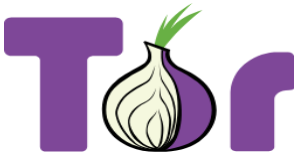
Rewrite metrics-lib in Rust

GSoC 2025 | The Tor Project



Basic Information
Project Details

Project Name	Rewrite metrics-lib in Rust
Skills	Rust, Java
Mentors	Hiro, Sarthik
Project Size	350 Hours

Personal Details

Name	Harshita Roonwal
Date of Birth	09-01-2004
Country	India (Nationality & Current Residence)
Email	roonwal721972@gmail.com
GitLab	@harshita_roonwal
GitHub	harshita9104
Medium	harshita9104
Linkedin	harshita-roonwal
Matrix	harshita_roonwal:matrix.org
University	Atal Bihari Vajpayee Indian Institute of Information Technology and Management
Course	Integrated BTech + MTech in Information Technology
Contact Number	(+91) 8306655190
Time Zone	UTC+05:30 (Asia/Kolkata)
Preferred Language	English

Introduction

I am currently pursuing an Integrated B.Tech + M.Tech (Dual Degree) in Computer Science at ABV - Indian Institute of Information Technology and Management, Gwalior (2021–2026). My journey into coding began with a deep curiosity for problem-solving, which gradually evolved into a passion for building scalable systems and contributing to open-source communities like Kubeedge. Over the years, I've developed strong programming skills in languages such as Rust, Java, Golang, and C++, alongside hands-on experience in web development with ReactJS, Node.js, and backend technologies like MongoDB and PostgreSQL. My comfort with DevOps tools like Docker and Jenkins, combined with a solid foundation in core CS subjects like Operating Systems, DBMS, and Computer Networks, allows me to approach problems with both depth and versatility.

Recently, I have been contributing to the Tor Project, particularly to the eRPC and metric library, where I utilized Rust to enhance functionality and performance. This experience not only strengthened my understanding but also helped me write robust and maintainable code in production-level environments. I'm excited to deepen my involvement in impactful open-source projects and further apply my skills in a collaborative ecosystem like Tor.

Contributions to The Tor Project

As part of my contributions to The Tor Project, I have actively engaged with two key repositories: **eRPC** and library.rs. In eRPC, I improved the relay scanning tool by implementing metadata-based filtering (by country, flags, and ASN), adding structured logging to trace exclusion paths, and I introduced detailed error classification during two-hop builds in one of my merge requests and I also added comprehensive documentation support by integrating both mdBook for high-level wiki-style guides under doc/book and Rust's built-in cargo doc for detailed API references. Parallelly, I designed and developed a full-featured Rust-based parser in library.rs to handle bridge-pool-assignment descriptors published by BridgeDB. This included version-aware parsing, digest computation, CLI integration, PostgreSQL and CSV/Parquet export support, and structured error handling which aligned with Tor's long-term goal of migrating away from Java-based metrics-lib. My contributions span code, documentation, design proposals, and test coverage, demonstrating a strong alignment with the project's vision for a secure and modernized metrics infrastructure

My Contributions to The Tor Project:

PR	Repository	Contribution Summary
#21	**eRPC**	Implemented relay filtering in the eRPC Scanner using RelayFilterConfig, allowing selection of relays based on country codes and flags. Integrated allow_filtering at the task level to toggle filtering as needed. Added unit tests to validate both filtering logic and default non-filtered behavior, ensuring correctness and flexibility. This also addresses Issue #53 by supporting relay selection based on descriptor properties.

#23	eRPC	Introduced detailed error classification during two-hop builds using tor_circmgr::CircuitBuilder. This improvement replaces generic failure handling with granular categories such as Partition, Timeout, Transient, and FirstHopFailed, enabling clearer diagnostics. It also distinguishes between Tor-specific errors (e.g., ChannelTimeout, CircuitExtend), system-level issues (e.g., too many open files), network errors (e.g., unreachable routes), and cases involving missing relays.
#12	eRPC	Introduced comprehensive documentation support by integrating both mdBook for high-level wiki-style guides under doc/book and Rust's built-in cargo doc for detailed API references. Additionally, configured the GitLab CI pipeline to automatically build and deploy these resources to GitLab Pages, ensuring accessible, up-to-date documentation
1	`library.rs`	Proposed modular architecture design and phase-wise layout for library.rs crate.
3	`library.rs`	Complete implementation of bridge-pool-assignment parser with CLI, PostgreSQL, and CSV.

My Contributions to Project KubeEdge:

PR	Project	Description
#680	KubeEdge	Improved the table layout by utilizing full-width formatting for better visual balance, expanded the description column for clarity, replaced long URLs with clickable hyperlinks, and aligned content consistently to enhance overall readability and structure.
#681	KubeEdge	Adjusted the sidebar to improve readability and utilize space effectively and enhanced scrollbar styling and introduced spacing with distinct sections for better navigation
#674	KubeEdge	Improved mobile responsiveness for better usability across different screen sizes and enhanced overall alignment of elements

PROJECT IDEA

Synopsis:

This proposal outlines a plan to develop “rewrite metrics-lib in Rust” project idea for the The Tor Project as part of Google Summer of Code (GSoC) 2025. This project focuses on creating a high-performance, Rust parser and CLI tool for processing Tor’s `bridge-pool-assignment` documents, which describe how sanitized bridge entries are assigned to various distribution pools. The tool supports file parsing from both online and offline sources, fingerprint validation via regex, digest generation using SHA-256, and structured output in multiple formats including PostgreSQL, CSV, and Parquet. Designed with modularity and testability in mind, the parser supports version-aware header handling and is equipped with rich CLI options to facilitate flexible usage. With structured logging, custom error types, async runtime support, and full test coverage, the implementation is tailored to replace the equivalent logic in the legacy Java-based `metrics-lib.`

Summary of the Project:

The Tor Project fosters internet freedom by creating privacy-preserving tools, and BridgeDB plays a critical role in Tor’s censorship circumvention strategy by distributing bridge relays to users in restricted regions. BridgeDB publishes periodic bridge-pool-assignment files containing sanitized metadata—such as SHA-1 fingerprints, transport protocols (e.g., obfs4), and distribution rings (e.g., moat, email, telegram)—used for research, monitoring, and distribution fairness.

This project proposes to replace the current Java-based metrics-lib with a performant, modular, and secure Rust-based library. The new crate will provide full parsing, validation, and export functionalities for Tor descriptors, beginning with bridge-pool-assignment files, and aligned to the CollecTor specification. It will be developed for long-term maintainability and integration into the Rust-based Tor metrics ecosystem.

