_column", y: "y_column"}, text: "text_column", neighbors: "neighbors_column" }` [17] [18].

- **Metadata and schema flexibility:** Apart from those, all other metadata columns (like file type, date, etc.) can exist in the table and Atlas will automatically create sidebar charts for them. For example, the Atlas paper describes loading a wine reviews dataset: it had columns for country, points, price, etc., and Atlas automatically showed distributions for each metadata field in a sidebar for filtering [5]. While we might not use that full capability in Semantik initially, it’s good to know Atlas is built to ingest rich metadata and allow interactive queries on it.

Semantik’s data model: Semantik’s backend deals with documents and their chunks. Key elements of Semantik’s model:

- **Embeddings in a vector DB:** When Semantik ingests documents, it breaks them into chunks and computes an embedding vector for each chunk. These vectors (e.g. 768-dimensional if using a transformer model) are stored in a vector database (like Qdrant). Each vector is associated with metadata such as the document ID it came from, the chunk index or position, the source or folder, MIME type, timestamp, etc. Essentially, Semantik has a collection of embedding points with rich metadata, but these live server-side.

- **Precomputed projections:** Semantik already runs a projection pipeline as a background operation. This pipeline pulls a set of embeddings (for example, all chunks in a workspace or a sampled subset), then applies dimensionality reduction (PCA, UMAP, or t-SNE) on those vectors to get 2D coordinates for each point. The result is stored as a set of artifact files or data structures:

- An array of X coordinates (float32) for each point.
- An array of Y coordinates (float32) for each point.
- A **category** array (uint8) assigning each point to a category (which could represent something like “which document this chunk belongs to” or other grouping).

- An **ids** array that holds a unique identifier for each point (so that Semantik can map the point back to the underlying chunk or document in the database).
- A metadata JSON object describing the projection (e.g. what algorithm was used, any sampling performed, and a legend mapping category indices to human-readable labels and counts).

- **Document & chunk metadata:** Outside of the projection artifact, Semantik maintains detailed metadata about each chunk (in the database and index). For any given chunk ID (or document ID + chunk index), Semantik knows the chunk's text content, the document it's from, the document title, etc. This is accessible via Semantik's APIs (for example, an API to resolve an ID to the chunk's full metadata or content snippet).

Mapping between the two models:

- **Data point correspondence:** A Semantik projection is essentially a dataset of points that can be visualized. In Embedding Atlas terms, one Semantik projection corresponds to an **Atlas dataset/table**. Each chunk in Semantik (within that projection) is one row in the Atlas table. The precomputed `x` and `y` arrays from Semantik become the "projection_x" and "projection_y" columns in Atlas (or are passed directly to the `EmbeddingView` component) [15]. Because Semantik computes these coordinates server-side, we will treat them as fixed inputs to Atlas (so Atlas doesn't need to run UMAP itself).

- **Identifiers (IDs):** Semantik provides an `ids` list for the projection points. This plays the role of Atlas's **id column**. In Atlas, each row should have a unique identifier so that selections can be mapped back to data [19]. We should ensure that the `ids` array we pass (or the equivalent) is used so that when a user selects a point in Atlas, we know exactly which Semantik chunk it corresponds to. If we use the low-level `EmbeddingView` component, we might not explicitly label the id in the data prop; however, Atlas will still emit an `identifier` field in selection events if it's aware of one. In the Mosaic-based components, you can specify `identifier="id_column"` [20] so that `DataPoint.identifier` equals whatever unique ID you provided. For Semantik, that should be the chunk's unique id or index. In short: **Semantik's `ids` array = Atlas's row identifiers.**

- **Embedding vectors:** Semantik's original embedding vectors (the high-dimensional ones in the vector DB) are not passed to Atlas in our integration – and that's fine. Atlas only needs them if it were to compute UMAP or run its own nearest-neighbor search. Since Semantik does those on the backend, we can omit sending 768-d vectors to the frontend. We give Atlas just the 2D coordinates. (If we ever wanted to let Atlas handle re-projection or local KNN, we could send vectors or use Atlas's WASM UMAP and KNN functions [21] [22], but in our setup, we keep those responsibilities on the server.)

- **Categories and legends:** Semantik often assigns each point a category, for coloring purposes. For example, all chunks from Document A might be category 0, from Document B category 1, etc., or categories might represent other groupings (file type, year, cluster ID from some analysis, etc.). In Atlas's model, this directly maps to a **category column** – specifically a numeric categorical column that the embedding view can use to color points. Atlas expects category values as integers starting from 0 [23], which matches what Semantik already does. The legend (mapping category index to label and count) that Semantik produces in its metadata JSON is analogous to Atlas's legend in the UI

(Atlas will show a color legend if using the full app). In our integration, we can use the `categoryColors` prop to supply a palette so that Atlas uses consistent colors for each category index [24], and we'll maintain our own legend UI or reference using the meta JSON (since using just the EmbeddingView component doesn't automatically show a legend).

- **Text and metadata columns:** Atlas has the concept of a text column (used for tooltips and search) [25] and can include arbitrary metadata columns (used for filtering and to display in a table view). In Semantik's current integration, we likely did not send all metadata to the frontend due to data volume. We might not have a "text column" at all in Atlas if we're not loading chunk text into the browser (which would be heavy). Instead, Semantik's backend can provide tooltip info on the fly. However, conceptually, if we were to mirror Atlas's model: each chunk could have a text field (e.g. the chunk's content or an excerpt) and other fields (document name, file type, etc.). If we ever decided to leverage Atlas's full capabilities, we could create a lightweight table with just key metadata (like `docTitle`, `fileType`, etc.) and use DuckDB in WASM to enable local filtering. But given Semantik's scale (potentially many chunks) and that the backend already handles queries, our approach is to keep most metadata server-side and fetch it as needed.

Summary of mapping: A **Semantik projection** can be viewed as an **Embedding Atlas dataset** where: - The **points** are the chunk embeddings (rows in Atlas). - `x` and `y` from Semantik become Atlas's 2D coordinates for the scatter plot [15]. - Semantik's `ids` are used as each point's unique identifier (Atlas `id`) [19]. - A Semantik **category** (numeric) is passed as Atlas's category column [23] for coloring; Semantik's legend maps to Atlas's category legend. - **Metadata** like chunk text or document name is not initially loaded into Atlas's table (to minimize data transfer), but Atlas's architecture could accommodate it if needed (and it uses such metadata for tooltips/labels when available [26] [27]).

Understanding this correspondence ensures that when we feed data to the Atlas visualization component, everything lines up: the point index in the `x/y` arrays can be traced back to a unique chunk ID, categories are correctly interpreted, and (if we extend the integration) we know where text or other fields would fit in Atlas's schema.

C. Embedding Atlas React API (Focus on EmbeddingView)

Semantik integrates Embedding Atlas on the frontend via its React component library. The main component of interest is the **EmbeddingView**, which is essentially the scatter plot viewer. Embedding Atlas offers a few React components – `EmbeddingView`, `EmbeddingViewMosaic`, `Table`, and a higher-level `EmbeddingAtlas` – but since Semantik is handling data and computation on the backend, we likely use `EmbeddingView` directly for simplicity. Let's break down how to use `EmbeddingView` and relevant API details:

EmbeddingView component: This component renders up to a few million points with given X/Y coordinates on a WebGL/WebGPU canvas [28]. In React, you import it from `"embedding-atlas/react"` and use it like a typical controlled component (passing props for data and event handlers). For example:

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

<EmbeddingView
```

```
    data={{ x: xArray, y: yArray, category: categoryArray }}
    tooltip={tooltipState}
    onTooltip={setTooltipState}
    selection={selectionState}
    onSelection={setSelectionState}
    labels={labelsArray}
    config={viewConfig}
    onRangeSelection={handleLasso}
    /* ...other props... */
  />
```

The `data` **prop** is where we supply our coordinate and grouping data: - It's an object with at least `x` and `y` fields, and optionally `category`. For example: `data={{ x: xColumn, y: yColumn, category: categoryColumn }}` [29]. Here `xColumn` and `yColumn` are Float32Array objects containing all the X and Y coordinates, and `categoryColumn` is a Uint8Array of the same length containing category indices [30] [31]. These correspond exactly to Semantik's projection outputs (the arrays we produce server-side). - **Important:** The arrays must be the correct types (Float32Array for coordinates, Uint8Array for category) as the library expects. If Semantik's data is in plain JS arrays or a different type, we should convert/cast to these typed arrays before passing in. - We do *not* provide an `id` or `text` field in this `data` object when using `EmbeddingView`. Unlike the high-level `EmbeddingAtlas` component, `EmbeddingView` doesn't have a built-in notion of an id or text column in the `data` prop. It purely takes coordinates (and category). This means the EmbeddingView itself isn't automatically aware of chunk identifiers or textual content – we have to manage that via event callbacks.

The `labels` **prop** allows us to overlay text labels on the scatter plot. Each label is an object with at least `{ x: number, y: number, text: string }` [32]. These coordinates are in the same data space as the points. For instance, we could supply cluster labels or document names as static labels. In Semantik's context, we might use this if we want to label clusters of points (perhaps computed server-side) or identify a particular region (like a big document cluster named by the document title). If we do not provide `labels`, Embedding Atlas can generate labels automatically *if* it has textual data and clustering info (more on that in Section E), but since our `EmbeddingView` doesn't have direct access to chunk text, we likely provide our own labels or disable auto-labeling. We can also control label appearance via the theme (font, color, etc.) [33].

Tooltip and selection events: EmbeddingView supports interactive tooltips (on hover) and selections (on click or lasso). These are controlled via the `tooltip` and `selection` props and their corresponding callbacks:

- **Tooltip:** The `tooltip` prop can hold the current tooltip data (or `null` if no tooltip). The `onTooltip` callback will be invoked whenever the user hovers over a point or moves off a point [34] [35]. The tooltip data is an object (of type `DataPoint`) with fields: `{ x, y, category, text, identifier }` [34]. For EmbeddingView (non-mosaic usage), what do these mean?
- `x` and `y` are the coordinates of the point hovered.
- `category` is the category index of that point (if we provided the category array).
- `text` and `identifier` – these would be present if the component had a text or id context. In our case, since we didn't supply a text field or id field in data, these might be `undefined` or some

default. By default, `identifier` in a DataPoint might actually be an index of the point (Atlas often uses the row index as an identifier if no explicit id is given). We should verify this, but typically if using the Mosaic version with `identifier_column`, that value populates `DataPoint.identifier` [20]. With plain EmbeddingView, it's safe to treat the index of the point as its identifier (the index in our `x/y` arrays). The library likely uses internal indexing such that the first element of `xArray` is point 0, second is 1, and so forth.
- Because we care about mapping a hovered point to Semantik's chunk metadata, we will use the `onTooltip` event to capture when a user hovers. We can then map the hovered coordinates back to an index in our data (for example, by maintaining the assumption that order is the same, or by a near-match if needed). A simpler approach: we could store the index in the `tooltip` state by piggybacking on `identifier` if it exists. E.g., if `tooltip.identifier` is present (say it's 42), that's the index of the point, which we can then use to look up the chunk ID from our `ids` array and display a tooltip with the appropriate info (like document name or snippet).

- In practice, we might *not* want to call the backend on every hover (that could be too slow/heavy). A good pattern is to include some basic info client-side for tooltips. For instance, Semantik could pass a small string per point for tooltips (like the document title or section name). If we had that, we could populate `text` for each point (Atlas would show it by default in its tooltip). Without that, we might handle `onTooltip` by manually setting a tooltip element in our React app (for example, showing document title from an in-memory map of id→title, which we can store since doc titles are not too large).

- **Selection:** The `selection` prop represents points that are selected (clicked). The user can click on a point to select it; shift/cmd-click to multi-select; or drag a lasso (if enabled) to select multiple points [36]. The `onSelection` callback gives us the new selection whenever it changes [35]. In EmbeddingView, the selection is provided as an array of `DataPoint` objects (or `null` if cleared) [36]. Each `DataPoint` in the selection has the same shape as for tooltip (x, y, category, text, identifier). For multiple selections, it's an array of these objects. For a single selection, it might still be an array of length 1 (or possibly they allow a single object – docs indicate it's always an array or null).

- We will use `onSelection` to know when the user has clicked or multi-selected points. The typical Semantik action on selection might be: highlight those documents in the document list, or open a detail view showing those chunks, or trigger a semantic search over that set. Because our selection can be multiple, we might provide an interface (e.g. a "Compare" or "Group Actions" if many points).
- To integrate with the backend, we need to map the selected DataPoints to Semantik's entities. As with tooltip, the key is the identifier or index. If our `ids` array is loaded on the frontend, we can map index→id easily. (It is recommended to have the `ids` array available client-side so we don't have to call backend just to resolve indices on selection – since we already downloaded the projection, including an `ids` list is trivial overhead). Once we have the actual IDs of selected points, we can call a Semantik API to fetch metadata or content for those, or simply store them as an "active selection" state that other parts of the UI (like a document viewer) can react to.

- **Note:** Embedding Atlas's Mosaic variant can integrate selection more deeply (with a `Selection` object and coordinate filtering across components) [37] [27]. In our simpler use of EmbeddingView, we'll manually handle the selection. That means we might also be responsible for visual highlighting of selected points. However, the EmbeddingView does have a notion of selection internally – when points are selected, they remain visually distinguished (likely with an outline or different color) as long as the `selection` prop reflects them. We just need to feed back the selection (the onSelection gives us selection, and we can set it back into the component's `selection` prop state to keep it controlled). This is already done if we use something like `useState` to bind selection prop.

- **Range selection (lasso):** The `rangeSelection` prop and `onRangeSelection` callback relate to lasso or rectangular selection areas [38] [39]. If a user clicks and drags in the EmbeddingView, they can select an area. Atlas will then provide either a rectangle or polygon (for lasso) representing that region [38]. By default, EmbeddingView might not automatically select all points in that range – instead, it gives you the geometry and it's up to us to decide what to do. Typically, one would handle `onRangeSelection` by computing which points fall within that polygon and then setting those as the new `selection`.

- Since Semantik's backend could potentially do spatial queries (e.g. find all points in this rect), we have options. But doing a call for that may be overkill if we already have all coords in the front. We can do it in the frontend: take the polygon, iterate or use a spatial index to find points inside it, then set `selection` to those points. (This might be non-trivial for millions of points in JS – though basic rectangle queries are fine, complex polygons might require a small utility).
- Alternatively, we could ignore fine lasso selection and only use clicking/multi-click for simpler use cases. However, lasso is a powerful way to grab a cluster of points, so it's worth supporting. We might implement a simplified approach: e.g., if > N points are in the lasso, maybe we limit or sample, etc., to avoid overwhelming the UI or backend.
- Embedding Atlas Mosaic can handle this in a linked way (it would translate a lasso into a filtered selection automatically via DuckDB). Since we're not using that, we manage it manually.

Configuration and options: EmbeddingView offers a `config` prop to tune visualization settings [40] [41] : - `mode`: can be `"points"` or `"density"` [42]. "Points" means draw each point individually; "density" means show a density map (useful when there are a ton of points overlapping). Atlas can automatically switch to density mode for very large datasets or when zoomed out. We can decide to expose this or set it based on point count. For example, if Semantik knows the projection has, say, >1 million points, we might default to density mode to get a better overview (it will render smooth contours and high-density areas will show up as blobs with contour lines [43]). If < say 50k points, point mode is fine. The default if not set is probably points, but with auto density overlay if extremely dense (the `minimumDensity` config sets threshold for when to show contours [44]). We can keep default auto behavior or explicitly control it. - `pointSize`: can manually override point size [45]. Atlas auto-calculates point size based on density (it tries to avoid overplotting by making points smaller when there are many, etc.). Usually the defaults are okay, but Semantik could tweak if needed (e.g., if points are too small to see on high-DPI screen). - `colorScheme`: `"light"` or `"dark"` [46]. We should match Semantik's UI theme. Atlas can also be given separate theme configs for light/dark. - `autoLabelEnabled`: boolean to turn off Atlas's automatic cluster labeling [47]. Because we may not provide the needed text for Atlas to do it, and we might want full control, we might set this to false to prevent any surprise labels. If we do want Atlas to attempt it (perhaps if we supply a `queryClusterLabels` callback or text), we can leave it true. (More in Section E). - There are other options like `autoLabelDensityThreshold` and stop words for labeling [48], which we likely don't need to fiddle with unless we enable auto labeling.

Custom rendering: Atlas allows custom tooltip and overlay components via `customTooltip` and `customOverlay` props [49] [50]. For example, if we want a richer tooltip (HTML content with styling, images, etc.), we can provide a class that Atlas will use to render the tooltip element [51] [52]. Similarly, a custom overlay could draw additional shapes on top of the scatter (perhaps highlight a region, or draw connectors, etc.) [53] [54]. Initially, Semantik might not need these – our integration can start with default tooltips (simple text) and default overlay (none). But it's good to know this extension point exists. For instance, if we wanted to draw bounding boxes around clusters or highlight certain documents with a polygon, a custom overlay could use the provided `proxy.location(x,y)` method [55] to convert data coords to screen coords and draw on a canvas or SVG.

