ChatGPT

Executive Summary

SQUI (SPARQL Query UI) is a modern SPARQL 1.2 client built with Svelte 5 and Carbon Design. It demonstrates strong foundations – a modular architecture in TypeScript, reactive stores, and integration of CodeMirror 6 and a virtualized data grid for high-performance results. **Key strengths** include its clean separation of concerns (editor vs. service vs. UI), rich autocompletion (SPARQL keywords and prefix IRIs), and a polished IBM Carbon UI theme with multiple color schemes [1] [2]. The design is forward-looking, with hooks for multi-tab management, localization, and an accessible UI that aims for WCAG 2.1 AA compliance [2] [3]. Automated tests (Vitest, Playwright) and code quality tools (ESLint, Prettier) are set up, though coverage is not yet 80% as targeted [4].

However, **several weaknesses/gaps** need attention. Crucially, the current Run Query logic still uses a stubbed service (returning mock results) instead of performing real HTTP requests [5] [6]. Result viewing is limited to JSON SELECT/ASK results – other formats (CSV, TSV, RDF) are fetched but not rendered, and there is no "raw" view or download link yet (despite claims of multiple format support) [7] [8]. Advanced features like multi-query tabs, saved query history, and shareable links are only partially implemented (flags exist, but UI is missing) [9] [10]. Accessibility, while stated as a goal, has not been fully verified in interactive components (e.g. keyboard resizing of the split pane, screen reader-friendly table semantics). Performance for very large result sets (>>100k rows) remains untested in practice (infinite scroll and chunked loading are planned but incomplete [11]).

Top 5 Risks: (1) **Non-functional query execution:** As shipped, the UI does not actually query endpoints – a major blocker for users [5] [6]. (2) **Incomplete spec compliance:** Missing support for SPARQL 1.2 protocol features (e.g. dataset URI parameters, update operations via proper HTTP methods) could lead to incorrect query behavior. (3) **Unbounded memory usage:** Without result size limits or chunking, a careless query might try to load millions of rows, risking browser crashes (the code's `maxRows` limit is not yet enforced) [12]. (4) **Accessibility shortcomings:** The virtualized results grid may be invisible to screen readers (no ARIA grid roles), and dynamic updates (query executing, errors) might not be announced, failing WCAG 2.1 criteria. (5) **Security and CORS:** Many public endpoints lack CORS; the UI will silently fail on those. Users may perceive this as "broken" unless clear error messaging or proxy workarounds are provided (currently only a console warning and a generic message) [13] [14].

Top 5 Opportunities: (1) **Implement multi-tab and persistence:** By leveraging Svelte stores and localStorage, SQUI can surpass YASGUI with a smoother multi-tab experience (faster load, modern UI) – a key feature to prioritize (task 39–41) [15] [16]. (2) **Enhanced results handling:** Adding a "Raw" view for CONSTRUCT/DESCRIBE (with syntax highlighting) and one-click downloads in various formats will set SQUI apart as a more versatile SPARQL client [17] [18]. (3) **Better autocompletion:** Integrating Wikidata's entity suggester or the LOV API for class/property suggestions (beyond prefix completions) can make the query editor smarter than legacy YASQE [19] [20]. (4) **User-friendly error and timeout handling:** Providing clear inline notifications with details and suggestions (already partly done in code) will greatly improve DX when queries fail [21] [22]. (5) **Packaging and integration:** Finalizing the library build (NPM package and a Web Component bundle) will open SQUI to broader adoption – e.g. easy embedding in docs or integration into Jupyter-Lab – giving it an edge over YASGUI's heavier setup.

Decisions for the next 2 weeks: Fix the critical query execution bug (replace the stub with real HTTP calls) [5] [6]. Verify basic SPARQL 1.2 compliance: queries, updates (non-GET), and Accept header behavior with a real endpoint. Implement a minimal "raw results" panel to display non-table results so

users can at least see Turtle/CSV output. Harden the documentation (note current limitations honestly) and set up a live demo page (GitHub Pages) for feedback. **Decisions for the next 2 months:** Deliver multi-tab support with persistence (ensuring separate state per tab [23]). Add a download menu and sharing option (perhaps encode endpoint+query in URL hash for permalink). Polish the UI with prefix management dialogs and optional query templates selection. Expand test coverage, including performance benchmarks on large datasets and an accessibility audit (axe). By end of 2 months, aim for a beta release (v1.0.0) with full YASGUI parity in features and a clear migration guide.

Documentation Review (README & Guides)

Overall, the README is well-structured and covers installation, usage, and inspiration. It effectively highlights SQUI's features and technology stack [24] [25]. However, some sections overstate current functionality or lack crucial details for new users. Below are identified gaps and suggested textual patches:

- **Clarify Feature Status:** The "Features" list claims *"Intelligent Autocompletion"*, *"Multiple Formats"*, and *"Fully Accessible"* [26], which are partially implemented. To manage user expectations, annotate experimental features or planned improvements. For example:

```
- - **Multiple Formats**: Download results as JSON, CSV, TSV, or raw RDF
- - **Fully Accessible**: WCAG 2.1 AA compliant, keyboard navigation, screen
reader support
+ - **Multiple Formats**: JSON and SPARQL XML results in-app; CSV/TSV/RDF via
download (coming soon)
+ - **Accessibility-Focused**: Keyboard navigation and screen reader support
(WCAG 2.1 AA target)
``` [27]

This edit retains ambition but notes which formats are fully supported in the UI and that accessibility is a goal rather than a completed fact.

- **Quick Start & Installation:** The README shows how to install dependencies and run the dev server [28] but does not yet instruct how to include SQUI in another project (since no NPM package yet). As packaging is completed (task 48 [29]), update with installation instructions via npm. E.g.:

```diff
 ## Quick Start

 ### Installation
-
- ```bash
- npm install
- ```
+ Install via npm (coming soon):
+ ```bash
+ npm install sparql-query-ui
+ ```
```

```` [30]

And if distribution as a web component is supported, mention the `&lt;script&gt;` tag inclusion or import.

- **Usage Examples:** The README provides a basic Svelte usage snippet [31]. This is good. We suggest adding an example demonstrating a few more props (endpoint catalogue, custom prefixes, disabling features) to showcase configurability. For instance:

````diff
 ### With Configuration
 ```svelte
 <SparqlQueryUI
   endpoint="https://query.wikidata.org/sparql"
-  defaultPrefixes={{
+  prefixes={{
     rdf: 'http://www.w3.org/1999/02/22-rdf-syntax-ns#',
     rdfs: 'http://www.w3.org/2000/01/rdf-schema#',
     wd: 'http://www.wikidata.org/entity/',
     wdt: 'http://www.wikidata.org/prop/direct/'
   }}
-  showEndpointSelector={true}
-  theme="white"
-  maxRows={100000}
+  features={{ enableTabs: false, enableDownloads: false }}
+  theme={{ theme: 'white' }}
+  limits={{ maxRows: 10000, timeout: 30000 }}
 />
 ```
```` [32] [33]

This adjusted snippet uses the actual prop structure (notice `prefixes` instead of `defaultPrefixes`, matching the component code [34], and grouping theme and limits into objects as per SQUI's API). It also demonstrates toggling feature flags (e.g. disable tabs/downloads). Clarifying the prop names and types here helps avoid confusion. In fact, naming consistency is an issue: the README used `showEndpointSelector` prop, but the component expects `endpoint.hideSelector` in the object (or not) [35] [36]. We suggest updating to reflect the actual usage:

````diff
-   showEndpointSelector={true}
+   endpoint={{ url: 'https://query.wikidata.org/sparql', hideSelector: false }}
```` [35] [37]

- **Documenting API Props:** Apart from examples, the README should enumerate key props and their default values. A new **Configuration** section can summarize `endpoint`, `prefixes`, `templates`, `theme`, `localization`, `features`, `limits` (all defined in `SquiConfig` [38] [39]). For example:
````

> **Configuration Options:** *(all are optional)*
> - `endpoint`: object with `url` (string) and `hideSelector` (boolean). Also an optional `catalogue` of endpoints. If not provided, defaults to none (user must input an endpoint).
> - `prefixes`: object with `default` (Record of prefix: URI) and an optional `discoveryHook` (function to fetch prefixes from an endpoint) [40] [41]. Defaults to common prefixes built-in.
> - `templates`: object to configure query templates. Use `default` (template name) and/or `custom` (array of templates) to populate the “New Query” content [42] [43].
> - `theme`: object with `theme` key (one of `'white' | 'g10' | 'g90' | 'g100'` for Carbon themes). Default is `'white'` (light theme) [44].
> - `localization`: object with `locale` (default `'en'`) and `strings` (custom translation overrides) for internationalization [45].
> - `features`: object toggling UI features – `enableTabs`, `enableFilters` (column filters), `enableDownloads`, `enableSharing`, `enableHistory` [46]. All are `true` by default, but note some features are under development.
> - `limits`: object for query execution limits – `maxRows` (max results to display; *note:* engine still retrieves all results, but UI will warn/truncate beyond this), `chunkSize` (for future incremental loading), and `timeout` (ms before abort; default 30000) [12].

Such a section, cross-referenced with the code definitions, guides integrators on using the component effectively. (For example, without this, one might not realize how to supply a list of known endpoints or custom i18n strings).

- **Examples and FAQ:** It would be helpful to include a couple of real-world examples: e.g., using SQUI to query DBpedia vs. Wikidata (illustrating switching endpoints), or embedding SQUI as a Web Component (once available). Additionally, a short **FAQ** could address common issues: *“Why do I get CORS errors?”* (answer: the endpoint must allow cross-origin; mention the InlineNotification will show a CORS error and possible solutions) [47] [48]; *“How to persist queries/tabs?”* (explain auto-save plans or localStorage), etc.

- **Changelog/Contributing:** As development progresses, start a CHANGELOG.md to record changes once versioning stabilizes. The README’s Contributing section currently points to agent docs [49] – consider adding direct guidelines (how to run tests, coding style, how to propose features). Also, update the “Inspiration” to link to the maintained YASGUI fork (Triply’s) instead of the old TriplyDB repo, for accuracy.

Below is a diff illustrating some of the above additions in the README context:

```diff
 ## Inspiration
- This project is inspired by [YASGUI](https://triply.cc/docs/yasgui) (Yet Another SPARQL GUI) and aims to provide a modern, high-performance
```

alternative built with Svelte 5.
+ This project is inspired by [YASGUI](https://triply.cc/docs/yasgui) (Yet Another SPARQL GUI) and aims to provide a modern, high-performance alternative built with Svelte 5. Key differences include a more accessible UI, out-of-the-box theming, and support for the emerging SPARQL 1.2 standard.
``` [50]

```diff
 ## Contributing
- Contributions are welcome! Please see the agent documentation for development guidelines.
+ Contributions are welcome! To get started:
+ - **Setup**: Fork and clone the repo, then `npm install`. Run `npm run dev` to start a local demo.
+ - **Testing**: Use `npm run test:unit` and `npm run test:e2e` to run unit and end-to-end tests (Playwright). Ensure new features have test coverage.
+ - **Guidelines**: Follow our ESLint/Prettier rules. Use conventional commit messages. For major changes, please open an issue for discussion first.
+ - See [CLAUDE.md](.claude/CLAUDE.md) and `.claude/agents/` for AI agent-assisted development notes (optional reading).
``` [49]

These amendments improve transparency (especially about partially implemented features) and provide practical info for users and contributors. In sum, the documentation should evolve from a basic orientation to a more comprehensive guide as SQUI nears a stable release. Currently, adding the above clarifications and correcting prop naming inconsistencies are high priority to reduce user confusion.

SPARQL 1.2 Protocol Conformity – Gap Analysis
The table below compares SQUI's current behavior against SPARQL 1.2 Protocol requirements, citing evidence and proposing fixes. Effort for each fix is estimated as **S**mall, **M**edium, or **L**arge:

Spec Requirement	**SQUI Behavior**	**Evidence**	**Proposed Fix**	**Effort**
Query via HTTP GET – MUST accept `query` as URL param if below size limit.	**Supported.** SQUI uses GET for shorter queries by default.	Code uses `determineMethod()` to choose GET when URL < 2000 chars [51] [52].	Ensure GET is only used for SELECT/ASK/construct/describe queries, not updates (currently no check). Add a condition: `if(queryType==='UPDATE') method='POST'` [51].	S
Query via POST (URL-encoded) – MUST accept `application/x-www-form-urlencoded` with `query=` param.	**Not implemented.** SQUI only uses direct POST. No code path for URL-encoded body.	`executePost()` sends raw query with `Content-Type: application/sparql-query` [53]. No usage of `application/x-www-form-urlencoded`.	Likely **won't fix** – Most modern endpoints accept direct POST. Document this limitation. Optionally, detect if endpoint URL is a form gateway and use URL-encoded POST if needed (edge case).	M
```

Query via POST (direct) – MUST accept raw query with `Content-Type: application/sparql-query` (+ optional `version=1.2`).	**Supported (partial).** SQUI sends direct POST with the correct content type but does not set the `version` parameter.	Direct POST implemented [53]. No setting of `version` param (not in headers or body).	Add SPARQL 1.2 `version` parameter when known. For instance, append `; version=1.2` to the Content-Type header [53] or add `version=1.2` URL param if a new config flag `useSparql12` is true. (Since most endpoints treat unannotated queries as 1.1, this is low priority unless using 1.2-only syntax).	S
Exactly one `query` param – Protocol requires exactly one query per request.	**Compliant.** SQUI always sends one query.	SQUI never splits or batches queries. The UI has no concept of multiple queries in one request (aside from updates with multiple ops).	*No action needed.* (The UI could validate that the editor contains a single query at a time, but SPARQL allows multiple queries separated by `;` only in update context – handled below).	–
Dataset specification (Query) – Support `default-graph-uri` and `named-graph-uri` params (0 or more). If both protocol and query `FROM` are given, use protocol-specified dataset.	**Not supported.** User cannot specify default/named graphs outside the query text. SQUI does not append any `default-graph-uri` params in requests. It also doesn't fetch Service Description to auto-set default dataset.	No code handling `default-graph-uri` in fetch calls [54] or query params. The editor relies on `FROM` clauses in query only.	Provide UI fields or an "Advanced" section in Endpoint selector to input default graph(s). If specified, append each as `default-graph-uri` (or `named-graph-uri`) in GET URL or POST URL params. Also, ensure if user manually writes `FROM` in query plus also chooses a dataset in UI, to follow spec: prefer protocol dataset (we can warn user in UI or just rely on endpoint behavior). *Effort:* adding a simple text field is **S**, but a full multi-graph picker is **M**.	M
SPARQL Update via POST – MUST send updates with `update=` param (URL-encoded or direct) and **not** via GET.	**Partial compliance.** SQUI's `detectQueryType` flags updates [55], but the execution path does not enforce POST (it might incorrectly use GET for short updates). No UI for `using-graph-uri`.	`determineMethod()` could return GET for an `INSERT` if short (bug) [51]. No special handling of updates beyond setting Accept (which is not even needed for updates).	**Critical fix:** Force POST for updates. E.g.: ```ts if (queryType==='UPDATE') { method='POST'; } ``` [51]. Also ensure `executePost` uses `Content-Type: application/sparql-update` for updates. Currently it always uses `application/sparql-query` [53]. Update logic to: ```ts const ct = queryType==='UPDATE' ? 'application/sparql-update':'application/sparql-query'; ``` and use that in headers [53]. UI: If an Update query is run, we should not expect a JSON results response. The code currently will treat any response as text (or JSON if endpoint returns a message), which is fine. Possibly set a flag to not call `parseResults` at all, just indicate success.	S
`using-graph-uri` / `using-named-graph-uri` (Update) – Should allow specifying dataset for update operations. Must error if both present in