- **Objective:** Re-implement the parsing, validation, and export functionalities of theJava-based metrics-lib as a standalone, modular Rust crate.
- **Descriptor Coverage:** Focus on parsing Tor network documents, particularly sanitized descriptors like `bridge-pool-assignment` files, which include SHA-1 bridge fingerprints and distribution metadata such as method, transport type, IP version, blocklist information, and performance metrics.
- **Rust-First Design:** Leverage Rust’s memory safety guarantees, async runtime capabilities, and zero-cost abstractions to ensure security and performance in data processing pipelines.
- **Export Backends:** Support structured output to PostgreSQL for relational storage using `tokio-postgres`, as well as to CSV and Parquet formats for analysis in downstream systems.
- **Integration-Ready:** Design the crate for integration into the broader Rust-based Tor Metrics pipeline, with compatibility for CollecTor-driven data ingestion and future descriptor formats.
- **Alignment with Metrics Pipeline Goals:** Directly supports the Tor Project's ongoing strategy to phase out legacy Java infrastructure in favor of safer and more maintainable Rust and Python systems.
- **Future-Proof and Extensible:** Provides a reusable architecture that can be extended to support other Tor descriptor types beyond bridge-pool-assignment files, such as server-descriptors or consensuses.
- **Specification-Driven Development:** Implements all parsing logic and schema transformations in accordance with the Tor directory specification and CollecTor’s documented descriptor structure, ensuring reliability and long-term maintainability.

Benefits to the Community:

This project directly contributes to the Tor Project's ongoing transition from a Java-based to a Rust-based metrics infrastructure by replacing the legacy `metrics-lib` with a modern, efficient, and secure Rust implementation. By providing a well-documented, modular crate for parsing bridge-pool-assignment files and exporting data to PostgreSQL, CSV, and Parquet, the tool improves performance, enhances memory safety, and aligns with Tor's broader ecosystem. The crate's reusable design will enable seamless integration into other components like OnionPerf or future metrics collectors, while also ensuring that sanitized bridge data is more readily available for censorship research, real-time analysis, and dashboard visualizations. This not only benefits Tor developers but also empowers researchers, integrators, and new contributors through accessible, maintainable tooling.

Deliverables:

Phase 1: Community Bonding + Proof of Concept

- Initial Architecture and Research
- BridgeDB Format and Java Metrics-lib Study
- Initial Rust CLI and Parser Implementation
- Digest Abstraction for File and Entry Hashing
- Dry-run and Local Mode for Testing
- Initial PostgreSQL Schema & Insertion Tests

Phase 2: Parser Expansion

- Version-aware Parsing and Validation:
- Online Descriptor Fetching with Retry Logic
- CLI Usability Enhancements

Final Phase: Analytics Export and Integration

- Implement CSV and Parquet Exporters
- End-to-end parsing and exporting pipeline with CLI UX
- Extensive Unit Testing
- Final polished and documented crate ready for merge

Prior Work: Implementation of a Bridge Pool Assignment Parser

To demonstrate feasibility and build domain expertise before applying for GSoC, I implemented a comprehensive Rust-based parser for the sanitized bridge-pool-assignment files published by BridgeDB. This work was based on the official project issue and aligned directly with the goals of the proposed GSoC idea of rewriting Java-based Tor descriptor parsing infrastructure in Rust. Below are the key features of the implementation, including how each feature was developed, tested, and validated with commands and results.

1. Core Bridge Assignment Parsing

Implemented a version-aware line-by-line parser that extracts bridge fingerprints, distribution methods, and metadata using Rust enums and pattern matching. Fingerprints are validated using a strict regex based on Tor's

SHA-1 format. The parser identifies the `@type bridge-pool-assignment` version directive at runtime and can handle both version 1.0 and 1.1 formats with fallbacks.

File : src/transformer/parser.rs

2. Multi-Format Export System

Supports exporting parsed data to PostgreSQL, CSV, and Parquet formats using a trait-based Exporter interface. CSV uses the csv crate, PostgreSQL uses tokio-postgres, and Parquet output is feature-gated using the parquet and arrow crates.

Files:

- PostgreSQL: src/exporter/pg.rs
- CSV: src/exporter/csv.rs
- Parquet: src/exporter/parquet.rs

Verify PostgreSQL Contents

Commands:

```
psql -U postgres -d tor_metrics -c "SELECT COUNT(*) FROM bridge_file;"
psql -U postgres -d tor_metrics -c "SELECT COUNT(*) FROM bridge_entry;"
```

```
harshita@DESKTOP-K77E48K:~/bridge_parser_rs$ sudo service postgresql start
[sudo] password for harshita:
harshita@DESKTOP-K77E48K:~/bridge_parser_rs$ psql -U postgres
Password for user postgres:
psql (14.17 (Ubuntu 14.17-0ubuntu0.22.04.1))
Type "help" for help.

postgres=# \c tor_metrics
You are now connected to database "tor_metrics" as user "postgres".
tor_metrics=# SELECT COUNT(*) FROM bridge_file;
 count
-------
    10
(1 row)

tor_metrics=# SELECT COUNT(*) FROM bridge_entry;
 count
-------
 23822
(1 row)

tor_metrics=#
```

