ChatGPT

SPARQL Query UI Web Component Specification

Overview

This specification describes a **SPARQL Query Web Component** inspired by YASGUI (Yet Another SPARQL GUI). The component provides a web-based interface for composing and executing SPARQL queries against any SPARQL endpoint, and for visualizing the results. It is implemented in **Svelte 5** and uses the **IBM Carbon Design System** for a clean, consistent UI. High performance is achieved through the **SVAR Svelte DataGrid v2** for rendering results, enabling virtual scrolling and dynamic data loading for large result sets [1] [2]. The component is designed to be lightweight, minimalistic, and fast, while offering rich features akin to YASGUI. All functionalities adhere to the **SPARQL 1.2 Protocol** for query execution and result formats, ensuring compatibility with current and future SPARQL endpoints [3].

Scope and Objectives

Scope: This component focuses on SPARQL **query** composition and result visualization. It supports SPARQL 1.2 Query operations (SELECT, ASK, CONSTRUCT, DESCRIBE) and can be extended to handle SPARQL Update if needed. The UI is suitable both as an embeddable component in larger web applications and as a standalone query interface (similar to yasgui.org). No authentication mechanisms are included in this version (public endpoints assumed).

Objectives and Goals:
- Provide a **modern SPARQL query editor** with syntax highlighting, error checking, and autocompletion to assist users in writing queries accurately.
- Enable intuitive **prefix management** and autocompletion for CURIEs (compact URIs), making it easy to use common RDF prefixes.
- Allow specifying and switching **SPARQL endpoints** easily, supporting integration into various deployment contexts (fixed endpoint in an app or user-provided endpoint in standalone mode).
- Present query results in both a **tabular view** (with rich interaction features) and a **raw view** (showing the raw response in various formats), with options to download results.
- Ensure **high performance** on large result sets through virtualization (infinite scroll) instead of traditional paging [1] [2].
- Maintain a **clean, consistent UI/UX** by following Carbon Design System guidelines (typography, spacing, theming) and ensuring responsiveness and accessibility.
- Achieve **lightweight** implementation (minimal external dependencies aside from Carbon and DataGrid libraries) and **fast load times**, suitable for embedding in other projects.
- Enforce **code quality** by including automated tests (unit, integration, and UI tests) and following Svelte 5 best practices for maintainability.

Core Technical Requirements

- **Framework:** Implemented in **Svelte 5** (latest version, e.g. v5.41.x) to leverage its reactivity and compile-time optimizations. The component should be packaged for reuse (e.g., as an npm library and optionally as a Web Component custom element if needed).
- **Design System:** Use **IBM Carbon Design System** (Svelte Carbon components) for all UI elements (buttons, dropdowns, modals, tabs, etc.). This ensures a polished look-and-feel and

consistency across the interface. Carbon's grid and responsive layout principles should be followed. The component must support Carbon's standard themes ("White", "Gray 10", "Gray 90", "Gray 100") for both light and dark mode compatibility, allowing easy theming switches.

- **Data Grid Library:** Utilize **SVAR Svelte DataGrid v2** (npm package `wx-svelte-grid`) for rendering tabular results. This grid provides **virtual scrolling** and **dynamic data loading** for performance [2], plus rich features like sorting, filtering, and editing. Its use ensures that even result sets with tens of thousands of rows can be handled smoothly in the browser [1] [2].
- **Code Editor Library:** Integrate a syntax-highlighting text editor for SPARQL. The editor should provide SPARQL **syntax highlighting** and basic **error indication** (e.g., mismatched braces or quotes). Using **CodeMirror 6** (or another lightweight editor) with a SPARQL mode is recommended, as YASGUI's YASQE was based on CodeMirror [4]. The editor should be configured within a Svelte component for seamless integration.
- **Testing & Tooling:** Use modern build tools (e.g., Vite) and include **automated testing** frameworks (e.g., Vitest or Jest for unit tests, and Playwright or Cypress for end-to-end testing of the UI). Ensure the component can be built and bundled for distribution via npm. Provide TypeScript typings (if using TS) for all public APIs of the component. Continuous integration should run tests and linters to enforce quality.

Functional Requirements

1. SPARQL Editor & Query Composition

1.1 Syntax Highlighting and Editing: The query editor must highlight SPARQL keywords, functions, strings, and punctuation in context. It should support indentation and multiline editing. Basic **syntax error checking** is required – for example, flagging an unclosed brace or quotes. This helps users catch mistakes early.

1.2 Autocompletion: Provide intelligent autocompletion in the editor to speed up query writing:
- **Prefix Autocomplete:** When typing a prefix declaration or using a prefix in a query, suggest common prefixes or those already declared. For example, typing "`rdf:`" could suggest expanding to "`PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>`". Leverage a service like prefix.cc or a built-in list of popular prefixes for suggestions [5]. The editor should automatically append prefix declarations if the user selects a suggested prefix not yet in the query (similar to YASQE behavior [6]).
- **IRI Autocomplete:** After a known prefix (e.g., `rdf:`), support autocompletion of terms if possible. (This could be an extensible hook – e.g., using Linked Open Vocabularies (LOV) APIs for class/property suggestions, similar to YASQE's property/class autocompletion [7].) Out of the box, the component can include basic support for RDF/RDFS/OWL vocabularies and allow pluggable autocompletion sources for specific domains.
- **Syntax Hints:** Provide completion for SPARQL keywords (SELECT, WHERE, FILTER, etc.) and functions as the user types. Keyboard shortcuts (e.g., **Ctrl+Space** to trigger suggestions) should be available.

