Bachelor's Thesis

How to Split a Long-Lived Monolithic Class in a Live Codebase: Refactoring QLever's Server

Tomáš Dámek

September 21st, 2026

Examined by Prof. Dr. Hannah Bast

Submitted to the University of Freiburg

Department of Computer Science

Chair for Algorithms and Data Structures

universität freiburg

Declaration

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used.

Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.

I also hereby declare that my thesis has not been prepared for another examination or assignment, either in its entirety or excerpts thereof.

Statement on the Use of Generative AI

The generative AI tool Claude Code (Anthropic) was used during the preparation of this thesis for feedback on the organization of chapters and sections, and for sentence-level phrasing suggestions on paragraphs I had already drafted myself. It was not used to generate or prewrite any complete passage of this thesis. All factual and technical claims discussed with the tool were verified by me against the QLever source code prior to their inclusion in this thesis.

Freiburg, September 21st, 2026

Place, date

Signature

Abstract

This thesis applies and evaluates refactoring on QLever's SPARQL query engine codebase's `Server` class. This part of the code is supposed to handle requests and responses, but it also covered the query engine's logic. Besides this mixed logic, this class had very long, hard-to-read functions with high cognitive complexity and code paths unreachable for unit testing. The refactoring let us add more unit tests for previously unreachable paths. We also improved readability, which we support with the cognitive complexity numbers gathered in the evaluation. `Server`'s longest function, `process`, had a cognitive complexity of 104 before the refactoring; after being decomposed into several small, named functions, its cognitive complexity dropped to 12. This way, we were able to show what refactoring a long-lived core class inside a live codebase could look like, what approach can be chosen, how progress can be tracked, and how to evaluate the success of such a refactoring.

Zusammenfassung

Diese Arbeit wendet Refactoring auf die `Server`-Klasse in der Codebasis von QLevers SPARQL-Query-Engine an und evaluiert dessen Wirkung. Dieser Teil des Codes soll Anfragen und Antworten verarbeiten, enthielt jedoch auch Logik der Query-Engine selbst. Neben dieser Vermischung von Zuständigkeiten hatte diese Klasse sehr lange, schwer lesbare Funktionen mit hoher kognitiver Komplexität sowie Codepfade, die für Unit-Tests nicht erreichbar waren. Die von uns verwendeten Refactoring-Methoden ermöglichten es uns nachweislich, weitere Unit-Tests für zuvor unerreichbare Codepfade hinzuzufügen. Außerdem verbesserten wir die Lesbarkeit, was wir mit den in der Evaluation erhobenen Werten der kognitiven Komplexität belegen. Die längste Funktion von `Server`, `process`, hatte vor dem Refactoring eine kognitive Komplexität von 104; nach der Zerlegung in mehrere kleine, benannte Funktionen sank ihre kognitive Komplexität auf 12. Damit konnten wir zeigen, wie das Refactoring einer langlebigen Kernklasse in einer aktiv weiterentwickelten Codebasis aussehen kann, welcher Ansatz dafür gewählt werden kann, wie sich der Fortschritt verfolgen lässt und wie sich der Erfolg eines solchen Refactorings bewerten lässt.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Contribution . . . 3

2 Related Work **5**

3 QLever Background **8**
3.1 `Server`'s Original Responsibilities . . . 8
3.2 Engine-Layer Concepts . . . 8
3.3 HTTP-Layer Mechanics . . . 9

4 Methodology **11**
4.1 Refactoring Methods . . . 11
4.1.1 Behavior-Preserving Change . . . 11
4.1.2 Extract Function . . . 12
4.1.3 Move Function . . . 13
4.1.4 Move Field . . . 14
4.1.5 Replace Error Code with Exception . . . 14

5 The Refactoring **16**
5.1 Moving Engine Logic from Server into libqlever . . . 16
5.1.1 Move of Server's engine-layer members into the Qlever class . . . 16
5.1.2 Query planning's transfer from Server class into the Qlever class . . . 17
5.1.3 Extraction of writing a materialized view logic from Server class into the Qlever class . . . 18
5.1.4 Move of the application logic of a SPARQL update from Server into the Qlever class . . . 19
5.1.5 Extraction of delta-triples management logic from Server into the Qlever class . . . 20
5.2 Extracting Server's Request/Response Helpers . . . 20
5.2.1 Extraction of request-parameter interpretation logic from Server into a named namespace . . . 21
5.2.2 Extraction of JSON response composition out of Server . . . 22
5.3 Decomposing Server's Long Functions . . . 22
5.3.1 Server's run function decomposition . . . 22
5.3.2 Extraction of Server::process's inlined command handlers . . . 23
5.3.3 Extraction of Server::process's inlined ping, metrics, and runtime-parameter handling . . . 24
5.3.4 Extraction of Server::process's SPARQL-operation dispatch . . . 24

5.3.5 Extraction of Server::process's cmd= command dispatch 24
5.4 Smells Uncovered Along the Way 25
5.4.1 Replacing empty-optional signaling with a thrown exception 26
5.4.2 Extraction of the logging and highlighting logic for a failed operation's error message 26

6 Evaluation **28**
6.1 Testability 28
6.2 Readability and Cognitive Complexity 29
6.2.1 Non-Comment Lines of Code (NLOC) 30
6.2.2 Cyclomatic complexity (CCN) 30
6.2.3 Cognitive complexity 30
6.3 Coupling 33

7 Future Work **35**

8 Conclusion **37**

9 Acknowledgements **38**

References **39**

Glossary **41**

List of Figures

1 `Server::run`'s call decomposition after the refactoring: it now fans out to two named functions instead of doing everything inline. 31
2 `Server::process`'s call decomposition before the refactoring: already 14 direct calls, but flat and undifferentiated. Ten of them are command-specific handlers (`writeMaterializedView`, `processUpdate`, `processQuery`, and others) sitting as direct siblings of `process` itself, with no single entry point grouping them. 32
3 `Server::process`'s call decomposition after the refactoring: down to 7 direct calls, with the command-specific handlers consolidated behind one new entry point, `processCommands`, instead of sitting as flat siblings of `process`. 33
4 `Server`'s direct engine-layer members, before and after the refactoring: 6 direct members collapse into 1. 34

List of Tables

1 The 18 newly added tests, by the change that introduced them. 29
2 `Server::run`, before and after the refactoring. 30
3 `Server::process`, before and after the refactoring. 31

List of Code Listings

1 Schematic excerpt of `Server::process` before this refactoring. The full function has 14 `cmd=` branches and 479 lines; three representative branches are shown here. . . . 2

2 `libqlever`'s `QueryPlan` and `Server`'s own `PlannedQuery`, before the preparatory step: two separate types for the same concept, an already-planned query. . . . 11

3 The single `qlever::PlannedQuery` struct after unification, now the only representation, moved into `libqlever`'s `QleverTypes.h`. . . . 12

4 The duplicated exception-handling tail inside `Server::run()`, before and after the refactoring. Both `catch` branches collapse into one call each to the new `reportHttpError`. . . . 13

5 `verifyUserSubmittedQueryTimeout`, before and after the refactoring. An exact instance of Replace Error Code with Exception. . . . 14

6 `Server::initialize`, before the refactoring. This logic duplicates what `Qlever`'s own constructor already does for embedded use. . . . 16

7 `Server`'s constructor, after the refactoring. The separate `initialize` function is gone; the steps shown above now run inside `Qlever`'s own constructor instead. . . . 17

8 `Server::planQuery`, before and after the refactoring. The query-planner invocation itself now lives in `Qlever::planQuery`; `Server` keeps only the surrounding logging. . . . 18

9 The correctness check added to `Qlever::applyUpdate`, the one behavior addition disclosed in this move. . . . 19

10 The correctness check added while parsing the `send` parameter, the one behavior addition disclosed in this extraction. . . . 21

11 The command metadata array added to the new `serverProcessHelpers` namespace. This is a metadata table, not a function-pointer dispatch table: the if-else chain in `process` itself is unchanged; the array only centralizes each command's logging and authorization requirement. . . . 23

12 `processCommands`'s command branches, before and after introducing `makeCommandResult`. The repeated `ProcessCommandsResult{...}` wrapping collapses into one call. . . . 25

13 The error-logging-and-highlighting block, before and after the refactoring. The whole block collapses to one call. . . . 26

14 The `requireNoOperation` check, an example of enforcing this decision at the code level, inside `processCommands`, rather than at the request's own structural level. . . . 35

1 Introduction

Working on a coding project can be divided into three phases. The first is the phase programmers are most passionate about: what we call the "burning urge" to create, when progress feels fast and comes easily. Small projects can stay in this phase for their entire lifetime. Bigger projects, with greater potential, have to move on to a second phase: Here the "magic" happens and more permanent foundation has to be built. This phase is less exciting and often frustrating, since most people would rather be building the actual feature than the structure underneath it, but it pays off over the project's lifetime.

Even this second phase does not last forever, and eventually gives way to a third: maintenance. As a project keeps growing, progress eventually starts to feel harder than it did at the beginning, not because the people working on it got slower, but because the project itself has grown too complicated. Newcomers can no longer make sense of what has been built, and even the original developers struggle to explain why a given part works the way it does. This is the point where developers start asking why adding a small feature takes so long, or why one class ended up responsible for everything. The answer is usually the same: change is inevitable. Left alone, this complexity keeps causing bugs and can eventually stall the project entirely. The usual causes are coding smells: long, hard-to-read functions, classes with too many responsibilities, or code that cannot be tested or reached in isolation. Addressing them is the goal of a maintenance technique called refactoring.

Fowler and Beck define refactoring as "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior" [4].

In this thesis, we investigate QLever, a rapidly growing, fast SPARQL engine for querying knowledge graphs, developed as an open-source project at the University of Freiburg's Department of Computer Science. Concretely, we take a closer look at one of its core classes, the `Server` class.

1.1 Motivation

QLever's codebase has been under constant development since 2014. Today, QLever serves live SPARQL endpoints over some of the largest knowledge graphs available, including the complete Wikidata and OpenStreetMap datasets. It can index and query datasets with more than one trillion triples on a single machine, and it outperforms other SPARQL engines, such as Virtuoso, GraphDB, and Blazegraph, on most queries [1]. Most client-facing usage goes through the `Server` class, which computes and returns the requested

resources back to the client. It is one of the core parts of QLever's codebase, and its maintainability is important for the project's future development.