This Should return a non-zero count if insert worked

```
   Compiling bridge-parser v0.1.0 (/home/harshita/bridge_parser_rs)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 3m 06s
     Running `target/debug/bridge-parser --db 'host=localhost user=postgres password=abcd12345 dbname=tor_metrics' --clear`
2025-04-05T11:22:06.587652Z  INFO bridge_parser: 🫧 Starting bridge parser
2025-04-05T11:22:30.365299Z  INFO tokio_postgres::connection: NOTICE: relation "bridge_file" already exists, skipping
2025-04-05T11:22:30.367514Z  INFO tokio_postgres::connection: NOTICE: relation "bridge_entry" already exists, skipping
2025-04-05T11:23:35.955706Z  INFO bridge_parser: ✅ Done
harshita@DESKTOP-K77E48K:~/bridge_parser_rs$ cargo run -- --format csv --csv-output bridges.csv
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 2.36s
     Running `target/debug/bridge-parser --format csv --csv-output bridges.csv`
2025-04-05T11:26:21.023575Z  INFO bridge_parser: 🫧 Starting bridge parser
2025-04-05T11:26:53.154456Z  INFO bridge_parser: ✅ Done
harshita@DESKTOP-K77E48K:~/bridge_parser_rs$ head -n 5 bridges.csv
d001ca7092e5985276607fcadacea505b181e285262736071f460332a1a06af9,0004f8aea55fe852194674c8554d68cc5e7a5bba,email,vanilla,"4,6",,true,functional,untested,
7bb5a42214edbc61f624a1351164052657377acd1871408feefd0b0db1d988c2,0009fd594d4f125e3300826adade11964c925dd5,https,webtunnel,6,,true,functional,untested,
b75c0bff31230dde06a5e8f1d1bb64ef9841500b0a6042ab4d231a03b52ccec3,001b4642a32171ac0360dc0c1fa2f1ae130ee99f,settings,vanilla,4,,false,dysfunctional,untested,
443f488ec29a052eb28a82ff4bdbb65ba2b19c75178cda7a19fba3c62eea870b,00382ab470bc22674984bea11b740c553496a5e3,settings,obfs4,4,,true,functional,untested,
ffddc146e39a2c456cff2966c2e7d169e8bd9737061ca087d5fe47c32cd18d5e,0082e9f98381722fe14fae3248548f98ec485780,reserved,vanilla,"4,6",,false,untested,untested,
```

Run and then check CSV structure

If you want to test writing to a file:

Run: `cargo run -- --local-dir test_data --format csv --csv-output local_bridges.csv`

Then inspect the result:

Command: `column -s, -t local_bridges.csv | less -S`

```
sha                                                              fingerprint                                      distribution_method  transport  ip  blocklist  distributed  state     bandwidth   ratio
29cb100fbdc5f00d320d4ed1a67d51889c9b02c7d2c1f48e16d2978c2c412e31  0004f8aea55fe852194674c8554d68cc5e7a5bba  email                vanilla    "4  6"                      true      functional  untested
959967e11dbf1aefa32e2bbc60503b8912abc535c39bc37359c7cdd6dd78082b  0009fd594d4f125e3300826adade11964c925dd5  https                webtunnel  6                   true        functional  untested
d6e76a4d18fdd12431bf90ce5494217a615012d07fb62c63bedb8fbcb42df6c6  001b4642a32171ac0360dc0c1fa2f1ae130ee99f  settings             vanilla    4                   false       dysfunctional  untested
a04a6840cf77df81d17bf48c7e0cc6bdfb0f568498b096c5d498d6be40bdf915  00382ab470bc22674984bea11b740c553496a5e3  settings             obfs4      4                   true        functional  untested
64a0c034b8a0a56c854411f06d7790682bbfb91f1487b0dcc5a24fe905e28e1af  0082e9f98381722fe14fae3248548f98ec485780  reserved             vanilla    "4  6"                       false       untested    untested
a4b5875abb6ef5fc7e3bf9ab5da1b641177be58a4c00172d6f1bd23a384955cd  009cda476053b5df4414211cb25f9b2190bb0128  settings             obfs4      4                   true        functional  untested
56add0e4772a02a72a7f4ca710bdb045a000f8266d112d969cfa88a98fe47264  00b6545499d9aeaebd6e0e228b92e557d640eb75  email                obfs4      4                   true        functional  untested
1708981e86e571752c6abe298be0d7e3c137cbdd814e089e2d88a85a55989ca6  00e1ae6cb75e47e363e6aef9f67a49c0e854fde7  https                obfs4      6                   true        functional  accepted    1.49
d368ec4fe30fa490a55c0cdf0ed8b56706ee84efc322900ede1cd236ad8c64336  00f61a4a92bc38221b3a509fbe260fed2dbfb20b  https                vanilla    4                   false       dysfunctional  untested
84101ca96fbb04a6002970abb63c0d08919264634f52df2ad96d2dd5898873b3c  01211ae573a28fff727c837ba8f4d7e9e667c74d  settings             webtunnel  6                   false       dysfunctional  untested
d753b8bbb1ae14808536be42345b3d3b28e4b9c33815d2f02c1f04d1993b78dc  01360b087e3659600bdc8ea969610d83ebb055b4  settings             webtunnel  6                   true        functional  untested
18a7c8656eb3c3319b8348f74fb98360887c8664d8ec977e99d2cf12ee2da38f  01471e73d4197d8e7f7d9014211138f4fa4e7de7  https                obfs4      6                   true        functional  untested
bbcf57a18d685201cd5c8829c6abb5df4016672b91cb20b8b1252ff56ded130d  015564c3c036fd723a57510445568c7167774c316  email                vanilla    4                   true        functional  untested
b1fdc82869702807d4879a2918f4987f8d6f10c8c983c55e76825f989d7415ed  018f98665c2092075c03ae0e01e2b911e4e79599  email                obfs4      4                   false       dysfunctional  untested
49d1c399e280c8c300cf13bb03f7cc39afbd466968fc4b8e750f6412e7b71d67  01bf4279de9fc549ffd0fdf6ba39bff8ec63ce2b  settings             obfs4      4                   true        functional  untested
7e716df3a0c671db962efeea5b6c4c6abc423ebefb3feb64095962c0304104419  01c4f4c5dc312e0959eed76e520e899e8b971552  https                obfs4      4                   true        functional  rejected    0.898
c9f23057e0ce9babb593484163d86fe1f1c2425e6a42f0b44a7bdf8fd5c92271  01c6574d62a56f428fa649834d2bb21646f2a78e  moat                 obfs4      6                   true        functional  accepted    1.579
ed2eef754d6437b3a7060ddc102dcc1a2f3deab52afd03275506be07bac2946d  01d7b24a1260a8fb41cb9a66a5ba108937a4a362  telegram             obfs4      4                   true        functional  accepted    1.471
52222b8e9e9f4945d11e04886f6cc3a05bc52e5b44690586282c38ae275403d8  01ff41dbc553407ca420683c57d7eb226eeaee57  https                obfs4      6                   false       dysfunctional  accepted    1.286
cc30ae9b4b5f39aa45faaa9211a93b4a29899906448daa0b3627db22d47b1f7d  020b4b26acb6fff00a4fb01f88d2e9118f3f08b2  settings             obfs4      4                   true        functional  untested
(END)
```

`cargo run` with `--format csv --csv-output bridges.csv` confirms tabular data export. Manual validation using `column -s, -t bridges.csv` shows correct structure.