protocol and `USING` present in update string. | **Not supported.** No UI or param for these. Users must embed `USING` clauses in their update query text if needed. No check for conflicts. | No code for this (since no UI). | Similar to query dataset, add optional fields for “Update using graph URIs”. Given updates are less common in UI, this can be low priority. Possibly document that SQUI doesn’t support setting these via UI. If added, ensure to validate (e.g., if user’s update string contains `USING`, ignore/disable the UI fields to avoid conflict). | L |
Acceptable SELECT/ASK Formats – MUST support at least SPARQL JSON and XML; SHOULD support CSV, TSV.	**Supported (back-end).** SQUI’s fetch sets Accept header to JSON by default, but user can request XML, CSV, TSV via the `format` option. All those MIME types are recognized [56] [57]. The response parser handles JSON and XML explicitly, treating CSV/TSV as text (which is acceptable) [58] [59].	Accept header logic covers JSON, XML, CSV, TSV for SELECT/ASK [60] [57]. Parser: JSON -> `.json()`, XML -> `.text()` (left to user to interpret), others -> `.text()` [58] [59].	**UI gap:** Add a format dropdown so users can choose JSON vs XML vs CSV/TSV for SELECT queries. Also, implement a results “Raw” view for non-JSON outputs. Currently, if CSV is fetched, SQUI creates an empty results table and logs that raw data is not shown [8] [61]. Fix: store raw text in `ResultsState` (e.g., new field `rawData`) when `format` is not JSON, and allow user to switch the results component to a raw text viewer. This aligns with planned Task 35/36 [17] [62].	M
Acceptable CONSTRUCT/DESCRIBE Formats – Turtle and RDF/XML SHOULD be supported; also N-Triples, JSON-LD, etc.	**Partially supported.** Accept header logic for CONSTRUCT/DESCRIBE defaults to Turtle, with JSON-LD as secondary and wildcard [63]. N-Triples and RDF/XML are included in MIME map and can be requested if `format` is set [64]. SQUI will fetch and store the text, but again has no UI to display it nicely.	Accept header for non-SELECT uses `preferredMime` or Turtle [63]. E.g., if user selects JSON-LD, header `application/ld+json` is sent. The fetch `parseResponse` currently treats JSON-LD as text (since it doesn’t match `application/json`) [58].	Same **Raw view** solution as above. For RDF graph results, the UI should default to Raw text view with syntax highlighting (perhaps auto-detect Turtle vs JSON-LD by Content-Type). Provide a download button to save the data. In future, could integrate a simple RDF graph visualizer for small graphs (nice-to-have).	M
HTTP Accept Handling – MUST base response format on Accept header; if none, service chooses default.	**Compliant.** SQUI always sends an Accept header explicitly covering the supported formats (with quality weights) [57]. If the endpoint doesn’t support the preferred format, it can choose an alternate from those listed (`*/*;q=0.7` ensures something is returned).	Verified in code [57]. Example: for SELECT+CSV preference, header is `"text/csv,application/sparql-results+json;q=0.9,application/sparql-results+xml;q=0.8,*/*;q=0.7"` [57].	*No change needed.* (Optional enhancement: allow user to specify a custom Accept in config if needed for non-standard endpoints – low priority.)	–
Successful Query Response Codes – SHOULD return 200 (or 3XX redirect) on success.	**Handled.** SQUI doesn’t explicitly check the status code for success beyond `response.ok`. It assumes any OK (200-299) yields data. It doesn’t handle 3XX internally – those would be followed by fetch

automatically. | `if (!response.ok) throw createErrorFromResponse()` [65]. This covers non-2xx. 3xx would be followed by fetch, so code likely never sees 3xx unless redirect fails. | *No action needed.* (Ensure that redirect flags in fetch are default, which they are. In rare cases of 303 redirect to a different results URL, fetch handles it). | – |
Error Status Codes – 400 for malformed query, 500 for internal error/refusal.	**Compliant behavior.** SQUI surfaces endpoint errors via HTTP status and messages. If the endpoint returns 400 or 500, it is caught and mapped to a `QueryError` with message and details [66] [67]. The InlineNotification then labels it appropriately (e.g., “SPARQL Syntax Error” if message contains "syntax") [68] [69].	`createErrorFromResponse()` checks status and sets `message = 'Bad Request: Invalid SPARQL query'` when status==400 [70], etc. It preserves response text in `details` for display [71]. ErrorNotification uses those to show “Client Error (4xx)” or “Server Error (5xx)” [72], and even specifically catches "syntax" in details to label as syntax error [68].	Improve minor detail: include the server’s error message (if any) more prominently. Currently, for JSON error bodies, SQUI tries to parse and use `errorData.message` field [71]. This is good. For non-JSON errors, it includes the full text in `details`. The UI could expand the `<details>` by default for 400 errors to ensure users see the syntax issue (since endpoints often put a message like “line 1: syntax error”). But that’s discretionary. Overall, the error mapping is robust.	S
Error Response Body – Implementation-defined. Could be HTML, JSON, etc.	**Handled gracefully.** SQUI doesn’t assume a format for error bodies beyond attempting JSON parse. It captures any text. If an error response is HTML (e.g., a Virtuoso error page), that entire HTML gets put into `details` string [73]. In the UI, that will be shown as plain text in the `<pre>` block (HTML tags not rendered) – safe from XSS [74] [75].	`createErrorFromResponse` reads `response.text()` for non-JSON [73], so HTML is stored. ErrorNotification displays `details` inside `<pre>` (so HTML is escaped, not executed) [74] [76].	Possibly **improve presentation**: If `details` contains `<body>` or looks like HTML, we could strip tags for clarity. But as is, showing raw HTML source at least gives the savvy user some info. This is a minor DX improvement.	S
CORS Preflight (OPTIONS) – Not a spec requirement, but necessary for browsers. The service should handle OPTIONS.	**Relies on endpoint.** SQUI itself does not perform preflight; the browser does. If an endpoint fails CORS, SQUI’s fetch throws a network error. SQUI catches it and categorizes as `type:'cors'` with message "CORS Error: Cross-origin request blocked". This is then shown to the user as a Carbon InlineNotification titled “CORS Error” [77].	`handleError()` in sparqlService explicitly checks if error.message contains ‘fetch’ or known CORS phrases, and returns a QueryError with type ‘cors’ and helpful details about enabling CORS.	**No fix in SQUI** (the fix is on the server side to enable CORS). SQUI appropriately informs the user. One enhancement: provide a link or tooltip in the error notification “Learn about enabling CORS” to guide users. For advanced use, SQUI could allow configuring a proxy URL to funnel requests (YASGUI offers `Yasgui.YASQE.defaults.sparql.endpoint` proxies). However, that’s an architectural decision beyond core spec compliance.	M
Request Cancellation – Should allow users to abort long-running queries (not mandated by spec, but good practice).	**Supported.** SQUI’s Run button

turns into a Cancel button during execution [78], which calls `AbortController.abort()` via `queryExecutionService.cancelQuery()` [79] [80]. The resultsStore then flags the error as "Query cancelled" [81] [82]. The UI displays this as a warning notification (since message contains 'cancelled', ErrorNotification maps it to “Request Timeout” kind=warning) [83]. | Cancel logic: `AbortController` is used in sparqlService and QueryExecutionService [84] [85]. On abort, an Error with name 'AbortError' is thrown; SQUI catches it and treats as cancellation [81]. | Improve UX: After cancellation, perhaps clear the results pane or indicate it was canceled (currently it shows an error notification “Request Timeout” which might be slightly confusing vs “Query cancelled by user”). We can differentiate user cancel vs real timeout by checking the message. E.g., if message == "Query execution cancelled" (set in code [86]), use title "Query Cancelled". This is a tweak to ErrorNotification’s logic [83]. Also, maybe return to an idle state (so user can edit query) immediately. This is small polish. | S |
| **SPARQL Service Description** – Not a protocol must, but very useful (especially in SPARQL 1.2). Clients should GET `<endpoint>?service-description` or use `OPTIONS` to retrieve supported features, default dataset, etc. | **Not implemented.** SQUI does not automatically fetch service descriptions. There is a `prefixService.discoveryHook` facility for endpoint-based prefix discovery [87] [88], but no default hook is provided. Features like autocomplete for classes or functions from the service are not present. | No mention of service description usage in code. The Endpoint selector could in future fetch `?service-description` to pre-populate info (like available default graphs or prefix mappings), but currently it doesn’t. | **Future enhancement:** Provide an option to fetch the SPARQL Service Description when an endpoint is selected. From it, SQUI could auto-add common prefixes (if listed in sd:prefixes), warn if endpoint is read-only (no updates), or list supported result formats. This is a larger feature (parsing RDF). Perhaps use a lightweight JSON-LD parser if the endpoint returns Turtle or RDF/XML. This would be **Large** effort and can be deferred. | L |

Legend: S = Small (a few lines change or minor feature), M = Medium, L = Large (significant new UI or logic).

Summary: The core query operations (GET and direct POST) are well-supported by SQUI [51] [53], but **SPARQL 1.2 updates require urgent fixes** – currently an `INSERT` might be sent via GET (violating the protocol and likely failing) [51]. Adjusting that and using the correct content type for updates is high priority. **Result format handling is another gap**: SQUI can retrieve all recommended formats, but lacks UI to utilize them fully. Implementing a raw results view and download options will close this gap. **Dataset specification** via UI is currently absent; while not strictly required by the spec (users can put `FROM`/`USING` in query text), adding this would improve SQUI’s completeness. Less urgent is service description integration and URL-encoded POST support – nice-to-haves for edge cases. By addressing the above, especially the update handling and results view, SQUI will conform to SPARQL 1.2 protocol and offer users the full flexibility the spec envisions.

UI/UX & DX Evaluation

SQUI’s user interface is clean and minimal, but to compete with YASGUI and delight users, several enhancements are needed in the **editor**, **results viewer**, and overall **workflow**. Below we detail each area with actionable recommendations (each could be a GitHub issue with acceptance criteria):

Query Editor (CodeMirror 6 Integration)
- **Autocomplete Enhancements:** The editor currently suggests SPARQL keywords and known prefixes/terms (from built-in vocabularies) [89] [20]. To improve UX:
  - *Issue:* Autocomplete doesn’t include user-defined prefixes in the query, until after typing them fully.
    - **Acceptance Criteria:** If a user has `PREFIX ex: <...>` in the query text, typing `ex:` should trigger term suggestions (if available) or at least acknowledge the prefix.
    - **Proposal:** Upon query load or prefix addition, dynamically include those prefixes into the completion logic. The `prefixService.parsePrefixesFromQuery()` is already available [90]; integrate it to update completion sources when the query changes or when an endpoint’s known prefixes are discovered.
  - *Issue:* Lack of class/property autocompletion. YASQE offered autocompletion of RDF terms by querying LOV or the endpoint’s ontology.
    - **Acceptance Criteria:** When the user types a prefix and colon (e.g. `rdf:`), suggest common terms from well-known vocabularies (already partly done via `commonTerms` in prefixCompletions.ts [91] [20]) *and* consider fetching from the web if not in the list.
    - **Proposal:** Implement an async completion that queries the [LOV API](https://lov.linkeddata.es/dataset/lov/api) or the endpoint’s vocabulary for terms. For example, upon `foaf:` trigger, call an API to get FOAF terms. This can populate additional `Completion` items. Use the already async-enabled CodeMirror completion interface (the `prefixCompletion` function is async-ready [92]). *Effort:* Medium (requires external API usage and caching).
  - *Issue:* No suggestion of function signatures or snippet expansions (e.g., no help for `CONSTRUCT WHERE { … }`).
    - **Acceptance Criteria:** Pressing Ctrl+Space at an empty line shows a menu of templates like “SELECT * WHERE { }”, “ASK { }”, etc.
    - **Proposal:** Leverage the existing templateService which has sample queries [42] [93]. Integrate it so that if the editor is empty (or user types a “/” or special trigger), a list of templates is offered. For instance, trigger on typing “sel” to suggest the full SELECT template.

- **Snippets and Formatting:**
  - SQUI has a TemplateService for initial query text but no UI to apply templates after initialization. Consider adding a **“Templates” dropdown** in the toolbar.
    - **Acceptance Criteria:** A “Templates” button in the toolbar opens a list of query templates (default plus any user-provided). Selecting one replaces the current query text with that template (after user confirmation if the editor is non-empty).
    - **Proposal:** Use Carbon’s OverflowMenu or a ComboBox for templates. The translations file already has entries like `'editor.selectTemplate': 'Select a template'` [94], indicating this was planned. Implement a

`TemplateSelector` component using those strings and TemplateService.getAllTemplates().
  - No one-click query formatting is provided. A large query can be hard to read if pasted in.
    - **Acceptance Criteria:** Clicking a “Format Query” button re-indents the SPARQL query in the editor (e.g., each clause on a new line, proper indent for nested braces).
    - **Proposal:** Integrate an existing SPARQL formatter library (if available in JS), or implement a simple rule-based formatter (e.g., after each `{` or `.` insert a newline). This could be a utility in `utils/` triggered by the button. (Label as enhancement – not critical, but improves DX.)

- **Error Highlighting:** Currently, syntax errors are only caught when the endpoint returns an error. In-editor linting would be ideal.
  - **Acceptance Criteria:** If the user types an obviously invalid token (e.g., `SELLECT`), the editor underlines it or shows a red squiggly.
  - **Proposal:** Use CodeMirror’s support for linting. A simple approach is to run a SPARQL parser in web worker; however, implementing full parsing is complex. Instead, we might integrate a third-party SPARQL JS parser (like `sparqljs` by Ruben Verborgh) for client-side checking of basic syntax. If it throws, mark the error location. This is a **large** effort; as an interim, we can highlight common mistakes (e.g., unmatched braces, prefix not defined) with custom regex checks.

- **Keyboard Shortcuts & Accessibility:**
  - Execution via **Ctrl+Enter** is already implemented [95] – great. Document this in a tooltip or help modal (there is a translation string for it [96]). Ensure on Mac it listens to Meta (the code uses `Mod-Enter` which should handle both).
  - Provide a shortcut for **focus switching** – e.g., Alt+E to focus editor, Alt+R to focus results pane. This aids power users. Add appropriate `accesskey` or documentation of such in the help.