QLever's `Server` class is a strong candidate for refactoring. Before this work, the class carried two main problems. First, it mixed two responsibilities: handling HTTP requests and managing the query engine. This coupling meant `Server` managed the engine's index and cache directly as its own members, instead of building on `libqlever`, the facade QLever already provided for engine-only use. Second, some of its functions, such as `process`, had grown too long and handled more responsibilities than a single function should. Both problems hurt readability. This coupling and length meant that understanding or changing one responsibility required tracing through the other, since the two were not separated: a developer working on `process`'s HTTP-handling logic still had to read past its inline engine-layer branches, and the reverse held for the engine side. This made new features harder to add, since a developer had to hold the whole tangled function in mind before safely changing any part of it. These long, monolithic functions also made the class hard to test, and in some cases made certain code paths impossible to reach at all. Listing 1 shows a representative excerpt of `process` before this refactoring, illustrating this tangle directly.

Listing 1: Schematic excerpt of `Server::process` before this refactoring. The full function has 14 `cmd=` branches and 479 lines; three representative branches are shown here.

```
Awaitable<void> Server::process(Request& request, Send&& send) {
  // HTTP concern: parse the request.
  auto parsed = SparqlProtocol::parseHttpRequest(request);

  if (checkParameter("cmd", "clear-cache")) {
    cache_.clearUnpinnedOnly();  // engine concern
    response = createJsonResponse(composeCacheStatsJson(), request);

  } else if (checkParameter("cmd", "clear-delta-triples")) {
    // Self-contained: could be pulled out into its own function.
    auto handle = std::make_shared<CancellationHandle>();
    co_await computeInNewThread(updateThreadPool_, [this] {  // execution
      return index().deltaTriplesManager()  // engine concern
          .modify<DeltaTriplesCount>(/*...*/);
    }, handle);
    response = createJsonResponse(/* ... */, request);

  } else if (checkParameter("cmd", "write-materialized-view")) {
    // Self-contained: could be pulled out into its own function.
    auto [name, query] = /* ... */;
    co_await computeInNewThread(queryThreadPool_, [name, query, this] {
      writeMaterializedView(name, query);  // engine concern
```

```
      }, handle);
      parsed.operation_ = None{};  // side effect on shared state

    } // ... 10 more 'cmd=' branches, same pattern

    auto visitQuery = [&](Query q) -> Awaitable<void> { /* ... */ };
    auto visitUpdate = [&](Update u) -> Awaitable<void> { /* ... */ };
    co_return co_await processOperation(
        std::move(parsed.operation_),
        OverloadCallOperator{visitQuery, visitUpdate,
                             visitGraphStore, visitNone},
        requestTimer, request, send);
}
```

The codebase under study is alive: it is under constant development. When refactoring, we cannot simply stop everything, as if closing a road for the next few months. The refactoring has to happen alongside the new development, and stay ready for new changes as new features keep coming in. This can be frustrating: a part of the code gets cleaned up only to need cleaning up again a few pull requests later. This friction is part of what refactoring a live codebase is about, and it has to be taken as a challenge to work with, rather than around.

We take the view that a server's role should be to serve, not to compute. In the ideal case, the server takes the client's request, gets the resource that was asked for, and serves it back via a valid response. Simple as that. Our aim here is to split `Server`'s responsibilities along this line, following Parnas's principle of information hiding: each module should hide a design decision that is likely to change from the rest of the system [10]. Concretely, we want to decouple its long functions into small, testable units. This should make it possible to test code paths that were previously unreachable, and increase both readability and overall code quality.

1.2 Contribution

We proceed through a sequence of small, independently reviewed changes, following Fowler's refactoring methodology [4]: restructuring existing code without changing its observable behavior. The refactoring follows four parts. First, engine logic that does not depend on HTTP moves out of the `Server` class into `Qlever`, QLever's engine-layer class, which already supports being used without a `Server` instance (e.g., in QLever's WebAssembly build). This reduces how much of that logic remains locked inside `Server` and unavailable to other consumers of `Qlever`. Second, logic that still touches HTTP-specific types, but does not need to stay inside `Server` itself, is extracted into its own independent helper

units. Third, the `Server` class's remaining long functions are decomposed into small, named, independently testable units, which increases code path coverage and readability. Fourth, further smells that these first three parts uncover along the way are fixed as they surface.

We evaluate the result along three dimensions. Testability: 18 new tests now cover logic that was previously unreachable, for example `run`'s inlined session-handling code, which used to require a live, bound socket to test. Readability: `Server::process` shrank from around 479 lines to 104, and `Server::run` from around 88 lines to 19, with the removed logic distributed across eight new named functions in total, three from `run` and five from `process`, instead of collapsed into one. Coupling: the number of engine-layer types `Server` depends on directly dropped from six to one, a single `qlever::Qlever` instance. Beyond the `Server` class itself, this refactoring works as a practical example for untangling similarly long-lived, coupled core classes in live codebases.

2 Related Work

This section reviews five empirical studies on software refactoring: how and why developers refactor, how refactoring happens during code review, how it differs in data-intensive systems, and an automated approach to decomposing an overloaded C/C++ file. Each is contrasted with the manual, pull-request-reviewed case study this thesis presents.

Golubev et al. [5] surveyed 1,183 users of IntelliJ-based IDEs on how, how often, and why they refactor their code. The researchers also wanted to find out why many avoid the automated refactoring features built into their IDEs. Their primary aim was to inform the future development of JetBrains's own refactoring tools, while also contributing empirical insight to the research community. They asked the following questions: What does the refactoring process actually look like for developers? Which refactoring types are used most often? Why do some developers stay away from their IDE's built-in refactoring tools? The results showed that almost two-thirds of developers had refactored code for an hour or more in a single session. Manual refactoring still remains common despite high IDE maturity. Developers tend to substitute general-purpose tools for dedicated refactoring commands, e.g., Find-and-Replace for Rename, Copy-and-Paste for Move. The respondents say that for simple changes, manual refactoring is often easier. In contrast to this paper, we did not use any IDE refactoring tools. Instead, we relied on nvim's general-purpose text-editing commands, such as the substitute command applied across a quickfix list, to make these changes.

A related line of work asks not how developers refactor, but why. Silva et al. [11] monitored Java projects on GitHub to detect refactorings as they were applied, then asked the developers who performed them to explain their reasoning. From 463 refactorings across 222 commits in 124 popular projects, they built a taxonomy covering 44 different motivations behind 12 well-known refactoring types. Their central finding is that new feature requests and bug fixes drive most refactoring activity, while eliminating code smells plays a much smaller role. For Extract Method, their most versatile case with 11 distinct motivations, only two, readability and removing duplication, target a code smell at all. One developer explained an Extract Method refactoring simply because the method "didn't fit onto the screen anymore." The paper also reports that the IDE a developer uses affects their adoption of automated refactoring tools, echoing Golubev et al.'s findings above. In contrast to this paper, our refactoring of the `Server` class was not a side effect of adding a feature: it was a deliberate, smell-driven decomposition, following Fowler's refactoring methodology from the start.

The next study, by Paixão et al. [9], looks at refactoring specifically during code review, rather than as a general method. They examined 1,780 code changes that had gone through review, drawn from 6 systems spanning two sizeable open-source communities. They then structured their findings into 7 distinct categories, such as feature work, bug fixing, and platform updates. The study presented the following results: developers often combine several non-trivial refactorings, spread across unrelated parts of the codebase, while working toward a single new feature. They also found that a developer's intention for a change often shifts during the review itself, which then shapes which refactorings get selected to match the new goal. In contrast to this study, we focus our attention on the refactoring of a single class instead of a broad study. However, we share one finding with them: several code smells only became visible once earlier changes had already isolated the surrounding code.

A different study, by Muse et al. [8], focuses directly on refactoring data access classes: the parts of a codebase that talk to a database, whether through driver calls or an API. They gathered data across 12 data-intensive subject systems and manually reviewed 378 sampled instances of data access refactoring to determine which functionality these changes actually target. The results showed that refactorings inside data access classes are frequent and vary in type, with Rename Variable being the most common. Data fetching and insertion were the primary targets. However, only 0.45% of the refactorings directly touched SQL queries. The rest stayed at the surrounding code level. QLever, the project this thesis focuses on, is itself a data-intensive system, so this study is closer to our work than the ones discussed above. The distinguishing factor here is that our work concentrates on the `Server` class, which sits at the HTTP dispatch boundary, rather than in the data-access layer this study covers.

The most contrasting study to ours discussed here is definitely the work that automates the decomposition of so-called God Header Files: header files that grow so large and get included by so many other files that a single change forces a costly, wide-reaching recompilation. Wang et al. [12] state that such God Header Files hurt code comprehension, maintainability, and compilation time. Existing God Class refactoring methods do not transfer well. The reason, they say, is that a header file's contents consist mainly of brief declarations, and that decomposing it has to account for build dependencies across the whole system. Their approach first constructs a code-element graph from the header file, simplifies it, and then applies a novel multi-view graph clustering algorithm to identify decomposition clusters. A heuristic step helps resolve the cyclic dependencies introduced by the split. For the evaluation, both a synthetic and a real-world God Header File dataset were used, with the following findings: this automated approach showed an 11.5% higher decomposition accuracy than the best-performing existing God Class refactoring method.

It also produced better results architecturally, improving modularity and cutting down on cyclic dependencies, and reduced recompilation time by 15 to 60%. Similar to this thesis, they also target a long-lived C++ artifact. However, our thesis uses a manual approach of incremental changes, verified pull request by pull request against unchanged behavior. They also specifically target header files, whereas we instead focus on the system's definition files and function-level changes.

3 QLever Background

In the following, we explain the background needed to understand the later refactoring steps. We first take a short look at `Server`'s original responsibilities and its main structure before the refactoring. We then cover the engine and HTTP layer concepts that the subsequent chapters rely on.

3.1 `Server`'s Original Responsibilities

Before we touched `Server`'s class structure, it had two responsibilities: handling the HTTP layer, and orchestrating some of the engine layer's jobs.