Which components to use: We've focused on `EmbeddingView` because Semantik's backend-centric approach means we don't rely on Atlas for data management. The other components are: - `EmbeddingViewMosaic`: similar to EmbeddingView but integrated with the Mosaic coordinator and data table. It expects a `table` name and column names (x, y, category, etc.) instead of raw arrays [56] [57]. This is what you'd use if you had loaded data into DuckDB in the browser. Semantik could theoretically use this if we decided to load an Arrow/Parquet of our data into DuckDB-WASM and let Atlas manage filtering. But that duplicates a lot of what our backend does, so we've avoided it. The Mosaic version does conveniently handle `identifier_column` so that selection events include the identifier directly [20]. But we can achieve the same mapping by our approach with the plain EmbeddingView and `ids` array. - `Table`: A React component to display a data table (with virtualization for performance) [58]. Atlas's table can show rows of data and link with selection. Semantik currently might have its own document list or not need a full table of chunks in the UI. We've not integrated Atlas's Table component yet – but if in future we want to show a list of all chunks or search results synchronized with the scatter, we might consider it. - `EmbeddingAtlas`: This is the full UI container – it includes the scatter, the table, search bar, and charts, orchestrated together [59] [60]. It requires a `coordinator` (to sync Mosaic state) and the data table to be defined (as mentioned above, you pass in table name, id column, etc.) [61]. We essentially implemented a custom version of what `EmbeddingAtlas` does by handling state in our app and feeding just the scatter component. We did this because our data lives in our backend, not a DuckDB in the browser. In practice, if we wanted to quickly get all of Atlas's UI features, we could try to use `EmbeddingAtlas` component and implement a custom `Searcher` that calls Semantik's APIs, and hook up our data via a DuckDB connection to our backend. However, that's a complex path and not necessary for a first integration. It's more practical that we continue with `EmbeddingView` and possibly a small custom UI around it (like our own legend and tooltip panel), leveraging Atlas where it excels (rendering the points and capturing interactions).

Practical usage in Semantik: Given that we have precomputed coordinates and categories: - We will use `<EmbeddingView data={{x, y, category}} />` to render the scatter. This should be done inside a React component that fetches or receives the projection data from the backend (perhaps via a REST API). - We maintain React state for `tooltip` and `selection` (e.g. `const [tooltip, setTooltip] = useState(null); const [selection, setSelection] = useState(null);`). - Pass those states to EmbeddingView: `<EmbeddingView ... tooltip={tooltip} onTooltip={setTooltip} selection={selection} onSelection={setSelection} ... />`. - Now, when the user hovers or selects, our state updates. We can use `tooltip` state to display a custom tooltip UI (like a popup div showing the doc name). Or we could rely on Atlas's default tooltip which might just show the `text` field if present. Since we likely don't have a `text` field, the default tooltip might be minimal (perhaps showing the numeric category or nothing). We can intercept the tooltip event and fill in meaningful info manually. - For selections, when `selection` state changes (say user clicked a point and `selection` becomes an array of one DataPoint), we can take that and trigger Semantik-specific actions. For example, we could call an API to get the content of that chunk or navigate the user to the document view focused at that chunk. If multiple points are selected, maybe we show an aggregate view or allow the user to run a search within that subset.

In summary, `EmbeddingView` is the core component we use, and its API revolves around feeding it **data arrays** and handling **interaction callbacks**. We should ensure the data we pass is correctly formatted (typed arrays) and that we properly map Atlas's event data (like the DataPoint indices) back to Semantik's world (document IDs, metadata). By doing so, we leverage the best of Atlas (rendering performance, interactive selection, etc.) while keeping our backend logic in play.

D. Interaction Model: Selection, Tooltips, Cross-Filtering

Embedding Atlas's interaction model is rich: it was built to let users explore data from multiple angles. For Semantik's integration, we are mostly concerned with the scatter plot interactions (hover tooltips and selections) and how those tie into Semantik's features (like showing document info or performing searches). Even if we're not using all of Atlas's coordinated multi-view interactions now, it's useful to understand them for future expansion.

Hover tooltips: When a user hovers over a point in the embedding view, Atlas can show a tooltip near the point. By default, if a **text** column is specified in the data, the tooltip will display that text [26]. For example, if we had a "snippet" or "title" column, Atlas would show that string. In our current use, since we haven't provided a text per point to Atlas, the default tooltip might only show the point's category or nothing meaningful. Instead, we handle the tooltip via the `onTooltip` callback (as discussed in Section C).

Recommendations for tooltips in Semantik: - Provide immediate, useful info to the user without overwhelming them. A good tooltip might be the document name and perhaps an excerpt of the chunk's text. This gives context about what the point represents. For instance: "**Document:** Project Apollo Plan\n**Snippet:** …first few words of the chunk…". - **Latency considerations:** Because hover is frequent (users may move the mouse across many points quickly), we should avoid round-trips to the server on every hover. It's best if we have the essential tooltip info preloaded. Options include: embedding a short snippet or title in the data sent to the frontend (perhaps we can afford, say, 100 bytes per point for a title or summary) or having a precomputed mapping from id to tooltip text that's loaded once. If not feasible, we might simplify the tooltip to just "Document [Name] (Chunk #)" which can be derived from metadata we have (doc name can be loaded once, and chunk number might be part of the id). - Atlas's own tooltip rendering can be customized. If we set up `customTooltip`, we could design a nicer HTML tooltip that maybe shows the content with formatting. But a simpler path: use Atlas's default tooltip mechanism by setting the `text` field in the DataPoint. We could potentially intercept `onTooltip` and augment the DataPoint with a text before setting tooltip state, but Atlas likely expects us to give the text upfront. So possibly, using Mosaic mode with a text column might be needed for Atlas to automatically show it. Without that, we likely implement our own tooltip box in React that follows the mouse position (Atlas gives pixel coordinates via `proxy.location(x,y)` if using customOverlay, or we could derive it from the event). - In short, for now: capture hover events, find the chunk's id, look up its info (from a preloaded map), and display a small tooltip. Keep it fast – no backend calls on hover ideally.

Selection (click & lasso): - **Click selection:** A single click on a point toggles its selection. Atlas supports multiple selection: shift-click (or meta-click) can add/remove points from the selection set [36]. So a user can select several disjoint points. The selection is an array of DataPoints as discussed. - In Semantik's UI, when a point is selected, we should highlight it and possibly show details. Atlas will already highlight selected points visually on the scatter (different stroke or opacity). We should also reflect selection in other parts of the UI – for example, perhaps listing the documents selected or enabling actions (like "Open all selected documents" or "Compare these chunks"). - When a user selects points, we might also want to display aggregated info: if many points from the same category are selected, maybe show "5 chunks from Document X selected". These are product decisions, but technically we have the info via the ids and our metadata.

- **Lasso/Area selection:** Atlas supports dragging a rectangular brush or an arbitrary lasso. In Mosaic mode, dragging a rectangle might create a range filter that Atlas applies to the dataset automatically (e.g. filter X between a and b, Y between c and d). In our manual EmbeddingView usage, when `onRangeSelection` fires and gives a polygon or rect [38], Atlas does not automatically select all those points. We have to handle it. A straightforward approach is:
- Compute which points fall inside the polygon. This can be done in JavaScript. For efficiency, if needed, one could approximate by first filtering points inside the bounding box then doing a point-in-polygon test for each, etc.
- Then call `setSelection` with the array of DataPoints for those points. We can construct DataPoints for them (with x,y,category – and we can include identifier if we have it). Alternatively, since we know indices, we might simply create minimal DataPoint objects with those fields.
- This will cause Atlas to highlight those points as selected.
- We might also directly trigger some UI response: e.g., show a popup "Selected N points. [View Selection] [Search Selection]".
- **Performance tip:** Selecting thousands of points at once is possible; Atlas's rendering can handle it (it will just mark them as selected). But our React state and any subsequent processing should be careful. We probably don't want to individually list a thousand items in the UI; instead maybe show summary or allow exporting selection.

- If the selection is extremely large (say the user lassos half a million points), we should be cautious. It might be better to treat that as a filter operation rather than a "selection" which is meant for a more moderate number of points. In Atlas's full app, a lasso that selects 100k+ points essentially acts like a filter (the rest gets de-emphasized). We might consider imposing a limit or giving a warning if too many points are selected ("Zoom in further or refine selection").

- **Linking selection to backend:** Once points are selected, Semantik can leverage that in a few ways:

- **Metadata lookup:** Fetch info about those points. For instance, call an API with the list of IDs to get their document names, or fetch the actual text if we need to present it.
- **Search within selection:** Perhaps Semantik might allow running a semantic search constrained to the selected subset, or using the selected points as query context.
- **Opening documents:** If one point is selected, perhaps double-click or a separate action could open that document at the chunk's location. We can use the id to route to the document viewer.
- We must ensure the mapping from Atlas selection -> Semantik ID is robust. Best practice: use the `ids` array that came with the projection. For example, if `ids[42] = chunk1234` and the DataPoint we got has identifier 42, we know to fetch info for chunk1234. If Atlas doesn't give the index directly, we can find the index by comparing coordinates (x,y) to our arrays – but floating point comparisons can be tricky. That's why maintaining the `ids` and assuming order consistency is safest. (If we use Mosaic mode with `identifier_column`, Atlas would directly give us the actual

chunk id in DataPoint.identifier, which is ideal [20]. We may consider that by hacking an "identifier" in somehow, but it might require going through EmbeddingViewMosaic.)