1.3 Query Templates & Examples: The component should allow starting with a default query template. For a new query tab, if no prior query is present, it can pre-fill a basic template (including some default PREFIX declarations and a simple `SELECT * WHERE {  } LIMIT 100` as an example). This gives users a starting point. The design should also allow integrators to provide **predefined example queries** (optionally): e.g., a list of queries that users can load with one click, which then populate the editor. (This is inspired by YASGUI and tools like Wikidata Query Service that offer example queries, though implementing a gallery of examples is optional and can be added by the host application.)

1.4 Prefix Management: The editor must make it easy to manage prefixes:
- There should be a way to **define common default prefixes** (configurable via props or initialization).

For instance, the integrator can supply a set of prefix mappings (prefix -> URI) that will always be present in new query tabs. Common prefixes like `rdf, rdfs, owl, xsd, foaf, skos` etc. can be included by default.
- The UI should allow viewing and editing prefix declarations. If many prefixes are declared, an option to **collapse/expand the prefix list** at the top of the query (similar to YASQE's prefix folding) is desirable to keep the editor uncluttered [8].
- **Retrieving prefixes from endpoint:** Although not mandated by the SPARQL protocol, some endpoints provide service descriptions or custom endpoints to list defined prefixes. The component will **support an extension hook** for prefix discovery: for example, it could accept a function or callback that, given the current endpoint URL, returns a set of prefixes (perhaps by querying a special graph or using a known convention). This way, integrators can implement fetching prefixes from a Fuseki config dataset or other store (if available) and then programmatically add those prefixes via the component's API. By default, no automatic prefix fetching is done (to remain protocol-agnostic), but the design makes it possible to integrate.

1.5 SPARQL 1.2 Compliance: The editor should recognize SPARQL 1.2 syntax. This includes support for new syntax/features from SPARQL 1.1 and 1.2 (like subqueries, VALUES, etc.). It should also allow switching between **Query** mode and **Update** mode if needed: for example, if a user starts a query with `INSERT` or `DELETE`, the component should treat it as an Update operation. (In practice, this means sending it to the update endpoint if one is configured, and possibly not expecting a result set but just success/failure.) Initially, focus is on query operations; update support can be minimal (just sending the request and showing success or errors, without needing a result view).
- The component must send queries following the SPARQL Protocol: support **HTTP GET** (with URL-encoded query parameter) and **HTTP POST** (with either URL-encoded form or direct posting of query) as appropriate [9]. By default, SELECT/ASK queries can use GET (for smaller queries), falling back to POST for very large queries or for UPDATE operations. The HTTP method should be configurable.
- Include appropriate HTTP headers (Accept header, etc.) based on the selected result format (see Output Formats below). Also support setting **request headers** if needed (e.g., for custom authentication tokens in the future or required headers). The configuration should allow the integrator to specify headers globally or per request if necessary [10].
- Follow redirects or HTTP 3XX responses from endpoints, and handle CORS (the component should make requests via browser fetch/XHR; thus the endpoint must permit cross-origin calls or be same-origin). If a query fails due to network issues or returns a SPARQL error, display a clear error message to the user (including any error details from the server).

1.6 Query Execution Controls: Provide a **"Run Query"** button (Carbon primary button) to execute the query. Also support a keyboard shortcut (e.g., **Ctrl+Enter** or **Cmd+Enter**) to run the query from the editor focus [11]. When a query is running, indicate activity (e.g., a spinner on the button or a progress bar at top). If a previous query is still running, decide whether to cancel it (if the endpoint supports cancellation via aborting the HTTP request) or disallow a new run until it finishes. For long-running queries, consider a **cancel/abort** control if feasible (though SPARQL protocol doesn't define cancellation, aborting the HTTP request might suffice).

1.7 Multiple Query Tabs: The interface should allow managing multiple queries simultaneously in a tabbed UI (mirroring YASGUI's tabbed design). Each **query tab** has its own editor content, endpoint selection, and result view state. Users can open a new tab (e.g., via a "+" icon on the tab bar). The default behavior for a new tab is to start with a blank or default query and use the same endpoint as the current tab (this was a YASGUI feature: optionally copy endpoint on new tab [12]). The user can close tabs and switch between them; the component should persist each tab's content as the user navigates.
- The tab strip should be a Carbon Tabs component styled to fit the application (likely positioned at the top of the editor pane). Tabs can be labeled (untitled tabs can default to "Query 1", "Query 2", or a user-

specified name if editing the tab name is allowed).
- Internally, maintain state per tab (query text, selected endpoint, selected result format, etc.). Optionally, use **browser local storage** to persist open tabs and their queries, so that if the user revisits the page, their last queries are restored (this behavior can be enabled by default, similar to YASGUI's 30-day persistency for queries [13], with an option to disable for privacy).

2. Endpoint Selection and Configuration

2.1 Endpoint Input: The component must allow the user or integrator to specify the SPARQL endpoint URL to query. In **standalone mode**, the UI should prominently show an **Endpoint URL input field** (for example, a Carbon TextInput at the top of the interface, possibly within an endpoint selector component). The user can paste or type an endpoint URL here. If the endpoint requires a specific dataset or path, the full URL including any path should be used.

2.2 Endpoint List (Catalogue): To improve usability, the component can offer an **autocomplete dropdown** of known endpoints. For instance, as the user types, it might suggest well-known public endpoints (DBpedia, Wikidata, etc.) or any endpoints provided via configuration. The integrator of the component can supply a list of endpoints (with optional metadata like a friendly name or description) to populate this suggestions list [14]. If no list is provided, the field just behaves like a normal URL input. In a controlled environment (e.g., inside a custom app), the endpoint input might be hidden or fixed to a single endpoint (the integrator should have the option to disable the endpoint chooser entirely, similar to YASGUI's configuration to hide it when a single endpoint is targeted [15] [16]).