Results Viewer (Table & Visualization)
- **Virtualized Table Performance:** The use of SVAR DataGrid (wx-svelte-grid) with virtual scrolling is a major strength [97] [98]. It allows 10k+ rows smoothly. However, test at scale: 100k rows have been set as a soft limit [99] [100].
  - Confirm the grid’s performance at that limit on typical hardware. If issues, consider lowering default `maxRows` or adding a warning if row count exceeds, say, 50k (there is a warning string `'warning.largeResult'` in translations [101], but it’s not triggered yet).
  - **Acceptance Criteria:** For a query returning >N rows (configurable, default 100k), the UI should display an inline warning (“Result set is too large to display completely…”) [101] and not attempt to render beyond N rows.
  - **Proposal:** Implement logic after parsing results: if `rowCount > maxRows` (from limits config), then (a) show the warning notification, (b) truncate the `rows` array to `maxRows` entries so the grid only renders that many [102] [103], and (c) provide a “Download full results” link for power users. The user can then get the complete dataset as file if needed.

  - This approach balances UX (avoid freezing the browser) and user need (access to all data). The tasks had similar ideas (Task 34) [104] [105].

- **Raw Results & Format Switching:**
  - As noted in the protocol analysis, implementing a **"Raw" view tab** in the results pane is crucial for non-SELECT queries and alternative formats. YASGUI's YASR had this ("Raw response" plugin) [106].
  - **Acceptance Criteria:** A toggle (perhaps Carbon SegmentedControl or Tabs) above the results allows switching between "Table" and "Raw" for SELECT queries, and between "Graph" and "Raw" for CONSTRUCT/DESCRIBE results. (ASK could simply show the boolean in a large font as now, that's fine). The "Raw" view shows the entire response text or JSON in a scrollable `<pre>` with appropriate formatting.
  - **Proposal:** Create a component `RawView.svelte` that displays `resultsStore.data` or `resultsStore.rawData`. If the format is JSON and query type SELECT, RawView could pretty-print the JSON (or just show it raw, since Table covers it). For XML, Turtle, etc., show as syntax-highlighted text (perhaps reuse CodeMirror in read-only mode with XML/Turtle mode). Carbon's design suggests maybe use Tabs: e.g., a Tabs component with labels "Table" and "Raw". The translations file already has `'results.table': 'Table', 'results.raw': 'Raw'` [107]. This indicates it was planned. Implement this toggle, and bind it to `ResultsState.view` (which exists: `'table' | 'raw' | 'graph'` in state [108]).
  - **Acceptance Criteria (Download):** When on Raw view or at least somewhere in UI, a "Download as [format]" button is visible to save results to a file.
  - **Proposal:** Add a Download button in the toolbar or results header. On click, if the results are already fetched in the desired format, simply create a Blob and trigger download (the JSON/CSV text is available; for the current JSON results, we can re-serialize pretty or allow the user to pick CSV re-fetch). If the user wants a format that wasn't fetched, either re-run the query with that format (prompt: "Re-execute query for CSV?") or if format was already chosen before execution (better). To start, perhaps tie the Download options to the Format selection: e.g., if user picks CSV format in a dropdown and runs, then the table is not shown (since we can't parse CSV to table easily yet) but raw CSV is available to download. This workflow can be refined in iterations.

- **Linkification and RDF Term Rendering:**
  - *Issue:* URIs in results are shown as full strings, which is hard to read and not interactive. E.g., `<http://example.org/resource/1>` is displayed in full [109], and cannot be clicked.
    - **Acceptance Criteria:** URIs in the results table appear as hyperlinks with a prefix abbreviation if possible (e.g., `ex:resource1`), and clicking opens the link in a new tab (with `rel="noopener noreferrer"` to prevent opener issues).
    - **Proposal:** The DataGrid allows custom cell renderers. We can check the cell's `type`. In `DataTable.svelte`, when constructing `gridData`, instead of storing the plain string via `getCellDisplayValue()` [110], we could store an HTML string or a small component reference. However, `wx-svelte-grid` likely expects plain data. Another way: post-process the cell values in

the DataGrid by styling known patterns. Perhaps easier: use the grid's slot for cell content if available. If not, we may need to incorporate hyperlink logic in `getCellDisplayValue()`: e.g., if `cell.type==='uri'`, set the value to a clickable form. But injecting HTML through `gridData` might not be straightforward. Alternatively, we could overlay an onClick handler: use the grid's event system to detect if a cell value looks like a URL and then open it on cell double-click. This needs exploration. The simplest solution: at least abbreviate the URI using prefixService (we have `prefixService.abbreviateIRI()`, which is currently unused in display [111] [112]). Setting `abbreviateUri=true` in `getCellDisplayValue` would do nothing now (it returns full URI still [111] [113]), but we can implement that to use prefixService [114] [115]. So as a first step, **show abbreviated URIs** in table cells. Then make them copyable (user can copy cell text easily). The clickable links can come next.
  - *Issue:* Literals with language or datatype are shown with `@en` or `^^xsd:int` in the same cell text, which is good for completeness but might be verbose. Possibly allow toggling a "simple view" vs "full view" of literals. The translations hint at this (`results.simpleView`, `results.fullView` strings exist) [116].
    - **Acceptance Criteria:** A toggle in results (maybe a button "Full literals info") that switches between showing `"Hello"@en` vs just `Hello` for example, and between full URIs vs prefixes.
    - **Proposal:** This can map to the existing design: the ResultsState might incorporate a setting for "abbreviate URIs" and "show data types". In code, `getCellDisplayValue` already takes options `showDatatype` and `showLang` [117]. We can connect those options to UI controls. For instance, a toolbar above the table with checkboxes "Show datatypes" and "Show language tags". If unchecked, the display function would output just the literal value without ^^ or @ parts (by tweaking lines [118] to skip adding them). Similarly for URIs: an "Abbreviate URIs" toggle (on by default perhaps) would use prefix abbreviations.
    - This gives users flexibility, especially useful if one is just scanning values (simple view) versus debugging RDF data (full view).

- **Pagination vs Scrolling:** YASR did not implement pagination; it relies on scrolling. Given virtualization, SQUI can handle large sets. But some users might prefer page-by-page viewing (especially if they want to see "50 rows at a time").
  - This is a low-priority since virtual scroll is effectively better in most ways. We might not need actual pagination controls. But an optional "Go to top/bottom" could be handy if user scrolls a lot. Or a text field to jump to a row by index? Possibly overkill.
  - For now, focus on making scrolling smooth (which it is, due to the grid).

State Management & Multi-Tab UX
- **Multiple Tabs:** The ability to have multiple query editor tabs is *in progress* (the code base has a Tabs folder placeholder and feature flag) [9]. This is a vital feature: YASGUI users heavily use tabs for different queries.
  - **Acceptance Criteria:** Users can click a "+" button to open a new tab

(with a fresh editor and results area), and switch between tabs without losing their query or results on each. Tabs should be named (“Query 1”, “Query 2”, or by a PREFIX title if possible). They should be closable. On refresh, the tabs and their contents are persisted.
  - **Proposal:** Implement `QueryTabs.svelte` using Carbon Tabs component (or a custom minimal tab strip). Use the existing `enableTabs` flag to conditionally show this around the Toolbar. The data model could be: each tab has its own QueryStore and ResultsStore instance, managed by a TabsStore mapping tabId -> {queryState, resultsState}. Simpler: augment current stores to handle multiple sessions (but that could complicate reactive bindings). Instead, use context or store with currentTabId, and make queryStore/resultsStore tab-scoped.
  - Effort is significant (Large), but much of it was planned (Tasks 39-41) [15] [119]. Notably, localStorage persistence (Task 41) can use JSON serialize of all tabs every time a query or endpoint changes. YASGUI does this (stores each tab in localStorage).
  - **Quick win**: If full multi-tab is too heavy for immediate release, consider at least a **“Clone Tab / New Tab”** functionality that opens the current query in a new browser tab via a permalink. For example, a “Open in new window” button that constructs a URL with `?endpoint=<…>&query=<encoded>` so the user can use the browser’s native tabs as a workaround. This is less elegant but easy to implement using the share link logic.

- **Saved History:** The `enableHistory` flag suggests plans for a query history (like a list of past executed queries). YASGUI doesn’t have an explicit history UI (it persists tabs instead). Having a history could be a unique value-add for SQUI.
  - **Acceptance Criteria:** Users can access a list of past N queries (with timestamps, maybe truncated text or a title), and clicking an entry loads it into the editor (possibly opening a new tab or replacing current).
  - **Proposal:** On each successful query execution, push the query (and endpoint) into an array in localStorage. Provide a “History” panel or dropdown (maybe under a 🕑 icon) showing the last, say, 10 queries. This feature should be optional (some users might not want localStorage usage for privacy – YASGUI addressed that with a config to disable persistence). Use the translation strings like `'shortcuts.hint'` and others to perhaps show a hint like “(History available)” in UI. This is a medium effort but a nice DX improvement.

- **Sharing & Permalinks:** SQUI aims to include sharing (feature flag exists) [120]. This is a highly useful feature: YASGUI provides a “shareable link” that includes the query and endpoint encoded in the URL (or uses a shortlink service).
  - **Acceptance Criteria:** Clicking a “Share” button generates a URL that when visited loads SQUI with the same endpoint and query pre-filled (and ideally the same results, though we could simply auto-run the query on load).
  - **Proposal:** Implement by updating the standalone page to read URL parameters. For instance, define `?endpoint=<url>&query=<urlencoded query>&format=<fmt>` in the standalone deployment. The SQUI component could on mount check `window.location.search` (if running in standalone mode) and

apply those props. Alternatively, generate a base64 or shorter token of the query and embed in hash (some SPARQL queries can be very long for a URL). Perhaps use `LZString` compression + base64 in the fragment identifier. YASGUI's public share uses the yasgui.triply.cc service which might shorten a bit. Initially, focus on the simpler approach: full query in URL (ensuring it works for moderately sized queries).
  - Also, consider adding a copy-to-clipboard for the generated link, and a small toast "Link copied!".

- **Theming & Customization:** With Carbon, SQUI already supports four themes. We should ensure an easy way for integrators to switch theme (which is via the `theme` prop as documented [44]). Possibly mention in docs that theme can also be changed at runtime via the `themeStore.setTheme()` (if integrator wants to add a theme switcher UI in their app). It's more DX than UX.
  - Also, ensure the CSS is encapsulated. Currently, the component's styles are mostly local (Svelte scopes them) except Carbon's global styles and CSS variables. When packaging as a Web Component, we might consider using `:global` wisely or shadow DOM. Perhaps an issue: "Provide Shadow DOM mode for Web Component to avoid host page CSS interference".
  - **Acceptance Criteria:** If SQUI is used on a page with different styling, it should still look and function as intended (no leakage of styles in or out).
  - **Proposal:** Utilize Svelte's upcoming support for shadow DOM custom elements (Svelte 5 can compile to custom elements; by enabling that in package config, we get `<sparql-query-ui>` element which encapsulates internal CSS). This might require some adjustments (e.g., Carbon's styles might not penetrate the shadow without being inlined – possibly use the `::part` or inject Carbon CSS inside shadow). This is advanced, so maybe backlog it as a "post-v1" improvement for DX.

- **Internationalization (i18n):** SQUI is built with i18n in mind, but currently only English is provided [121] [122].
  - **Acceptance Criteria:** It should be straightforward to add translations for other languages. Test by adding a dummy locale (say 'fr') via `addTranslations('fr', {...})` and `setLocale('fr')`, and ensure all UI strings update.
  - **Proposal:** Provide a guide or tool for contributors to supply translations. Possibly integrate with a platform or at least list the keys. Since the keys are many (and cover even warnings and shortcut hints), a contributor might translate the basics (like button labels) and skip some advanced messages. That's okay.
  - On UX side, if multiple locales are bundled, add a locale switch in a menu (maybe under a "Settings" or language icon). Not urgent until more locales exist, though.

- **User Assistance and Help:** Consider adding a small help modal or documentation link within the UI. YASGUI doesn't have this in-app (since it expects users to know SPARQL), but given SQUI's broader aim, a "?" icon that lists keyboard shortcuts and maybe links to SPARQL reference could be nice DX.

  - E.g., “Ctrl+Enter to execute. Tab through inputs. Visit [W3C SPARQL Tutorial](https://www.w3.org/TR/sparql12-query/) for query syntax.” etc.

In summary, **immediate UX priorities** are: multi-tab support, raw results view, and linkable/downloadable results. These directly address gaps where users would otherwise feel restricted compared to YASGUI. Simultaneously, some **DX improvements** (like query formatting, history, share links) will greatly enhance the developer experience of using SQUI in other projects. Each of the above has clear criteria and can be tackled incrementally – for example, start with a basic Raw view (just a `<pre>`), then enhance with syntax highlight; or start with tabs without persistence, then add persistence. The goal is to maintain a tight feedback loop with users as these features roll out.

(Actionable ticket examples are provided in the Roadmap section for implementation.)

Architecture & Code Quality Review
SQUI’s codebase is logically structured and largely adheres to Svelte best practices. The project structure (components, stores, services, utils) is very clear [123] [124]. The use of Svelte 5’s `$state` and `$derived` stores yields concise component logic. TypeScript types are well-defined for SPARQL data and config, aiding maintainability [125] [126]. The separation of `sparqlService` (protocol logic) from UI is especially good [127] [128], as it will facilitate testing and potential reuse outside the Svelte context. There are, however, some areas for refactoring to reduce complexity and eliminate bugs:

- **Duplicate Query Execution Paths:** There are two parallel implementations for running queries – `sparqlService.executeQuery()` and the older `QueryExecutionService` stub [129] [127]. Currently, RunButton still calls `queryExecutionService.executeQuery` (with its mock) [6], meaning the real HTTP logic in `sparqlService` is unused for UI interactions. This is a critical mismatch.
  Refactor: Remove or integrate `QueryExecutionService`. The simplest fix is to call the results store’s `executeQuery` (which internally uses sparqlService) directly from the UI. For example, in `RunButton.svelte` we can do:

  ```diff
  - import { queryExecutionService } from '../../services/queryExecutionService';
  + import { resultsStore } from '../../stores/resultsStore';
  ```
  ```diff
    async function handleRunQuery(): Promise<void> {
      if (!canExecute) return;
      try {
  -      await queryExecutionService.executeQuery({
  -        query: queryState.text,
  -        endpoint: endpoint,
  ```

```
  -      });
  +      await resultsStore.executeQuery({ query: queryState.text, endpoint:
endpoint });
       } catch (error) {
         console.error('Query execution error:', error);
       }
     }
  ```
  ```diff
    function handleCancelQuery(): void {
  -    queryExecutionService.cancelQuery();
  +    resultsStore.cancelQuery();
     }
  ``` [6] [80]

  This change will route all query execution through the unified sparqlService, and the resultsStore will properly set loading state and handle errors. After this, `QueryExecutionService` (the class and file) can be removed entirely to avoid confusion. Its abort logic is already in sparqlService with AbortController [84], and resultsStore.cancelQuery calls sparqlService.cancel internally (we should update resultsStore.cancelQuery accordingly to call `sparqlService.cancelQuery()` instead of the old service – currently it does call sparqlService, so that's fine [82]).
  Impact: High – fixes the main functionality bug. Effort small (few lines). This simplifies maintenance (no duplicate query logic).