Cross-filtering and linked views: - In Atlas's full application, cross-filtering means that all views (charts, table, scatter) are linked by a global filter state. For example, selecting a range in a histogram of "date" will filter the scatter to only points in that date range, and vice versa selecting points in scatter filters the table, etc. [5]. This is achieved by Mosaic's coordinator and DuckDB queries. - In our current integration, we **do not have automatic cross-filters** because we didn't set up the data table in the browser. However, we can mimic some behaviors manually: - For example, we could have UI controls (dropdowns or checkboxes) for metadata filters (like a filter to only show chunks from selected folders or a date slider). When the user changes those, we could trigger the Semantik backend to recompute or filter the projection and send back a subset of points or a new projection. But doing a full round-trip and recompute is heavy; an alternative is simply to hide/filter points on the frontend. We could do that by manipulating the data arrays (e.g., blanking out points or using an Atlas Selection as a filter). - Atlas's EmbeddingViewMosaic component can accept a `filter` prop which is a Mosaic Selection predicate [37]. That would allow showing only a subset without removing data from the array. With plain EmbeddingView, there's no direct filter prop. We would need to remove points from the data and update the component (which is not trivial since it expects static arrays). - One trick: we could maintain a bitmask or selection of active points and use `customOverlay` to dim or hide others. But that's hacky. - Given that Semantik's backend can do filtering anyway (like "only chunks from Document X"), the more straightforward approach is to simply trigger a new projection generation for that subset or reuse the existing projection but just tell the viewer which categories to show. For example, if category already encodes document ID, filtering by document is just selecting that category. - In a future enhancement, we might integrate the **Table** component, which could list chunks and allow multi-select or filter, and tie that to the EmbeddingView. If we used the `Table` with Mosaic coordinator, filtering a column in the table (like search or sort) could filter the scatter. We would need to have the data loaded in DuckDB for that though.

- **Brushing and faceting:** Atlas can handle more advanced interactions like brushing over one chart to filter another, or facet by categories. For instance, one could imagine facetting the embedding by a category (like splitting into small multiples per document). Atlas's Mosaic might allow that (though not sure if built-in). These are not trivial to implement in our custom integration without using more of Atlas's infrastructure, so we likely skip them for now. However, if designing Semantik's future features, it's good to know Atlas is capable if we feed it the data. It means we should structure our code such that adding Mosaic coordination later (if needed) is not impossible.

Summary of interactions for Semantik:

- **Tooltip:** Use it to provide quick, cached info about a point (no heavy calls). Possibly include doc title or chunk summary. Ensure it's fast to update as the user moves the pointer.
- **Single click selection:** Use it to drive detail views or simply highlight that chunk. Possibly treat single-click as "focus this document" in the UI.
- **Multi-selection:** Support shift-click and lasso to allow selecting a set of points, and decide what Semantik does with a set (e.g., compare documents or group actions).
- **Lasso behavior:** Implement capturing the region and updating selection. Keep an eye on performance for large lassos.
- **No built-in cross-filter yet:** If the user wants to filter by metadata, they might do that through Semantik's other UI controls (outside of Atlas). For now, consider the embedding view showing

whatever data set was requested (maybe all chunks, or maybe some pre-filtered subset the user chose before opening it).

- **Coordinate with search:** Atlas provides a search bar in the full app which can do keyword search on the text column and highlight results, or vector search on a query point [6]. In our integration, we haven't exposed that UI, but we can achieve similar outcomes:
- A user might type a query in Semantik's own search bar, which returns results (chunks). We could then highlight those chunks in the embedding view by providing their IDs to the viewer as a selection or a filter. For example, if a search returns 50 chunks, we could set the EmbeddingView's `selection` to those 50 points to highlight them (or use a different mechanism like an overlay to circle them). The Atlas design encourages such linking – e.g., if we had a Searcher object, Atlas could directly integrate it. Without it, we do it manually.
- Alternatively, if a user selects a point and wants nearest neighbors, we can call Semantik's backend for nearest neighbors (embedding search) and then perhaps highlight those in the view or bring them into view. Atlas's UI natively, if given a `neighbors` column or a `searcher`, would show a list of neighbors in a sidebar or tooltip when a point is selected [7] [8]. Since we're not using that, we can simulate it: e.g., on point double-click, call backend for neighbors, and then maybe flash those points or list them in an adjacent panel.

The core concept is **linking** – linking user interactions on the visualization to Semantik's data and actions. Atlas was built with linking in mind (points <-> metadata <-> search queries), so even though we haven't plugged in all the pieces (like the Mosaic linking or built-in searcher), we can still follow that philosophy. As we extend the integration, we should try to maintain a seamless experience: hover to see info, click to select, lasso to filter, and search to highlight – all working in concert. With careful state management and utilization of Atlas's API, this is achievable.

E. Labels, Legends, and Categories

Visualizing an embedding isn't just about points; it's also about providing context – which points belong together, what groups exist, and what they might represent. Embedding Atlas provides mechanisms for coloring points by category, showing legends, and labeling clusters or groups of points. Semantik already computes categories and legends as part of its projection pipeline, so aligning those with Atlas's expectations is key to a coherent visualization.

Categories and color-coding: - In Atlas, the `category` concept is used to assign colors to points [31]. Each point's category value is an integer (0-indexed) that Atlas maps to a color. By default, Atlas has a built-in color palette (often a repeating list of distinct colors). We can override this by providing a `categoryColors` array of color strings [24]. For example, if category 0 = red, 1 = blue, 2 = green, etc., we pass `categoryColors={["#e41a1c","#377eb8","#4daf4a", ...]}` or any colors we choose. Semantik's legend metadata can guide this – if we want consistent coloring (e.g., Document A is always blue across sessions), we can generate or store a palette accordingly. - Semantik's `category` array is already a `Uint8Array` of category indices. One thing to note: if Semantik has more than 256 categories, a Uint8 might overflow (because max 255). However, in many cases the category is by document and the number of documents visualized might be under 256. If not (say 1000 documents), we might need a larger type. Atlas's EmbeddingView strictly mentions `Uint8Array` [31]. This is a limitation if categories > 256. Possibly, they assumed categories would be a cluster ID or similar, not extremely high cardinality. If we truly need more, we might have to find a workaround (e.g., reassign top 255 categories and lump others into "other", or use multiple category channels). Alternatively, the Mosaic version might allow string categories which it internally maps to ints. For now, if we foresee >255 categories, we should consider using fewer (for example, if coloring by document and there are 1000 docs, maybe choose a different coloring scheme, or group smaller ones). - **Legend:** In Atlas's full UI, when a category column is present, a legend is shown listing the categories with their color and count. The Atlas paper even mentions automatic handling of too many categories (like in genomics, "legend capabilities to handle numerous categories") [62]. At the moment, the EmbeddingView component by itself does not display a legend – it's up to us to show one if needed. Semantik's backend already provides a legend mapping (category index → label, and the count of points). We should use that to render a legend in our UI. For example, a small panel next to the chart listing each category's name (document title, or cluster name, etc.) with a colored swatch. We might limit it to top N categories if there are many. - We should ensure the colors we use in `categoryColors` align with how we display the legend. If Atlas's default colors are acceptable, we can replicate those in the legend labels (the Atlas default palette is likely d3 category10 or similar, but to be safe, explicitly set them). - If no category is provided, Atlas would render all points the same color (with maybe a single legend entry). But in Semantik, we typically have a meaningful category, so we should always provide one. However, we might allow the user to change what the category represents (e.g., color by document vs. color by file type). If that's a feature, then Semantik would recompute the `category` array (or have multiple arrays prepared) and update the viewer. The EmbeddingView can be updated with new props – if we give it a new `category` array and `categoryColors`, it will recolor accordingly (we have to call `component.update()` if using imperative API, or just re-render with React which triggers diff).

Label overlays (cluster or group labels): - Embedding Atlas can annotate the visualization with text labels that float over clusters of points. This can be done manually via the `labels` prop (as discussed, we supply an array of positions and text) or automatically. The automatic labeling works when: - We have not provided explicit labels, - and either there's a text column in Mosaic (so Atlas can figure out representative words), - or we provide a `queryClusterLabels` callback that returns a label for given cluster regions [63]. - By default, if using the full app with text, Atlas will find dense regions and pick frequent words to label them [10]. If using EmbeddingView non-mosaic, we can hook into that by providing `queryClusterLabels`. - For Semantik, automatic cluster labeling could be tricky because it would require sending all chunk text to the frontend or at least a significant amount of data for analysis, which we want to avoid. Instead, we might do our own cluster labeling on the backend: for example, we could cluster points (maybe via HDBSCAN or k-means on the embedding) and compute representative terms from the chunks in each cluster. Those labels could then be sent as part of the projection meta. In fact, if Semantik's operations pipeline eventually includes an automated labeling step, we could directly use those labels. - If we have such labels and coordinates for them (perhaps using the cluster's centroid or densest point), we can feed them into the `labels` prop. We should also consider using Atlas's **level and priority** fields in labels [64]. These allow multiple layers of labels (for multi-resolution labeling, e.g., big clusters vs sub-clusters) and controlling which labels have priority in case of overlap. Atlas's auto labeling uses these concepts (level 0 might be coarse cluster labels, level 1 finer, etc., and priority to avoid too many overlapping labels). - If we do not implement any labeling ourselves, we might disable Atlas's auto labels to avoid it attempting something with possibly insufficient data. This can be done via `autoLabelEnabled: false` in config [47]. - Another approach: Use `queryClusterLabels` prop to ask the backend for a label when Atlas finds clusters. How this works: Atlas can perform a density clustering on the fly (it has `findClusters` in WASM [65] [66]). It will then pass the cluster regions (approximate boundaries) to our `queryClusterLabels` function and expect back an array of labels (strings or null) [63]. We could implement `queryClusterLabels` to call a Semantik API, giving it perhaps the points within each cluster (we'd have to invert from region to points – or maybe just give the bounding boxes to backend and let it find points in those bounds via the vector DB). This is a bit complex, and could be slow if clusters are large and many. But it's a neat way to integrate backend smarts (like maybe use an LLM to summarize each cluster's content). This is a potential future improvement; initially we might stick to simpler static labels. - **Choosing good labels:** Whether automated or manual, labels should be concise and meaningful. If labeling by document, using the document title is straightforward. If labeling algorithmic clusters, using a few keywords that frequently appear in that cluster's chunks is effective (Atlas by default probably picks up to 3 words that are frequent and not stopwords [67]). We should ensure stopwords or common terms (like "the", "report", etc.) are filtered out – Atlas uses NLTK stop words by default [68] . - Also consider not overcrowding the view: too many labels can clutter the visualization. The Atlas algorithm tries to place them nicely and avoid overlap by "de-overlapping" (as if on a map) [10] . If we supply too many, we can assign lower priority to some. Alternatively, only label clusters above a certain size (e.g., only label a cluster if it has at least 20 points, so that outlier points don't all get labels). - Color of labels: by default, I believe cluster labels are drawn in a contrasting color with outlines (white text with black outline or vice versa, depending on theme) [33] . We can adjust via theme if needed (properties like `clusterLabelColor`, `clusterLabelOutlineColor` [69]).