The first, the HTTP layer, covers parsing an incoming SPARQL request, preparing the resources it asked for, and dispatching the appropriate response. Listing 1 shows a schematic excerpt of this function before the refactoring.

The second responsibility managed some of the engine layer's instances directly: `Server` owned the index, cache, and materialized views as member variables on itself. We explain each of these engine-layer concepts in subsection 3.2. Since a dedicated structure for these engine aspects already existed in `libqlever`, this was a clear sign of a mixed-concerns smell.

The following sections explain QLever's engine and HTTP layers in their current, after-refactoring state, as opposed to the before-state described above. This lets us prepare for the upcoming refactoring steps and shows which aspects this thesis touches.

3.2 Engine-Layer Concepts

QLever stores the basic units of Resource Description Framework (RDF)-style statements as triples. Clients query this stored information using the SPARQL language.

Index The `Index` holds the on-disk, permutation-based representation of the loaded triples, together with the vocabulary mapping their terms to IDs. It gives the results of the scans the query engine runs against it. For operations that update the state of the index, such as `clearDeltaTriples`, `vacuumDeltaTriples`, and `writeMaterializedView`, it is important to take one `indexAndViewsSnapshot()` at the beginning of the operation. This gives the operation a consistent view to work against for its whole duration, even while other queries or updates run concurrently, so the index never needs to be locked.

Materialized Views QLever stores so-called views: an already planned query that, depending on the operation, is either written, loaded, deleted, or unloaded. All of these can be executed via different flag commands sent to the server via an HTTP request.

Cache The `QueryResultCache` is a thread-safe least-recently-used (LRU) cache storing (partial) query results for later reuse.

The `Qlever` Class From here on, "QLever" refers to the project as a whole, while `Qlever` (typeset in code font, lowercase "l") refers specifically to this one C++ class. `Server` now owns a `qlever::Qlever` instance, which bundles the engine-layer members mentioned above. `Qlever` holds most of them, the cache, named-result cache, allocator, and sort-performance estimator, as plain direct members. The index and the materialized-views manager are the exception: they live together inside an internal `IndexAndViews` structure, reachable only through a synchronized snapshot pointer, so both can be swapped atomically whenever the index gets rebuilt. This `Qlever` class lives in `libqlever`, a library QLever provides for using the engine as an embedded database, without spinning up an HTTP server. It also exposes `PlannedQuery` as its own type for representing an already-planned query.

Query Planning `QueryPlanner` turns a `ParsedQuery` into a ready-to-execute `QueryExecutionTree` by selecting the plan with the lowest estimated cost. The `PlannedQuery` holds the two together, exposing access to both.

Delta Triples Instead of running the rebuild of an index each time triples get deleted or inserted, `Qlever` has `DeltaTriples`. These represent the newly inserted or deleted triples since the index was built. `clearDeltaTriples` clears all delta triples of the `Qlever` instance's active index snapshot. `vacuumDeltaTriples` works the same way, but only removes redundant delta triples.

SPARQL Query and Update A SPARQL query is read-only: it matches a pattern against the triple store and returns the triples (or bindings) that satisfy it, similar to a SELECT in SQL. A SPARQL update is the write counterpart: instead of retrieving data, it changes it, via INSERT DATA, DELETE DATA, DELETE/INSERT WHERE, etc. It adds, removes, or modifies triples in the store.

3.3 HTTP-Layer Mechanics

`Server` exposes QLever via the HTTP protocol, and clients use HTTP requests to ask for resources from it, such as running SPARQL operations or one of QLever's own custom commands, for instance `rebuild-index` or `vacuum-delta-triples`.

Request Lifecycle The cycle starts in the server's `run` function. There, we build a session-handler lambda forwarding each incoming request to the specified function `handleHttpRequest`. Following that, we set up the server, bound to the configured port and thread count, with the WebSocket session supplier provided by `makeWebSocketSessionSupplier`. Once that setup is finished, `run()` actually starts the server listening and logs that information. `handleHttpRequest` adds CORS (Cross-Origin Resource Sharing) headers to every response and answers an HTTP `OPTIONS` request directly, or otherwise tries to call the `process()` dispatch function. Any error or exception here gets turned into a response via `reportHttpError`. This function logs the error, records an HTTP-error metric, and returns the response.

Process Function `process()` is the central entry point for the request handling. It deals with the user's demands and produces the response. It calls the `processCommands` function. This dispatches on the request's `cmd=` parameter, one `if`/`else if` branch per command, such as `stats`, `cache-stats`, `clear-delta-triples`, and `write-materialized-view`. Each command is checked and logged against a metadata table listing its name, description, and whether it requires an access token. This table only centralizes auth-checking and logging; it does not itself route requests to handlers. The `process` function also branches for handling metrics requests, setting runtime parameters, ping, and lastly handling the SPARQL operation.

SPARQL Operation Type The type of a SPARQL operation exists as one of four variants: `Query` and `Update`, already spoken about earlier, the `GraphStoreOperation`, which works on a named graph via the Graph Store HTTP Protocol, and finally the `None` operation. The last one should not be mistaken for an error. It actually signals requests using one of QLever's own custom commands, e.g., `cache-stats`, which carry no SPARQL query or update, but are still valid requests.

Response Construction Commands such as `stats` or `cache-stats` return a JSON-structured response, composed via the functions `composeIndexStats` and `composeCacheStats`, both located in `ResponseJson.h`.

Timeout A timeout parameter can be sent along with a query or update. There is a configurable default timeout value in the server. The requested timeout may be higher only if the request carries a valid access token. In case of a higher timeout than the default with no such access token, the request is rejected with a 403 Forbidden error.

4 Methodology

In the following section, we explain the refactoring methods we used, based on Fowler's refactoring methodology [4]. Rather than one single procedure, this methodology is a catalog of individually named techniques, such as Extract Function or Move Function. Each technique has its own precise mechanics and is applied one at a time, preserving behavior at every step. We use concrete examples from the actual refactoring process. The ordering follows each method's number of usages. But before that, we first clarify the core of refactoring itself.

4.1 Refactoring Methods

4.1.1 Behavior-Preserving Change

While refactoring, we want to keep the actual external behavior unchanged while making changes to the internal structure [4]. This is the key to good refactoring: find code smells, prepare a solution, apply it, and do not break anything else. While this might seem like a simple, logical process, it can sometimes be very difficult. For example, we might have a code snippet that requires function extraction. However, if this is moved out, we break the dependency of the next piece, which then also needs refactoring, and this can cascade. The usual solution, central to Fowler's methodology, is to do this step by step, following Kent Beck's own advice, quoted directly in Fowler's book: "for each desired change, make the change easy (warning: this may be hard), then make the easy change" [4].

Before a later refactoring could move all query-planning logic into `Qlever`, an earlier, preparatory step first had to unify two separate types that both represented an already-planned query, `QueryPlan` and `PlannedQuery`, into one. Only once `Qlever` and `Server` shared the same representation could the planning logic that used it move as a single unit. The two types were not just named differently: `QueryPlan` was an unnamed, positional tuple, while `Server`'s own `PlannedQuery` was already a proper struct with named, encapsulated fields.

Listing 2: `libqlever`'s `QueryPlan` and `Server`'s own `PlannedQuery`, before the preparatory step: two separate types for the same concept, an already-planned query.

```
// libqlever's own representation, an unnamed positional tuple:
namespace qlever {
using QueryPlan =
    std::tuple<std::shared_ptr<QueryExecutionTree>,
               std::shared_ptr<QueryExecutionContext>, ParsedQuery>;
}

// Server's own, separate representation, a named struct:
struct Server::PlannedQuery {
```

```
 private:
  std::shared_ptr<const QueryExecutionContext> qec_;
  ParsedQuery parsedQuery_;
  QueryExecutionTree queryExecutionTree_;

 public:
  PlannedQuery(ParsedQuery pq, QueryExecutionTree qet,
               const QueryExecutionContext& qec);
  // ... (getters, unchanged)
};
```

Listing 3: The single `qlever::PlannedQuery` struct after unification, now the only representation, moved into `libqlever`'s `QleverTypes.h`.

```
namespace qlever {
struct PlannedQuery {
 private:
  std::shared_ptr<QueryExecutionContext> qec_;
  ParsedQuery parsedQuery_;
  std::shared_ptr<QueryExecutionTree> queryExecutionTree_;

 public:
  PlannedQuery(ParsedQuery pq, QueryExecutionTree qet,
               QueryExecutionContext& qec);
  // ... (getters, unchanged)
};
}
```

The unification carried `Server`'s struct over almost unchanged: the main difference is that `queryExecutionTree_` now lives behind a shared pointer instead of being stored by value. The positional `QueryPlan` tuple disappears entirely, and `Qlever`'s `parseAndPlanQuery` gains the extra parameters, `datasetClauses`, a cancellation handle, and a time limit, that `Server::planQuery` already needed, so the later move of planning logic would not lose any capability `Server` still relied on.

4.1.2 Extract Function

One of the most common refactoring methods is Extract Function, found by Silva et al. to be the most frequently occurring refactoring type in their study of GitHub projects [11]. It describes the process of selecting a part of code that can be factored out into its own dedicated, named function, for multiple reasons, as also observed in the related work: Silva et al. [11] found Extract Method to be the most versatile refactoring type, covering 11 distinct motivations. We use this method with the following motivations: a common reason for function extraction is reusing the same code elsewhere to avoid code duplication. Another reason is, e.g., overloading the function with extra or differently typed parameters.

Separating mixed concerns into one specified function is another aim of this method. Sometimes the goal is a simple relocation, which occasionally ends up revealing further code smells along the way. We also find this method useful for making a diff smaller by introducing a local helper, which shortens many lines and increases the overall readability of the code. This method is also beneficial for testing and increases the project's code coverage: if the not-yet-extracted code was untested, e.g., due to its unreachability, this method allows easier test integration. In the following example, the code snippet from the now-extracted `Server::handleHttpRequest` is shown, previously part of `Server::run`. This extraction enabled unit testing that was not possible before. We also removed code duplication by extracting the `reportHttpError` function, which at the same time increased readability.