2.3 Default Endpoint: The component should accept a default endpoint setting (via prop or initialization option). This endpoint will be pre-filled when the component loads (especially if local storage has no remembered endpoint yet). In standalone usage, this might default to a common endpoint (or none, requiring user input). In integrated usage, the host application will likely set this to their SPARQL service URL.

2.4 Endpoint Validation: Basic validation on the endpoint input should be done (e.g., ensure it is a valid URL). If the user tries to run a query with no endpoint specified, an error should be shown ("No endpoint specified"). Optionally, the component might test the endpoint by performing a lightweight request (like an OPTIONS or a service description query) to check availability and CORS, but this is not strictly required on every entry (could be noisy).

2.5 Multiple Datasets/Graphs: (Optional for future) If the endpoint is a SPARQL service with multiple datasets (like Fuseki with multiple named datasets), the endpoint URL might include the dataset name. This component will treat each endpoint URL as a separate endpoint; managing multiple datasets is up to the user to input the correct URL. (We do not include a specialized UI for selecting datasets or graphs in this version, aside from writing FROM/FROM NAMED in the query itself.)

3. Query Result Visualization – Tabular View

After a query is executed (particularly SELECT queries), results should be displayed in a **tabular format** by default for user-friendly browsing. The component will use the Svelte DataGrid to render this table. Key requirements for the tabular results view:

3.1 Data Parsing: The component must parse the SPARQL query results from the HTTP response into a structured format for the grid. It should support all standard SPARQL Result Set formats for SELECT/ASK: **JSON**, **XML**, **CSV**, **TSV** [17]. The recommendation is to use JSON as the default (as it's compact and easy to parse), but the component can handle others if needed. For convenience, the application might

request JSON from the endpoint and then internally convert to table format. (For CSV/TSV, simple parsing can be done; for XML, an XML parser may be needed – but since the component can request JSON, we primarily focus on JSON parsing.) The table should have one column per variable and one row per result binding [18]. If a variable is unbound in a result, the cell can be left blank or marked as "NULL" (for clarity, an empty cell or a `–` symbol can indicate no value). ASK queries return a boolean; these can be displayed as a single cell or a textual "True/False" message.

3.2 Virtual Scrolling (Infinite Loading): Instead of conventional pagination, the results table must implement **infinite scroll / dynamic loading**. This means initially only a portion of results may be rendered, and as the user scrolls down, more rows are loaded dynamically [1] [2]. This improves performance and user experience for large result sets. Two possible approaches can be combined:
- **Frontend virtualization:** The DataGrid by default only renders visible rows, which is already taken care of by the virtual DOM. This allows smooth scroll with, say, 100k rows, without DOM overload.
- **Chunked data fetching:** For extremely large results, loading all results in the browser might not be feasible memory-wise. The component should support fetching results in chunks (e.g., 1000 rows at a time). If the SPARQL endpoint supports streaming or chunked responses, that's ideal, but most don't. Instead, the component can internally re-issue the query with an adjusted `LIMIT/OFFSET` when the user scrolls near the bottom of the loaded data, to retrieve the next batch. For example, initial query could use `LIMIT 1000`; when 90% scrolled, fire another query for `OFFSET 1000 LIMIT 1000`, and so on. This continues until no more results are returned. This logic should be abstracted such that the user simply scrolls and sees new data appear, without manual page switching. (Integrators should be warned that using ORDER BY without a stable order may lead to inconsistent overlapping data when chunking; documentation can note best practices, but implementing chunking is still valuable).
- The grid should display a loading indicator when fetching additional data. If an error occurs during additional fetch, it should alert the user (and possibly allow retrying).
- Note: If chunked loading is complex to implement initially, the component may default to loading the full result into memory (if it's not too large) but must still **display** it via virtual scrolling to avoid rendering all rows at once. In either case, there will be no page buttons – the user just scrolls. This aligns with modern UX expectations and leverages the DataGrid's capabilities.

3.3 Columns and Cells: Each result variable corresponds to a column. The column header should display the variable name (sans the leading `?`). For readability, consider styling variable names in lowercase or as-is from query. The grid should **auto-size column widths to fit content** by default, so that each column is just wide enough for the longest value in it (within some maximum) [19]. This prevents excessive empty space and reduces the need for horizontal scrolling. If values are very long (e.g., long URIs or literals), an **ellipsis truncation** mechanism should kick in: cells should show a truncated value with "…" (ellipsis) if they exceed a certain length, and a tooltip on hover should show the full content. This is the "ellipse" feature mentioned – it ensures the table remains readable without huge cells. Users can toggle this if needed (see "Simple vs Full view" below).

3.4 IRI Display and Linking: Many query results will contain IRIs (resources, predicates, etc.). The component should display IRIs intelligently:
- By default, use **prefix abbreviations** for IRIs when possible. If the query had a prefix declaration that matches the IRI, use the prefix form. For example, `http://www.w3.org/1999/02/22-rdf-syntax-ns#type` would display as `rdf:type` if `rdf:` prefix is known [20]. If no prefix is available, the component may display the full URI or just a QName-like contraction (e.g., showing only the local name and an ellipsis for the namespace). Since the user specifically mentioned "simple view" and "ellipse", we interpret **"Simple view"** as showing a shortened version of IRIs (prefixed or truncated forms) for simplicity. There should be an option to toggle between **Simple view (abbreviated)** and **Full view** of IRIs. In full view, each cell would show the entire URI string (with wrapping or horizontal scroll in the cell if necessary). In simple view (the default), use prefixes or truncate with ellipsis to keep cells compact.