Legends (category labels): - If categories correspond to discrete known groups (like documents or file types), we should present a legend to the user. Atlas's full UI would do this automatically in the sidebar: each category value's label and count. In our integration, we have the data from Semantik's `meta` JSON. For example, `meta.legend` might be an object: `{ "0": {"label": "Document A", "count": 50}, "1": {"label": "Document B", "count": 30}, ... }`. We can easily turn that into a legend UI component. - Keep the legend readable: if there are only, say, up to 10 categories, list them all with color swatches. If there are dozens or hundreds (like coloring by document and a user loaded 100 documents), we may need to make the legend scrollable or collapsible. Atlas itself had an open issue about scrollable legends for many categories [70] . A sensible approach is to show top N categories by size and group the rest as "Other" if needed. - If we allow changing the coloring scheme (like user chooses "color by file type" vs "color by document"), we should update the legend accordingly. We might want to reuse the same EmbeddingView but just swap out the category array. That means if currently category encodes doc, and we want to color by file type, we'd recompute a category array of same length where each point's category = that point's file type code. We'd also update the `categoryColors` and legend mapping. This is doable by re-rendering the component with new props. The EmbeddingView will detect changed arrays and update colors.

Naming conventions and palettes: - Atlas likely uses a fixed palette (like Tableau or d3 Category10 for first few, then repeats or uses variations). Semantik might already have a color scheme (maybe from the old projection view if one existed). It could be nice to align for consistency. For example, if Semantik traditionally showed PDFs as blue and Word docs as green, we can set those explicitly. - For categories that are ordinal or bucketed (like age buckets, or year), sometimes one might use a gradient or ordered palette. Atlas's category color mapping is categorical (discrete). If we needed something like a continuous color scale, we would have to map the values to category bins ourselves. For now, assume categories are nominal. - The legend labels themselves should be user-friendly. Use short names, maybe with count in parentheses. E.g., "Document A (50)". - The ordering of categories in the legend – perhaps descending by count or alphabetical by label. If one category is a catch-all (like "Other"), maybe list it last.

In summary, to align Semantik with Atlas: - Use **categories** to color points in a meaningful way (document, cluster, etc.). - Provide a **legend** in the UI from Semantik's metadata so users know what each color means. - Utilize **labels** on the embedding to annotate important clusters or groups. This could be as simple as labeling each document's cluster with the doc name (if points cluster by doc), or something more advanced later. - Follow Atlas's guidance for label placement and generation: avoid too many labels, focus on larger clusters, and use clear, short text. The end result should be that a user can glance at the map and identify major regions by name, and use the legend to decode colors, which greatly improves interpretability over an unlabeled sea of points.

F. Performance and Scaling

Embedding Atlas was explicitly designed to handle large datasets (millions of points) in the browser, but there are practical limits. Semantik, being a document analysis tool, might have to visualize anywhere from a few hundred points (small document set) to potentially a million or more (a huge corpus with many chunks). We should understand Atlas's performance characteristics and how to optimize our integration for smooth experience.

Rendering performance (points vs. density): Atlas leverages GPU rendering (WebGPU if available, otherwise WebGL2) to draw points very efficiently [71] . In point mode, each point is essentially a tiny drawn shape (Atlas uses a shader to draw circles, with special handling for transparency to avoid overlap issues [72]). The system is capable of maintaining interactive frame rates (>= 60 FPS) up to a certain number of points. According to Apple's info, Embedding Atlas can handle *a few million points* smoothly [71] . The research paper's benchmark indicated it could go to around 4 million points with 60 FPS and about 10 million with ~25 FPS on a modern machine (M1 Pro) [11] . So, practically: - If we keep point counts under, say, 1-2 million, users with decent hardware should see very fluid interactions (panning, zooming). - Between 2-5 million, it might still be fine, but we should test on target hardware (some Semantik users might not have a GPU as powerful as an M1 Pro; WebGPU support may vary, and WebGL2 could be slightly slower). - Above ~5 million, we might start to see frame drops or memory issues. 10 million is mentioned as possible but at reduced FPS – that's an extreme case.

Memory constraints: Each point consists of two float32 (8 bytes total for x and y) plus one byte for category (if Uint8). So roughly 9 bytes per point (not counting overhead). 1 million points ≈ 9 MB of raw coordinate data, 5 million ≈ 45 MB. That's just the data arrays; the GPU buffers might duplicate some of this, and if there are other structures (like indexes or if text data is present, etc.), memory usage increases. Most modern browsers can handle tens of MB easily, but pushing towards hundreds of MB could strain lower-end machines or cause GC pauses. Also, if many points are selected or have tooltips, those structures are small per point but negligible in comparison.

Density mode benefits: Atlas's density mode is useful when points completely overcrowd the view. In density mode, instead of drawing each point, Atlas computes a density grid and draws contour lines and a heatmap of point density [43] . This is done with a GPU compute shader for efficiency [73] . The performance benefit is that rendering a fixed grid is actually lighter than rendering millions of individual points, especially when zoomed out (when many points overlap). Also, it provides a clearer visualization of clusters when you have massive overlaps. - Atlas can dynamically switch to density mode based on a threshold of average point density (points per pixel) [44] . We can rely on that or manually switch if we detect extremely large datasets. - When zooming in, it will switch back to points as points become sparser. - For Semantik, if a user loads, say, 500k+ points, the initial view (zoomed out) might be extremely dense. Instead of seeing a solid blob or a lot of overplotting, Atlas might show a nice contour map highlighting high-density regions. This is great for identifying hotspots. We should ensure this feature is enabled (it is by default unless turned off). - We might also explicitly encourage its use in documentation to users: e.g., "When viewing more than a million points, the map will display density contours for clarity. Zoom in to see individual points." - **Point size:** Atlas auto-adjusts point size to manage density (smaller points when dense to reduce overdraw) [45] .

This, combined with density mode, ensures performance stays reasonable. We generally shouldn't override `pointSize` unless we find the auto logic isn't suitable.

Sampling strategies: On the Semantik backend, we have control over how many points we include in a projection. In some cases, showing *all* chunks might not be necessary or desirable. For example, if a user has a million chunks, showing all might overwhelm the visualization. Semantik might choose to sample or limit to, say, the 100k most relevant or a uniform sample across documents. The operations pipeline could include such sampling info (which it already might – the meta likely includes a note if sampling was applied). - If sampling is done, Atlas doesn't need to know; it just sees whatever points it's given. But we should communicate to the user (in the UI or meta) that this is a sampled view if that's the case. - If not sampling, rely on Atlas's capability but be mindful of the user's machine limits. It might be worth providing an option: maybe Semantik could allow the user to choose the level of detail (e.g., "View all points" vs "view sampled for performance"). However, given Atlas's impressive performance, we might err on including as much as possible until proven problematic.

WebGPU vs WebGL: Atlas uses WebGPU when available [71]. WebGPU is the modern graphics API that offers better performance and more advanced compute (like the density calculation) on new browsers. Chrome, Edge, and Safari have WebGPU support as of 2025 in varying degrees. If WebGPU isn't available (older browser or settings), Atlas falls back to WebGL2 (which might handle the point rendering but possibly not the exact same features like order-independent transparency as well, but the fallback is said to exist [6]). - We should test on the range of browsers our users use. It's possible some corporate environments might not have the latest browser with WebGPU enabled. Atlas on WebGL2 is still likely faster than typical JS plotting libraries, but its performance ceiling might be lower (maybe a million points instead of multi-millions). - If we detect performance issues on certain setups, we could proactively downsample or encourage density mode usage.

Large dataset considerations: - Atlas allows using a remote DuckDB (server) for data heavier than what the browser can handle, via the `--duckdb server` mode [74]. That's more relevant if we used the full app. For us, since our backend is effectively the "server", we could in future implement something where Atlas queries our backend for subsets (like tile-based loading or chunk streaming). But currently, our approach is to load everything needed into memory at once. This is fine up to moderate sizes. For extremely large sets (e.g., 10 million points, which might be >80 MB of just coordinates), we might consider chunking – but it's non-trivial with Atlas's current API. - Alternatively, we could let the backend do a heavier aggregation. For instance, if a user wants to see an embedding of 10 million chunks, maybe first show a bird's-eye aggregated view (like cluster centroids or a heatmap) rather than every single point. Atlas's own density approach somewhat does this visually, but we could go further by not even sending all points (just send clusters or a sampled subset). - The Atlas paper suggests that one of its aims is to remove the need for pre-sampling by handling it in visualization [75]. So if we trust it, perhaps we can try pushing it and rely on density mode to carry the load. But always keep an eye on memory.

Performance tuning from Atlas docs: - Use of **cache:** There is a `cache` prop that can store intermediate results [76]. Possibly relevant if reusing the same data repeatedly – maybe not crucial for us. - Using `viewportState` to sync multiple EmbeddingViews, not so much an issue unless we have multiple views (we don't currently). - The heavy computations available (UMAP, KNN) are in WASM and multi-threaded. We bypass those by doing them server-side. That means the initial load time in our case is dominated by fetching data from the server and rendering, rather than computing the UMAP in browser. This is good for reproducibility and consistency [77], and likely faster for large sets if our server is beefy. Atlas's WASM UMAP is quite fast though (C++ via Emscripten). - **Threading:** When WebAssembly threads are enabled (usually on desktop browsers), Atlas can utilize multiple cores for computing clusters or etc. But for just rendering, the GPU does the work.