Listing 4: The duplicated exception-handling tail inside `Server::run()`, before and after the refactoring. Both `catch` branches collapse into one call each to the new `reportHttpError`.

```
// Before: Server::run(), the inline try/catch tail (repeated per branch):
catch (const HttpError& e) {
  httpResponseStatus = e.status();
  exceptionErrorMsg = e.what();
  metrics_->httpErrors_->Add(1, {HttpErrorType::http});
} catch (const std::exception& e) {
  exceptionErrorMsg = e.what();
  metrics_->httpErrors_->Add(1, {HttpErrorType::internal});
}
if (exceptionErrorMsg.has_value()) {
  AD_LOG_ERROR << exceptionErrorMsg.value() << std::endl;
  auto response = createHttpResponseFromString(
      exceptionErrorMsg.value(), status, request, MediaType::textPlain);
  co_return co_await sendWithAccessControlHeaders(std::move(response));
}

// After: Server::reportHttpError, called once per catch branch:
HttpErrorResponse reportHttpError(std::string_view message, http::status status,
                                  const RequestT& request,
                                  const MetricLabel& errorType) const;
// ...
catch (const HttpError& e) {
  errorResponse = reportHttpError(e.what(), e.status(), request, HttpErrorType::http
    );
}
```

4.1.3 Move Function

The Move Function method allows us to again make the individual concerns of the code crystal clear. First, we identify a function's concerns and responsibilities, and then

relocate it into a different class or namespace, or create a new dedicated one, to preserve a logical structure. This usually increases the number of files, but on the other hand creates a code structure that is easier to follow and maintain. We apply the simple rule "put things where they belong." In our thesis, this method is used to move the non-server responsibilities out of the `Server` class, mostly into the engine-layer class `Qlever`. We apply this to `Server::processUpdate`: the moved function's own body is unchanged (apart from its rename to `applyUpdate` and one added correctness check). From a caller's point of view, only the call site itself changes: `Server::processUpdate` now calls `qlever().applyUpdate(...)` instead of its previous free-standing implementation.

4.1.4 Move Field

The Move Field method is a direct companion of Move Function. They happen at the same time: if we move a function, e.g., from one class to a different one, we also need to move along the members it operates on. Otherwise, we would introduce more complexity by introducing new getters for this function, and the goal of the refactoring would be lost. This can be observed in the following example, where `Server`'s original members moved together with the move functions into the `Qlever` class.

`Server::initialize`, which set up the index and materialized views using exactly these fields, is a case in point: that logic now runs inside `Qlever`'s own constructor instead of a separate function. Most steps carry over unchanged, though not strictly in the same order: the sort-performance estimate now runs before the materialized-views preload, reversed from before.

4.1.5 Replace Error Code with Exception

Another refactoring method we applied is replacing an error code with an exception [3]. Unlike the other refactorings in this section, this one is not part of Fowler's printed catalog [4], but of his web-only catalog. In this case, we have a function that used to return a special sentinel value, which signaled to the caller the need to check and interpret the function's outcome. The caller then also needed to decide how to handle this state. The problem with this approach is that if a deep, lower-level function needs to populate this state, only an upper-level caller actually has access to resolve it. Therefore, we decided to get rid of this problematic propagation and instead throw an exception directly at the lower level, which can simply be caught and handled generally. This can be observed with the timeout check and process functions.

Listing 5: `verifyUserSubmittedQueryTimeout`, before and after the refactoring. An exact instance of Replace Error Code with Exception.

```
// Before: sends the response itself, signals "stop" via an empty optional:
if (timeoutCandidate > defaultTimeout && !accessTokenOk) {
```

```
  co_await send(ad_utility::httpUtils::createForbiddenResponse(
      "User submitted timeout was higher than what is currently allowed...",
      request));
  co_return std::nullopt;
}

// After: throws instead, caught in one place further up the call chain:
if (timeoutCandidate > defaultTimeout && !accessTokenOk) {
  throw HttpError(boost::beast::http::status::forbidden,
                  "User submitted timeout was higher than what is currently allowed
    ...");
}
```

The following chapter applies these refactoring methods to `Server`'s actual transformation.

5 The Refactoring

In this section, we go through the actual changes applied to QLever's codebase and examine them in detail. Our procedure is as follows: first, we briefly state the before state and the code smells we found. Second, we discuss the actual changes applied. Finally, we look at the resulting after state. This process is applied to each of the four logical groups to which we applied refactorings: moving engine logic from `Server` into `libqlever`, extracting `Server`'s request/response helpers, decomposing `Server`'s long functions, and fixing smells uncovered along the way.

5.1 Moving Engine Logic from Server into libqlever

This subsection explains the work done to move engine logic from the `Server` class into `libqlever`'s `Qlever` class (recall the `QLever`/`Qlever` distinction from subsection 3.2). We look at the change in ownership of core member variables from `Server` to `Qlever`, using the Move Field method (subsubsection 4.1.4) together with its companion, Move Function (subsubsection 4.1.3), for the correct placement of the corresponding functions. The logic moved here is the code that is HTTP-agnostic, relocated from `Server`'s structure to `Qlever`'s class.

5.1.1 Move of Server's engine-layer members into the Qlever class

In the before state, `Server` directly managed what are now the `Qlever` instance's components, the same engine-layer concepts introduced in subsection 3.2: index, cache, named-result cache, materialized-views manager, allocator, and sort-performance estimator. It held each as its own private member. This introduced a mixed-concerns smell: `Server` was responsible for HTTP-layer handling but also for engine-layer management. The initialization of those members happened inside `Server`'s prior `initialize` function, shown here:

Listing 6: `Server::initialize`, before the refactoring. This logic duplicates what `Qlever`'s own constructor already does for embedded use.

```
void Server::initialize(const std::string& indexBaseName, bool useText,
                        bool usePatterns, bool loadAllPermutations,
                        bool persistUpdates,
                        std::vector<std::string> preloadMaterializedViews) {
  index().usePatterns() = usePatterns;
  index().loadAllPermutations() = loadAllPermutations;
  index().createFromOnDiskIndex(indexBaseName, persistUpdates);
  if (useText) {
    index().addTextFromOnDiskIndex();
  }
  for (const auto& viewName : preloadMaterializedViews) {
    try {
```

```
      materializedViewsManager_->loadView(viewName);
    } catch (const std::exception& ex) { /* ... (logged, startup continues) */ }
  }
  sortPerformanceEstimator_.computeEstimatesExpensively(
      allocator_, index().numTriples().normalAndInternal_() *
                      PERCENTAGE_OF_TRIPLES_FOR_SORT_ESTIMATE / 100);
  // ... (access-token check, unchanged)
}
```

This is the mixed-concerns smell the Move Field method (subsubsection 4.1.4) addresses: the members above move from `Server` into `Qlever`, and their initialization moves with them, directly into `Qlever`'s constructor. This collapses `Server`'s former two-phase initialization (a constructor, followed by a separate `initialize` function) into the single constructor shown below. `Server` now holds one `Qlever` instance, initialized from a `qlever::EngineConfig` inside its own constructor. To keep the diff small and avoid rewriting every call site to go through `qlever()` directly, thin forwarding getters for each member were added to `Server`, delegating to the `Qlever` instance. As a consequence, `Server` now accesses the engine internals through a single `qlever_` member and its getters, a clear simplification that resolves the mixed-concerns smell.

Listing 7: `Server`'s constructor, after the refactoring. The separate `initialize` function is gone; the steps shown above now run inside `Qlever`'s own constructor instead.

```
Server::Server(unsigned short port, size_t numThreads, std::string accessToken,
               const qlever::EngineConfig& config, bool noAccessCheck)
    : qlever_(config),
      numThreads_(numThreads),
      port_(port),
      accessToken_(std::move(accessToken)),
      noAccessCheck_(noAccessCheck),
      queryThreadPool_{numThreads} {
  // ... (access-token check, unchanged)
}
```

5.1.2 Query planning's transfer from Server class into the Qlever class

Before this refactoring, `Server` itself directly invoked the query planner: it constructed a `QueryPlanner` and assembled the resulting `PlannedQuery`, rather than delegating this to the engine layer. The actual plan-selection algorithm always lived in `QueryPlanner` itself (see subsection 3.2); what moved here is the surrounding orchestration that invokes it. This introduces the same mixed-concerns smell as before: `Server` should be responsible for the HTTP layer, while HTTP-agnostic responsibilities such as invoking the query planner belong in `libqlever`'s `Qlever` class. As introduced in the Behavior-Preserving Change

discussion (subsubsection 4.1.1), this move first needed a preparatory step: unifying the two structs that represented an already-planned query, `QueryPlan` (used inside `libqlever`) and `PlannedQuery` (used inside `Server`), into the single `qlever::PlannedQuery` struct now defined in `libqlever`'s `QleverTypes.h` (see Listing 2 and Listing 3). This removed the duplicate representation and let `Qlever::parseAndPlanQuery` support the same capabilities `Server::planQuery` already needed, namely cancellation, time limits, dataset clauses, and pinning.

With that preparation in place, this query-planner orchestration can move out of `Server` and into `Qlever`. Since `Server` already holds a `Qlever` instance, as shown in the previous subsubsection, its `planQuery` function can replace its own query-planner invocation with a call to that instance, shown below.

Listing 8: `Server::planQuery`, before and after the refactoring. The query-planner invocation itself now lives in `Qlever::planQuery`; `Server` keeps only the surrounding logging.

```
// Before:
QueryPlanner qp(&qec, handle);
auto executionTree = qp.createExecutionTree(operation);
PlannedQuery plannedQuery{std::move(operation), std::move(executionTree), qec};
handle->throwIfCancelled();
plannedQuery.queryExecutionTree().getRootOperation()
    ->recursivelySetCancellationHandle(std::move(handle));
plannedQuery.queryExecutionTree().getRootOperation()
    ->recursivelySetTimeConstraint(timeLimit);
plannedQuery.queryExecutionTree().isRoot() = true;
// ... (timing, logging, unchanged)

// After:
PlannedQuery plannedQuery = qlever().planQuery(
    std::move(operation), timeLimit, qec, std::move(handle), requestTimer);
// ... (timing, logging, unchanged)
```

As a result, two things change. First, `Server` no longer defines its own `PlannedQuery` type, a struct that never belonged to the HTTP layer in the first place. Second, `Server::planQuery` shrinks down to little more than a delegating call into `libqlever`, with logging as the only work left in `Server`'s own hands, a genuine `Server` responsibility.