- All IRIs displayed should be **clickable links**. Clicking on a link opens it in a new browser tab (to avoid losing the query UI) [21] . This allows users to navigate to resource pages if the IRIs dereference to useful information. For safety, use `target="_blank"` and `rel="noopener"` for such links. (The open-in-new-window behavior can be configurable, but default to true as per YASR's approach [21]).
- If the SPARQL result includes **literals** with datatypes or language tags, display them with their value and an indication of type/lang. For instance, `"Alice"@en` or `5.3^^xsd:decimal`. The UI could format these subtly (perhaps color code or smaller font for the type) or simply include them in text. This is a detail to ensure that users see exactly what the data is.
- If the result includes an **RDF term that is an HTML literal** (datatype `rdf:HTML`), the component **must not** render the HTML directly without sanitization (to prevent XSS). By default, such content can be shown as raw HTML source or safely sanitized and rendered. (Triply's YASGUI, for example, sanitizes and renders HTML result cells [22]). Initially, we can simply display the HTML as text or provide a toggle to render it, using a sanitizer like DOMPurify if needed. Security is paramount here.

3.5 Sorting: The table should support **column sorting** by clicking on a column header. The DataGrid provides multi-column sorting, but for SPARQL results it might be simpler to allow single-column sort by default (to avoid user confusion). Clicking a header the first time sorts ascending by that column's values, clicking again sorts descending, and a third click clears sorting. The sort should be performed **client-side** on the loaded data. (If data is partially loaded in chunks, we can either sort only within loaded portion, or fetch all for sorting – to keep it simple, once a user triggers sort, we might choose to load the full result to ensure correctness, or disable sorting until all data loaded. This design decision should be documented). Sorting should handle different data types properly if possible (e.g., numbers vs strings). The UI should indicate the sort order (e.g., an arrow icon up/down in the header [23]).

3.6 Filtering: By default, the results view is shown without any filters (all results visible). The user should have the option to **toggle on filters** for refining the visible data. When filters are enabled, a filter row or filter UI appears:
- The simplest approach is a **per-column filter** input at the top of each column (DataGrid supports header filters [24] [25]). For text columns, this could be a text input that filters rows to those containing the typed substring. The DataGrid's default is AND logic for multiple columns (a row must match all active filters) [24] ; this is acceptable. Advanced users could configure OR logic if needed, but our UI can keep it simple.
- Alternatively (or additionally), a single **global search** field can be provided (as YASGUI did [26]). However, since we have per-column filters available, the global search might be redundant. We will prioritize column filters for precision.
- The toggle to show/hide filters can be a button (e.g., a "Filter" icon in the toolbar above the table). When activated, the filter row appears. Users can then enter filter text under any column. Filtering happens client-side on the data already fetched. (If only a portion of data is loaded and the user tries to filter, we might load all data to ensure the filter applies to the full result set, or at least warn that only the loaded subset is being filtered. For a complete solution, best is to load everything upon filter activation, as filtering on a partial set could be misleading.)
- Filters should match case-insensitively by default for strings, and perhaps allow simple operators for numeric columns (like ">5" to filter greater than 5, etc., if the grid supports it easily). We can implement basic substring matching initially.

3.7 Column Resizing & Reordering: The user must be able to **resize columns** and **reorder columns** for better viewing, leveraging DataGrid features:
- **Resizable:** Each column header can be clicked and dragged at its edge to adjust width [27] . The grid should update layout accordingly and remember the new width (at least during the session). Double-clicking the edge could trigger auto-fit to content for that column (if supported).
- **Reorderable:** Users can drag a column header left or right to rearrange the column order [28] . A visual

indicator should show the insertion point. This does not change the underlying data, just the presentation order. The new order persists for that tab's session.
- **Show/Hide Columns:** Provide a UI for toggling visibility of columns (especially useful if there are many variables). For example, a "Columns" dropdown listing all variables with checkboxes can allow the user to hide certain columns [29]. Hidden columns should not be lost – just not displayed until re-enabled. The DataGrid may have built-in support for column visibility toggling (it mentions extensibility and perhaps a context menu). A simple approach: a button in the toolbar opens a small menu listing columns. The state of hidden/visible should be tracked. By default, all columns from the query are visible.

3.8 Pagination (Not used): The component explicitly avoids traditional pagination controls (page numbers). YASGUI's original table view paginated results 50 at a time by default [30], but here we prefer continuous scrolling. Therefore, **no page size selector** or page number display is shown. Instead, the dynamic loading covers that use case. Internally, however, we may use page-like chunks for loading as described in 3.2. If needed, an integrator could impose a maximum rows limit or re-enable pagination, but by spec default we use infinite scroll.

3.9 Simple View vs Full View: As noted, **Simple View** will be the default presentation (with prefix abbreviations and truncated values with ellipses). The UI should allow switching to **Full View** where all cell contents are shown in full (no ellipsis truncation, and IRIs shown as full URI strings). This could be a toggle button labeled "Show full IRIs/values" or similar. In Full View, columns might expand significantly, potentially requiring horizontal scrolling; the component should allow a horizontal scrollbar if content exceeds the container width. Full View helps when users need to copy full URIs or read long literals, whereas Simple View is for quick browsing. The toggle should be easily accessible (perhaps in the toolbar above the table).

3.10 No Results & ASK results: Handle two special cases gracefully:
- If a SELECT query returns **zero results**, the table should still render the columns (if any were projected) or a message "No results found." Possibly show an empty table with just headers, and maybe a placeholder row with "–". The user should clearly see that the query ran but nothing matched.
- If an ASK query is executed (which returns a boolean true/false), the tabular view could either show a single cell with "True" or "False", or simply display a labeled message in the results area (e.g., a Carbon notification or large text "Result: True"). YASGUI handled ASK via a special plugin that just wrote "True/False" [31]. We can implement a simple handling: if query type is ASK, bypass the grid and show a centered text result. The raw view (if chosen) can show the raw boolean JSON/XML as well for completeness.