Limits in selection and interactions: - Selecting extremely many points (100k+) might slow down JavaScript side (large arrays to process). Atlas likely optimizes the rendering of selection by using bitmasks on GPU. But transferring 100k points selection to some UI element or processing them in JS (like building a list) could choke. So we should handle big selections carefully (maybe just show summary stats instead of listing all points). - The tooltip event fires every time you move to a new point; if you fly over thousands of points quickly, that's thousands of events. Our handling should be lightweight to not backlog (i.e., avoid heavy DOM updates on every event; perhaps use a throttle or just replace the same tooltip element's content). - Cross-filtering (if implemented later) means running queries on possibly large data. DuckDB in WASM can handle fairly large data in-memory, but complex filters might be slower. But since we likely push filtering to backend or avoid heavy local queries, this might not apply.

Recommended point counts and strategies: - For typical usage, aiming at **hundreds of thousands of points** will work well out of the box. If expecting to regularly go into millions, test and consider a fallback (like a warning or auto-sample above a threshold). - If a user's dataset is extremely large (multiple millions), consider chunking the analysis: maybe separate by document or time, or ask them to refine query. In Semantik's UI, it might be logical not to try to visualize an entire massive archive at once, but rather one collection or one query's results at a time. - Use **density mode** advantage: it keeps the UI informative and responsive for large N. Ensure the `minimumDensity` threshold is not set too high so that it engages when needed [44] (by default it's likely tuned well – it measures points per square pixel). - Keep an eye on **browser memory**: We should test memory usage. If we see significant memory usage, we might need to free resources (Atlas does have a `destroy()` method to clean up when unmounting [78] – we should call that if the user closes the projection view to avoid memory leaks). - Also, instruct future integrators that adding more data to the browser (like adding the text for all chunks for tooltips) can balloon memory. One reason we avoid loading chunk text is not just privacy but also size – a million chunks of text could be gigabytes. So we consciously limit what goes to the front.

In conclusion, Embedding Atlas can handle Semantik's data scales impressively, but we should **use its features wisely**: leverage density rendering for cluttered views, possibly limit extremely large visualizations through sampling, and always test on realistic hardware. By doing heavy lifting on the backend (embedding computation, projection), we've already offloaded some work from Atlas, which will help it remain fluid in the browser [79]. The combination of a precomputed projection and Atlas's optimized rendering should give users a fast, smooth experience even as data sizes grow.

G. Feature Map: Atlas Features vs. Semantik Integration

Embedding Atlas is a broad toolkit, but Semantik only needs parts of it, since Semantik already has its own backend capabilities. Let's enumerate key features of Embedding Atlas and categorize how Semantik uses or could use them:

1. **High-dimensional embedding computation** – *Atlas capability:* Given raw data (text, images, etc.), Atlas (via its Python package) can compute embeddings using default or custom models (e.g., SentenceTransformers) [80]. *Semantik status:* Semantik already does this during ingestion. We do **not** use Atlas to compute embeddings; we feed Atlas the already-computed embeddings (or rather, the

projection of them). **Action:** No need to use Atlas's embedding computation. (If anything, we might ensure we use similar models so that results are comparable, but that's a data science detail.)

2. **Dimensionality reduction (UMAP projection)** – *Atlas:* Can run UMAP in-browser (WASM, multi-threaded) to reduce embeddings to 2D [81] [82]. *Semantik:* This happens in our projection pipeline (server-side UMAP/PCA). We pass Atlas the 2D coords. We **do not use** Atlas's `createUMAP` or projection algorithms at runtime. **Action:** Stick with Semantik's projection for consistency and because we may use custom settings. We benefit from Atlas's ability to accept precomputed coordinates.

3. **Interactive scatter plot (EmbeddingView)** – *Atlas:* Core UI component to visualize points, with GPU acceleration and options for millions of points, density mode, etc. *Semantik:* We fully **use this**. This is the centerpiece of our integration – we embed the scatter plot in our web app. Atlas's ability to handle large point sets and efficient zoom/pan is crucial. **Action:** Continue leveraging EmbeddingView for visualizing chunk embeddings.

4. **Semantic search (text search & vector nearest neighbors)** – *Atlas:* The UI supports searching text (full-text search on the text column, powered by DuckDB's full-text search or similar) and finding nearest neighbors for a selected point (if precomputed or via an approximate KNN index in WASM) [9] [8]. *Semantik:* Semantic search is a core Semantik backend feature (we query the vector DB for nearest neighbors and re-rank, etc.). Currently, we have not enabled search through the Atlas UI. We likely have a separate search interface outside the projection. **Potential integration:** We could utilize Atlas's `Searcher` interface to plug in Semantik's search: implement a `Searcher` object whose `search(query)` calls Semantik's text search API and returns results, and whose `nearestNeighbors(id)` calls our vector DB for similar chunks. If we did this and passed it to the `<EmbeddingAtlas searcher={...} />` prop, Atlas might provide a search bar and highlight results. Right now, we haven't done that, so **not used today**. But this is a valuable feature for the future – it would make the projection view more dynamic (type a keyword, see relevant chunks light up on the map, or click a point and see neighbors directly). So, consider it **"could be valuable"** to implement a custom searcher integration.

5. **Cross-filtering across multiple views** – *Atlas:* Enables coordinated multiple views: e.g., histograms of metadata, a table view, and the scatter, all linked by a common filter state [5]. You can click a bar in a histogram to filter points, or lasso points to filter the table, etc. *Semantik:* Currently, we do not use Atlas's built-in charts or automatic cross-filter. We rely on Semantik's own filtering UI (if any) and typically would run a new query or projection for filtered data. So currently **not used**. **Could be valuable:** Yes, as Semantik grows, having on-the-fly filters (by document type, date, etc.) on the visualization would be great. Implementing that would mean bringing some of the data into the front-end (or making fast backend endpoints and tying them in). Possibly, we could incorporate a lightweight version: e.g., a filter dropdown that simply filters categories (which we can apply by hiding certain categories). But the full power (like arbitrary numeric filters, multi-value filters) is not yet realized. This remains a **future enhancement** idea.

6. **Table view of data points** – *Atlas:* Has a virtualized table component that can list metadata for each point, allow sorting, etc., and is linked to selection [83]. *Semantik:* We haven't integrated Atlas's table. Semantik has its own ways to view documents and chunks (like a document detail view). We could consider showing a list of chunks (maybe with snippet) corresponding to what's visible or selected.

But currently, **not used**. If we wanted a quick way to implement that, using Atlas's Table with our data might be possible (if we load data in the front end). However, since chunk listing is something our backend can do (and maybe with richer info and actions), we might stick to custom UI. So table is **not used today** and perhaps **not a priority** unless we aim for that cross-filter experience.

7. **Automated clustering & cluster labels** – *Atlas:* Automatically finds clusters in the 2D distribution and labels them with representative terms [4] [10]. *Semantik:* We have our own notion of grouping (e.g., by doc) but we haven't been doing unsupervised clustering for label suggestions on the fly. We do color by known categories, but Atlas's idea of surfacing "interesting clusters" could highlight patterns that aren't just one document. For example, maybe chunks from different docs that talk about the same topic cluster together – Atlas might label that region "budget" if that word is common. We are **not using** Atlas's auto-label yet (likely disabled or it doesn't trigger without text data). **Could be valuable:** Yes, especially if users dump many documents and want to discover themes. To utilize this, we'd either need to supply a text column or implement `queryClusterLabels`. It's a feature to consider down the road, once basic functionality is solid. It might require sending some textual data or doing backend calls to label clusters. So medium priority, interesting for "discovery" use cases.

8. **Local (client-side) operation vs. Backend operation** – *Atlas:* Can function fully offline, doing heavy computation (embedding, UMAP, clustering, search) in the browser via WASM and WebGPU [1] [13]. *Semantik:* We've chosen a hybrid – heavy computations on backend, using Atlas for visualization. So in terms of features:

9. Local embedding & UMAP: **not used** (Semantik does it on server).
10. Local KNN search (HNSW in WASM): **not used** (Semantik uses vector DB).
11. Local SQL filter (DuckDB): **not used** (Semantik's backend handles queries).
12. These are replaced by Semantik's services, which is fine. The advantage is we control those better and can scale them on the server side. The disadvantage is we don't get the full "offline mode" that Atlas could provide (but likely not needed in our web app context).

13. This means certain Atlas UI elements (like the search box or filter UI) either aren't present or need to be wired to our backend if we want them.

14. **Exporting and state saving** – *Atlas:* The EmbeddingAtlas component supports exporting the current selection (as JSON/CSV) and even downloading the entire application state as an archive [84]. This is part of the built-in UI (for example, a user can save the state and load it later, or export selected points' data). *Semantik:* We have our own mechanisms (if any) for exporting data. We haven't integrated Atlas's export callbacks. We could potentially wire `onExportSelection` to trigger a backend export (like deliver a JSON of those chunks) [85], but we have not done so. It's **not used currently**. For a future agent: if asked, they could implement those callbacks to perhaps allow one-click CSV download of selected results using our backend data. Not a priority right now.

15. **Multi-dataset comparison** – *Atlas:* Not explicitly mentioned in docs, but one could imagine loading two embeddings side by side (maybe using two EmbeddingView components or some toggle). Atlas doesn't directly support multiple datasets in one view, but you can switch state. *Semantik:* If we needed to compare two projections (say before/after some action), we might do that in our UI by having two viewer components. Atlas can share viewport state between multiple EmbeddingViews if

needed (via the `viewportState` prop) [86]. Right now, we don't have this feature in Semantik. Potentially, though, comparing different runs could be useful (like two versions of the corpus). Consider it a niche use-case.

To summarize the feature mapping: - **Already handled by Semantik backend (Atlas not used):** Embedding generation, projection (UMAP/t-SNE), vector search (KNN), data filtering. - **Atlas features we actively use now:** The interactive scatter plot (with points/density), category-based coloring, basic selection & tooltip interactions. - **Atlas features not used yet but potentially useful:** Atlas's integrated search bar and nearest-neighbor UI, automatic cluster labeling, and coordinated multi-view filtering (charts & table). - **Atlas features likely not needed or lower priority:** The built-in Table (since Semantik has other ways to show data), and one-click export (we can implement if requested, but not core to analysis).

By understanding which parts of Atlas we lean on and which we replace with our own, we can avoid conflicts (e.g., make sure to disable Atlas's duplicate functionalities) and identify opportunities (like maybe turning on Atlas's search UI with our backend). This helps ensure we get the best of both worlds: Semantik's robust backend and Atlas's polished frontend.