How to test the Parquet export functionality:

First, build with parquet feature enabled

Command: `cargo build --features parquet_export`

Run with limit to process just 2 files

Run: `cargo run --features parquet_export -- --format parquet --parquet-output test.parquet --limit 2`

```
   Compiling bridge-parser v0.1.0 (/home/harshita/bridge_parser_rs)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 1m 48s
     Running `target/debug/bridge-parser --format parquet --parquet-output test.parquet --limit 2`
2025-04-05T22:52:11.057197Z  INFO bridge_parser: 🔍 Starting bridge parser
2025-04-05T22:52:28.578669Z  INFO bridge_parser: 🔢 Truncated input to 2 files due to --limit
2025-04-05T22:52:28.763062Z  INFO bridge_parser::exporter::parquet: 📊 Exporting 2 assignments to Parquet format...
2025-04-05T22:52:28.763175Z  INFO bridge_parser::exporter::parquet: 📝 Processing 4766 total entries...
2025-04-05T22:52:28.771845Z  INFO bridge_parser::exporter::parquet: 💾 Creating Parquet file: test.parquet
2025-04-05T22:52:28.824889Z  INFO bridge_parser::exporter::parquet: ✅ Successfully exported to Parquet format
2025-04-05T22:52:28.825302Z  INFO bridge_parser: ✅ Done
harshita@DESKTOP-K77E48K:~/bridge_parser_rs$
```

Parquet Export tested

3. SHA-256 Digest System

Every file and line is hashed using a digest trait that implements SHA-256. These digests are used as primary keys and for deduplication. File: src/helper/digest.rs

Command: `cargo test test_digest_trait_hash_entry_works`

```
     Running tests/parser_test.rs (target/debug/deps/parser_test-71b4d2caea4cdfa2)

running 1 test
test test_digest_trait_hash_entry_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 4 filtered out; finished in 0.02s

harshita@DESKTOP-K77E48K:~/bridge_parser_rs$
```

4. Resilient Network Fetching

Fetches bridge-pool-assignment files from CollecTor with retry logic using `tokio-retry`. Supports `.txt.gz` compression, HTTPS with `reqwest`, and exponential backoff.

File: src/collector/fetch.rs

Run: `cargo run -- --base https://collector.torproject.org --path recent/bridge-pool-assignments`

Test with poor network conditions:

```
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 1m 28s
     Running `target/debug/bridge-parser --base 'https://collector.torproject.org' --path recent/bridge-pool-assignme
nts --db 'host=localhost user=postgres password=abcd12345 dbname=tor_metrics'`
2025-04-06T19:34:28.961749Z  INFO bridge_parser: 🔍 Starting bridge parser
2025-04-06T19:34:49.752706Z  INFO tokio_postgres::connection: NOTICE: relation "bridge_file" already exists, skipping

2025-04-06T19:34:49.753963Z  INFO tokio_postgres::connection: NOTICE: relation "bridge_entry" already exists, skippin
g
2025-04-06T19:35:06.653801Z  INFO bridge_parser: ✅ Done
harshita@DESKTOP-K77E48K:~/bridge_parser_rs$
```

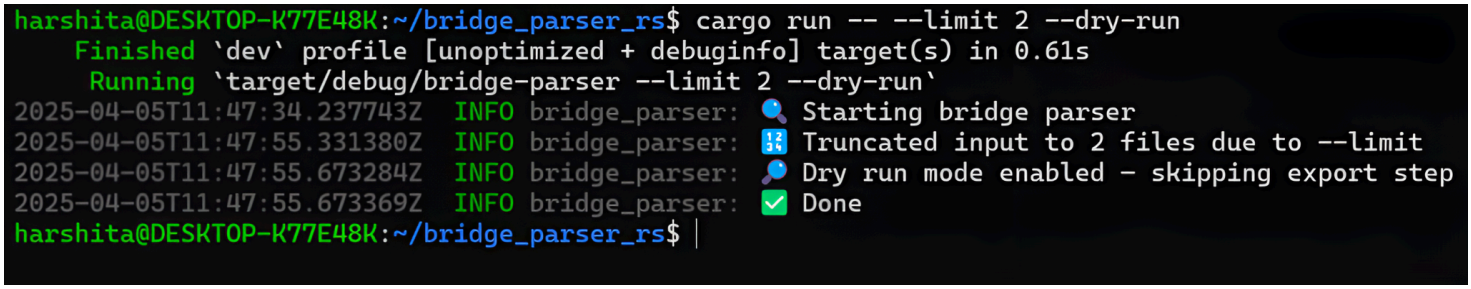
The successful execution and clean exit indicate that the resilient network fetching worked as designed, successfully handling any temporary network issues

5. Offline Mode with Local Files

Provides offline support for parsing pre-downloaded `.txt` or `.gz` files.

File: src/collector/local.rs

Run: cargo run -- --local-dir test_data --dry-run

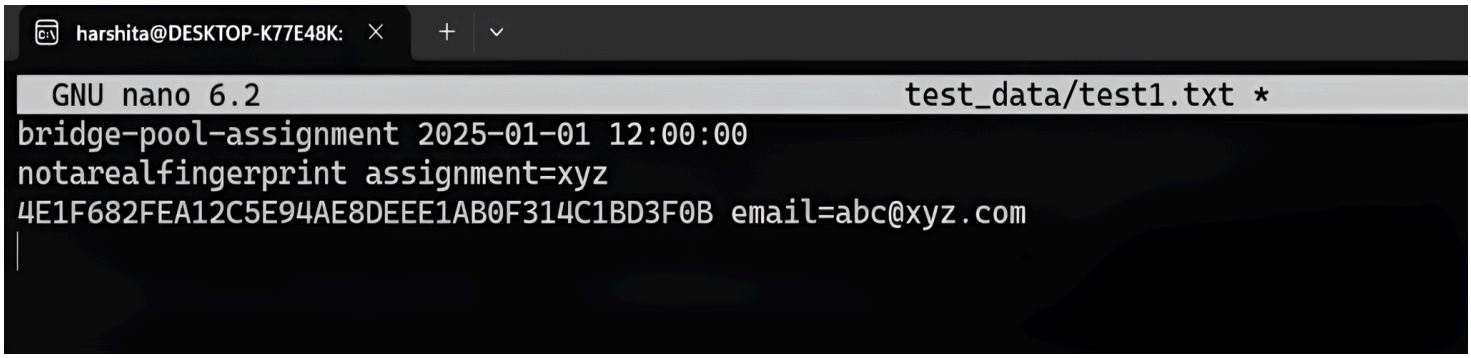


When running with --dry-run, logs confirm files are parsed and validated without touching the network.