3.11 Performance Considerations: The table should be able to handle at least tens of thousands of rows without freezing the UI. The virtualization helps here, but we also ensure that heavy operations (like converting JSON to grid data or applying filters) are done efficiently. Use requestAnimationFrame or debouncing for operations like scrolling events if needed to maintain 60fps scroll [32]. Also, large result sets might require a lot of memory; we may consider streaming parsing of JSON results if possible. However, given the use of dynamic queries or chunk loading, memory use can be controlled. The UI should warn or limit if a result set is extremely large (e.g., over 100k rows) and perhaps suggest adding a LIMIT to the query.

4. Query Result Visualization – Raw View and Downloads

In addition to the interactive table, users should be able to view the **raw results** as returned by the endpoint, and download them in various formats.

4.1 Raw Response View: Provide a **"Raw" results mode** where the raw response from the endpoint is shown as text (or formatted accordingly). This is useful for developers who want to see JSON, XML, or RDF directly. The raw view should use a monospace font and possibly syntax highlighting for JSON/XML. We can reuse the CodeMirror editor in read-only mode for this, or a simpler `<pre>` block with styling. If the result is JSON or XML, format it with indentation for readability. For RDF formats (Turtle, etc.), just display as plain text (with syntax highlight if available).

4.2 Switching Views: The user can switch between **"Table" view and "Raw" view** using a toggle or tabs in the results pane. A Carbon tab set or segmented control can be used with two options: **Results** and **Raw**, for example. By default, after running a query, **Results (Table)** is selected for SELECT queries, and **Raw** might be automatically selected for graph queries (CONSTRUCT/DESCRIBE) since those don't naturally form a table. The user can switch at any time. The component should remember the chosen view per tab so if they run a new query in the same tab it can stick to whichever view was last used.

4.3 Raw Content Formats: The raw view should reflect exactly the content returned by the server, or the content that would be returned for a chosen format. If the query was executed with a certain Accept header (e.g., JSON for SELECT, Turtle for CONSTRUCT), then the raw view shows that response. Optionally, we can allow the user to **choose the raw format** from a dropdown and re-run the query to get that format. For example, in a SELECT query scenario, a dropdown of "JSON, XML, CSV, TSV" could be offered. If the user switches to CSV, the component would refetch the query with `Accept: text/csv` (or a `?output=csv` param for endpoints like Fuseki) and then show the raw CSV text. Similarly for graph results: offer "Turtle, JSON-LD, RDF/XML, N-Triples" [17]. This gives power users control over the format. The UI should indicate if switching format triggers a new query execution. It might be wise to only enable this on the Raw view (since the Table view inherently needs a structured format like JSON to populate it properly; if a user explicitly wants CSV, they likely intend to download it or view raw).

4.4 Download Results: Users must be able to **download the query results** easily in their desired format. This can be implemented as a **Download button** or menu in the UI. For instance, a "Download" button in the Raw view (as suggested by YASGUI docs [33]) that directly downloads the currently viewed raw data as a file. Alternatively, a dropdown menu "Download as..." with options (JSON, CSV, etc.) can be provided in the results toolbar. When a user selects a format to download, the component should either use the already fetched data (if we have it in that format) or perform a new GET request with `Accept` header for that format and prompt a file download.
- Filenames can be automatically generated (e.g., `results.csv` or `results.json`), perhaps including the query name or endpoint name for clarity.
- If the raw data is already available (e.g., we have the JSON from initial fetch and user wants JSON), we can create a Blob and trigger a download without an extra request. For other formats, it might be simpler to do a direct window location download (with the endpoint's `?output=` param if supported).
- Ensure that downloads include all results (if we used chunked loading in the table, we should re-run the full query for download to not miss data). The UI should warn if it's about to fetch a potentially very large file.

4.5 Result Size and Timeout Warnings: If a query returns an extremely large result (e.g., millions of rows or very large RDF graph), the component should handle it gracefully. It might not be feasible to display millions of rows even with virtualization due to memory limits. The component can set a configurable **maximum rows threshold** (say 100,000) beyond which it will not automatically load everything. If the threshold is exceeded, it can show a warning like "Result set is too large to display completely. Please refine your query." and only load the first chunk. The user could still download the full results if needed. Also, if a query takes too long, consider implementing a timeout (configurable, e.g., 60 seconds) after which the request is aborted and an error shown. These safeguards maintain responsiveness.

5. User Interface & Interaction Details

5.1 Layout: The component's UI can be organized as follows for a standalone usage: a top toolbar (with Endpoint input and perhaps buttons), a main area split into query editor (top pane) and results view (bottom pane), and possibly a resizable splitter between them so the user can give more space to results or the editor. If multiple tabs are enabled, the query editor area will include the tab bar above the editor. The Carbon grid or flexbox can be used for the layout. All sections should be responsive: on smaller screens, it may stack (editor on top of results) or allow toggling visibility (maybe hide results until query run, etc., but primarily this will be used on desktop).

5.2 Toolbar Controls: In the header/toolbar of the component, include:
- The **Endpoint selector** (as described in section 2).
- The **"Run" button** to execute the query (with an icon, e.g., play icon).
- A **format selector** (for output format) if we want the user to choose result format before running (this could also be placed next to the Run button or in the results section as discussed). Possibly simplify by always fetching JSON for table display, and let downloads handle other formats, to avoid confusion.
- Optionally, buttons for **"New Tab"**, **"Delete Tab"** (if tabs are not shown in a separate UI element). If using Carbon Tabs for query tabs, those have their own UI for adding/removing which we can integrate (Carbon doesn't natively have a plus on tabs, but we can add a custom tab or a button).
- A **toggle for view mode** (Simple/Full) and **toggle for filters** in the results toolbar rather than the main toolbar. It might make sense to have a secondary toolbar above the table for actions related to result view (like Filter toggle, Simple view toggle, Download menu). We should keep the main top toolbar focused on query execution and endpoint selection.