- **Handling of QueryError vs string:** In resultsStore, `setError` expects a string [130]. Consequently, when sparqlService throws a rich `QueryError` (with type, status, details), resultsStore currently reduces it to just the message string [131] [132]. This loses error details that the UI could show (and indeed ErrorNotification is prepared to handle a `QueryError` object with `error.details`) [133] [134].
  Refactor: Change resultsStore.setError to accept a `QueryError`. For example:
  ```ts
  setError: (error: string | QueryError): void => {
    update(state => ({
      ...state,
      error,
      loading: false
    }));
  }
  ```
  And adjust ResultsState.error type to `string | QueryError | null`. Then in executeQuery's catch, instead of `errorValue = queryError.message` and storing that [135], directly do `resultsStore.setError(queryError)`. Similarly handle unknown errors. This way, the UI gets the full object.
  - *Example diff (in resultsStore.executeQuery)*:
    ```diff
           } catch (error) {
  -        let errorValue: string;
```

```
  -        if (error && typeof error === 'object' && 'message' in error) {
  -          const queryError = error as QueryError;
  -          errorValue = queryError.message;
  -          // Store the full QueryError for detailed display
  -          // For now, we store just the message, but the error
notification
  -          // will be enhanced to show full QueryError details
  -          update((state) => ({
  -            ...state,
  -            loading: false,
  -            error: errorValue,
  -          }));
  -        } else {
  -          errorValue = 'An unknown error occurred';
  -          update((state) => ({
  -            ...state,
  -            loading: false,
  -            error: errorValue,
  -          }));
  -        }
  +        const errorObj: QueryError = error instanceof Error
  +            ? { message: error.message, type: (error as any).type ||
'unknown', details: (error as any).details }
  +            : { message: String(error), type: 'unknown' };
  +        update(state => ({ ...state, loading: false, error: errorObj }));
           }
```
136 137

In ErrorNotification, since we now pass a QueryError, it will go into the detailed branch and display everything nicely (title, message, details) 69 138.
Impact: Medium – richer error info for users (e.g., syntax error line numbers from Fuseki, etc.). Very low risk (just data plumbing).

- **Global Store Scope (Multiple Instances):** SQUI uses module-scoped stores (queryStore, resultsStore, etc.) that behave like singletons 139 140. This means if two `<SparqlQueryUI>` components are mounted on the same page, they would share state – likely unintended. Ideally, each instance should isolate its state.
Refactor Options:
1. Encapsulate stores in the component: Convert `queryStore` etc. to be created per component instance (e.g., via context or by instantiating in the component's script). But given Svelte's reactivity, using context might be needed so that child components (RunButton, Editor) use the instance-specific store instead of the import. Svelte 5 could allow passing stores as props or using new runes.
2. Alternatively, make the component not directly use the global store in a way that conflicts – in current code, `SparqlQueryUI.svelte` sets `defaultEndpoint` store on mount to the prop endpoint 141, which is fine for one instance but a second instance would override it globally. Similarly, queryStore is imported and used in Editor – multiple editors share it, clobbering each other's text.

  Proposed Solution: Use Svelte's `context=` API: e.g., in SparqlQueryUI on creation, do something like `setContext('squiStores', { queryStore: createQueryStore(), resultsStore: createResultsStore(), ... })` (we would need to expose factory functions from each store module). Then each child component (Editor, RunButton) would first try `getContext('squiStores')` and use that instead of the imported singleton. This is a non-trivial refactor but improves reusability.
  Given most users likely use one instance, this is not urgent. We should document that multiple instances aren't supported (if we decide not to fix it now). Effort is moderate.
  Impact: If done, allows scenarios like comparing two endpoints side by side on a page, which could be a strong feature.

- **Project Build & Packaging:** The use of `@sveltejs/package` in devDeps [142] indicates intention to package as a library. Ensure the `package.json` has correct `exports` (for ESM). Check if the build outputs types (maybe run `tsc --declaration`). Also, currently the package name is "sparql-query-ui" v0.1.0 [143] – before 1.0, consider publishing under an @beta tag. No glaring issues in build config from what's visible (vite.config wasn't shown in code, but presumably standard for lib mode). Just verify that Carbon and CodeMirror are properly externalized or bundled as needed. Possibly mark `carbon-components-svelte` and `codemirror` as peerDependencies if the library expects host to provide them (but more likely we bundle them to simplify usage). This is more for release engineering.

- **CSS and Styling Isolation:** Inside components, many styles use Carbon CSS variables and `:global()` selectors (for Carbon classes) [144] [145]. This is fine as long as Carbon's base styles are loaded. With Svelte's preprocess for Carbon, presumably those get injected globally. We should double-check that if someone uses SQUI as a library, they don't have to manually import Carbon CSS. The Carbon Svelte preprocess might embed necessary styles in the components. If not, we should instruct users to import Carbon's CSS in their app. For a Web Component build, we might want to inject Carbon's styles into shadow DOM or use light DOM for Carbon parts.
  - Also, `wx-svelte-grid` likely has its own styles – currently it looks unstyled except what we override. Actually, it does (the global `.wx-grid` styles, etc. are likely from its library's CSS). Are we including those? Possibly the library's styles are inlined or required as side effects. Confirm and document if user needs to include any CSS.
  - No action needed in code per se, just ensure packaging includes necessary CSS.

- **Memory Management & Cleanup:** Svelte 5's reactive stores and CodeMirror instances – ensure they are disposed to avoid leaks. Noticed in `SparqlEditor.svelte`, `editorView.destroy()` is called on component destroy [146] – good. Also event listeners on stores are unsubscribed (EndpointSelector does that on unmount) [147]. This shows good practice. Just verify that abort controllers are properly cleared – after query finishes, sparqlService clears the timeout and sets controller to null [65] [148], which is fine.

- **Code Patterns:** Overall, code is clean. Some minor observations:

  - In several places, the code uses `$derived(store)` to get values (Svelte 5 reactive statements) – which is fine, but in RunButton.svelte, they do a pattern `$state($queryStore)` and then manually $effect to update it [149] [150]. This is a bit verbose; Svelte 5 might allow directly using store values in markup or $derived for the boolean conditions. Perhaps the pattern was due to an earlier bug. Not a big issue, but one could refactor it for brevity. (No change required, as it works.)
  - The tasks mention >80% test coverage target [4]. Currently, it's unclear how close we are. We encourage adding tests especially for the critical path (executing a real query and getting results). Possibly integrate a CI with a dummy SPARQL endpoint (maybe use a public endpoint with a known small dataset for test assertions). This is more QA than architecture, but ties in.

- **Priority of Refactors:**
  1. **Fix QueryExecution stub (High priority) –** must-do for real queries to run [6].
  2. **Error handling object (Medium) –** improve debugging and user feedback [135] [68].
  3. **Multi-instance safety (Low for now) –** note in docs, fix later.
  4. **Completing multi-tab architecture (High, after core fixes) –** requires introducing a TabManager store or context; plan carefully to avoid overly complex reactive logic. Possibly use Svelte store that holds an array of tab states and an active index. The component could subscribe and render active tab's editor and results accordingly. Leverage Svelte's keyed each to recreate components per tab.

Below is a short diff illustrating the **RunButton.svelte** refactor to remove the stubbed QueryExecutionService usage (Priority ✱✱✱✱✱):

```diff
<script lang="ts">
   import { Button, InlineLoading } from 'carbon-components-svelte';
-  import { Play, StopOutline } from 'carbon-icons-svelte';
-  import { queryStore } from '../../stores/queryStore';
-  import { resultsStore } from '../../stores/resultsStore';
-  import { defaultEndpoint } from '../../stores/endpointStore';
-  import { queryExecutionService } from '../../services/
queryExecutionService';
+  import { Play, StopOutline } from 'carbon-icons-svelte';
+  import { resultsStore, queryStore } from '../../stores';
+  import { defaultEndpoint } from '../../stores/endpointStore';
   import { t } from '../../localization';
   // ... (props interface etc.)
</script>
```

```
 async function handleRunQuery(): Promise<void> {
    if (!canExecute) return;
    try {
-      await queryExecutionService.executeQuery({
-        query: queryState.text,
```

```
-        endpoint: endpoint,
-      });
+      await resultsStore.executeQuery({ query: queryState.text, endpoint:
endpoint });
    } catch (error) {
       console.error('Query execution error:', error);
    }
 }
 function handleCancelQuery(): void {
-    queryExecutionService.cancelQuery();
+    resultsStore.cancelQuery();
 }
```
6 80

After this change, real HTTP requests will be issued and the mock is fully removed. This exemplifies how a small diff can have a large impact on functionality.

Another example diff for **passing QueryError to resultsStore** (improving error detail):

```diff
 // In sparqlService.executeQuery, on error:
 catch (error) {
-    throw this.handleError(error);
+    throw this.handleError(error);
 }
```

(No change above, but ensuring we actually throw QueryError) 151