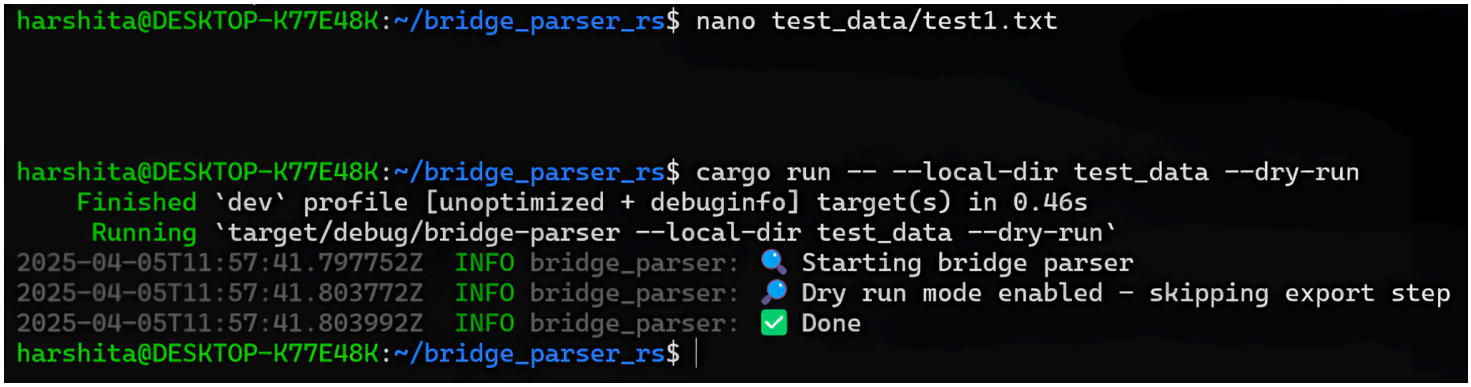
6.Test Invalid Fingerprint Handling

Fingerprint fields are regex-validated to match 40-character hex patterns. Lines failing validation are skipped, ensuring only compliant data is processed.

create a sample file and Paste this inside it:
Line 2 is invalid (not a 40-char hex string)
Line 3 is valid (40-char uppercase hex)



The invalid line should be skipped due to regex validation



7. CLI Interface

Flexible CLI designed using `clap`. Supports flags like `--dry-run`, `--format`, `--limit`, `--local-dir`, `--db`, and `--csv-output`.

8. Error Handling System

Created `BridgeError` enum using the `thiserror` crate to categorize and trace parsing, network, and database errors.

File: src/error.rs
`Run: cargo run -- --base invalid-url`

```
   Compiling bridge-parser v0.1.0 (/home/harshita/bridge_parser_rs)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 1m 32s
     Running `target/debug/bridge-parser --base invalid-url`
2025-04-06T19:52:55.778823Z  INFO bridge_parser: 🔍 Starting bridge parser
Error: Fetch("builder error: relative URL without a base")
harshita@DESKTOP-K77E48K:~/bridge_parser_rs$
```

An invalid URL ("invalid-url") was caught early by the `request` builder. This builder error was then transformed into a more contextual `BridgeError::Fetch` variant. Finally, this specific error was successfully formatted and displayed, demonstrating a clear error handling process.

9. Unit Testing Framework

Developed comprehensive unit tests for regex validation, digest hashing, and descriptor parsing. Implemented using Rust's `#[test]` framework.

File: parser_test.rs

`Test Command: cargo test`

```
    Finished `test` profile [unoptimized + debuginfo] target(s) in 2m 04s
     Running unittests src/lib.rs (target/debug/deps/bridge_parser-356b2cf1e7fa9ae2)

running 1 test
test tests::test_parse_line_valid ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

     Running unittests src/main.rs (target/debug/deps/bridge_parser-eaae08ab0a1b3272)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

     Running tests/parser_test.rs (target/debug/deps/parser_test-71b4d2caea4cdfa2)

running 5 tests
test test_digest_trait_hash_entry_works ... ok
test test_parse_line_invalid ... ok
test test_fallback_version_if_type_missing ... ok
test test_parse_line_valid ... ok
test test_invalid_fingerprint_rejected_by_regex ... ok

test result: ok. 5 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.01s
```

Parser validation, Digest verification, Regex checks

10. Feature Flags

Enabled optional features like Parquet export using Rust's `cfg (feature)` system in `Cargo.toml`.

Command to Enable: `cargo build --features parquet_export`

11. Structured Logging

Using the `tracing` and `tracing-subscriber` crates, all major operations (fetching, parsing, exporting) are logged with structured spans and levels. Logging behavior is customizable via RUST_LOG.

Command: `RUST_LOG=debug cargo run`

12. PostgreSQL Integration:

Export logic is implemented using the `tokio-postgres` crate. Parsed data is inserted into two relational tables (`bridge_pool_assignments_file`, `bridge_pool_assignment`) using prepared statements and batch insertion strategies.

13. Performance Optimizations

Parsing and export are fully asynchronous using the `tokio` runtime. PostgreSQL writes are batched, and database indexing is used for performance.

Benchmark Command: cargo run -- --format postgres --clear

```
harshita@DESKTOP-K77E48K:~/bridge_parser_rs$ cargo run -- \
    --format postgres \
    --clear \
    --db "host=localhost user=postgres password=abcd12345 dbname=tor_metrics"
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 2.38s
     Running `target/debug/bridge-parser --format postgres --clear --db 'host=localhost user=postgres password=abcd12345 dbn
ame=tor_metrics'`
2025-04-07T05:10:33.164193Z  INFO bridge_parser: 🔍 Starting bridge parser
2025-04-07T05:10:57.120364Z  INFO tokio_postgres::connection: NOTICE: relation "bridge_file" already exists, skipping
2025-04-07T05:10:57.123386Z  INFO tokio_postgres::connection: NOTICE: relation "bridge_entry" already exists, skipping
2025-04-07T05:12:07.382137Z  INFO bridge_parser: ✅ Done
```

Technical Details and Implementation Plan

The proposed rewrite of the Tor metrics-lib in Rust is structured around modularity, correctness, and extensibility. While preserving the core functionality of the existing Java-based metrics-lib, the Rust version will incorporate modern architectural practices and leverage the robustness of Rust's ecosystem. The end goal is a maintainable, performant, and verifiable library for parsing and exporting various Tor network descriptors from CollecTor.