5.3 Carbon Design Elements: Use Carbon components for all UI elements where applicable: - Buttons (Run, Download, etc.) using `<Button>` with appropriate kind (primary, secondary).
- Text Input for Endpoint field, possibly with an embedded dropdown (Carbon's ComboBox or Dropdown can be considered for endpoint suggestions).
- Tabs for query tabs and possibly for switching between Table/Raw views (or a toggle instead of tabs for the view mode if preferred).
- DataGrid is not a Carbon component, but it can be styled to match (we might need to ensure its default styles mesh with Carbon's themes, adjusting colors for header background, text, etc., to look like a Carbon data table).
- Modal dialogs (if any, for example if we provide an "About" or help modal, or confirm on large downloads, etc.).
- Notifications or Toasts for error messages or warnings (Carbon has a Notification component that can be used to display query errors, e.g., a red error notification with the error text).

5.4 Accessibility: All interactive elements must be accessible (proper ARIA labels, focus order, keyboard controls). The Carbon components already handle a lot of accessibility. The DataGrid v2.1 is stated to be WAI-ARIA compliant [34] . We must ensure that:
- The editor textarea is focusable and usable by keyboard (CodeMirror typically is). Provide a label (visually hidden) for the editor (e.g., "SPARQL Query Editor").
- Tab navigation works: user can Tab to endpoint field, Run button, editor, results, etc. and activate controls via keyboard.
- Color contrast should meet AA standards (Carbon themes handle this generally).
- If possible, ensure screen readers can read out results in a logical way (the grid might need to be in an accessible table format; DataGrid's accessibility features [35] should cover row/column roles, etc.).
- Provide visible focus indicators on focusable elements (Carbon does this by default).
- Avoid using color alone to convey information (e.g., use icons or text for error vs success).

5.5 Localization: All text in the UI must be externalizable for translation. The component should not have hard-coded English strings. Instead, use a localization approach (for example, a dictionary or i18n library). Svelte 5 doesn't have built-in i18n, but we can use a simple store or context with translations. At minimum, provide a JSON or JS object of strings (English by default) that can be replaced or extended. Strings to localize include: UI labels like "Run", "Endpoint", "Download", "No results found", "True/False", tooltips, error messages, etc. We will start with English, but ensure the mechanism to switch languages is documented. Also consider number and date formatting for any data that might need it (though SPARQL results are mostly strings and numbers as strings). The Carbon components might also need locale setting (Carbon has locales for things like date pickers if used, but we likely don't need those here).

5.6 Theming: The component will support IBM Carbon's theming. The four default themes ("White" – light theme, "Gray 10" – light with gray backgrounds, "Gray 90" and "Gray 100" – dark themes) should all display correctly. This likely means using Carbon's CSS classes or theme context. We should verify the DataGrid and CodeMirror editor also adjust for dark backgrounds (e.g., CodeMirror may need a dark theme CSS if Carbon dark theme is used, to ensure text is visible). The component should either follow a global Carbon theme automatically (if the host app sets the CSS variables) or accept a `theme` prop to force a theme. It should load any required Carbon theme styles (Carbon Svelte likely provides a way to include styles for each theme or a default). Ensuring the **contrast and styling of custom parts** (like the DataGrid cells) align with the theme is important. For example, Carbon tables in dark theme have darker backgrounds for cells; we should apply similar style to DataGrid cells in dark mode. Possibly, we can use Carbon's tokens or CSS custom properties for colors.

6. Integration & Deployment

6.1 Usage as Component: The solution will be delivered as an NPM package (e.g., `sparql-query-ui`) that can be installed and imported into Svelte (or other) projects. The component can be used in a Svelte app like: `<SparqlQueryUI endpoint="https://example.com/sparql" />` with various props for customization. It should also be buildable as a standalone bundle (and potentially as a Web Component via Svelte's custom element build, if needed, so it could be dropped into non-Svelte pages). Documentation will be provided to show how to integrate it.

6.2 Standalone Application/Page: In addition to being a reusable component, it's desirable to have a standalone deployment (similar to yasgui.org) where the component runs on its own. This could be a simple HTML page that imports the compiled JS/CSS and initializes the component in a full-page view. The user would then just input an endpoint and use it. This page should be able to take an endpoint (and optionally a query) from the URL, so that links can be shared. For example, the page URL could accept query parameters like `?endpoint=<URL>&query=<encoded SPARQL>` to load a specific query on startup. This allows sharing queries by URL, akin to YASGUI's share functionality. Implementing the encoding (likely base64 or URI-encoded) for longer queries may be necessary. At minimum, ensure that if the page is refreshed, the state (endpoint and query) can be restored from the URL or local storage so the user doesn't lose their work.

6.3 Deployment Targets: The component's bundle should be optimized and tree-shakable. Svelte 5 ensures a small runtime overhead. Rely on Carbon's CSS (possibly Sass) and the DataGrid's code. We should avoid pulling in heavy dependencies beyond those – CodeMirror is perhaps the largest for editor. The final JS and CSS bundle should be as small as reasonably possible for a feature-rich app (aim for a few hundred KB gzipped, ideally). We will use dynamic imports or code-splitting if certain features (like advanced autocompletion or big libraries) are optional.

6.4 Testing & Quality: Every core feature will have associated tests. For example: - A unit test for the function that parses SPARQL JSON results into table data. - A test that typing a prefix and selecting autocompletion inserts the prefix declaration. - Integration test that simulates running a sample query against a known endpoint (maybe a mock or a public endpoint) and checks that results appear in the table and raw view. - Visual regression tests for the UI could be done with screenshot comparisons (optional).
Additionally, before release, perform manual testing across major browsers (Chrome, Firefox, Safari, Edge) to ensure compatibility. The component should also degrade gracefully if JavaScript is disabled (likely it won't function, but the page should at least inform the user that JS is required).