5.1.3 Extraction of writing a materialized view logic from Server class into the Qlever class

Before this refactoring, `Server` executed writing a materialized view as its own direct method, `Server::writeMaterializedView`. This logic had no HTTP-layer needs, making

it another good candidate for the Move Function method (subsubsection 4.1.3). `Qlever` already had a simpler version of `writeMaterializedView`, but it lacked the arguments `Server`'s version needed: dataset clauses, a cancellation handle, and a time limit. We extended `Qlever::writeMaterializedView` with these as optional arguments, all defaulted so existing embedded callers keep compiling unchanged, moved `Server`'s extra logic into it, and removed `Server`'s own copy entirely.

As a result, `Server` is lighter by another piece of engine-layer responsibility. One change is worth flagging explicitly: the behavior was not entirely preserved, unlike the Behavior-Preserving Change examples in this thesis (subsubsection 4.1.1). Before, only `Server`'s HTTP-triggered writes read the `materialized-view-writer-memory` runtime parameter; embedded/library callers of `writeMaterializedView` silently fell back to a fixed 4 GiB default instead. After the merge, both paths share the same function, so embedded callers now also honor the runtime parameter.

5.1.4 Move of the application logic of a SPARQL update from Server into the Qlever class

Applying a SPARQL update to the delta triples was, before this refactoring, logic that `Server` carried out itself, even though none of it touches HTTP concerns. This is the same criterion the Move Function method (subsubsection 4.1.3) already applies, so we moved this logic out of `Server`'s scope. The right place for it is `Qlever`'s class, since even before this refactoring it already operated on `Qlever`'s members through `Server`'s getters. This was a simple relocation, a body-for-body move with small adjustments. One of these adjustments is a new safeguard: verifying that the update was planned against the very index the given delta triples belong to:

Listing 9: The correctness check added to `Qlever::applyUpdate`, the one behavior addition disclosed in this move.

```
UpdateMetadata Qlever::applyUpdate(
    const PlannedQuery& plannedUpdate,
    ad_utility::SharedCancellationHandle cancellationHandle,
    DeltaTriples& deltaTriples, ad_utility::timer::TimeTracer& tracer) {
  AD_CORRECTNESS_CHECK(plannedUpdate.parsedQuery().hasUpdateClause());
  AD_CORRECTNESS_CHECK(
      &plannedUpdate.getIndex().getImpl() == &deltaTriples.getIndex());
  // ... (executeUpdate call and cache clearing, unchanged body-for-body move)
}
```

The result of this refactoring is one less engine-layer concern inside `Server`. It also brings a testing benefit, though not the one it might seem to at first: running an update through `Server` already exercised this logic via the existing end-to-end tests, so it was

not untestable before. The distinction is precision, not possibility. Before, exercising this logic meant going through the full HTTP path: building a request, dispatching it through `Server`, and parsing the response, which entangled it with `Server`'s threading, timing, and response-building. `Qlever::applyUpdate` is now a public method on the engine class itself, so the new tests, `LibQlever.applyUpdate` (a GoogleTest suite name, distinct from the `libqlever` library despite the similar spelling), `LibQlever.clearDeltaTriples`, and `LibQlever.vacuumDeltaTriples`, instantiate a `Qlever` directly and call it with no HTTP request, no `Server`, and no response formatting involved: a narrow, focused unit test of just the engine logic instead of an end-to-end one.

5.1.5 Extraction of delta-triples management logic from Server into the Qlever class

At the start of this refactoring, one of the biggest challenges was the `process` function itself. The change described here is one of several preparation steps toward decomposing that function, which we cover in subsection 5.3. This function had, before this change, the execution logic for clearing and vacuuming delta triples inlined inside its branching logic: a lambda capturing `indexAndViewsSnapshot()` and calling `deltaTriplesManager().modify(...)` directly inside each command's if-branch. This made the function needlessly long and mixed engine logic into the HTTP dispatch logic. Moreover, no test exercised the correctness of these paths directly. We applied the Move Function method (subsubsection 4.1.3) here, following the same pattern as the move of `writeMaterializedView`: the engine logic now lives in `Qlever::clearDeltaTriples` and `Qlever::vacuumDeltaTriples`, and `Server` gets two new thin coroutine wrappers, `processClearDeltaTriples` and `processVacuumDeltaTriples`, that just call them. Each branch inside `Server::process` shrinks to a single call to its wrapper, with no engine logic left in the dispatch function.

With this, we have finished moving logic out of `Server` that belonged somewhere else, mainly to `Qlever`, for this refactoring phase. We followed Parnas's information-hiding principle here [10], letting `Server` shed non-concerns to the places better suited to them. In the next section, we look at parts that touch the HTTP layer, helping `Server`'s cycle, but that do not need to keep that load on `Server`'s class and can be extracted into their own helper modules.

5.2 Extracting Server's Request/Response Helpers

Similarly to the previous section, we are again concerned with the location of code. This time, however, the logic keeps touching HTTP-specific types instead of moving into the engine layer, but it still leaves `Server`'s class scope, extracted into its own, independent helper units.

5.2.1 Extraction of request-parameter interpretation logic from Server into a named namespace

Before we dive into the long-function refactorings themselves, we prepare the ground by simplifying the interpretation of request parameters. The earlier state of `Server` had five private, static methods that turned an already-parsed parameter-value map into typed results. None of them depends on anything specific to `Server` itself, which is exactly why this logic can safely move out of its class scope. For this reason, we introduced a new namespace, `qlever::http_api_helpers`, and moved `Server`'s media-type determination, result-pinning, and send-limit logic into it. Two smaller improvements rode along: `determineMediaTypes` drops its templated `Request` parameter for a plain `acceptHeader` string, making it request-type-agnostic, and its if-else chain becomes a lookup table iterated in a loop; the send-limit logic splits into single-responsibility functions, with applying the resolved limit moving onto `ParsedQuery` itself as `updateExportLimit`, since that belongs to the query's own scope rather than to parameter interpretation.

This extraction also uncovered a missing check: nothing rejected a negative value for the send limit. Passed to `std::stoul`, a negative value was silently accepted and converted to a huge unsigned number. This is not a functional problem: omitting `send` entirely already resolves to the same effectively unlimited default, `std::numeric_limits<uint64_t>::max()`. It is a contract violation: the function accepted an out-of-domain input instead of rejecting it. We now throw a clear error for this case, backed by new test cases, shown below.

Listing 10: The correctness check added while parsing the `send` parameter, the one behavior addition disclosed in this extraction.

```
// Before (Server::adjustParsedQueryLimitOffset):
if (sendParameter.has_value() && considerSendParameter) {
  exportLimit = std::stoul(sendParameter.value());
}

// After (qlever::http_api_helpers::detail::parseSendLimit):
// The parameter cannot be negative. ‘std::stoull‘ would otherwise accept
// a leading ’-’ and convert it to an unsigned value.
AD_CONTRACT_CHECK(!sendParameter->starts_with('-'));
return std::stoull(sendParameter.value());
```

As a result, the five methods above, and the tests that already covered them, move into a new, independently testable `HttpApiHelpersTest.cpp`. The single test for the send-limit logic splits into three focused ones, `parseSendLimit`, `considerSendParameter`, and `determineSendLimit`, and one of these gains the new negative-value rejection case described above.

5.2.2 Extraction of JSON response composition out of Server

Logic touching JSON is usually connected to the HTTP layer, and this case was no different: the construction of JSON responses for index statistics, cache statistics, but also a function composing the JSON error message. Composing a JSON response is a legitimately different concern from running `Server`'s own logic: it concerns only the very last step of the dispatch. This step can later be extended with more and more JSON response types, so it makes sense to extract it into its own logical module, structured solely for composing JSON responses. This extraction gained us two new tests, plus one moved out of `Server`'s own test suite.

We have now covered the logic that still touches the HTTP layer but can still be moved logically out of `Server`'s class scope into its own independent helpers. This stands in contrast to the next section, where we no longer leave `Server`'s class scope at all: there, we mainly decompose its long functions into small, testable logical units.

5.3 Decomposing Server's Long Functions

Unlike the previous two subsections, here we no longer move logic out of `Server` at all. Instead, we decompose logic that legitimately stays as `Server`'s own member functions, into more structured, more easily testable units. Our focus is on the `run` and `process` functions. Both are important HTTP-dispatch logic functions that need to stay in `Server`, but have accumulated many concerns since the early days of QLever's development. We therefore apply refactoring steps for better organization, readability, and testability.

5.3.1 Server's run function decomposition

The pre-refactoring state of `Server`'s `run` function was responsible for more than its name suggests. Both the full HTTP session handler and the WebSocket handler factory lived inline inside it. Besides that, it also had duplicated `catch` branches, ready for extraction as well. Its testability was never exercised, because `run` requires a real bound socket. We extracted this function into three named functions, as follows: the HTTP session handler becomes member function `handleHttpRequest`; the duplicated exception-handling collapses into its own named function, `reportHttpError`; and finally the WebSocket factory becomes its own named function, `makeWebSocketSessionSupplier`. Directly comparing the two versions, `run` shrinks from 88 to 19 lines, the cleanest before/after line-count story of this refactoring phase. With these changes, we restored the purpose implied by `run`'s own name, and were able to add two new tests covering paths that were not covered before: `ServerTest.handleHttpRequest` and `ServerTest.makeWebSocketSessionSupplier`. The extraction of the duplicated `catch`

tail into `reportHttpError` is the same Extract Function example already shown in Listing 4.

5.3.2 Extraction of Server::process's inlined command handlers

`Server`'s `process` function was the head and heart of the `Server` class in one. It was a very long function handling the dispatch process, and the logic for most commands was handled there inline. This contributed heavily to the function's length. We therefore applied extraction steps to this inlined logic. The first part of the function deals with the URL command parameter: a long, 14-branch if-else chain. At first glance, it seems complicated to read and to make sense of, so we tried to extract as much as possible to clean it up. First, we introduced a new namespace for these extracted `Server`-process helpers. There, we extracted the lambda logic for checking a valid access token, and the duplicated logic of dispatching a log message. To make the branches more readable, we added an array storing each command's metadata: its name, its description for logging, and whether it needs authorization. This made the branching logic much cleaner and easier to work with. For the command branching itself, we added a new lambda, `commandIs`, which saved us duplicated lines for checking the parameter inside the parameter-value map together with the dispatching logic we extracted. This way, we were able to shrink each branch to a call to that checking lambda, containing only the short command name and the branch's actual execution lines.