The plan follows a descriptor-centric modular architecture where each descriptor type (e.g., consensuses, bridge-pool-assignments, server descriptors) will have its own parser, digest logic, validation module, and exporter interfaces. Each component will be implemented using best practices and feature-rich crates documented in the official Rust ecosystem.

Descriptor Parsing Framework

Each descriptor type will be processed through a dedicated DescriptorParser trait. This abstraction allows the implementation of version-aware, line-based parsers that adapt to structural changes over time. Descriptor files such as those published under the CollecTor bridge pool assignment specification will be mapped into structured Rust types that maintain semantic compatibility with the original Java metrics-lib.

Line-by-line transformation is handled using idiomatic match patterns and regular expressions via the regex crate, ensuring structural validity of fields such as SHA-1 fingerprints, transport type, or IP metadata. Parsing correctness will be validated through unit tests targeting header recognition, fingerprint format, and field extraction.

CLI Interface and Command Argument Control

All user input and behavior configuration will be handled via clap to build a user-friendly and declarative command-line interface. The CLI will support arguments such as --dry-run, --limit, --format, --csv-output, and --local-dir, which will influence the runtime workflow and output format. These inputs will allow flexible switching between test-mode parsing, offline development, and full-scale data export.

Asynchronous File Processing and Streaming

All I/O operations will be asynchronous and non-blocking, powered by the tokio runtime. Asynchronous readers will be implemented for loading local files, decompressing archives, and performing inserts into databases. This enables robust pipeline performance and responsiveness even when handling hundreds of thousands of descriptor entries or large historical archives, especially when parsing real-time CollecTor data in production settings.

Specification-Driven Development

All parsing and transformation logic will be aligned with the official Tor Directory Specification and the published CollecTor descriptor format. The PostgreSQL schema will closely follow the schema defined at bridge_pool_assignment_tables.sql, ensuring compatibility with historical storage pipelines and analysis tools.

Flexible Export Architecture

A polymorphic export system will be developed using an Exporter trait. Each export backend will implement this trait and support conversion from parsed bridge entries to a target output. The supported formats include:

- PostgreSQL Export via `tokio-postgres`: For normalized relational storage and indexed querying.
- CSV Export via `csv`: For flat file output and human-readable inspection.
- Parquet Export via `arrow` and `parquet`: For efficient binary columnar analytics. Parquet will be gated under the parquet_export Cargo feature.

Parquet support is feature-gated and toggled at build time, following the Cargo features design model.

Content-Addressable Digest Logic

To enforce integrity, uniqueness, and deduplication across descriptors, each parsed document and individual entry will be hashed using a trait-based digest framework. The hashing is implemented using `sha2` and hex conversion via `hex`. Digest results are used as primary keys in PostgreSQL and as unique identifiers for indexing.

This architecture can later accommodate alternate hash algorithms like BLAKE3.

Online and Offline Operation Modes

The crate will support both offline development (via --local-dir) and online mode (via CollecTor endpoints). Files will be downloaded using reqwest and optionally retried using tokio-retry in case of transient network failures. Both `.txt` and compressed file types will be processed, depending on file extension detection.

Version Extensibility

The parser is designed to detect and support multiple versions of descriptors using a simple match logic on the @type header. Parsing strategies for newer versions such as 1.1 can be introduced non-disruptively in the future, allowing for long-term maintainability.

Schema-Driven PostgreSQL Integration

The database schema used for inserts is based directly on Tor's published schema and will be dynamically created at runtime using asynchronous migration logic. Indexing will be applied on high-frequency lookup fields like fingerprint and published timestamp, ensuring query efficiency. Insertions will be batched where possible to reduce round trips and leverage PostgreSQL's performance features.

Logging and Runtime Observability

Runtime observability will be implemented using `tracing` and `tracing-subscriber`. Each stage - parsing, exporting, fetching—will be wrapped in spans to enable structured logs with level filtering. This provides observability for large-scale usage and eases debugging in CI/CD environments.

Structured Error Handling

Errors throughout the system will be handled using statically typed enums defined via thiserror. Rather than relying on dynamic error models, the system will use contextual BridgeError variants such as ParseError, NetworkError, and DbError, returning strongly typed Result<T, BridgeError> outcomes. This promotes maintainability and robustness by enabling compiler-level exhaustiveness checking.

Compression and Decompression Support

Many CollecTor documents are distributed in `.gz` or `.xz` formats. To support streaming decompression, the crate will use:

- `flate2` for streaming GZIP decompression.
- `xz2` for XZ-based decompression.

These libraries are compatible with Rust's Read and AsyncRead traits, enabling efficient streaming decompression suitable for both local and network-based workflows, making it suitable for future integration with object stores or streaming HTTP fetches.

Integration with Environment Config

Configuration management for credentials, file paths, and database URIs is handled through `dotenvy`, which automatically loads `.env` files into environment variables, ensuring clean separation of config from source logic.

This promotes reproducibility and portability, especially when deploying across development, testing, and CI/CD environments.

Component	Crate used	Purpose
CLI	clap	Argument parsing for flexible user input
Async Runtime	tokio	Powers non-blocking PostgreSQL and file I/O
HTTP(future use)	reqwest	Optional fetching from remote CollecTor
DB Export	tokio-postgress	Async insertion of entries using parameterized queries
Serialization	Serde, serde_json	Struct serialization for JSON, DB bindings, and file output
Export (CSV)	csv	Fast, memory-efficient export of tabular records
Export (Parquet)	Arrow, parquet	Feature-gated columnar data export for analytics
Regex	regex	SHA-1 fingerprint and key=value

		validation
Error Handling	thiserror	Typed error variants for each logical layer of the codebase
Logging	tracing, tracing-subscriber	Span-based observability during fetch/parse/export stages
Hashing	sha2, hex	Implements digest-based primary keys and deduplication for bridge entries
Retry Logic	tokio-retry	HTTP retry support for resilient file fetching
Compression	flate2, xz2	Optional support for `.gz` and `.xz` file decompression

Flow of the System

The following diagram outlines the control and data flow across the parsing pipeline developed for the bridge-pool-assignment descriptors. It captures the logical flow of operations, from ingestion and configuration, through transformation, to final export. This layered, decoupled architecture improves maintainability and ensures each component performs a single, focused task.