6.5 Documentation: Alongside the code, provide documentation including: how to install, how to use in an app, description of props/options, how to style (theming), how to localize, and how to run tests. Also document any extension points (like custom autocompletion or prefix loading callbacks). Ensure the specification of supporting **SPARQL 1.2 Protocol** is met by referencing W3C specs in documentation for advanced behavior (like how to set up an update endpoint if needed, or service description usage).

Non-Functional Requirements

Performance and Efficiency

- **Client Performance:** The UI must remain responsive even with large queries and result sets. The use of virtualization (only rendering what's needed) is crucial. All heavy computations (parsing, data conversion) should be done efficiently in JavaScript, possibly offloading to web workers if parsing very large files (not initially, but consider for future). Scrolling performance should target 60 FPS [32] – meaning avoid layout thrashing in scroll handlers, and use fast DOM update patterns (Svelte's reactivity and the DataGrid's internal optimizations will help). Memory usage should be optimized by cleaning up data from closed tabs and not holding onto huge data sets unnecessarily (unless needed for download).
- **Endpoint Load:** The component should be cautious not to overwhelm the endpoint with too many requests. For infinite scroll, throttle the loading so it's one chunk at a time. If the user scrolls very fast to the bottom, queue sequential requests rather than firing dozens concurrently. Also, ideally, use the same HTTP connection if possible (keep-alive) or pipeline requests to be efficient.
- **Bundle Size:** Keep the component's footprint as small as possible. Use production builds of Carbon and DataGrid. Do not include unnecessary polyfills (if targeting modern browsers primarily). If needed, provide polyfills separately (e.g., fetch or Promise if truly needed for older IE – but likely not needed in 2025 context). The Carbon Design System and CodeMirror may add to size, but ensure to import only necessary languages/modes for CodeMirror (just SPARQL mode, rather than all).

Reliability and Robustness

- **Error Handling:** The component should handle error cases gracefully: network errors, endpoint not reachable (show a message "Failed to reach endpoint"), SPARQL query syntax errors (the endpoint will return a 400 with a message – display that to the user), timeouts, or non-OK HTTP statuses. Also handle JSON parse errors if the endpoint returns invalid JSON. The UI should not freeze or crash on errors; instead, show an error notification and allow the user to try again.
- **Automated Tests:** Maintain a high test coverage for critical logic (target > 80%). Every release should pass all tests. Use GitHub Actions or similar CI to run tests on multiple platforms to catch environment-specific issues.

- **Cross-Browser:** Support latest versions of Chrome, Firefox, Safari, Edge. Use feature detection or graceful degradation for any cutting-edge features. Svelte and Carbon are generally fine on modern browsers. Ensure CSS prefixes or fallbacks are applied if needed for grid or flex features in older browsers (e.g., IE11 is not targeted due to Svelte 5 likely requiring modern JS).
- **Extensibility:** While not a primary focus, design the component in a modular way. E.g., separate subcomponents for Editor, ResultsTable, ResultsRaw, etc., with clearly defined interfaces. This can allow future replacement or extension (e.g., adding a new type of result visualization plugin such as a chart for specific queries, akin to YASR plugins like map or graph visualizations). The core should support registering new "result renderers" in the future, although initially we have table and raw (and implicit boolean handler, maybe a simple gallery if HTML widgets are detected as in Triply, but that's an edge case).

Security

- **XSS Prevention:** As mentioned, any content that could include HTML (especially from data) must be sanitized. The application itself doesn't execute user-provided code, but the results from endpoints could potentially contain malicious scripts if someone stored them in RDF as literal. Our default is to treat all result data as plain text unless explicitly safe. This especially applies to the Raw view: we will render JSON and XML as text, not let the browser execute any embedded `<script>` that might be in an XML literal, etc. Using `textContent` or a code editor rendering ensures that. For clickable links, use `rel="noopener noreferrer"` to prevent opened pages from accessing `window.opener`.
- **SPARQL Injection:** If the user inputs their own query, there's not a typical injection issue as it's not interacting with a database through dynamic query building on our side; they are literally writing the query for the endpoint. We just send it as-is. We must ensure not to accidentally allow injection through any internal query composition (like if we ever use user input in a system query for prefixes – but since we leave that to integrators, not much risk in our code).
- **Integrity and Trust:** When fetching data from external endpoints, we rely on browser CORS for security. We should make no assumptions about the endpoint's trustworthiness. Possibly log or display the endpoint's URL clearly so the user knows where data is coming from (to avoid confusion if some UI is embedded that could be spoofed, etc.).

Maintainability and Best Practices

- Follow Svelte 5 best practices (e.g., use `$effect` and `$trigger` appropriately instead of older Svelte patterns, as noted in SVAR UI 2.0 changes [36] [37]). Use bind:this to get component references rather than deprecated patterns [38].
- Keep the code modular: for example, the logic for executing queries can be in a separate module or store, making it easier to test (mocking an HTTP fetch). The UI components (editor, table) should not themselves contain the fetch logic – they can call a controller or store action. This separation improves testability (e.g., we can simulate a response and test that the table component correctly renders it).
- The styling should primarily rely on Carbon classes and the DataGrid's theming. Avoid excessive custom CSS; where needed, use Svelte's scoped styles or global theming tokens. This will ease upgrades of Carbon or DataGrid.
- Document the code with comments, and possibly use JSDoc/TSDoc for functions and public APIs. This will help AI agents or developers in the future to understand the intent of each part.