H. Recommendations for Semantik

Given what we've learned from Embedding Atlas's capabilities and Semantik's current approach, here are concrete recommendations to guide the integration and future improvements:

1. **Leverage Atlas's strengths in visualization and interaction:** Semantik should continue to use Embedding Atlas primarily as a visualization **widget** – meaning, let Atlas handle rendering millions of points, showing density or scatter as appropriate, and capturing user interactions (zoom, hover, select) with high performance. We should avoid reimplementing things like custom canvas rendering or manually managing WebGL – Atlas does this well out of the box [71]. Focus engineering effort on connecting those interactions to Semantik's logic.

2. **Ensure robust ID mapping (one-to-one) between Atlas and Semantik data:** This is critical. Every point in Atlas must correspond to a unique Semantik chunk or document. If using `EmbeddingView`, we rely on array index alignment – so never shuffle or filter the arrays in ways that desynchronize them from the `ids` mapping. If using `EmbeddingViewMosaic`, set the `identifier` column so Atlas directly gives the chunk IDs on selection [20]. An invariant to maintain: *the index of a point in the x/y arrays is the index of that point's ID in the ids array.* If any processing (like filtering points) changes that, update all arrays consistently. This avoids any confusion when user selects a point – we can immediately resolve it to the correct chunk. **Pitfall to avoid:** Off-by-one or mismatched indexing. For example, if you were to drop some points (say filtering by quality), make sure to drop their entries in all arrays, not just coordinates.

3. **Use Atlas's category mechanism for coloring, but avoid excessive categories:** Semantik should continue to use the `category` array to color points by meaningful group (document, cluster, etc.). However, avoid feeding Atlas a scenario with thousands of unique categories, as a legend of that size is not usable and might degrade performance. If the natural grouping (like doc ID) yields too many, consider grouping smaller ones under an "Other" category or allow the user to pick a higher-level grouping. Atlas expects category as Uint8 (0-255) [23]; if we risk exceeding that, we may need to

upgrade to Mosaic mode or break the dataset. **Pitfall to avoid:** If category values exceed 255, Atlas's behavior might be incorrect (could wrap around). So put in place a check or enforce limit.

4. **Implement a custom legend UI on the frontend:** Since we aren't using the full Atlas app's sidebar, Semantik's frontend should display a legend based on the data in the `meta` JSON (category labels and counts). This legend should update if the coloring changes (e.g., if in future we allow switching color by different fields). It should use the same colors as Atlas (`categoryColors` prop). This ensures users know what each color means – a fundamental part of cluster interpretation. Keep it simple and scrollable if categories are many. This is relatively easy to implement and greatly improves usability.

5. **Improve tooltip content without heavy backend calls:** We recommend populating the tooltip with at least some identifying information that can be obtained with low overhead. Options:

6. Include a short text per point when preparing the projection data. For example, Semantik might store the first sentence of each chunk or a summary. Even a 10-15 word excerpt could be enough. This could be included in a "text" array parallel to x/y (though EmbeddingView doesn't accept that directly, we could manage it side-by-side and use a custom tooltip).
7. Or, store just the document title and maybe section heading for each chunk. Document title is highly useful context and often just a few bytes per chunk if repeated (it could be repeated across many chunks of the same doc, but we could compress that by storing an index to a title).
8. If including any text for every point is infeasible memory-wise, at least ensure we have a quick way to map an ID to a title or doc name (maybe preload a map of docID -> title, then on tooltip use the chunk's docID to get it).
9. **Avoid** doing a network request for each hover. That would be slow and also put load on the backend. If absolutely necessary (like user pauses on a point), we could lazy-load a snippet, but try to push as much as possible in front.

10. Possibly use Atlas's `customTooltip` with our own component that, upon receiving a tooltip event with an identifier, looks up the info in a cached map and displays nicely formatted text [51] [52]. This gives full control over layout (we could bold the title, etc.).

11. **Use Atlas's selection model to drive Semantik actions (don't reinvent selection logic):** Atlas already handles multi-select and lasso; rely on its `onSelection` to get what's selected [35] rather than building separate UI controls for selection. Once we get the selected IDs, integrate with Semantik's existing features:

12. Provide a button or context menu for "Open selected documents" or "Compare selected chunks" if applicable.
13. Perhaps integrate with Semantik's search: e.g., a button "Find similar to selection" could take the embeddings of selected points (we have their IDs, can fetch vectors or use centroid) and do a new search. But that's a new feature idea.

14. The main point: **the pipeline for selection should be** – Atlas -> our React state -> call backend if needed or update other components. **Invariant:** Atlas's selected DataPoint indices map to exactly those chunks – maintain the mapping as per point 2.

15. **Disable or override Atlas features that conflict with Semantik's backend responsibilities:** Specifically:

16. Disable Atlas's own full-text search if we're not using it. By default, if a `text` column is present and no custom `searcher`, Atlas enables a search UI that does client-side filtering [9]. In our case, we likely did not expose that UI. If we use the high-level `EmbeddingAtlas` component in the future, we should provide `searcher={null}` to disable it if we plan to handle search externally [9]. Or better, provide a custom searcher that calls our backend.
17. If we use `EmbeddingAtlas` or Mosaic components, set `searcher.nearestNeighbors` to use our backend or else Atlas might use any provided neighbor column to show neighbors [8]. If we have not provided neighbors and no searcher, it might fall back to nothing, which is fine.
18. Turn off `autoLabelEnabled` if we do not provide proper data for it [47]. Otherwise Atlas might attempt to generate labels from whatever it has (which could be weird or blank if no text is there).
19. Ensure that if we do not want Atlas to try computing UMAP (say we accidentally pass vectors and not coords), we explicitly disable projection (`--disable-projection` in CLI terms, or simply always pass coords in UI so it never tries) [87].

20. Essentially, **avoid double-computation**: Semantik computes, Atlas displays. Keep that separation clear.

21. **Consider implementing a custom Searcher to unify search experience:** If we want users to seamlessly jump between visual exploration and semantic search, hooking into Atlas's search interface could be great. This means writing a `Searcher` object with methods for fullTextSearch, vectorSearch, and nearestNeighbors (based on Atlas's `Searcher` interface, which likely expects certain function signatures). This `Searcher` would call Semantik's backend APIs (which might return IDs of results and maybe snippets). Atlas would then highlight those results in the scatter and possibly show them in the table or a side panel. This might be a moderate effort but would leverage Atlas's front-end for search UI rather than building our own highlight mechanism. The trade-off is complexity and ensuring the async calls are handled. If we find it too complex, we can do a simpler approach: have an external search UI and then use Atlas selection to highlight results (which we essentially manage ourselves). But long-term, using Atlas's built-in pipeline (the search bar in the UI) could make for a cleaner integration.

22. **Performance tuning on large data:**

23. If we know a projection has extremely many points, prefer to start in density mode for clarity and speed. We can set `config.mode = "density"` initially [42]. The user can always toggle to points by zooming in. This prevents initial overload.
24. Keep an eye on the browser's memory. For very large projections, consider chunking data transfer (maybe stream the arrays if needed). Right now, likely we send the whole arrays in one go over HTTP. For a million points, that's fine (tens of MB). For 5 million, that might be ~45 MB, which is still borderline but maybe okay on a LAN or modern connection. If networks or memory become an issue, we might compress the arrays (they're pretty compressible if many zeros or patterns) or consider sending only a sample for initial view.
25. Atlas's WASM backend (DuckDB, KNN, etc.) is not used, but if we ever move some filtering to front, remember to use `--duckdb wasm` (default) or appropriate mode. But likely not needed.

26. Use `window.requestIdleCallback` or similar to defer heavy JS processing (like computing lasso inclusion) until after UI renders, to keep interface snappy.

27. Test on different devices, including an average Windows laptop (which might have worse WebGL performance than a Mac with Metal/WebGPU). If some users have to use CPU rendering fallback, then perhaps limit point count or encourage using density more.

28. **Mimic Atlas's UI patterns in Semantik's wrapper:**

    - Atlas's UI shows a status bar (with point count, selection count, etc.) at the bottom by default [88]. In our integration, we might not have that (especially using only EmbeddingView, which might not show it unless we enable via `theme.statusBar=true`). We might consider adding a simple status line ourselves: e.g., "Showing 50,000 points. 200 selected." This gives user feedback. Atlas can show it if we toggle that theme option, but since we didn't incorporate the whole UI, we might do it manually.
    - Provide clear affordances for switching modes if needed (Atlas auto switches to density, but perhaps a legend indicating "Density mode" could be useful so user knows those contours are not additional data but a visualization).
    - Maintain consistency: if we use Atlas's dark mode, ensure our surrounding UI is also dark, etc. (Atlas can adapt to light/dark via `colorScheme` or follow system).
    - Pay attention to **order-independent transparency (OIT)**: Atlas handles overlapping points nicely so that dense areas don't just become an opaque blob [89]. To utilize this, keep `config` default (it's already on). If we overlay any custom drawing, we should be mindful not to ruin that. For example, if highlighting selected points by drawing on top, better to use Atlas's built-in selection highlighting or a custom overlay that still leverages the `proxy` to draw with proper blending.

29. **Avoid duplicating features unless needed:** For example, we might think "let's implement our own cluster detection on backend and draw polygons around clusters." But note, Atlas already finds clusters and can give boundaries [65] [90]. If we want to visualize clusters, maybe we can ask Atlas for them (via `findClusters`) or rely on its density contours. Only implement separately if we need more semantic clustering than density-based. Basically, use Atlas's algorithmic features if they meet our needs (they are quite optimized and integrated).

    - Conversely, avoid using Atlas's internal features if they conflict with Semantik's approach or data guarantees. For example, don't use Atlas's internal nearest neighbor if we require results to come from our vector DB which might have more up-to-date data or a custom ranking. Always decide where a feature should live (front vs back) and stick to one to prevent divergence.

30. **Plan for future UI expansions with Atlas:**

    - If we anticipate adding charts (histograms of metadata) in the UI, consider gradually moving toward using Mosaic. For instance, maybe we want a histogram of document dates alongside the scatter. We could either custom-build that or use Atlas's Mosaic and Chart components if they exist. The Atlas paper indicates automatic charts for numeric and categorical fields [5]. If we go down that route, we might need to actually load data into DuckDB (which could be heavy). Alternatively, have backend precompute histograms and just plot with a normal chart