```
 // In resultsStore.executeQuery catch:
-      if (error && typeof error === 'object' && 'message' in error) {
-         const queryError = error as QueryError;
-         errorValue = queryError.message;
-         // (Currently only storing message)
-         update(state => ({ ...state, loading: false, error: errorValue }));
-      } else {
-         errorValue = 'An unknown error occurred';
-         update(state => ({ ...state, loading: false, error: errorValue }));
-      }
+      const errObj: QueryError = error && typeof error === 'object' &&
'message' in error
+           ? error as QueryError
+           : { message: String(error), type: 'unknown' };
+      update(state => ({ ...state, loading: false, error: errObj }));
```

And ensure `ResultsState.error` type is updated (and components expecting string adjust to handle object). 131 132

These code adjustments, along with thorough testing after each, will incrementally improve the reliability and maintainability of the code. Many planned features (templates, history, etc.) already have scaffolding (e.g., translation keys, config flags). The strategy should be to **remove deprecated code (stubs)** and **flesh out the scaffolded features one by one**, keeping the design consistent. The codebase is in a good state to support rapid addition of functionality without major refactor, aside from the multi-tab context which needs careful design to avoid a state management tangle.

In conclusion, aside from the specific tweaks above, the overall architecture is sound – logically separated and using modern frameworks correctly. Addressing the identified code smells will eliminate the last "dead code" and set a strong foundation for adding the remaining features on the roadmap.

Performance Profile & Benchmark Plan

Current Performance: SQUI's architecture appears performant for typical use – CodeMirror 6 is efficient for editing, and the virtualized DataGrid handles large results far better than a raw HTML table (which YASGUI used). The critical performance paths are: initial load, query execution latency (mostly network/endpoint-bound), results parsing, and rendering of results.

- **Initial Load (First Query):** Load time is dominated by library size and CodeMirror initialization. Bundle analysis is needed: including CodeMirror modules and Carbon likely produced a few hundred KB of JS/CSS. The target was <500KB gzipped [152]. Using dynamic import for heavy modules (e.g., CodeMirror) could improve initial load if we only instantiate editor on user focus, but that might complicate things. Likely okay as is. We should measure the time from component mount to editor ready. CodeMirror's mount is synchronous in onMount, so negligible (couple of ms). Carbon CSS injection similarly.

- **Query Execution Throughput:** For small queries, overhead is minimal (one fetch and JSON parse). For large SELECT results, performance depends on streaming parse and rendering:

- **JSON Parsing:** The fetch `.json()` call will block until the entire response is downloaded. Parsing 10MB JSON (roughly 100k results) in V8 might take ~100-200ms. This is fine. Memory-wise, that JSON object of 100k×3 cells ~ can be ~ tens of MB (our test showed ~13 MB string for 100k×3 results). After parsing into JS objects, maybe ~2-3x that in memory (so ~30-40 MB). Manageable on modern devices, but not infinite.
- **Data Conversion:** `parseResults()` then iterates bindings and creates our `rows` array with ParsedCells [153] [154]. That's another 100k iterations – should be under 100ms in JS. No heavy computations there (just constructing small objects).
- **Rendering:** The DataGrid only renders what's visible (~ maybe 20-50 rows at a time). So initial render cost is low (constant). Scrolling will render new rows on demand, which is fine.
- The grid allows sorting on client. If a user clicks a column to sort 100k rows, what happens? Likely `wx-svelte-grid` sorts its internal data array. That's 100k comparisons – probably okay (< 0.1s maybe). But if multiple columns or repeated sorts, just be aware. (If needed, we could implement a more efficient sort or let the endpoint handle ordering via SPARQL if really needed).

- Filtering (if enabled later) would similarly loop over data. That might be heavier, but we can optimize with indices or simply note that filtering 100k in JS might be slow – but acceptable if user knows what they're doing. We can add a debounce on filter input.

- **Memory Footprint:** As discussed, 100k rows ~ tens of MB. 1 million rows would be ~ 10x more, which would strain memory and performance (plus no browser will handle 1e6 HTML elements

even virtually without slowdown in scroll, and JSON parse 130MB might crash). Thus, the limit is necessary. The default maxRows 100k is a bit high but okay. We might consider reducing default to 50k to be safer for average machines, unless testing shows 100k still feels smooth.

- **Streaming & Backpressure:** SQUI currently waits for full response then parses. Some endpoints (Virtuoso) stream results progressively. In the future, to improve perceived performance, we could parse JSON chunks as they come (using fetch Streams API and a streaming JSON parser). That's complex and usually not needed if endpoints are reasonably fast. Alternatively, the Task 33 approach (infinite scroll by re-querying with OFFSET) is simpler for extremely large datasets, but that complicates user experience (they must scroll to trigger new queries, which might produce duplicates if underlying data changed). Probably not urgent.

Benchmark Plan: We recommend establishing automated performance benchmarks to guard against regressions. A combination of unit tests (for parsing) and integration tests (for end-to-end throughput) will be used:

- **Parsing Benchmark (Node):** Use a Node script to generate a synthetic SPARQL JSON result of a given size (similar to the one we did) and measure `parseResults()` time. For example, generate 10k, 50k, 100k bindings and ensure parsing scales roughly linearly. This can be integrated with Vitest (perhaps as a special test that is skipped or only run on demand due to time). Target: parse 100k results under 200ms on a modern dev machine.
  If it's slower, consider optimizations: e.g., avoid unnecessary object creation. One micro-optimization: currently we create a new `ParsedCell` object for each cell [155] [156]; that's fine. We could reuse some structure or precompute if needed. Unlikely to need.

- **Rendering Benchmark (Headless browser):** Use Playwright to automate a scenario:

- Launch a minimal HTTP server serving a known large SPARQL result. We can include a static JSON file of, say, 10k results, and make the endpoint respond with it (or spin up a lightweight SPARQL engine like Apache Jena Fuseki with a dataset – but that's heavier). Simpler: implement a dummy `/sparql` in Node that always returns our prepared big JSON with a slight delay to simulate network.
- In Playwright, configure SQUI's endpoint to that dummy endpoint and issue a query. Start measuring time when "Run" is clicked, and stop when the results table renders (could detect the DOM element of first result row).
- Measure memory usage if possible. Playwright's page API has `metrics()` that might give JS heap size. Or use Performance API from inside the page to inspect memory. We could at least log `window.performance.memory.usedJSHeapSize` after rendering.

Key metrics: - Time to **first row rendered** after query response (this includes network + parse + initial render). Ideally under 1 second for 10k results (not counting network). - Time to allow user interaction again (i.e., the UI should not freeze too long). - Peak memory usage on 100k results should ideally stay under ~ hundreds of MB.

- **Baseline vs Changes:** We should run these benchmarks now (with current implementation) and then after implementing features like multi-tab or raw view to ensure they don't introduce leaks or slowness. For example, if we allow multiple tabs with huge results open in each, memory could balloon – maybe limit number of tabs with heavy data loaded.

- **Target budgets:**

- Initial load (bundle download + init): aim < 2s on a 3G network (which implies keeping bundle small, ~<500KB gzipped as planned).
- Query execution: obviously depends on endpoint. But SQUI overhead should be negligible compared to SPARQL engine time. Still, we can set: "UI adds < 10% overhead to query latency for 10k results". If a query on endpoint takes 2s and returns 10k rows, SQUI should ideally render them in <0.2s after receiving (so total ~2.2s).

- For large results (100k), if endpoint streaming is not considered, maybe the user expects a wait. We can set a budget like "100k results fully displayed within 5s" (which would include network for 10MB result + parse ~0.2s + minimal rendering). 5s might be acceptable for such a large retrieval. If more, we might warn or encourage LIMIT usage.

- **Regression thresholds:** Incorporate these into CI if possible. Perhaps run a Playwright test on a fixed dataset and fail if time > X or if an operation times out. Because performance on CI can vary, we might not make it fail on slight differences but could record metrics over time.

- **Memory leak checks:** Use Playwright to open SQUI, run a query, then navigate away or close component, then force GC (if possible via Chrome dev protocol) and see if memory is freed. If not, we might have leaks. Already, ensuring `editorView.destroy()` and unsubscribes will help. We should also check if abort controllers from past queries could accumulate (but since we nullify them after use [148], it's fine).

- **Web Worker for Parsing:** If parseResults ever becomes a bottleneck (say on extremely underpowered devices), consider offloading it to a Web Worker thread. It's relatively straightforward: send the JSON text to a worker, parse and process into an array, post back. But given the <0.5s parse times expected, it might not be necessary. Monitor if UI freezes during big parse in testing – if so, a worker is Plan B. (Our tests should try e.g., 200k results on a decent PC to simulate worst-case; that might freeze the main thread for ~1s which some might find borderline.)

To implement the above plan, we can add a `benchmarks/` folder with a Node script for parse, and integrate a Playwright scenario in the test suite that prints timing. Possibly not run every commit on CI to avoid long times, but run on demand or nightly.

In summary, SQUI is already leveraging efficient techniques (virtual scroll, incremental rendering). The main performance concerns – large result handling and multi-tab resource use – can be addressed by the combination of limiting, lazy-loading (for infinite scroll if chosen), and prompt user feedback (warnings for large results, etc.). By instituting automated benchmarks, we can ensure new features (like raw view or additional processing for autocompletion) do not regress performance beyond acceptable budgets.

Accessibility Audit (WCAG 2.1 AA)

SQUI's UI uses Carbon Design System components, which generally adhere to accessibility guidelines (Carbon's React components are WCAG 2.1 AA compliant, and the Svelte port should be similar). Many good practices are evident: form inputs have labels (EndpointInput uses Carbon's TextInput with labelText prop [157]), the editor has an `aria-label="SPARQL Query Editor"` for screen readers

[158], and focus states are handled by Carbon styles. However, to truly achieve **WCAG 2.1 AA**, we need to address some gaps:

- **Structured Headings & Landmarks:** Ensure the component has clear structure for assistive tech. Currently, the layout is a `<div class="squi-container">` with toolbar, then main content split. There's no usage of ARIA landmarks (like `<nav>` or `<main>`). Given this is an embedded component, not a full page, it might not need landmarks, but if it's a standalone app, consider marking the results section with `role="region"` and `aria-label="Query Results"` (there is a translation for a11y.results) [159]. For example, add `role="region" aria-label={$t('a11y.results')}` to the results container.

- **Keyboard Navigation Order:** Verify tab order: likely, tabbing goes to the Run button, then Endpoint field, then possibly into the editor (CodeMirror content). Actually, Carbon's ComboBox might capture tab differently. We should test: Initially focus might go to endpoint input (since it's first in DOM in toolbar after Run? Actually in Toolbar snippet, RunButton is before EndpointSelector [160] [161], so tabbing from Run goes to endpoint input). From endpoint, hitting Tab might skip into the editor or go to SplitPane's first pane (editor) automatically? The editor is not an input element per se, but CodeMirror sets tabindex or role=textbox which should be focusable. We should ensure the editor is in tab order. CodeMirror's `.cm-content` gets role textbox; it likely is focusable via Tab by default (especially since we added contentAttributes with role/textbox, we might need to add `tabindex="0"` explicitly). Check if that's needed. If a screen reader cannot focus the editor via keyboard, that's a major issue. **Fix:** Add `EditorView.dom.setAttribute('tabindex', "0")` or similar content attribute. Possibly CodeMirror already does it, but we should verify.

- **Focus Visibility:** Carbon buttons and inputs have visible focus outlines by design. CodeMirror's focus outline was disabled with `outline:none` in our CSS [162] – reconsider that. We removed CodeMirror's default outline for better styling, but we should replace it with a clearly visible focus indicator (maybe a ring or border). Currently `.cm-focused` is set to no outline [162]. Suggestion: apply a focus style consistent with Carbon (e.g., a 2px solid highlight). WCAG requires visible focus for keyboard users.
  Acceptance Criterion: When the editor is focused, there is a visible indication (e.g., a glow around the editor area or a caret blinking which at least indicates focus inside). The caret does blink, so user might notice that. But an outline on the editor container might be clearer.
  Fix: Remove or override the `outline:none`. Possibly use a subtle border color change for the `.sparql-editor-container.cm-focused`.

- **Screen Reader Feedback:**

- When executing a query, currently the UI shows a loading state "Executing Query, please wait..." in the results area [163]. But a screen reader user might still be focused in the editor or elsewhere and not know something happened. We should use ARIA live regions for announcements. For instance, the "Executing Query" message could be given `role="status"` or `aria-live="polite"`. Carbon's InlineLoading might not automatically announce (it's just a spinner with text). Possibly wrap the loading text in a `<div aria-live="polite">`. That way, when it appears, SR will read "Executing Query. Please wait...".

- Similarly, when results are ready or an error occurs, announce it. The InlineNotification likely has `role="alert"` or similar (need to confirm Carbon's spec). If not, add `role="alert"` to ErrorNotification wrapper so it's read immediately. The translations include 'ask.true'/'ask.false'

for ASK results [164]. Perhaps announce the ASK result with an aria-live too ("ASK Query Result: TRUE").

- **Table Accessibility:** The results DataGrid currently generates a bunch of `<div>`s for cells and rows (since it's a grid, not a semantic table). This is problematic for screen reader users – they cannot navigate cell by cell or know headers. WCAG recommends using actual `<table>` for tabular data or implementing a complex ARIA grid pattern. The SVAR grid probably does not automatically set ARIA roles. We should inspect the DOM it outputs. If it's purely divs, then for SR it's just a big list of unrelated text. This fails Success Criterion 1.3.1 Info and Relationships (structure not conveyed).
  Potential Solutions:

- Use ARIA roles: e.g., add `role="table"` on the container, `role="row"` on row divs, `role="columnheader"` on header cells, `role="cell"` on data cells, and `aria-colindex/aria-rowindex` attributes. If the grid library doesn't do this, maybe we can extend it or post-process the DOM. This is quite involved but doable. Alternatively, provide an **accessible table alternative**: e.g., a hidden `<table>` that mirrors the visible grid for SR. For instance, if result row count is not huge (below some threshold, say 1000), we could render a off-screen HTML `<table>` with the same data (using `<th>` and `<td>`). If above threshold, maybe skip because it would be too heavy/infinite for SR to go through anyway. But at least for modest results, SR users can review the table properly.
- Given time constraints, a simpler short-term fix: when a screen reader user toggles to Raw view (which is just text), that might be easier for them to read with virtual cursor than the scattered grid. We could mention in documentation that screen reader users may prefer raw mode, or even automatically default to raw mode if we detect a screen reader (not trivial to reliably detect). But better to build actual accessibility in the table.

- *Plan:* Mark this as a known issue to tackle. Possibly raise an issue with `wx-svelte-grid` maintainers to add ARIA roles. If not, we might fork or wrap it. This is somewhat large effort, but necessary for full compliance.

- **Color Contrast:** Because we rely on Carbon themes, contrast is mostly sufficient. Carbon's g90 and g100 (dark themes) have proper text colors for dark background [165]. We should double-check one thing: the "placeholder" text (like Endpoint placeholder "Select or enter SPARQL endpoint") – Carbon likely uses a lighter gray. We should ensure it meets the 4.5:1 contrast ratio if it's considered input text. If not, we might not worry (placeholders are exempt from strict contrast rules but good to have if possible). Carbon being an enterprise design system is likely fine here.

- **Focus Trap/Modals:** If any modal dialogs are introduced (for settings, templates, etc.), ensure focus trap inside them and return focus to trigger on close. Right now, we have no modals, only the built-in Carbon ComboBox which handles its own focus (closing dropdown returns focus to input). The prefix addition maybe will have a modal – keep this in mind.

- **ARIA Labels & Hints:** Already a lot of translations cover ARIA (`a11y.sortColumn`, `a11y.filterColumn` etc.) [159] [166]. We must ensure these are actually used in the DOM. For example, if we implement column sorting, when a user focuses a header, the ARIA-label could read "Sort by ?name (ascending)" or something – presumably the translation key is intended for that. We should attach these to elements accordingly (like `aria-label={$t('a11y.sortColumn', { column: varName })}` on the clickable header div).

- **Testing with Axe and NVDA/JAWS:** We should run axe-core on the rendered output. Possibly integrate `@storybook/addon-a11y` (which is already in devDeps [167]) for automated checks. The Storybook or direct testing should reveal issues like missing landmark or table semantics. Use these reports to guide fixes.

Accessibility Checklist (WCAG 2.1 AA):
We can create a checklist to track compliance: - [x] **Keyboard Navigation** – All interactive elements reachable via Tab (Run, Endpoint, Editor, any toggles). *Status:* Likely yes, but verify Editor focus (add tabindex if needed).
- [x] **Focus Visible** – Clear focus style on all controls (need to address Editor outline).
- [x] **Meaningful Sequence** – DOM order is logical (Toolbar then Editor then Results). *Status:* Yes, UI flows top-down.
- [x] **Headings/Labels** – Endpoint input label is provided visually and via `labelText` (Carbon handles `for` association) [157]. Buttons have text or `title` (Run button has `title={$t('toolbar.runTooltip')}`) [168] – good. Check that the inline loading "Cancelling..." has accessible name (it's within the button, so yes screen reader will read "Cancelling...").
- [ ] **Role/Structure** – Add ARIA roles for results grid or provide alternative table. *Open*.
- [x] **Contrast** – likely all text vs background is > 4.5:1 given Carbon palette. We should test a couple sample screens with axe or manual calculation, but Carbon is designed for AA. The placeholder text might be ~#6f6f6f on #fff which is ~5.7:1 – okay. Dark theme texts seem fine by Carbon tokens.
- [x] **Live Updates** – Announce dynamic content: add `aria-live` to loading status and error notifications.
- [ ] **Error Identification** – If an input error occurs (e.g., invalid endpoint URL), Carbon shows an invalid state with red border and message. Ensure that is announced (Carbon's TextInput likely sets `aria-invalid` and links to error text with aria-describedBy). EndpointInput uses Carbon's mechanism with `invalid` and `invalidText` bound [169] [157], so it should announce "Invalid endpoint URL". Good.

From this, major outstanding tasks are making the results accessible (which is complex but crucial for screen reader users) and adding live region for query feedback. Everything else is minor tweaks.

Snippet for adding a live region for query status:
In `ResultsPlaceholder.svelte`, around the loading message:

```
{#if state.loading}
  <div class="placeholder-content" aria-live="polite">
    <h3>{$t('results.loading')}</h3>
    <p>{$t('results.loadingHint') /* e.g., "Please wait..." */}</p>
  </div>
{/if}
```

This way, when that content appears, SR will read "Executing query, please wait". For result ready, perhaps when loading turns false and results appear, we could inject an off-screen announcement like `<div aria-live="polite">Query complete. {state.rowCount} results.</div>` hidden visually. That could use the translation `'results.rowCount': '{count} rows'` [170].

Error notification ARIA: Carbon InlineNotification likely has `role="alert"` built-in when kind='error'. If not, we can wrap it or add manually: e.g., `<InlineNotification ... role="alert" />` (though if their component doesn't pass down unknown props, we might need a wrapper div with role alert). We should test with a screen reader to ensure it's announced.

By implementing the above, we aim to meet WCAG standards: - **Perceivable:** Text alternatives (all icons have text or are decorative – the Play/Stop icons in buttons are purely decorative since the button also has text “Run” and the icon is given via `icon={Play}` prop on Carbon Button which likely labels it appropriately). We might want to hide the icon from SR or ensure the button label covers it. Carbon’s API might do that by default. - **Operable:** Keyboard access handled; no timing issues. - **Understandable:** Consistent labels, no unusual jargon. Possibly ensure language is set (`<html lang="en">` in standalone). - **Robust:** Should work with various browsers and assistive tech.

We will iterate with actual accessibility testing tools (like axe) and screen readers to catch any missed issues. Accessibility is an ongoing effort; the above steps will significantly improve the experience for keyboard-only and screen reader users, moving SQUI closer to the “fully accessible” claim in README.

Security Review

As a client-side component, SQUI’s security considerations mostly revolve around **content handling** and **user-provided inputs**. There are no server components in SQUI, thus classes of vulnerabilities like SQL injection are not applicable. However, a malicious SPARQL endpoint or query result could potentially introduce risks if not handled properly. We evaluate key areas:

- **Cross-Site Scripting (XSS):**
  SQUI displays data from SPARQL endpoints, which might include arbitrary strings (including HTML snippets) in results. We must ensure such content is not executed as HTML/JS in the page. The current implementation is safe by default because:
- Query results are inserted as text content. For example, the DataTable creates text nodes for cell values via Svelte’s binding (we use `{cell.value}` in the grid, which will escape any HTML characters by default). In our DataTable, we actually pass values to the grid which then presumably does `textContent`. Even our error details, we wrap in `<pre>` with no `{@html ...}` usage, so HTML tags appear inert [74] [75]. This is good.
- We should double-check that we never use Svelte’s raw HTML binding (`{@html ...}`) with endpoint data. Quick grep: the codebase does not appear to (the only potential is if `InlineNotification` or others dangerously set innerHTML, but Carbon likely doesn’t for content we pass as props – it treats subtitle as plain string).
- Even when we implement clickable links for URIs, we must not use `innerHTML` to set them. Instead, create anchor elements properly with `href` and text. Svelte’s binding of attributes will escape as needed. So as long as we avoid injecting raw HTML, we’re safe from XSS.

Potential XSS via Service Description: If we fetch a service description (RDF) and maybe display parts of it (like default dataset name), ensure we treat it as data, not markup. Probably not an issue now as we don’t display any untrusted HTML.

Prefix.cc: If we integrate prefix.cc suggestions, note that comes from an external JSON. That should be fine (prefix and URI strings, unlikely to contain markup – but even if, we treat them as text in suggestion list).

Conclusion: SQUI currently appears to **sanitize by construction** (no raw HTML injection). This is a strong security stance. We should maintain that discipline when adding features: - E.g., when showing Raw RDF/XML, do **not** render it as actual HTML. Show it in a `<pre>` or editor component. If someone wants

to actually render that XML, it's out of scope; we treat it as code. - If providing a "Copy to clipboard" for results, ensure we copy text, not any hidden HTML element content.

- **Content Downloads & Injection:**
  When enabling downloads (CSV/TSV), consider using `Blob` and `URL.createObjectURL` to trigger a download. Set the `download` attribute on an `<a>` element to a safe filename, e.g., `results.csv`. One subtlety: if the CSV contains untrusted content, opening it in Excel is fine, but opening in a browser as file could theoretically run as HTML if content is crafted (though CSV doesn't execute HTML, but if user changes extension to .html then opens, that's on them). Not a big concern. One thing: If we allow downloading JSON results, maybe default to `.json` extension. If an attacker somehow got a victim to download a malicious JSON and then rename to `.html` and open… that's a very contrived scenario. JSON doesn't run by itself.
  Just ensure when constructing the Blob for download, use the correct MIME type (e.g., `text/csv` or `application/json`) so that the browser knows how to handle it (although it will likely just save it, but still good practice). For RDF like Turtle, use `text/turtle` etc.

- **CORS & Network Security:**
  SQUI must abide by browser CORS. It cannot access endpoints that don't allow it. We have detection for CORS errors and message telling user about it. There is no way to bypass CORS in the client (and we should not attempt shady workarounds like using JSONP – not relevant to SPARQL). The recommended approach for closed endpoints is use a proxy or have user enable CORS on their endpoint. We can mention that. The security implication: We do not override any browser security – which is correct. We should ensure the Endpoint URL validation forbids dangerous schemes:

- It currently requires `http:` or `https:` [171]. That prevents someone from doing `javascript:alert(1)` or `data:` or `file:` as an endpoint, which is good. So no XSS via endpoint input either (plus if they tried, nothing would happen except a fetch error maybe).

- Also by disallowing file://, we avoid local file access issues. Good.

- **Open Redirect or External Navigation:**
  The UI might allow opening a URL (when user clicks an IRI link in results, or the endpoint link maybe). We have to ensure `target="_blank"` and `rel="noopener noreferrer"` on those anchors to prevent any potential reverse tabnabbing attacks. The plan is exactly that (we noted it in Linkification fix) – e.g. `<a href="{uri}" target="_blank" rel="noopener noreferrer">prefix:term</a>`. This prevents the opened page from controlling our window. Also, ensure the link text is sanitized (we'll likely use prefix abbreviation or full URI, which won't contain executable code, and it'll be text in anchor, so fine). We might add `noopener` for safety on any programmatically opened windows (though we probably won't use `window.open` directly – just anchor clicks).

- **Endpoint URL Storage:** If we implement saving custom endpoints in localStorage (which we do in EndpointCatalogue when user enters a new URL, it adds to catalogue store which presumably persists?), actually I see `endpointCatalogue` uses a writable with default list [172] [173]. It does not persist to localStorage currently. It might be nice to store that so next session the custom endpoints remain. If we do, be mindful not to store sensitive info (like if some endpoint requires auth, though we have no auth support now, or if user accidentally enters a credential in the URL – which they shouldn't as SPARQL doesn't use user:pass in URL typically). Probably fine. The endpoints are public or at least not secret. If we ever support authenticating to endpoints

(OAuth, API keys), that's a whole new dimension – but currently out of scope. We can mention: *if endpoint requires HTTP auth or API key, SQUI doesn't directly support it. The user might use a proxy that injects auth.* So no secrets handling in our code, which simplifies security.

- **Dependency Security:** All dependencies are front-end. Carbon and CodeMirror are reputable. `wx-svelte-grid` – just be cautious if it's not as widely used; ensure no known vulnerabilities (shouldn't, it mostly manipulates DOM for grid). The code does not use eval or risky constructs.

- **Potential DoS vectors:** A malicious endpoint could try to crash the UI by returning extremely large results or very deeply nested JSON causing parser issues, etc. We mitigate by the maxRows limit as discussed. Perhaps also put a hard cap on response size: e.g., if Content-Length header is > some threshold (like 100 MB), we might auto-cancel and warn rather than attempt to parse. We do have a 60s timeout on queries [174] [175], which covers extremely large or slow responses. That's good (prevents endless hang). Additionally, AbortController abort is used at 60s, yielding a 'timeout' error to user. The user might then refine query.

- **Injection in Queries:** The user writes SPARQL; if they make a harmful query (like `DROP ALL`), that's on them. We are not protecting against that scenario because it's legitimate SPARQL, and SQUI's not running queries on behalf of untrusted users (it's an end-user tool). But if SQUI were integrated into a multi-user environment where some could input queries for others to run, that environment would need protections. Out of scope for us (SQUI is a client, not a shared service).

- **Privacy:** Not strictly security, but note that using SQUI will send queries and maybe parts of data to endpoints. We should inform users that queries might be logged by endpoint servers. Also, if the user enables things like prefix.cc suggestions, that means any prefix they type could trigger a call to prefix.cc (leaking that they're interested in a certain prefix). Minor issue, but perhaps mention in docs if needed (like a privacy note: "Autocomplete may fetch data from prefix.cc – disable if not desired"). This is akin to how browser search autocomplete works – usually acceptable.

Threat Model Summary:
The primary threat addressed is XSS – SQUI mitigates it by escaping all endpoint-provided content. Another threat: *Malicious endpoint causing UI misbehavior.* For example, endpoint could deliberately send back a huge number of tiny results to cause memory issues – we mitigate with row limit and can add a check on response size/time. Or endpoint could serve a slow streaming response to tie up the browser (but our 60s timeout handles that). An endpoint could also return misleading content (like a Data URL disguised as a result that if clicked does something) – but since we treat everything as text, even if an endpoint returned `<script>alert('XSS')</script>` as a literal, it will display as literal text, not execute. If the user manually copies it and pastes in console, that's on user. One could imagine an endpoint returning a "URI" value that is actually a `javascript:alert(1)` URI. If we naively made it clickable, clicking it would execute JS in context – **this is important**: when linkifying URIs, filter out dangerous URI schemes. We should only allow `http:` or `https:` for clickable links (and maybe `ftp:` if someone still uses RDF with ftp links?). We should definitely prevent `javascript:` or `data:` links. - Implementation: when creating anchor href, if `cell.value.startsWith('http')`, allow. Otherwise, do not hyperlink or at least prefix with a warning. But realistically, SPARQL URIs are usually http(s) or urn:. If `urn:`, we can't open that anyway (maybe no link). If `javascript:` improbable, but to be safe: e.g.,

`if(url.match(/^(http|https|ftp):/)) { create link } else { just display text }`. So add this check.

- **Endpoint URL validation and SSRF:** Since SQUI is client-only, SSRF (Server-Side Request Forgery) isn't a concern – we're not a server making requests. The user's browser will only request the endpoint they specify. If the user is running this on their own machine, they could potentially query internal endpoints (like `http://localhost:internalport/sparql`). That's by design (the user presumably wants to query their local endpoint). There's a slight scenario: if someone hosts SQUI on a public site and an attacker convinces a user to use it to query a local resource (like by embedding an endpoint URL that's internal), the attack could be to exfiltrate data from user's internal network (if CORS allowed from that internal resource to the site). This is a **less likely scenario** because internal SPARQL endpoints typically don't allow arbitrary origins. But if a user uses SQUI on example.com to query an intranet SPARQL endpoint that *does* allow cross-origin from `*`, then example.com (the site) could initiate queries to internal data. However, SQUI doesn't automatically fetch anything except when user actions occur (user enters endpoint and hits Run). If an attacker had control of a SQUI instance and set a default endpoint to a common internal address (like `http://localhost:5000/sparql`), they could attempt to get data. But they'd still need the user to open that page and click run (or maybe auto-run on load if coded so). This is somewhat similar to how one must be cautious with any client app that can connect to arbitrary URIs (like a web-based port scanner concept). We could mitigate by not auto-running queries without user interaction. Currently, queries only run on user click or Ctrl+Enter. So that's good – no malicious auto-run. We'll mention this risk academically but it seems low.

- **Future Auth:** If adding features like HTTP Auth or API keys, we must store credentials safely (probably in memory, not localStorage) and use secure contexts. But as we have none now, skip.

Security Countermeasures Implemented:
- Strict URL validation in EndpointInput (no dangerous schemes) [176].
- Global `mode: 'cors'` on fetch (so no silent failure if CORS disallowed; it will error out which we catch) [177].
- Content outputs sanitized by default (no `{@html}` usage).
- User actions required for network calls (no background fetching besides maybe prefix.cc on typing – which is triggered by user input anyway).

Recommendations:
- Add the link-scheme filter for result links (only allow http/https links to be clickable).
- When providing shareable links, be careful not to include any sensitive info (e.g., if in future we support an API key parameter, don't put it in URL by default). But currently, nothing sensitive in query or endpoint normally.
- Document that if the endpoint uses HTTPS with self-signed cert, the browser will block or ask user (just a note, as it's common in dev setups – user can still accept the cert manually). Not a security hole, just something to mention for DX.
- Provide guidance for deploying the standalone app in a secure way (e.g., if someone hosts it, serve over HTTPS especially if querying HTTPS endpoints to avoid mixed content issues – not exactly security, more reliability, but worth noting).

Given the above, SQUI is on track to be a **secure client** as long as best practices are followed. Security in this context is largely about not introducing XSS or leaks – which we have under control.

We will include these points in documentation: e.g., “SQUI does not store any sensitive data. Queries are executed in your browser and not sent to any third-party except the SPARQL endpoint you specify. (Autocomplete may contact prefix.cc – you can disable by …). Ensure your endpoint supports HTTPS and CORS for secure usage.”

By implementing the recommended checks (URI scheme filtering for links) and maintaining a strong content escaping policy, SQUI should maintain a high security posture.

YASGUI Comparison: Parity and Differentiation

The following table compares SQUI (SPARQL Query UI) with YASGUI (Yet Another SPARQL GUI, particularly the Triply-maintained version) across key features and developer experience aspects:

Feature	YASGUI (YASQE + YASR)	SQUI (Sparql Query UI)	Notes
Multiple Query Tabs	Yes – Tabbed interface with persistent tabs (saved in local storage). Tabs labeled “Query 1”, “Query 2”, etc., can clone/close tabs [178].	*Planned* – Single-tab in current version. Multi-tab UI scaffolding exists (flags, translations) but not yet functional [9].	SQUI’s roadmap includes full multi-tab with localStorage (Tasks 39–41) [15] [179]. Once implemented, it should match or exceed YASGUI’s tab UX (Carbon Tabs styling vs YASGUI’s custom).
Query Editor	YASQE (CodeMirror 5-based). Supports syntax highlighting, basic autocompletion of prefixes and some vocab terms (uses prefix.cc and LOV). Also does rudimentary syntax error underlining (CM5 lint) and keyboard shortcuts (Ctrl+Enter to run).	CodeMirror 6 with custom SPARQL mode [180] [181]. Autocompletion for keywords and known prefixes is implemented [89] [182]. No LSP or advanced lint yet. Has Ctrl+Enter execution and will have snippet/template insertion.	SQUI’s editor is more modern (CM6 offers better modularity). As SQUI adds async completions (LOV integration) and error checking, it can surpass YASQE’s capabilities. Also, SQUI can leverage CM6 features (multi-cursor, dynamic extensions) that YASQE (CM5) lacks.