Listing 11: The command metadata array added to the new `serverProcessHelpers` namespace. This is a metadata table, not a function-pointer dispatch table: the if-else chain in `process` itself is unchanged; the array only centralizes each command's logging and authorization requirement.

```
struct CommandMeta {
  std::string_view name_;
  std::string_view description_;
  bool requiresAuth_;
};
constexpr std::array commands = {
    CommandMeta{"clear-cache", "clear the cache (unpinned elements only)", false},
    CommandMeta{"clear-delta-triples", "clear delta triples", true},
    // ... (12 more entries, elided)
};
```

With this cleaner setup, we were able to see the branches' own execution lines that could still be extracted. This is where the execution logic of `processClearDeltaTriples` and `processVacuumDeltaTriples` (already covered in the first block, since their engine logic moved into `Qlever`), `processWriteMaterializedView`, `processLoadMaterializedView`, `processDeleteMaterializedView`, and `processRebuildIndex` belongs. This way, we

were able to shrink the branches' lines even further, reaching a better state than we started with and setting the ground for future simplifications.

5.3.3 Extraction of Server::process's inlined ping, metrics, and runtime-parameter handling

We continue in this subsubsection in the same vein as the previous one, with one distinction: the changes here do not come via the same URL-parameter dispatch, since the inlined `/ping` and `/metrics` requests each have their own entry point, distinguished from the general one. The code smell, however, is the same: we want to extract these inlined code snippets out of `Server`'s process-dispatching logic. For this refactoring step we cite the Extract Function method (subsubsection 4.1.2) again: we introduced three new extracted functions, `processPing`, `processMetrics`, and `processSetRuntimeParameters`, shrinking `Server::process`'s branches and, with them, its general overload. Setting a runtime parameter also repeated the same three steps, an access-token check, a log line, and applying the value, at every call site; we extracted that shape once into `checkAndLogParameterSetting` in the `serverProcessHelpers` namespace, so each call site now collapses to a single conditional call. Again, these extractions made it possible to add focused test coverage for these specific paths, which did not exist before.

5.3.4 Extraction of Server::process's SPARQL-operation dispatch

Another step towards a better `process` function is to extract yet another piece of inlined logic out of it. This time, we extract the SPARQL-operation-handling lambdas. We extracted the construction of the inline lambda visitors for query, update, graph store, and none into their own named function, `processSparqlOperation`. This extraction uncovered two smells inside those lambdas: the shared `visitOperation` lambda took an extra expected-operation/message pair that only `visitQuery`/`visitUpdate` actually needed, forcing `visitGraphStore` to fake values for it, since a Graph Store operation can legitimately be either a query or an update. It also uncovered a second smell: `visitQuery` built a throwaway tracer, `std::make_shared<ad_utility::timer::TimeTracer>("dummy")`, purely to satisfy `visitOperation`'s previously mandatory `tracer` parameter. Making that parameter optional, defaulting to `nullptr`, lets the call site drop it entirely. This is a nice example of a refactoring uncovering further smells along the way.

5.3.5 Extraction of Server::process's cmd= command dispatch

Last but not least, we want to take the simplification of `Server`'s `process` function further, by extracting its longest, already-simplified branching structure out of it: the commands dispatch logic. Command processing gets its own named function, `processCommands`, and undergoes further simplification along the way: with every branch now returning a `ProcessCommandsResult` instead of setting a shared `response` variable, each branch would

otherwise repeat the same `ProcessCommandsResult{...}` wrapping. We introduce a new lambda helper, `makeCommandResult`, that defines this wrapping once instead; `cacheStats`, a lambda already used the same way in `process()` before this change, now calls it too. Adding this helper creates more room and more readability along with it. Two smells surfaced along the way, both cases of an invalid input being silently ignored rather than rejected. Submitting a command that was not in the commands table previously had no effect and gave no indication anything was wrong; it now throws instead, making the failure visible rather than silent. Likewise, sending a request that combines a command with a query or update it does not support was silently ignored before; this case now throws as well, for the same reason. With these steps, we have clarified the intended behavior of `Server`'s dispatch logic further, and anchored it in new unit test.

Listing 12: `processCommands`'s command branches, before and after introducing `makeCommandResult`. The repeated `ProcessCommandsResult{...}` wrapping collapses into one call.

```
// Before: every branch wraps its own result the same way (two of several
// branches in the same if/else chain shown):
if (commandIs("stats")) {
  co_return ProcessCommandsResult{jsonResponse(composeIndexStats(index))};
} else if (commandIs("cache-stats")) {
  co_return ProcessCommandsResult{jsonResponse(cacheStats())};
}
// ... (repeated the same way in several more branches)

// After: makeCommandResult defines that wrapping once:
auto makeCommandResult = [&makeJsonResponse](const json& j) {
  return ProcessCommandsResult{makeJsonResponse(j)};
};
// ...
if (commandIs("stats")) {
  co_return makeCommandResult(composeIndexStats(index));
} else if (commandIs("cache-stats")) {
  co_return cacheStats();
}
```

With this, we have covered the decomposition of `Server`'s long, complex functions into more compact, modular units. In the following part, we inspect what these changes actually uncovered along the way, and what we did about it.

5.4 Smells Uncovered Along the Way

During this refactoring, some new code smells surfaced that we were not originally aiming for. We have already addressed some of these indirectly within the earlier sections. In

this one, we briefly discuss those that were not addressed indirectly, but rather directly, as by-products of the previous refactoring steps.

5.4.1 Replacing empty-optional signaling with a thrown exception

Only after we had already extracted `process`'s inline command handlers, in particular `processVacuumDeltaTriples` and `processWriteMaterializedView`, did we notice a smell we could not refactor as part of those earlier steps, since it concerned a different issue entirely and deserved its own treatment. So far, those functions needed to check, via `verifyUserSubmittedQueryTimeout`, whether a user-submitted timeout could be accepted. This function was called from inside the processing functions. In the case of an invalid user-submitted timeout, it returned an empty optional, signaling to its caller that execution should stop. Since those functions were called through coroutines, there was no way to signal this other than passing that empty optional through every function between it and `process` itself, which also meant threading the request and the `send` callable along with it. To change this, `verifyUserSubmittedQueryTimeout` no longer sends a response itself; it only throws an `HttpError`, which is then caught in one place together with other errors and exceptions, reducing the number of functions that need to know how to send a response. This change also let us drop the request/response template parameters on `processVacuumDeltaTriples`, `processWriteMaterializedView`, and `verifyUserSubmittedQueryTimeout` itself, simplifying their structure and shrinking the surrounding code. We also extended an existing test to cover this behavior directly. This gives us better control over the code's flow and makes future changes easier, since there is no longer an optional that needs to be passed along. This is the same Replace Error Code with Exception transformation already shown in Listing 5 (subsubsection 4.1.5).

5.4.2 Extraction of the logging and highlighting logic for a failed operation's error message

Another part that surfaced was code inside `Server::processOperation` that logged a failed operation's error message and the query with its offending clause highlighted, all inline. This had a try/catch fallback for the case where the highlighting itself fails, which can happen because of differing Unicode handling between QLever and ANTLR (ANother Tool for Language Recognition). Since this was a genuine mixed concern inside that function, we applied the Extract Function method (subsubsection 4.1.2) and moved it into a new free function, `logErrorAndHighlightedMetadata`, placed in the existing `util/ParseException.h/.cpp` rather than a new module of its own, keeping it next to `ExceptionMetadata`, the class it operates on. This way, we were able to add targeted unit tests that were missing before, including for the fallback path itself.

Listing 13: The error-logging-and-highlighting block, before and after the refactoring. The whole block collapses to one call.

```
// Before: inline in Server::processOperation:
AD_LOG_ERROR << exceptionErrorMsg.value() << std::endl;
if (metadata) {
  try {
    AD_LOG_ERROR << metadata.value().coloredError() << std::endl;
  } catch (const std::exception& e) {
    exceptionErrorMsg.value().append(/* ... failure reason appended ... */);
    AD_LOG_ERROR << "Failed to highlight error in operation. " << e.what()
                 << std::endl;
    AD_LOG_ERROR << metadata.value().query_ << std::endl;
  }
}

// After: moved into util/ParseException.h/.cpp, called with one line:
logErrorAndHighlightedMetadata(exceptionErrorMsg.value(), metadata);
```

In this chapter, we were able to see up close what has changed and, to some extent, what we were able to achieve by it. We have seen multiple instances of moving logic, extraction, decomposition, and also resolving challenges uncovered along the way in our work. In the next chapter, we concentrate our direction on how precisely we have improved the code by reaching this explained state.

6 Evaluation

In this section, we demonstrate our refactoring's impact on code quality by presenting the results of the following metrics. We first look at the increased number of code paths exercised by unit tests, which were not tested before. After covering testability, we then inspect what happened to the readability of the functions worked on, along with cyclomatic complexity as a metric. Next, we look at the results of the coupling measurement. Since QLever is a live, actively developed project, this refactoring itself landed as a series of incremental changes via individual pull requests on GitHub; we tracked these changes directly and took care to isolate their effect from unrelated changes landing in between, rather than simply comparing two arbitrary snapshots of the whole project. This should paint a fuller picture: not only how the code improved, as explained in the previous chapter, but also an objective, numeric confirmation of that improvement.

6.1 Testability

As a starting point, we cover the changes in the code's testability. During the refactoring, a lot of extraction, decomposition, and moving of code happened. These changes made code paths that were previously unreachable by tests reachable, and we exercised them with new unit tests. Out of the 18 newly added tests, we take a closer look at two representative examples.

Parsing send limit test Before the refactoring, parsing the send limit for a user's request lived entirely on `Server`'s side, all in one function. The three logically separate steps, 1. parsing the send limit, 2. considering whether the send limit applies for the given media type, and 3. determining the actual send limit, were all buried inside one function, `Server::adjustParsedQueryLimitOffset`. Only after extracting each of these into its own function did we uncover a gap in the check's correctness: nothing rejected a negative value, which was silently converted into a huge unsigned number instead (subsubsection 5.2.1). This was not a functional problem, since omitting the parameter entirely already resolved to the same effectively unlimited default; it was a contract violation, the function accepted an out-of-domain input instead of rejecting it. The extraction let us spot this gap and fix it, backed by new tests. Without decomposing the function this way, the gap was not nearly as visible from the surface as it became afterward.