lib. This is a design decision: do we try to harness Atlas as a full analytics UI, or just as a scatterplot widget? Right now, it's the latter, but we should keep the door open.
  - If future developers/agents work on adding such features, they should read this guide first (see Section I) for context on what's done on backend vs front.

Trade-offs to communicate: - *Trade-off 1:* **Using Atlas's full app vs. custom integration.** We chose custom integration to have control and use our backend. If Semantik ever wanted a quicker path to some features (like local usage without server), theoretically one could run Atlas's CLI and have a user use that. But that doesn't integrate with multi-tenant or our database, so likely not an option. It's good to explicitly note: **Do not attempt to use Atlas's CLI or Python server inside Semantik's environment** – it conflicts with our architecture. - *Trade-off 2:* **Search integration:** Using Atlas's search UI is nice but requires writing a `Searcher` wrapper and maybe dealing with differences (like Atlas expects results in certain format). Alternatively, keeping search separate might be simpler initially but less integrated visually. If we integrate it, we must also handle the case of the user typing a query that returns many results – Atlas might try to highlight all, which could be like selecting hundreds of points at once. Our backend might also return a relevance score – Atlas's default might not handle that except maybe by ordering results. So we have to adapt. - *Trade-off 3:* **Auto vs. manual cluster labels:** Letting Atlas auto-label might yield insightful labels but we lose some control (and it requires text data in front). Manual labeling (from backend clusters) gives us control but requires implementing clustering ourselves and might not be as dynamic (Atlas's are dynamic with zoom possibly). We decided for now manual or none. If in future someone wants auto, they must accept sending more data to front or doing more complex integration with `queryClusterLabels`.

To wrap up the recommendations: **prioritize the essentials** (IDs mapping, tooltips content, legend, selection mapping) to make the current integration solid. Then, **gradually enhance** by tapping into more of Atlas's features (searcher, auto labels, cross-filters) as needed, always ensuring those features align with Semantik's backend (so we don't duplicate effort or break consistency of data). And always test under realistic workloads to catch any performance pitfalls early (like memory spikes or event handling issues).

I. "For Future LLM Implementers" Summary

Hello, future implementer! If you are an AI agent or developer now looking at Semantik's codebase and this guide, here's a quick summary of the most important points and invariants you must uphold when integrating or modifying the Embedding Atlas functionality in Semantik:

- **Read Section B and C first:** They contain the mapping of Semantik data to Atlas data model, and how we use the React API. This will orient you so you don't mistakenly use Atlas in a way that conflicts with our backend. Always remember: *Semantik's backend is the source of truth for data and computation; Atlas is the frontend display.* Don't inadvertently offload something to Atlas that the backend already handles (e.g., computing new embeddings or doing filtering that bypasses our logic).

- **Key invariant: Index alignment for IDs.** The index of each point in the `x`/`y` arrays corresponds exactly to an entry in the `ids` array which maps to a chunk/document [19]. When Atlas gives you a `DataPoint` with `identifier` (or if you use the index), use that to retrieve the correct item. **Never break this alignment.** If you filter or reorder points for any reason, adjust all arrays (x, y, category, ids, etc.) consistently. Many integration bugs come from misaligned indices.

- **Labels and legends must reflect the category data:** If you change how categories are assigned or introduce a new coloring scheme, update the legend (user-facing labels for categories) accordingly. Keep the `categoryColors` prop in sync with those labels [24]. If you allow multiple category modes (like color by X vs Y), ensure the legend and colors update and that Atlas is given the updated category array promptly.

- **Do not rely on Atlas for heavy data operations that Semantik does.** For example, don't try to have Atlas compute a new UMAP on the fly for a different subset – Semantik should do that. Don't use Atlas's DuckDB to run complex queries on our data – our backend or search engine is better suited and more secure for that. Atlas's algorithms (UMAP, clustering, etc.) are great for a purely client-side scenario, but in Semantik we maintain control server-side for reproducibility and multi-user consistency [77].

- **When enabling new Atlas features, integrate with Semantik's backend logic:**

- If you enable **search** in the Atlas UI (e.g., via a `Searcher` object), make sure it calls Semantik's search endpoints. Validate that the results highlight or select the correct points (likely by returning their IDs or indices to Atlas). Test both text search and vector neighbor search if implemented.
- If you use `queryClusterLabels` for auto-labeling, ensure it calls a Semantik service that can provide a label. Possibly you'd send the IDs of points in a cluster to the backend for analysis. Make sure it's asynchronous and won't freeze the UI (Atlas expects a Promise) [91]. Also ensure the labels returned align with what the user understands (maybe use document titles or frequent terms).

- If you add **cross-filters** or incorporate Atlas's Table or charts, decide where filtering logic lives. If using Atlas's, you might need to load data into DuckDB. Alternatively, intercept filter events and call backend. Keep it consistent (don't have Atlas filter one way and backend another with conflicting states).

- **Pitfalls to avoid:**

- *Performance pitfalls:* Don't send extremely large data to the front unnecessarily (like full text of all chunks). This will bog down the browser. Use summaries or on-demand loading for detailed data (for example, only load a chunk's full text when the user explicitly opens it, not on hover).
- *Double selection state:* Ensure we treat Atlas's selection as the source of truth for what's selected in the scatter. We shouldn't maintain a separate selection state that can diverge. Instead, if other parts of the app (like a list of documents) allow selecting documents, consider linking that to the EmbeddingView by programmatically setting Atlas's `selection` prop to highlight those, or vice versa. One consistent selection model is easier to reason about.
- *Updating props incorrectly:* When updating the EmbeddingView (say user loads a new projection), either unmount and remount it with new data or use the `component.update()` if using the imperative API [92]. Changing large props like data arrays should ideally be done by remount for cleanliness (React unmount/mount will call Atlas's destroy and re-init). Memory leaks can occur if we, for example, forget to call destroy on an old component or keep old huge arrays around.
- *Mismatched array lengths:* If you pass arrays of different lengths to `EmbeddingView.data` (say x and y length differ, or category length doesn't match x), Atlas might error or behave unpredictably. Always ensure those arrays are same length (equal to number of points).

- *Incorrect type usage:* If you inadvertently pass normal JS arrays or the wrong TypedArray type, Atlas may not throw an obvious error but could malfunction or perform very poorly. Always use Float32Array for coords, Uint8Array for categories [93] [31].

- **Don't override Atlas's rendering logic with hacks:** For example, instead of trying to manually draw on the canvas for custom highlights, use Atlas's provided hooks (like selection or customOverlay) [94] [95]. They ensure things like coordinate transforms and layering are handled. If you need to add something visual (say an annotation), prefer `customOverlay` so you can get correct positioning via `proxy.location()` [96]. This avoids issues with differing zoom or pan.

- **State persistence:** If relevant, note that Atlas allows saving the UI state (zoom, pan, selection, etc.) and reloading it [97]. Semantik might implement its own state save (like user can save a projection state). If you implement such features, use Atlas's `onStateChange` to get the state and maybe store it, and `initialState` to restore [98] [99]. But ensure any state restoration aligns with current data (IDs might need to match, etc.). If not needed, you can ignore this.

- **Testing and debugging:** Use the documentation references:

- If something in the UI isn't working (e.g., selection not highlighting), consult Atlas docs for that prop or event to ensure we use it correctly (citations given above point to relevant lines).
- If performance is lagging, check if we accidentally disabled WebGPU or triggered a slow path (like a huge DOM element overlaying the canvas can slow things).

- If tooltips or labels look odd, ensure our data (like category indices or label positions) are correct and not NaN or mis-scaled.

- **Communication with backend:** Always consider the round-trip cost. If a user action in Atlas requires data from backend (like clicking a point to open document), design it asynchronously with feedback (e.g., show a loading indicator or at least don't freeze UI). Atlas's events are asynchronous-friendly (you can call backend and then update UI when ready). Use debouncing for continuous interactions (like maybe a lasso might fire many intermediate events – though likely it only fires once after completion).

- **Keep consistency:** If Semantik's backend categorizes or clusters data one way, do not apply a different clustering logic on the front that would confuse the user. For example, if we color by document, don't at the same time use Atlas's auto cluster labels that might label regions by topic – that could conflict (user sees color grouping by doc but a label that says "Topic X" spanning multiple docs – might be fine, might be confusing). If mixing, make sure to explain in UI (like a tooltip on the label to clarify what it represents). Generally, one dimension of grouping at a time is shown (color by X, labeled by X if possible).

To conclude, integrating Embedding Atlas into Semantik can greatly enhance the user experience, but it requires careful alignment between the front-end visualization and the back-end data logic. By following this guide – respecting data alignments, using Atlas's API as intended, and avoiding overlapping responsibilities – you'll ensure a smooth, powerful projection visualization that feels like a natural extension of Semantik. Good luck, and happy coding!

100 6

---

1 3 11 Embedding Atlas: Apple's Open-Source Tool for Exploring Large-Scale Embeddings Locally - InfoQ
https://www.infoq.com/news/2025/11/embedding-atlas/

2 58 100 Overview | Embedding Atlas
https://apple.github.io/embedding-atlas/overview.html

4 Embedding Atlas: Low-Friction, Interactive Embedding Visualization - Apple Machine Learning Research
https://machinelearning.apple.com/research/embedding-atlas

5 10 13 14 16 43 73 75 83 Embedding Atlas: Low-Friction, Interactive Embedding Visualization
https://arxiv.org/pdf/2505.06386

6 12 71 72 89 Embedding Atlas
https://apple.github.io/embedding-atlas/

7 15 74 77 79 80 87 Command Line Utility | Embedding Atlas
https://apple.github.io/embedding-atlas/tool.html

8 9 17 18 19 25 59 60 61 84 85 97 98 99 EmbeddingAtlas | Embedding Atlas
https://apple.github.io/embedding-atlas/embedding-atlas.html

20 23 26 27 37 56 57 94 95 EmbeddingViewMosaic | Embedding Atlas
https://apple.github.io/embedding-atlas/embedding-view-mosaic.html

21 22 65 66 81 82 90 Algorithms | Embedding Atlas
https://apple.github.io/embedding-atlas/algorithms.html

24 28 29 30 31 32 33 34 35 36 38 39 40 41 42 44 45 46 47 48 49 50 51 52 53 54 55 63 64 67 68 69 76 78 86 88 91 92 93 96 EmbeddingView | Embedding Atlas
https://apple.github.io/embedding-atlas/embedding-view.html

62 Apple Embedding Atlas: Making Sense of Millions Through ...
https://joshuaberkowitz.us/blog/github-repos-8/apple-embedding-atlas-making-sense-of-millions-through-interactive-visualization-1769

70 Issues · apple/embedding-atlas - GitHub
https://github.com/apple/embedding-atlas/issues