Feature	YASGUI (YASQE + YASR)	SQUI (Sparql Query UI)	Notes
Results Viewer	YASR (Result renderer). Default view is an HTML table for SELECT (pagination by DataTables for large sets, which can lag). Also provides a "Raw" tab for the raw response (esp. for CONSTRUCT/ DESCRIBE) [106]. Offers simple chart plugins (bar chart, pivot table) as optional extras on small results.	SQUI uses a virtualized grid (handles 10k+ rows smoothly) [183] [98]. Currently only a table view for SELECT and a boolean display for ASK [184]. No built-in charts yet. "Raw" view and format switching planned (not yet implemented).	SQUI's virtualization is a big performance win over YASGUI's DOM table (which tends to break on ~5k+ rows). This is a **USP**: SQUI can handle bigger results in-browser without crashing [185] [98]. Once SQUI adds the Raw view for CONSTRUCT results and perhaps copy/download, it will be on par. Charting might remain YASGUI-only unless we add a plugin system.
Prefix Autocompletion	Yes – As user types a prefix and ":", YASQE calls prefix.cc for suggestions if prefix not known locally. Also auto-suggests prefix declarations for common vocabs when typing "PREFIX".	Yes – SQUI has a built-in list of common prefixes and suggests them on "PREFIX …" or on typing known prefix "rdf:" etc. [186] [187]. Also has a mechanism to fetch suggestions from prefix.cc (function `searchPrefixes()`) but not yet wired into UI [188] [189].	In current state, YASGUI might fetch a broader array of prefixes dynamically. SQUI's approach is local but can be extended. SQUI's prefixService also allows adding custom prefixes via config, which YASGUI didn't expose as cleanly. Both ensure users don't have to remember long URIs.
Vocabulary Auto-complete	Yes – YASQE queries the LOV API to suggest ontology classes and properties after a prefix is resolved (e.g., after typing "foaf:" it might list "name", "mbox"). This is a distinct feature that uses LOV's search.	Not yet – SQUI currently only completes a fixed list of terms for a few prefixes (rdf, rdfs, etc.) [190] [191]. No dynamic lookup of ontology terms is implemented yet.	YASGUI has an edge here for now. SQUI's design could incorporate LOV suggestions in future. This is a potential enhancement – implementing it would achieve feature parity and improve UX for ontology exploration.

Feature	YASGUI (YASQE + YASR)	SQUI (Sparql Query UI)	Notes
Execution & Protocol	Supports SPARQL 1.1 (and 1.2 to the extent endpoints do) via GET or POST. Uses GET for queries below ~2k chars, POST otherwise (similar to SQUI's strategy). Does not officially support SPARQL Update in the UI (YASGUI is read-only by default, though underlying Yasqe can send updates if configured).	Supports SPARQL 1.2 (incl. new protocol features). GET vs POST logic implemented [51]. Will support updates (after bugfix) – SQUI intends to allow UPDATE queries (with appropriate POST) whereas YASGUI's UI had them disabled unless manually enabled.	If SQUI enables updates, it becomes a more general client out-of-the-box (caution: ensure user is aware when they execute an update!). YASGUI typically was used for queries; updates often were hidden behind config. So SQUI can differentiate by offering a full 1.2 experience (Query and Update and Graph Store perhaps).
Error Feedback	Basic – YASGUI shows endpoint HTTP errors or parse errors in an alert or in the results area (often as an HTML message from endpoint). Not very user-friendly messages (just whatever the endpoint returned).	Rich – SQUI categorizes errors (syntax vs network vs CORS) and shows a styled notification with details [192] [48]. Messages are more user-friendly (e.g., "Bad Request: Invalid SPARQL query" instead of just 400) [70].	SQUI provides a better DX here. The inline error notifications with a copy of details are a plus [72] [68]. This helps users debug queries. This is a **strong differentiator** – YASGUI's error handling often confuses novices (just sees "ParseException at line 1…" with no help). SQUI can further leverage this (maybe link to documentation when syntax error occurs, etc.).

Feature	YASGUI (YASQE + YASR)	SQUI (Sparql Query UI)	Notes
Result Format Options	YASGUI allows switching result format in the UI via a dropdown (JSON, XML, CSV, TSV). If switched, it re-runs the query and then either shows the raw or tries to display (for CSV it might just force download). It also provides a download button for results.	SQUI will have format options (not yet exposed). Internally can fetch those formats [56], but currently always uses JSON for display. No download button yet.	Once implemented, SQUI's format switching/download will be similar. One advantage SQUI can have: allow selecting format *before* query (so the user can get CSV directly without rendering). YASGUI's UI requires you to run then switch, or use the API. SQUI's planned UI could streamline that.
Saved Queries & Share	YASGUI auto-saves queries in local storage under each tab. It also has a "Share" functionality that generates a sharable link (through yasgui.org service or by embedding in URL). E.g., one can click "Copy Link" and it produces a URL with the query and endpoint encoded (possibly compressed).	SQUI not yet saving or sharing. It has no persistence beyond page refresh (once multi-tab with localStorage is done, that covers save). Share links not implemented yet.	This is a **must-have parity feature**. YASGUI's share is very useful for collaboration. SQUI's opportunity: implement share as pure client-side (maybe using URL fragment encoding to avoid server). This is on the 2-month roadmap. When done, SQUI will match YASGUI here, potentially with an even simpler UX (one-click copy link with built-in compression).

Feature	YASGUI (YASQE + YASR)	SQUI (Sparql Query UI)	Notes
Default Endpoints & Catalog	YASGUI by default comes with public endpoints (DBpedia, WikiData) in a dropdown for easy selection. It also remembers past endpoints used.	SQUI has a built-in catalogue of popular endpoints (DBpedia, Wikidata, LOD Cloud, UniProt) shown in the Endpoint selector [172]. It allows adding custom endpoints which persist for the session (and soon could persist across sessions via localStorage) [193] [194].	Both provide a quick start with known endpoints. SQUI's endpoint selector UI (Carbon ComboBox with name/description) is arguably more polished than YASGUI's simple dropdown. SQUI also validates the URL format and warns on HTTP vs HTTPS [176] [195] – YASGUI lacked this. This improves user experience and safety.
UI Theming	Minimal theming – YASGUI has light theme by default; a dark theme was not official (though users could override CSS). The Triply fork might allow some CSS variables but not documented widely.	Full theming via Carbon – SQUI offers 4 themes out of the box (white, g10, g90, g100 for various light/dark contrasts) [2] [44]. Easy to switch via prop.	This is a **SQUI advantage**. Many developers will appreciate a dark mode. YASGUI in practice often had to be restyled manually for embedding in sites. SQUI's design system approach yields a consistent look in different environments with minimal effort.
Plugin Ecosystem	Yes – YASGUI's YASR supports plugins. E.g., the Pivot Chart plugin or RDF graph visualizer plugin. These require additional JS includes and are not widely used, but exist. Also, YASQE had plugins (like autosuggest via external services).	No plugin system (yet). SQUI doesn't yet expose extension points for custom result renderers or editor plugins. However, SQUI's modular code could be extended by forking or contributing rather than runtime plugin.	It's unlikely a casual user writes a SQUI plugin at this stage. Instead, we can incorporate popular requests directly. If a need arises (say, visualize a geo SPARQL result on a map), we might add an API to plug that in. For now, not a focus. YASGUI's plugin usage in the wild is limited (most stick to table/raw). So lack of plugin support is not a deal-breaker.

Feature	YASGUI (YASQE + YASR)	SQUI (Sparql Query UI)	Notes
Integration & Embedding	YASGUI can be included via script or npm. It creates a global Yasgui object that you attach to a div. It doesn't naturally integrate with frameworks – devs sometimes face challenges embedding in React/ Angular due to it manipulating the DOM.	SQUI is designed as a Svelte component and can compile to a web component. This makes it easier to embed in modern frameworks (just include the custom element or mount the Svelte component). Also, typed API for props instead of imperative calls.	**DX win for SQUI:** Using a Web Component with Shadow DOM (planned) will isolate styles and allow dropping `<sparql-query-ui>` into any page. YASGUI could conflict with existing styles or require manual initialization code. SQUI's approach (once packaged) should be more plug-and-play.
Release Cadence & Support	YASGUI (Triply fork) is maintained but updates are infrequent (the core YASQE/YASR are fairly stable; last major update was to support RDF-star, etc.). Community support via issues is okay. Documentation is somewhat scattered between older YASGUI docs and Triply's site.	SQUI is new (under active development). Frequent updates expected as it approaches v1. There is a comprehensive CLAUDE dev log and the code is fresh which may yield quicker issue turnaround. However, it's not yet battle-tested by community.	For an enterprise considering adoption, YASGUI is proven, SQUI is promising. Over the next 2 months, as SQUI reaches feature parity and gets community usage, it could become the preferred choice thanks to modern tech and better UX. Active development is a plus (fast incorporation of user feedback). Documentation for SQUI is in progress but will be consolidated (unlike YASGUI where some info is outdated).

Migration Path: For current YASGUI users, adopting SQUI should be relatively straightforward once SQUI hits 1.0: - Embedding: Instead of instantiating Yasgui in JS, they can include the SQUI component. We can provide a small adapter if needed – e.g., a script that reads Yasgui's config (like default endpoint, saved queries from localStorage) and feeds them into SQUI's props. Since YASGUI already stores tabs in localStorage under a key, we could write a migration function to import those into SQUI's format. For example, YASGUI stores a tab's query in localStorage `yasgui._tabCache` – we could parse that and recreate tabs in SQUI. This could be an optional tool or just instruct advanced users how to manually copy queries. - Feature mapping: Everything a user did in YASGUI (writing queries, running, exporting results, switching tabs) will be doable in SQUI, often more easily. We should highlight improvements like dark mode and performance to entice them. - Potential pitfall: If someone wrote custom code on top of YASGUI's API (like using Yasqe/Yasr programmatically), they'd have to adapt to SQUI's API (which is more prop-driven or store-driven). We can assist by documenting SQUI's component methods (e.g., if we

expose methods like `.getQuery()` or `.setQuery()` via component or stores). Perhaps provide a mini migration guide: “In YASGUI you did X, in SQUI do Y”. - As SQUI stabilizes, we might reach out to known YASGUI users (like DBpedia’s public editor, etc.) to try SQUI.

Unique Selling Points (USPs) of SQUI: - Modern UI/UX (Carbon theming, dark mode, responsive design possibly better than YASGUI which wasn’t mobile-friendly at all – though SPARQL on mobile is niche, SQUI likely can adapt to smaller screens more gracefully due to Carbon’s grid and flex usage). - Performance on large results – no other web SPARQL client handles 100k results smoothly in-browser to our knowledge. - Accessibility – making SQUI the first SPARQL UI that’s truly accessible would open it to new user groups (this could be a selling point for government or academic projects requiring AA compliance). - SPARQL 1.2 readiness – features like property paths shorthand, new functions, etc., SQUI will incorporate those in highlighting and such sooner. - Developer-friendly integration – as noted, web component usage and a clean API.

In contrast, YASGUI’s moat was mainly familiarity and being “good enough” for many. SQUI can create a new standard if it achieves all planned features.

Matrix Summary: SQUI is on track to achieve parity with YASGUI on core features (multi-tabs, share links, autocompletion) within the next iteration, while already offering **significant improvements in UI theme, error handling, and result performance**. The migration path is mostly smooth: queries themselves move over 1:1, and the concepts are the same. We will ensure any unique YASGUI aspects (like how it saved unnamed queries or any default behaviors) are either replicated or consciously improved upon in SQUI.

Roadmap & Backlog

Based on the analysis, here is a proposed development roadmap with prioritized tasks. The roadmap is divided into **Quick Wins (next ~2 weeks)**, **Must-Haves (next ~2 months)**, and **Nice-to-Haves (later)**. Following that, I outline specific GitHub issues (with suggested labels and acceptance criteria) and even some PR diffs for immediate fixes.

Quick Wins (Next 2 Weeks) – High Priority Bugfixes & Essentials

1. **Enable Real Query Execution (Replace Stub)** – *Bugfix*: Remove the mock query service and route execution through the real `sparqlService`.
2. *Acceptance Criteria:* Running a query sends an HTTP request to the specified endpoint and displays actual results. No more hardcoded mock data.
3. *Steps:* Implement the RunButton.svelte changes discussed (using resultsStore.executeQuery) [6], remove QueryExecutionService references. Test against DBpedia and Wikidata endpoints for real responses.
4. *Label:* `bug`, `P0`. *Effort:* **S** (couple of lines fix).
5. **Fix SPARQL Update Handling** – *Bugfix*: Ensure UPDATE queries use proper HTTP POST with `Content-Type: application/sparql-update`.
6. *Acceptance Criteria:* If user enters an INSERT/DELETE/LOAD query and hits Run, the request is a POST with body (no GET), and the endpoint executes it. If endpoint returns success or error, SQUI shows appropriate feedback.

7. *Steps:* Modify `determineMethod` to force POST for updates [51], adjust `executePost` to use sparql-update content type when queryType is 'UPDATE' [53]. Possibly add a visual indicator (maybe the Run button could say “Run Update” if it detects an update query for clarity, though not required).
8. *Label:* `bug`, `protocol`. *Effort:* **S**.

9. **Error Detail Preservation** – *Enhancement*: Pass full `QueryError` to the UI so users can expand error details.

10. *Acceptance Criteria:* On a syntax error, the error notification “Error Details” section shows the server’s message (e.g., “Line 1: Unexpected '}'”). On CORS error, details show the suggestion text we set (already does). No regression in error messages.
11. *Steps:* Change resultsStore as per diff (store QueryError object instead of just string) [196] [68]. Update ErrorNotification prop types if needed. Test by causing a syntax error and a 500 error and ensure details appear.
12. *Label:* `enhancement`, `DX`. *Effort:* **S**.

13. **Add Download Button (Simple)** – *New*: Provide at least a basic “Download Results (JSON)” button for now.

14. *Acceptance Criteria:* After a query, a “Download” icon/button is enabled. Clicking it triggers the browser to download a file `results.json` containing the SPARQL JSON results of the last query. (Later we’ll extend to other formats, but JSON is straightforward since we already have it parsed – we can reserialize or use the original text via rawData if we store it).
15. *Steps:* Use Carbon’s Button with download icon in the toolbar or results header. On click, call `JSON.stringify(resultsStore.data)` and create a Blob, set `a.href = URL.createObjectURL(blob)` and simulate click or use FileSaver if needed. The user gets a JSON file. If resultsStore.data is null or an error, disable the button.
16. *Label:* `feature`, `easy-win`. *Effort:* **S** (one afternoon).

17. **Live Query Feedback (ARIA)** – *Accessibility*: Add `aria-live="polite"` to the query executing message.

18. *Acceptance Criteria:* When user runs a query, screen reader announces “Executing query, please wait”. When finished, it announces either “Query complete. X results.” or the error message.
19. *Steps:* Add `aria-live` as shown earlier, and possibly inject a status update on completion by using a visually-hidden div. Also mark the InlineNotification with `role="alert"`. Test with NVDA or VoiceOver.
20. *Label:* `accessibility`, `a11y`. *Effort:* **S**.

21. **Linkify URIs in Results** – *Feature*: Turn URI cells into clickable links (with safety measures).

22. *Acceptance Criteria:* URIs in the results table are displayed as shortened (prefixed) links. Clicking opens the resource in a new tab. Non-HTTP URIs remain plain text.
23. *Steps:* Implement abbreviation using prefixService for display (e.g., store both full and abbreviated in ParsedCell or compute on render). Possibly use a small Svelte component for a cell: `<UriCell value={uri} prefix={prefix} />` that returns `<a`

`href=...>prefix:local</a>`. Ensure `rel="noopener noreferrer"` and `target="_blank"`. Filter out dangerous schemes (only hyperlink http/https/ftp). Test with a known URI like `http://dbpedia.org/resource/Berlin`.

24. *Label:* `feature`, `UX`. *Effort:* **M** (because integrating with the DataGrid might be a bit tricky, but we can perhaps use grid's `cellRenderer` if available, or re-render links after grid loads using querySelector – hacky but for first iteration might do).