Server run function testing Earlier, `Server`'s `run` function could not be tested directly, since doing so would require a live, real bound socket. Its inlined logic, however, could have been tested if it were not trapped inside that function's scope. With the refactoring of this function, extracting the inlined logic that handles the HTTP request and creates the WebSocket session supplier (subsubsection 5.3.1), we were able to introduce new unit tests

for those two parts, which were previously unreachable. This made `Server`'s `run` function not only more modular, but gave us direct control over testing these infrastructure-level components.

With these two examples, we can see how important the refactoring was in these cases, and what part it played in increasing testability overall. We covered only these two examples in detail, but a similar story played out for the remaining new tests as well, some more and some less similar to the pattern shown here. Table 1 lists all 18, each tied to the subsubsection that introduced it.

Table 1: The 18 newly added tests, by the change that introduced them.

New test	Introduced in
`HttpApiHelpersTest.parseSendLimit`	subsubsection 5.2.1
`HttpApiHelpersTest.considerSendParameter`	subsubsection 5.2.1
`HttpApiHelpersTest.determineSendLimit`	subsubsection 5.2.1
`ResponseJsonTest.composeCacheStats`	subsubsection 5.2.2
`ResponseJsonTest.composeError`	subsubsection 5.2.2
`ServerTest.handleHttpRequest`	subsubsection 5.3.1
`ServerTest.makeWebSocketSessionSupplier`	subsubsection 5.3.1
`HttpApiHelpersTest.getViewNameParameter`	subsubsection 5.3.2
`ServerTest.clearDeltaTriples`	subsubsection 5.1.5
`ServerTest.vacuumDeltaTriples`	subsubsection 5.1.5
`LibQlever.applyUpdate`	subsubsection 5.1.4
`LibQlever.clearDeltaTriples`	subsubsection 5.1.4
`LibQlever.vacuumDeltaTriples`	subsubsection 5.1.4
`ServerTest.pingEndpoint`	subsubsection 5.3.3
`ServerTest.setRuntimeParameters`	subsubsection 5.3.3
`ServerTest.setIndexAndTextDescription`	subsubsection 5.3.3
`ParseException.logErrorAndHighlightedMetadata`	subsubsection 5.4.2
`ServerTest.processCommands`	subsubsection 5.3.5

6.2 Readability and Cognitive Complexity

Next, we move our focus to the improved readability of the refactored functions. We connect this readability property to the cognitive complexity metric, which measures how hard a function is for a human reader to follow by scoring nested decision points more heavily than flat ones. We also report cyclomatic complexity and line-count numbers alongside it, which support this chapter's testability findings and the Long Function smell addressed throughout the refactoring. We again closely inspect two examples, namely `Server`'s `run` and `process` functions. Before we get to that, however, let us first introduce the metrics used in this subsection.

6.2.1 Non-Comment Lines of Code (NLOC)

The number of lines of code is a somewhat naive metric. It does not tell us much about whether code is actually of high quality, since more lines can sometimes mean better readability for a programmer, rather than having to resolve encapsulated, inlined calls. On the other hand, in our case there is a point past which a function becomes too long to follow easily, this was the case for the `process` function, so we use this metric here as well.

6.2.2 Cyclomatic complexity (CCN)

McCabe's measure of cyclomatic complexity directly ties Fowler's Long Function smell to a number [7]. It captures the number of linearly independent paths through a function's control flow: concretely, 1 plus the number of decision points. It is an important measure because, unlike the previous one, two functions can have identical NLOC yet differ a lot in their CCN. This is a testability metric. It sets the lower bound on the number of test cases needed for full branch coverage.

6.2.3 Cognitive complexity

For human readers, multiple decision points, such as if-statements, can be harder to read when nested rather than laid out as a flat sequence. McCabe's original CCN rates both cases the same way; Campbell therefore introduced a newer metric, Cognitive Complexity, to account for this and give a measure that targets readability instead of testability [2]. Note that this is a SonarSource white paper, not a peer-reviewed academic publication like McCabe's own paper above.

For obtaining the NLOC and CCN we used the cyclomatic complexity analyzer tool `lizard` [13]. For the cognitive complexity, nesting-aware check we used the `clang-tidy` tool [6].

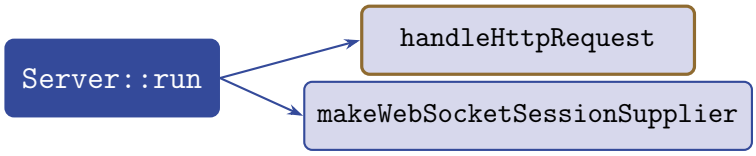
Server run function First, we look at the metrics of the changes discussed in subsubsection 5.3.1 for `Server`'s `run` function.

Table 2: `Server::run`, before and after the refactoring.

Metric	Before	After
NLOC	59	12
Length (lines)	88	19
Cyclomatic complexity (CCN)	6	2
Cognitive complexity	28	2

As we can see in Table 2, every metric drops from its before state to its current one, with the smallest drop of 4 at the cyclomatic complexity and the largest drop of 69 at

the actual length. The logic here simply did not vanish or get deleted. As explained in the refactoring section, it got redistributed into named pieces, each with much lower complexity on its own (`makeWebSocketSessionSupplier` has a cognitive complexity of only 1), while the whole `run` function's own cognitive complexity started at 28 and now shrinks to just 2.[1] These numbers show a concrete effect of the refactoring: lowering the function's complexity and thereby enabling better readability. This allows easier future development and faster code comprehension.



Thick border: has further calls of its own, not expanded here.

Figure 1: `Server::run`'s call decomposition after the refactoring: it now fans out to two named functions instead of doing everything inline.

Server process function Next, we look at the metrics of the changes discussed in subsubsection 5.3.2, subsubsection 5.3.3, subsubsection 5.3.4, and subsubsection 5.3.5 for `Server`'s `process` function.

Table 3: `Server::process`, before and after the refactoring.

Metric	Before	After
NLOC	357	54
Length (lines)	479	104
Cyclomatic complexity (CCN)	39	8
Cognitive complexity	104	12

As expected, the differences here are significantly higher compared to the previous `run` function table. Analyzing the data from Table 3, we can see that the cyclomatic complexity dropped by a factor of about 5. Again, the logic did not vanish, but got split up using the refactoring methods mentioned in section 5: we did not simply hide the complexity elsewhere and present the shrunk number as a nice result. What happened is that the newly split-up pieces also have, each on their own, a much lower complexity: `processCommands`, for instance, now holds the current highest complexity in `Server`'s code, at CCN 19. This is expected, as this function holds the dispatch branches for the known commands. Likewise, another named function introduced by extracting this logic from `process` is `processSparqlOperation`, with CCN 12, or, for example, `processPing`, with CCN 2. So we still have the same logic, only redistributed for

[1]The `lizard` cyclomatic complexity count misreads the `&&` in the lambda's forwarding-reference parameter (`auto&& send`) as a logical-AND operator, equally in both states, so the reported drop of 4 is accurate. The cognitive complexity count also includes a real branch per state from the `AD_LOG_INFO` logging macro, a deliberate efficiency tradeoff documented in its own header, not decision logic specific to `run`.

an overall better overview. The cognitive complexity in the table shrinks even more, dropping by a factor of about nine.[2] This can be explained by the difference in what the two measurement methods penalize: cognitive complexity specifically penalizes the deep nesting this function used to have more strictly. As already stated, nested decision points are much harder on a human reader's eye than flat ones. Across both functions, this redistribution produced eight new named functions in total: from `run`, `handleHttpRequest`, `reportHttpError`, and `makeWebSocketSessionSupplier`; from `process`, `processCommands`, `processSparqlOperation`, `processPing`, `processMetrics`, and `processSetRuntimeParameters`.

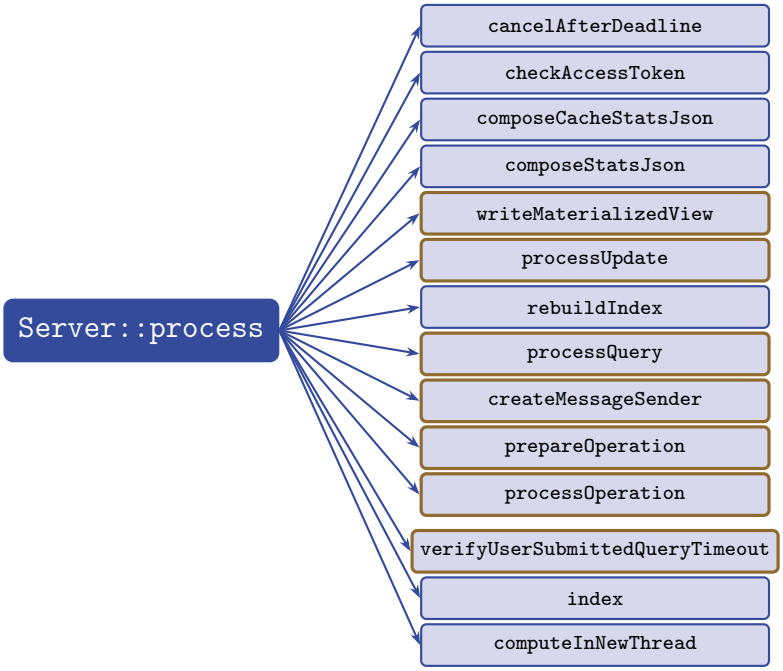


Figure 2: `Server::process`'s call decomposition before the refactoring: already 14 direct calls, but flat and undifferentiated. Ten of them are command-specific handlers (`writeMaterializedView`, `processUpdate`, `processQuery`, and others) sitting as direct siblings of `process` itself, with no single entry point grouping them.

[2]The same artifacts as in the previous footnote (one forwarding-reference `&&`, real per-call `AD_LOG_*` logging branches) appear in `process` too, without changing the ratios stated here.