Testability

- Design the component so that it's test-friendly. For instance, factor out pure functions for things like "parse SPARQL JSON to table data" so it can be unit-tested without a DOM. Similarly, allow

dependency injection or override of the network fetch function so tests can simulate endpoint responses (e.g., inject a dummy fetch that returns a known JSON for a given query).
- Provide sample tests in the repository as a reference for how to interact with the component in a test (e.g., using Svelte Testing Library to render it and simulate user input).
- Ensure deterministic behavior as much as possible (for example, avoid depending on actual prefix.cc network calls in the default autocompleter – better to use a static prefix list or allow mocking; otherwise tests can be flaky if offline).

Out of Scope

To clarify, the following are **not included** in the initial version of this component, though the design might allow adding them later:
- **User Authentication/Authorization:** No support for authenticated endpoints (no UI for login, API keys, OAuth, etc.). All queries are assumed to go to public or otherwise accessible endpoints. In the future, hooks could be added for including auth headers or tokens, but it's not handled now.
- **SPARQL Update UI Enhancements:** While the component can send SPARQL Update commands if typed, it does not provide a specialized UI for updates (no separate update form, no transaction management). It will simply show success or failure of an update operation.
- **Graphical Query Building:** We do not include a visual SPARQL query builder or forms to construct queries – the user must write SPARQL in the text editor.
- **Result Visualizations beyond Table:** The only result visualizations provided are the table and raw text (and a trivial true/false for ASK). We are not including fancy charts, maps, timelines, etc. Those could be added via a plugin mechanism in the future, similar to YASR's additional plugins (Triply's YASGUI has plugins for maps, gallery, etc., but those are not MIT licensed [39] and thus out of scope).
- **Query Scheduler or History beyond session:** We do not store a history of queries run (except in the open tabs persistence). There is no built-in version control or long-term query archive. Users can copy queries out manually.
- **Multiple Endpoint Joins/Federation:** The component targets one endpoint at a time (though user can have multiple tabs to different endpoints). It does not manage federated queries explicitly (other than allowing SERVICE clauses in the query text, which is purely handled by the endpoint).
- **Server-side components:** This is purely a client-side component. We assume the SPARQL endpoint exists and is managed elsewhere. We do not provide any server or database.

Quality Goals Summary

In summary, this SPARQL Query UI component aims to meet high standards of quality: a **feature-rich** yet **minimalistic** interface, high **performance** on large data, **usability** for both casual and power users, **consistency** via Carbon design, and **reliability** through thorough testing. By adhering closely to the proven model of YASGUI while updating the technology stack (Svelte 5, modern libraries) and refining the UX (infinite scroll, prefix handling, etc.), this component will serve as a robust foundation for querying RDF data on the web. The specification provided here is detailed enough to guide implementation by developers or AI agents, ensuring that all requirements, constraints, and quality considerations are understood upfront.

References and Sources

- YASGUI (Yet Another SPARQL GUI) inspiration and features: SPARQL editor with highlighting, prefix/IRI autocompletion [4] [5], and result set visualizer using tables and raw outputs [40] [33].
- Triply/YASGUI documentation for table view features (abbreviations using prefixes, filtering, sorting, pagination in original YASGUI) [20] [41].

- SVAR Svelte DataGrid documentation for performance (virtual scroll, dynamic loading) [1] and interactive features like filtering, resizing, infinite scroll, print, etc. [42] [43].
- Apache Jena Fuseki output format options, used as baseline for supported result formats (SELECT results in JSON, XML, CSV, TSV; Graph results in Turtle, JSON-LD, N-Triples, RDF/XML) [44].
- IBM Carbon Design System for Svelte components (ensuring a cohesive and accessible UI per Carbon's guidelines).

---

[1] SVAR Svelte DataGrid | Lightning-Fast Grid Component
https://svar.dev/svelte/datagrid/

[2] [19] [27] [28] [29] [43] GitHub - revolist/svelte-datagrid: Svelte data table spreadsheet best best features and performance from excel
https://github.com/revolist/svelte-datagrid

[3] SPARQL 1.2 Protocol - W3C
https://www.w3.org/TR/sparql12-protocol/

[4] [5] [7] [11] Making SPARQL fancy with YASQE and YASR | RDF JavaScript Libraries Community Group
https://www.w3.org/community/rdfjs/2014/06/10/making-sparql-fancy-with-yasqe-and-yasr/

[6] Extending custom prefix autocompleter · Issue #64 · Triply-Dev/YASGUI.YASQE-deprecated · GitHub
https://github.com/TriplyDB/YASGUI.YASQE/issues/64

[8] [9] [10] [12] [13] [14] [15] [16] [21] [31] [39] Yasgui API Reference - Triply Documentation
https://docs.triply.cc/yasgui-api/

[17] Source code for SPARQLWrapper.Wrapper
https://rdflib.dev/sparqlwrapper/doc/1.8.5/_modules/SPARQLWrapper/Wrapper.html

[18] [20] [22] [23] [26] [30] [33] [40] [41] Yasgui - Triply Documentation
https://docs.triply.cc/yasgui/

[24] [25] [34] [35] [42] SVAR DataGrid v2.1 - Powerful Data Table for Svelte 5 | SVAR Blog
https://svar.dev/blog/svelte-datagrid-v2/

[32] High-Performance Data Grid for Vue, React, Angular, Svelte
https://rv-grid.hashnode.dev/top-3-datagrids-performance-review

[36] [37] [38] SVAR UI v.2.0: Advanced UI Components for Svelte 5 | JavaScript in Plain English
https://javascript.plainenglish.io/svar-ui-v-2-0-advanced-ui-components-for-svelte-5-dc621a7abffe?gi=32690fdd5d88

[44] Apache Jena Fuseki - inspect dataset - IJS
http://semantichub.ijs.si/fuseki/dataset.html