(If 6 looks too heavy for 2 weeks, it could slip to next phase, but it's such a visible improvement that I'd try to include it.)

Must-Haves (Next 2 Months) – Major Features & Improvements

1. **Multi-Tab Interface** – *Feature*: Implement multiple query tabs with localStorage persistence.
2. *Acceptance Criteria:* Users can open a new tab (via a "+" UI), switch between tabs, and close tabs. Each tab maintains its own query text, endpoint, and results independently. Tabs are saved such that reloading the page restores them (including unsaved queries). Limit of ~10 tabs or so for performance.
3. *Steps:* Create `QueryTabs.svelte` using Carbon Tabs or custom. Manage state: perhaps a TabsStore that holds an array of {id, query, endpoint, results} or indexes to stores. Probably instantiate separate queryStore/resultsStore for each and switch the $state bindings based on active tab. Use localStorage (window.localStorage or Svelte's `persist` store if available) to save tabs on change and load on init. Handle naming: use "Query 1" or if query has a PREFIX with a base (no, that's too complex). Just enumerate.

4. *Label:* `feature`, `tabs`, `P1`. *Effort:* **L** (the biggest single feature, likely several days of coding and testing).

5. **Raw Results View & Format Selector** – *Feature*: Add ability to view raw response and switch output format.

6. *Acceptance Criteria:* A toggle (e.g., segmented control labeled "Table / Raw") appears above results. For SELECT/ASK, default is Table; for CONSTRUCT/DESCRIBE or when user chooses a non-JSON format, default to Raw. Raw view shows the raw text (with basic formatting or highlighting). Additionally, a dropdown or buttons to choose result format (JSON, XML, CSV, TSV, TTL, etc.) **before** running the query. When chosen, running the query will fetch that format and show in Raw view (or table if it's JSON/XML for SELECT). The Download button also reflects the chosen format (or offers multiple).
7. *Steps:* Implement `view` toggle in UI (bound to resultsStore.view which exists [108]). If user switches to Raw, and if we haven't stored rawData, we might need to reconstruct it (either by re-fetching with no parse or by stringifying JSON for JSON). Better: modify `sparqlService.executeQuery` to always keep a copy of raw response text if format != JSON (the code has a TODO about storing rawData in state [197] [198]). Do that. For format switching: perhaps a select element in the toolbar with options "JSON (default)", "CSV", etc. When changed, store it in resultsStore.format (exists [199]) and pass to executeQuery as option. Then re-run query if already executed (or wait for user to hit Run).

8. *Label:* `feature`, `formats`, `UX`. *Effort:* **M** (involves UI work and some refactoring of data flow).

9. **Query History Panel** – *Feature*: Provide a history of past executed queries.

10. *Acceptance Criteria:* A “History” button opens a side panel or modal with a list of last N queries (and endpoints, timestamps). Clicking a history item will either fill the current tab with that query or open a new tab with it (to avoid overwriting). Users can clear history.
11. *Steps:* Maintain a simple array in localStorage (maybe under key “squi_history”) of {endpoint, query, time}. Each executeQuery push to it (with max length). Implement a HistoryModal (Carbon has a Modal component we can use) that lists items (maybe just display first 50 chars of query plus endpoint host name). This interacts somewhat with multi-tabs (maybe if you click history while on a certain tab, it loads there).
12. *Label:* `feature`, `history`, `nice-to-have`. *Effort:* **M**.

13. **Sharing/Permalink** – *Feature*: Generate shareable link for current query.

14. *Acceptance Criteria:* A “Share” button creates a short link or a copyable URL that, when visited, loads SQUI with the query and endpoint pre-filled (and ideally runs it). The link should be as short as feasible (if not using an external service, base64 compress long queries into the URL fragment).
15. *Steps:* Implementation approach 1: encode JSON {endpoint, query} with LZ-string and append to `window.location.hash`. Then our app on startup checks if hash exists, decode it and populate state (set endpoint prop, queryStore.text, and auto-run resultsStore.executeQuery). Approach 2: Use a URL shortener API (less ideal due to external dependency). Probably do approach 1 for offline capability. Write utility functions compressQuery/decompressQuery.
16. *Label:* `feature`, `share`. *Effort:* **M**.

17. **Accessibility Improvements** – *Epic*: Continue a11y tasks: making DataGrid accessible or providing fallback table for screen readers, ensuring all controls have appropriate ARIA labels and roles.

18. *Acceptance Criteria:* Running axe on the app yields zero critical violations. Screen reader users can navigate results either through an ARIA grid or an alternative. Keyboard-only users can perform all actions (open/close tabs, toggle results view, etc.) without needing a mouse.
19. *Steps:* Possibly implement an offscreen `<table>` for SR with limited row count. Add ARIA attributes to the grid if feasible (need to dive into wx-svelte-grid’s DOM to add roles). Also, ensure focus order especially with new modals (history, share) – trap focus within modals and return on close. Use Carbon Modal’s built-in focus trap. Test thoroughly with NVDA.
20. *Label:* `accessibility`, `WCAG`. *Effort:* **M-L** (depending on approach taken for the grid).

21. **Documentation & Examples** – *Chore*: Expand documentation (in README or separate docs) for all new features.

22. *Acceptance Criteria:* README updated with multi-tab usage, format selection, etc. A small “Migration from YASGUI” guide added. Possibly publish a GitHub Pages demo site for SQUI so people can try without installing.
23. *Steps:* Use GitHub Pages to host the built `index.html` (since package has a deploy script). Update README usage examples for multi-tab (maybe none needed if it's internal, but mention share links, etc.).
24. *Label:* `documentation`. *Effort:* **S**.

These must-haves aim to reach feature parity with YASGUI and polish core UX by the 2-month mark, effectively making SQUI a drop-in modern replacement.

Nice-to-Haves (Future) – Further Enhancements

- **SPARQL Graph Store Protocol Support:** Possibly add ability to do Graph Store HTTP operations (CRUD on graphs). E.g., a secondary panel to upload/download RDF. This is pretty advanced and maybe not widely needed in the UI – likely low priority unless user requests.
- **Visualization Plugins:** For example, if result has ?subject ?predicate ?object, offer to visualize as node-link diagram (like YASGUI's experimental graph plugin). Or if results are numeric, offer a chart. This could be done via an optional toggle or a "Insights" tab that tries to guess a suitable viz. Nice, but can postpone.
- **Full SPARQL 1.2 Syntax Coverage in Editor:** Ensure new features like `VALUES ?x { UNDEF }` highlighting, new functions, etc., are recognized in the mode. Possibly update the mode to incorporate any 1.2 changes once the spec finalizes (small effort when needed).
- **Internationalization beyond English:** After structure is in place, get community contributions for languages (e.g., Spanish, German) and bundle them. Also maybe auto-detect browser locale to set default.
- **Theme Customization:** Maybe allow users to choose custom color themes beyond Carbon's four presets. But that's complicated given Carbon's design tokens. Probably not needed – Carbon's are enough.
- **Integration Testing on Multiple Endpoints:** building a CI matrix to run tests against a local Fuseki or Blazegraph to ensure compatibility and catch any differences (like some endpoints might not accept certain Accept header combos). This ensures broad compatibility.
- **Performance Monitoring:** Perhaps include a hidden debug panel to show how long last query took in parsing/rendering, or number of rows, as info for power users. Not necessary but could help users gauge if they hit limits.
- **Embed as React/Vue Component:** While Web Component should cover this, maybe provide wrappers for popular frameworks for better integration (especially if someone wants to directly manipulate SQUI's state from React, a small wrapper could expose those as props). Only do if requested.

Finally, here are **10 precise GitHub issues** (titles and acceptance criteria) and **5 suggested PR diffs** reflecting some of the above:

Issues:

1. **[P0 Bug] Query results using stub instead of real endpoint** – *Description:* Running any query returns the hardcoded example data, not the actual endpoint data. *AC:* Queries are executed against the provided endpoint and real results are shown. Mock data is removed. *(Fix by using sparqlService in RunButton – see PR #XX)*.
2. **[P0 Bug] SPARQL UPDATE queries sent via GET** – *Description:* Update operations (INSERT/DELETE) fail because they are sent with wrong method. *AC:* Update queries are always sent with HTTP POST and `Content-Type: application/sparql-update`. Updates execute successfully on compliant endpoints (or appropriate error is shown).
3. **[UX] Enable multi-tab query editing** – *Description:* Allow opening multiple query tabs and switching between them. *AC:* A user can click "+" to create a new tab (with default content), have multiple queries open, and close tabs. State in each tab is preserved on switch. Tabs persist on page reload (localStorage).
4. **[UX] Raw results view for non-table formats** – *Description:* Implement a toggle to view raw response text. *AC:* For CONSTRUCT/DESCRIBE queries or when user selects "Raw", the output is shown as preformatted text (with proper syntax highlighting for Turtle, etc.). Selecting "Table" returns to grid view (if applicable).

5. **[Feature] Download query results in multiple formats** – *Description:* Add options to get results as JSON, CSV, TSV, etc. *AC:* User can choose an output format before running the query (default JSON). After execution, a Download button saves the results in that format. If format is changed post-run, either auto-requery or prompt to re-run.
6. **[Autocomplete] Integrate LOV API for class/property suggestions** – *Description:* Enhance editor autocomplete to suggest ontology terms after prefix. *AC:* When user types a prefix and colon, suggestions include terms from the Linked Open Vocabularies repository (if prefix is recognized). E.g., typing "foaf:" suggests "Person, name, mbox, …".
7. **[Accessibility] Make results grid accessible** – *Description:* Screen readers cannot navigate the virtual grid. *AC:* Provide either ARIA roles on the grid or an alternative HTML table for assistive tech. Verify with axe – no violations regarding table structure. SR users can hear row/column headers for cells.
8. **[Accessibility] Keyboard navigation & focus indicators** – *Description:* Ensure all controls are keyboard-friendly. *AC:* Tab order flows logically through Run, Endpoint, Editor, Results. The editor and split-pane handles have visible focus outline. Users can resize split with keyboard (e.g., arrow keys on focused splitter) or at least we document if not.
9. **[Performance] Add large result warning and limit** – *Description:* Extremely large results can degrade performance. *AC:* If result set exceeds `limits.maxRows` (default 100k), SQUI displays a warning ("Result set too large, showing first N rows") and does not render beyond that. Provide option to download full results instead.
10. **[Docs] Publish SQUI usage guide & migration tips** – *Description:* Improve documentation. *AC:* The README (or wiki) contains sections: Configuration options (with examples), Keyboard Shortcuts, Troubleshooting (CORS, etc.), and a short guide "Coming from YASGUI? Here's what's new/different." Also deploy a live demo link.

Suggested PRs:

- *PR 1:* **Remove QueryExecutionService stub, use sparqlService** – Changes RunButton.svelte as per diff [6], deletes queryExecutionService.ts. (Closes issue #1 above.)
- *PR 2:* **Force POST for SPARQL Updates** – Modifies SparqlService.determineMethod() and executePost content type as discussed (with unit tests for queries starting with INSERT to ensure method selection is correct). (Closes #2.)
- *PR 3:* **Error handling: Preserve details** – Updates resultsStore and ErrorNotification to handle QueryError objects fully. Verify that "Error Details" now shows. (Closes #3.)
- *PR 4:* **Download JSON results** – Adds a download button in Toolbar.svelte that appears after results load. Utilizes Blob and triggers file save. (Partial work for #5; might mark #5 as resolved for JSON and leave CSV/other for later.)
- *PR 5:* **UI enhancements & minor fixes** – a batch PR addressing: adding aria-live to loading state, adding rel=noopener to external links (in preparation for linkify), fix CodeMirror focus outline (maybe by adding a subtle border on focus), and updating any label text as needed. (Addresses #7 and #8 partly.)

Each PR would include references to relevant lines (as we have in analysis) for easy review. We'll label them appropriately (bugfix vs feature) for clarity in changelog.

This roadmap, with issues and PRs, prioritizes critical functionality first, then parity features, then polish. By following it, in 2 months SQUI should achieve its goal of being a robust, user-friendly SPARQL UI that outshines YASGUI in performance, accessibility, and modern developer experience.

---

1 2 3 4 7 24 25 26 27 28 30 31 32 33 49 50 123 124 GitHub
https://github.com/arne-bdt/CHUCC-SQUI/blob/478799c1737e3c3e090368780e69038f62bbc1a5/README.md

5 79 81 85 86 109 129 148 GitHub
https://github.com/arne-bdt/CHUCC-SQUI/blob/478799c1737e3c3e090368780e69038f62bbc1a5/src/lib/services/queryExecutionService.ts

6 78 80 149 150 168 GitHub
https://github.com/arne-bdt/CHUCC-SQUI/blob/478799c1737e3c3e090368780e69038f62bbc1a5/src/lib/components/Toolbar/RunButton.svelte

8 61 82 130 131 132 135 136 137 196 197 198 GitHub
https://github.com/arne-bdt/CHUCC-SQUI/blob/478799c1737e3c3e090368780e69038f62bbc1a5/src/lib/stores/resultsStore.ts

9 10 34 35 36 37 44 99 100 120 141 160 161 165 GitHub
https://github.com/arne-bdt/CHUCC-SQUI/blob/478799c1737e3c3e090368780e69038f62bbc1a5/src/SparqlQueryUI.svelte

11 15 16 17 18 23 29 62 104 105 119 152 179 GitHub
https://github.com/arne-bdt/CHUCC-SQUI/blob/478799c1737e3c3e090368780e69038f62bbc1a5/.tasks/MASTER-TASKS.md

12 38 39 45 46 GitHub
https://github.com/arne-bdt/CHUCC-SQUI/blob/478799c1737e3c3e090368780e69038f62bbc1a5/src/lib/types/config.ts

13 14 47 51 52 53 54 55 56 57 58 59 60 63 64 65 66 67 70 71 73 84 127 128 151 174 175 192 GitHub
https://github.com/arne-bdt/CHUCC-SQUI/blob/478799c1737e3c3e090368780e69038f62bbc1a5/src/lib/services/sparqlService.ts

19 20 89 91 92 182 186 187 190 191 GitHub
https://github.com/arne-bdt/CHUCC-SQUI/blob/478799c1737e3c3e090368780e69038f62bbc1a5/src/lib/editor/prefixCompletions.ts

21 22 48 68 69 72 74 75 76 77 83 133 134 138 GitHub
https://github.com/arne-bdt/CHUCC-SQUI/blob/478799c1737e3c3e090368780e69038f62bbc1a5/src/lib/components/Results/ErrorNotification.svelte

40 41 87 88 90 114 115 188 189 GitHub
https://github.com/arne-bdt/CHUCC-SQUI/blob/478799c1737e3c3e090368780e69038f62bbc1a5/src/lib/services/prefixService.ts

42 43 93 GitHub
https://github.com/arne-bdt/CHUCC-SQUI/blob/478799c1737e3c3e090368780e69038f62bbc1a5/src/lib/services/templateService.ts

94 96 101 107 116 159 164 166 170 GitHub
https://github.com/arne-bdt/CHUCC-SQUI/blob/478799c1737e3c3e090368780e69038f62bbc1a5/src/lib/localization/en.ts

95 146 158 162 GitHub
https://github.com/arne-bdt/CHUCC-SQUI/blob/478799c1737e3c3e090368780e69038f62bbc1a5/src/lib/components/Editor/SparqlEditor.svelte

97 98 102 103 110 144 145 183 185 GitHub
https://github.com/arne-bdt/CHUCC-SQUI/blob/478799c1737e3c3e090368780e69038f62bbc1a5/src/lib/components/Results/DataTable.svelte

106 178 Yasgui API Reference - Triply Documentation
https://docs.triply.cc/yasgui-api/

108 125 126 199 GitHub
https://github.com/arne-bdt/CHUCC-SQUI/blob/478799c1737e3c3e090368780e69038f62bbc1a5/src/lib/types/sparql.ts

111 112 113 117 118 153 154 155 156 GitHub
https://github.com/arne-bdt/CHUCC-SQUI/blob/478799c1737e3c3e090368780e69038f62bbc1a5/src/lib/utils/resultsParser.ts

121 122 GitHub
https://github.com/arne-bdt/CHUCC-SQUI/blob/478799c1737e3c3e090368780e69038f62bbc1a5/src/lib/localization/index.ts

139 140 GitHub
https://github.com/arne-bdt/CHUCC-SQUI/blob/478799c1737e3c3e090368780e69038f62bbc1a5/src/lib/stores/index.ts

142 143 167 GitHub
https://github.com/arne-bdt/CHUCC-SQUI/blob/478799c1737e3c3e090368780e69038f62bbc1a5/package.json

147 193 GitHub
https://github.com/arne-bdt/CHUCC-SQUI/blob/478799c1737e3c3e090368780e69038f62bbc1a5/src/lib/components/Endpoint/EndpointSelector.svelte

157 169 GitHub
https://github.com/arne-bdt/CHUCC-SQUI/blob/478799c1737e3c3e090368780e69038f62bbc1a5/src/lib/components/Endpoint/EndpointInput.svelte

163 184 GitHub
https://github.com/arne-bdt/CHUCC-SQUI/blob/478799c1737e3c3e090368780e69038f62bbc1a5/src/lib/components/Results/ResultsPlaceholder.svelte

171 176 177 195 GitHub
https://github.com/arne-bdt/CHUCC-SQUI/blob/478799c1737e3c3e090368780e69038f62bbc1a5/src/lib/utils/endpointValidator.ts

172 173 194 GitHub
https://github.com/arne-bdt/CHUCC-SQUI/blob/478799c1737e3c3e090368780e69038f62bbc1a5/src/lib/stores/endpointStore.ts

180 181 GitHub
https://github.com/arne-bdt/CHUCC-SQUI/blob/478799c1737e3c3e090368780e69038f62bbc1a5/src/lib/editor/sparqlLanguage.ts