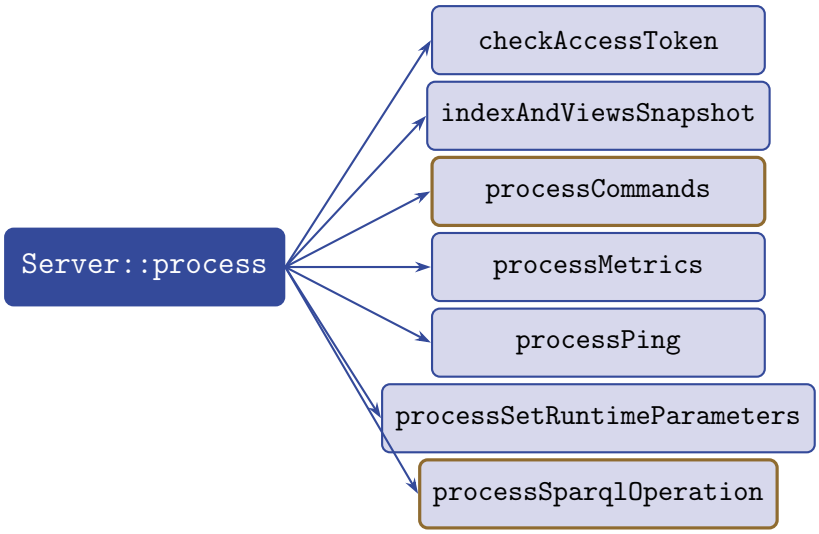


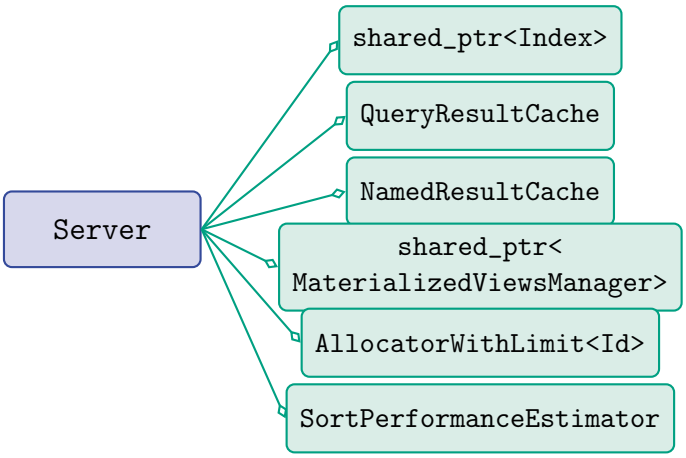
Figure 3: `Server::process`'s call decomposition after the refactoring: down to 7 direct calls, with the command-specific handlers consolidated behind one new entry point, `processCommands`, instead of sitting as flat siblings of `process`.

6.3 Coupling

Here, we investigate the work done on splitting responsibilities, mainly handled in subsection 5.1, where engine logic was moved out of `Server`'s scope to its more suitable location. We cover one concrete example here: the replacement of `Server`'s direct engine-layer members with a single `Qlever` instance.

Engine members replacement via Qlever instance As described in subsubsection 5.1.1, `Server`'s class body had 6 direct engine-layer members, mixing HTTP-layer and engine-layer concerns. Our refactoring resolved this by moving these members out and replacing them with a single, direct `Qlever` instance, as shown in Figure 4.

In summary, we were able to support the general, subjective points about readability and coupling with hard facts and metrics such as CCN, cognitive complexity, unit testing, and the direct engine-member count. These facts support our refactoring's original intentions of achieving better code quality, maintainability, and easier extensibility. Next, we provide future directions this refactoring could continue and build on.



Before: 6 direct engine-layer members

After: 1 direct engine-layer member

Figure 4: `Server`'s direct engine-layer members, before and after the refactoring: 6 direct members collapse into 1.

7 Future Work

This part concerns itself with possible directions and open ideas for future refactoring steps that surfaced with the help of this refactoring, but were no longer in scope for this thesis. So far, QLever distinguishes what the user requests mostly via a query parameter on a single, shared endpoint, for example, the following requests:

`?cmd=stats`, `?cmd=clear-cache`, `?cmd=rebuild-index`

All share the same `/` entry point. However, this does not hold for all of them: `/ping` and `/metrics` already have their own dedicated paths. The future idea is to continue this pattern and extend it to the commands as well, giving each command its own entry point, such as:

`/cmd/stats`, `/cmd/clear-cache`, `/cmd/rebuild-index`

The reason for proposing this as future work is that the current design allows multiple, mutually-exclusive instructions to be sent within the same request's parameter set. This forces the intended behavior to be checked at the code level, inside the `process` and now `processCommands` functions (subsubsection 5.3.5), so the layer at which this decision gets enforced sits one level lower than necessary. An example of this is the new check, `requireNoOperation`, which verifies that no query was sent along with certain commands, and throws if one was:

Listing 14: The `requireNoOperation` check, an example of enforcing this decision at the code level, inside `processCommands`, rather than at the request's own structural level.

```
void requireNoOperation(const sparqlOperation::Operation& operation,
                        std::string_view actionName) {
  if (!std::holds_alternative<sparqlOperation::None>(operation)) {
    throw HttpError(
        boost::beast::http::status::bad_request,
        absl::StrCat("cmd=", actionName,
                     " does not accept an additional query or update"));
  }
}
```

Instead of checking this at runtime, we would move this logic one level higher and eliminate the low-level need for checking this correctness edge case at all. As appealing as this sounds, it does not come without a cost, and some decisions need to be made before diving into this. This would break the current API for anyone already depending on the `?cmd=`

form; the QLever UI, too, would need to be updated to reflect this change. It touches many parts. We therefore propose the following approach: introduce the new path-based routes additively, alongside the existing parameter-based ones, enabling both versions to work at the same time without a hard cutover. Migrate existing clients, such as the QLever UI, to the new routes step by step, over time. Once ready, and once all external users have been informed of and are familiar with this change, move over to only one option.

The additive routing step itself is a small implementation effort, since the new routes can reuse the existing `processCommands` dispatch logic underneath. Migrating the QLever UI and coordinating the eventual cutover with external users is the larger, open-ended cost, because it depends on how quickly current users adopt the new routes.

8 Conclusion

QLever's `Server` class was a great candidate for refactoring, since it mixed HTTP-layer concerns with the engine, had long, high-cognitive-complexity functions, and under-tested code paths. We addressed these code smells using various refactoring methods: we moved HTTP-agnostic engine logic into the `Qlever` library, decomposed `Server`'s hard-to-read functions, added new unit tests, and resolved smells that surfaced along the way.

To prove our findings, we gathered evaluation metrics and showed lower complexity for the main targeted functions. `Server`'s `process` function, previously the single most complex function in the class at a cognitive complexity of 104, dropped to a cognitive complexity of 12; the remaining complexity moved into newly named, individually much simpler functions, such as `processCommands` and `processSparqlOperation` (section 6).

We also proposed future work addressing a limitation this refactoring surfaced: the shared `?cmd=` parameter design forces correctness checks, such as rejecting an unsupported combination of command and query, into the application code itself. We outlined a concrete, additive migration path toward per-command routes that would enforce this at the routing level instead (section 7).

As QLever is a live project, all of our proposed changes are now part of the codebase, creating more organized, more readable, and more extensible code for QLever's current and future contributors. We believe continuing this kind of refactoring work helps keep the project maintainable as it keeps growing.

9 Acknowledgements

I would like to give my biggest thanks to my supervisor M.Sc. Johannes Kalmbach. He always gave me some of the best code reviews and answers to my questions that I could have gotten. I would also like to thank Prof. Dr. Hannah Bast for taking on the role of my examiner.

Further thanks go to the corporate design team at the Albert-Ludwigs-Universität Freiburg and Leander Marius Bürkin for this LaTeX-template.

References

[1] Hannah Bast, Johannes Kalmbach, Robin Textor-Falconi, and Christoph Ullinger. Sparqloscope: A generic benchmark for the comprehensive and concise performance evaluation of SPARQL engines. In *Proceedings of the International Semantic Web Conference (ISWC)*, pages 22–40, 2025.

[2] G. Ann Campbell. Cognitive complexity: A new way of measuring understandability. White paper, SonarSource, 2023. Accessed September 20, 2026.

[3] Martin Fowler. Replace error code with exception. `https://refactoring.com/catalog/replaceErrorCodeWithException.html`. Refactoring catalog, web edition only; not included in the printed second edition of the book. Accessed September 9, 2026.

[4] Martin Fowler and Kent Beck. *Refactoring: Improving the Design of Existing Code*. The Addison-Wesley Signature Series. Addison-Wesley, Boston, second edition, 2019.

[5] Yaroslav Golubev, Zarina Kurbatova, Eman Abdullah AlOmar, Timofey Bryksin, and Mohamed Wiem Mkaouer. One thousand and one stories: A large-scale survey of software refactoring. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1303–1313, 2021.

[6] LLVM Project. Extra clang tools: Clang-tidy. `https://clang.llvm.org/extra/clang-tidy/`. Accessed September 20, 2026.

[7] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[8] Biruk Asmare Muse, Foutse Khomh, and Giuliano Antoniol. Do developers refactor data access code? an empirical study. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 25–35, 2022.

[9] Matheus Paixão, Anderson Uchôa, Ana Carla Bibiano, Daniel Oliveira, Alessandro Garcia, Jens Krinke, and Emilio Arvonio. Behind the intents: An in-depth empirical study on software refactoring in modern code review. In *Proceedings of the 17th*

International Conference on Mining Software Repositories (MSR), pages 125–136, 2020.

[10] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[11] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why we refactor? confessions of GitHub contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 858–870, 2016.

[12] Yue Wang, Wenhui Chang, Tongwei Deng, Yanzhen Zou, and Bing Xie. Decomposing god header file via multi-view graph clustering. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 112–124, 2024.

[13] Terry Yin. Lizard: A code complexity analyzer. `https://github.com/terryyin/lizard`. Accessed September 20, 2026.

Glossary

Notation	Description	Page List
RDF	A W3C standard data model that represents information as subject-predicate-object triples	8
SPARQL	The standard query language for data stored in the RDF data model	8
triple	A statement in the RDF data model, consisting of a subject, a predicate, and an object	1, 8, 9