Input Handling Layer:

The system begins by accepting descriptor data either from local directories `--local-dir` or by fetching from CollecTor endpoints via HTTPS. Remote fetching utilizes `reqwest` with retry logic via tokio-retry. Local ingestion supports pre-fetched and optionally compressed files (`.gz`, `.xz`) which are decompressed using `flate2` and `xz2`.

Runtime Orchestration via CLI:

All behaviors—source selection, output type, processing limits—are orchestrated through a structured CLI interface defined using the `clap` crate. Flags like `--dry-run`, `--format`, `--limit`, `--csv-output`, and `--parquet-output` make the tool highly customizable and CI-friendly.

Pre-Parse Decompression & Normalization:

Before parsing, input files undergo decompression if needed. This enables compatibility with typical archival formats used by CollecTor. The decompressed buffer is then cleaned and normalized line-by-line for consistency in parsing, which facilitates downstream regex validation and matching.

Parsing & Validation Core:

The bridge assignment documents are parsed into structured Rust types that mirror the schema defined in metrics-lib's BridgePoolAssignment.java. The parser distinguishes header metadata and bridge lines, applying validation logic using regex for SHA-1 fingerprint conformity and field integrity.

SHA-256 Digesting Layer:

For every validated line and file, a SHA-256 digest is computed using `sha2` and `hex` crates. These digests serve as primary keys in the schema defined in Tor's PostgreSQL schema reference. Digesting also enables content-based deduplication, ensuring that redundant documents are not stored or reprocessed.

In-Memory BridgeRecord Builder:

A validated, hashed bridge entry is then encapsulated in a unified `BridgeParsedAssignment` struct which abstracts over transport details, distribution method, IP type, bandwidth, state, and more. This struct is serializable via `serde` and is the pivot point between parsing and export logic.

Format-Agnostic Export Dispatcher:

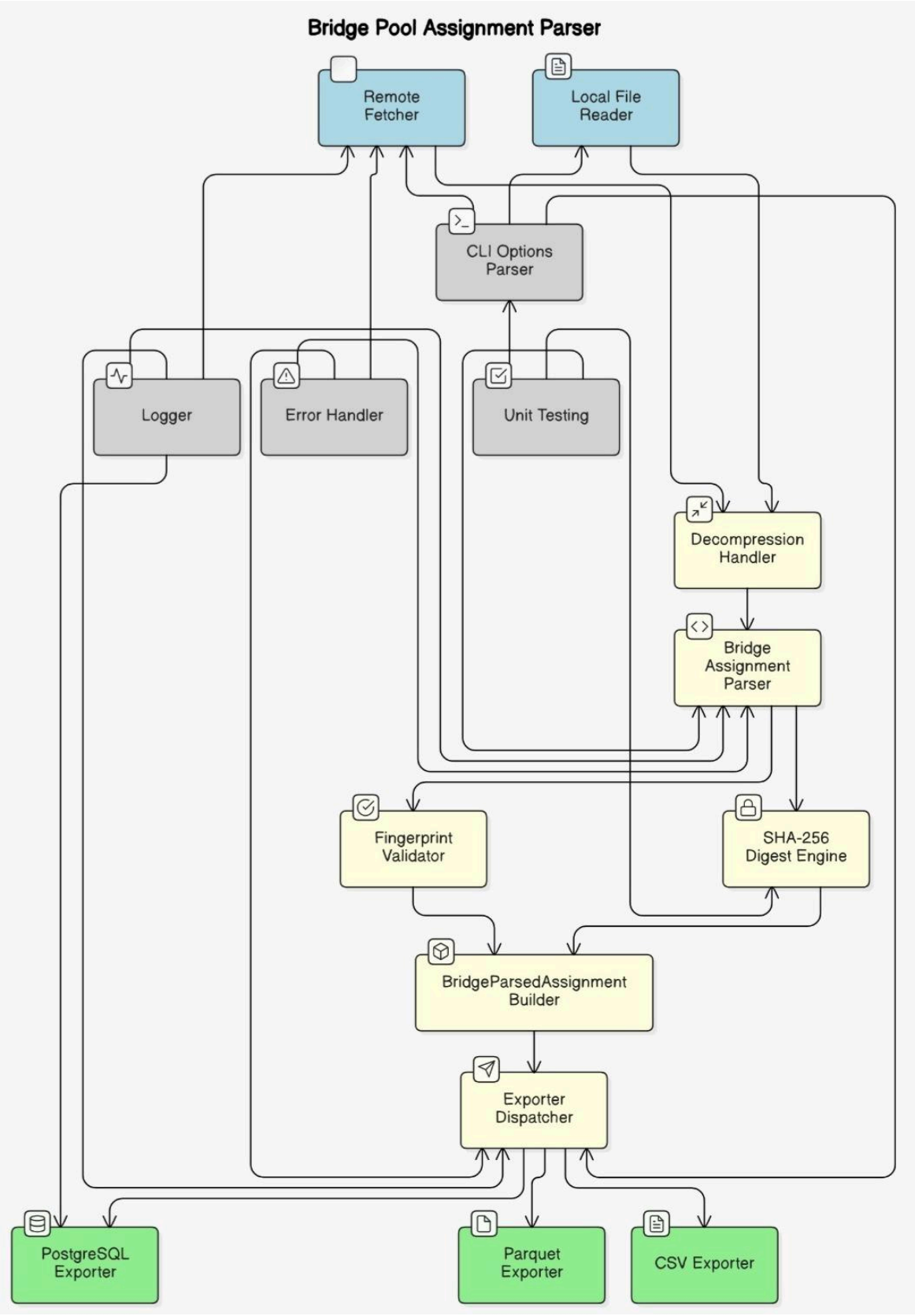
An exporter router then routes the parsed records to the appropriate backend depending on the user-defined `--format`. Export targets include:

- PostgreSQL: Inserted using `tokio-postgres`.
- CSV and Parquet: Written using `csv`. and stored using `parquet` and `arrow` for analytics workflows.

The export layer also supports batch inserts and transactions for atomicity and performance.

Logging, Errors, and Testing Feedback:

The entire flow is instrumented with structured logs via `tracing` and environment filters via `tracing-subscriber`. Errors are scoped and described cleanly using `thiserror` for modular diagnostics. A full suite of unit tests under `/tests/` ensures robustness at each transformation step.



This flow diagram encapsulates the modular, streaming-based architecture of the bridge assignment parser—from data acquisition to backend export.

Proposed Timeline:

The project is planned for the GSoC 2025 timeline (350-hour project). Below is a week-by-week breakdown, including the community bonding period, planning, development milestones for each task, testing phases, integration of components and buffer time for polishing and documentation. This timeline is tentative and may be adjusted in consultation with mentors, but it demonstrates a clear plan for steady progress. I will use an iterative approach: implement basic versions early, then refine and add features, ensuring that by mid-term evaluations a significant portion is functional, and leaving adequate time for testing and improvements towards the end.

Timeline	Activity	Key Tasks / Deliverables
Community Bonding *(May 8 – May 28)*	**Research & Setup**	Deep dive into the existing Java-based descriptor logic. Finalize the modular Rust design. Set up code scaffolding, PostgreSQL backend, and organize the local and remote file ingestion layout. Prepare a design document to guide the parser's architecture.
Week 1 *(May 29 – June 4)*	**Core Parser Development**	Build parsing routines for descriptor headers and entries. Define data structures for bridge metadata. Integrate pattern recognition for line segmentation. Validate parser logic using static sample files.
Week 2 *(June 5 – June 11)*	**Descriptor Version Awareness**	Extend parser to support version-based dispatching using the type identifier. Include graceful fallbacks and logging for unrecognized formats. Refactor file loading and internal struct construction accordingly
Week 3 *(June 12 – June 18)*	**Digest Trait and Hashing**	Create a trait-based digest abstraction for hashing file content and line entries. Compute and store SHA-256 digests for uniqueness. Ensure fingerprint-level deduplication is incorporated cleanly into record logic
Week 4 *(June 19 – June 25)*	**CLI Foundation**	Develop command-line argument handling using structured flags for dry run, local directory, and file limits. Add contextual argument validation and usage guidance. Enable toggling between parsing-only and export paths
Week 5 *(June 26 – July 2)*	**PostgreSQL Integration**	Implement asynchronous insertion logic for parsed bridge entries. Auto-create schema and indexes. Ensure transaction safety and connection pooling. Perform test insertions and validate row integrity

Week 6 *(July 3 – July 9)*	**Export Reliability Checks**	Validate record correctness via test queries. Cross-check record count against file digests. Enable optional clearing of database state for testing. Add basic integrity assertions for schema population
Week 7 *(July 10 – July 16)*	**CSV Exporter Integration**	Build lightweight CSV export layer with configurable output paths. Generate CSVs matching schema layout. Implement unit tests to validate row formatting and special-character escaping.
Week 8 *(July 17 – July 23)*	**Parquet Export Support**	Add feature-gated support for Parquet export. Define Arrow schema matching bridge record model. Implement batch writing and check compatibility with standard analytics platforms
Week 9 *(July 24 – July 30)*	**Offline and Retry Mode**	Implement offline mode for .txt and .gz input folders. Integrate retry logic for remote file fetching. Add file filtering options for large-scale testing
Week 10 *(July 31 – Aug 6)*	**CLI UX Polish and Format Switching**	Refine CLI help, flag consistency, and defaults. Introduce user feedback for unsupported formats or file issues. Ensure consistent formatting across dry-run, CSV, and Parquet paths
Week 11 *(Aug 7 – Aug 13)*	**Testing and Documentation**	Extend test coverage to edge-case files and malformed entries. Document CLI options, internal logic, and developer setup. Prepare test-driven usage example
Week 12 *(Aug 14 – Aug 21)*	**Final Review and Submission**	Finalize README, contributor guidelines, and module comments. Conduct integration tests for full flow. Submit merge-ready crate with complete logs, tests, and CLI interface

Availability and Commitments

I am fully available and committed to dedicating forty to fifty hours per week for the entire duration of the GSoC program. My academic calendar is fully aligned with the GSoC timeline, and I will not have any university obligations or examinations during the coding period. This ensures uninterrupted focus and consistent availability throughout the project timeline. I plan to follow a daily schedule that allows me to work primarily from 10:00 AM to 01:00 AM Indian Standard Time, which corresponds to 04:30 to 19:30 UTC. This range provides

flexibility for sustained deep work, asynchronous collaboration, and timely communication with mentors and contributors across time zones.

To maintain steady progress and ensure effective communication, I will share detailed weekly progress reports summarizing the implementation milestones, encountered issues, and next steps. These updates will be shared via the project's GitLab issues, proposal threads, or other communication channels preferred by the mentor. I will also maintain a structured changelog or development diary alongside my code commits to document implementation decisions and architecture updates in real-time. In case of any blockers or delays, I will notify mentors early and work collaboratively to find practical solutions or revise timelines when needed. This structured workflow will help ensure that deliverables are met on time and that the project remains aligned with expectations throughout the coding period.

Post GSoC Plans

My intention is to remain actively involved with the Tor Project even after the completion of the GSoC program. Over the past few weeks, I have gained meaningful familiarity with the bridge pool assignment infrastructure, descriptor parsing logic, and the direction in which the Tor metrics ecosystem is evolving. Post-GSoC, I plan to continue contributing improvements to the Rust-based parser, extend its support for additional Tor descriptor types such as server descriptors or consensuses, and participate in maintenance efforts for the codebase.

I am particularly interested in helping review and triage merge requests related to bridge parsing, contribute to test coverage and stability of the data pipeline, and support the broader integration of Rust-based tools into Tor's metrics infrastructure. Additionally, I plan to assist new contributors who want to get started with Rust-based projects under the Tor organization. To that end, I will help improve documentation, share technical write-ups or blog posts explaining project internals, and provide mentorship or peer support to future GSoC contributors. This long-term commitment reflects my interest in supporting open-source development and contributing to privacy-enhancing technologies that serve the public good.

Conclusion

The goal of this project is to build a clean, efficient, and extensible Rust implementation that replaces the legacy Java-based metrics-lib for parsing Tor network descriptors. By focusing on the bridge-pool-assignment files and adhering to the official Tor directory specifications, this project will directly support Tor's shift to a modern, scalable, and safe metrics processing pipeline. The implementation will feature version-aware parsing, digest-based deduplication, and multi-format export capabilities, allowing data to be exported to PostgreSQL, CSV, or Parquet. It will also support both online and offline usage modes and offer a structured command-line interface with tracing and error reporting built in.

With this foundation, the project aims not only to achieve its immediate goals within the GSoC timeframe but also to serve as a robust base for future contributions within the Tor ecosystem. By aligning with the project's long-term vision of phasing out Java and promoting secure and maintainable tooling in Rust, this work has the potential to make a lasting impact beyond GSoC. I am enthusiastic about contributing to this important transition and look forward to building a tool that meets the standards and goals set forth by the Tor Project.