

Verification of the ALICE Inner Tracking System readout chain

Bachelor project report 2023F12

Navn	Studienummer	Retning
Marc Beck König	201205935	SW
Supervisor: Jesper Michael Kristensen - jmkr@ece.au.dk		

Date: 31-5-2023

Characters excluding tables and figures: approx. 51500



ALICE

Abstract

The [ALICE Inner Tracking System \(ITS\)](#) team at [CERN](#) has completed commissioning the [ITS2](#) detector, related electronic systems, and data-taking for physics has started. With higher rates and [Pb-Pb collisions](#), increased data rates and radiation effects are expected to challenge the system's stability. The [ALICE ITS](#) upgrade project includes a new complex readout system that interfaces several other complex systems, and debugging the readout chain is challenging. This project consists of a requirement specification, technical analysis, and acceptance test from which a command-line utility named [fastPASTA](#) has been developed as a final product to verify readout protocols and inspect [raw data](#) to debug failures in the readout system. The project is exploratory in assessing the Rust programming language and ecosystem's impact on developing performant and reusable software. Source code and documentation are available at the GitLab repository [\[1\]](#).

Resumé

[ALICE Inner Tracking System \(ITS\)](#) teamet på [CERN](#) har gennemført installation af [ITS2](#) detektoren og relaterede systemer, og fysikeksperimenter er påbegyndt. Med højere frekvens og [Pb-Pb collisions](#), forventes højere datamængder og radioaktivitet at udfordre systems stabilitet. [ALICE ITS](#) upgrade projektet inkluderer et ny komplekst dataudlæsningssystem som er koblet sammen med flere andre komplekse systemer, og fejlfinding af systemerne er udfordrende. Dette projekt består af en kravspecifikation, teknisk analyse, og accepttestspecifikation for et endeligt produkt kaldet [fastPASTA](#). Produktet er et command-line værktøj udviklet til at verificere og inspicere rå data og tilhørende dataudlæsningsprotokoller med henblik på fejlfinding i dataudlæsningssystemet. Projektet har et udforskende aspekt til formål at vurdere effekten af at bruge Rust programmeringssproget og dets økosystem på udviklingen af genanvendeligt software med høj ydeevne. Kildekode og dokumentation er tilgængeligt på projektets Gitlab side [\[1\]](#).

Contents

1	Glossary	iv
2	Acronyms	v
3	Preface	1
4	Introduction	2
4.1	Problem domain	2
4.2	Problem statement	5
4.3	System description	6
5	Requirements	7
5.1	Functional requirements	7
5.2	Non-functional requirements	8
5.3	Delimitation	8
6	Process & methodologies	11
6.1	Methodologies	11
6.2	Process	11
7	Technical analysis	13
7.1	Choice of technology	13
7.2	Software architecture choices	17
8	ALICE ITS readout protocol	19
8.1	CRU protocol	19
8.2	ITS data protocol	20
9	Software architecture	21
10	Software design & implementation	23
10.1	Structure & module interdependence	23
10.2	Parallelism implementation	26
10.3	Tracking ITS payload protocol	29
10.4	Performance	30
11	Testing	34
11.1	Unit testing & code documentation	34
11.2	Integration testing	35
11.3	System tests	35
11.4	Continuous integration & regression testing	37

11.5 Code coverage	37
11.6 Acceptance testing	38
11.7 System benchmarking	40
12 Results	41
13 Discussion	44
13.1 Impact of Rust	44
14 Conclusion	46
15 Future development	47
References	48

1 Glossary

ALICE A Large Ion Collider Experiment. Particle detector in the [Large Hadron Collider \(LHC\)](#). [1](#), [2](#), [4](#), [6](#), [7](#), [11](#), [13](#), [19](#), [24](#), [31](#), [40](#), [44](#), [46](#), [47](#)

ALPIDE Silicon pixel sensors used for particle detection in [ALICE ITS2](#). [4](#), [10](#), [20](#), [47](#)

CERN European Organization for Nuclear Research. [1](#), [2](#), [5](#), [6](#), [12](#), [41](#)

fastPASTA fast Protocol Analysis Scanner Tool for [ALICE](#). [1](#), [2](#), [6–8](#), [11–13](#), [17](#), [19](#), [21](#), [23–29](#), [31](#), [33–35](#), [38–47](#)

Hermetic detector Particle detector designed to observe all particle products from a collision. Made up of multiple subdetectors built in layers around the point of collision. [2](#)

ITS2 Inner Tracking System **2**. The upgraded [ITS](#) installed during [LS2](#). [2](#)

Jira ticket An issue description in the bug tracking software Jira. [5](#)

LLVM Compiler backend toolchain used for optimizations, generating intermediate language representations and/or machine code for a specific platform. [37](#)

LS2 Long Shutdown **2**. The pause for upgrades in the [LHC](#) from Dec. 2018 to Jul. 2022. [2](#)

LS3 Long Shutdown **3**. The pause for upgrades in the [LHC](#) scheduled for Q4 2026 to Q2 2029 as of this writing. [47](#)

Pb-Pb collisions Lead ion collisions in the [LHC](#). [2](#)

Raw data Data from the [ALICE](#) readout system. [2](#), [4](#), [5](#), [8](#), [10](#), [11](#), [21](#), [22](#), [24](#), [28](#), [29](#), [31](#), [38](#), [39](#), [41](#), [42](#), [44](#), [46](#)

Sanity check Broad and shallow test. Sanity checks in data protocols test if the fields of a data word by itself contain valid values and do not compare with, e.g., the data word preceding or succeeding it. [10](#), [41](#), [44](#)

SEU Single Event Upset. Bit flips caused by charged particles colliding with digital electronics. [5](#)

2 Acronyms

CDP [Common Readout Unit \(CRU\)](#) Data Packet [7](#), [10](#), [19](#), [20](#), [25](#), [29](#), [41](#)

CI Continuous Integration [12](#), [37](#), [46](#)

CLI Command-Line Interface [8](#), [47](#)

CRU Common Readout Unit [4](#), [5](#), [7](#), [8](#), [10](#), [19](#), [39](#), [41](#), [44](#), [47](#)

CWD Current Working Directory [39](#)

DAQ Data Acquisition system [4](#), [5](#)

DDW Diagnostic Data Word [20](#)

FLP First Level Processor [4](#), [19](#)

FPGA Field-Programmable Gate Array [5](#)

FSM Finite State Machine [11](#), [30](#)

GBT Gigabit Transceiver [4](#), [8](#), [10](#), [25](#), [29](#), [39](#), [41](#), [43](#), [44](#)

HBF Heart Beat Frame [10](#), [19](#), [20](#), [39](#), [41](#)

IHW ITS Header Word [20](#)

ITS Inner Tracking System [1–4](#), [7](#), [8](#), [10](#), [19](#), [20](#), [29](#), [30](#), [40](#), [41](#), [44](#), [47](#)

LHC Large Hadron Collider [2](#), [44](#)

MiB Mebibyte [31](#)

MTBF Mean Time Between Failure [5](#)

MVP Minimum Viable Product [8–12](#), [42](#), [44](#)

NSA National Security Agency [15](#)

OOP Object-Oriented Programming [14](#), [15](#), [24](#)

PDF Probability Density Function [32](#), [33](#)

RDH Raw Data Header [7](#), [10](#), [19](#), [24–26](#), [29](#), [31–33](#), [39](#), [41](#), [43](#), [44](#), [47](#)

RHEL Red Hat Enterprise Linux [8](#), [42](#)

RU Readout Unit [4](#), [5](#), [10](#), [39](#), [41](#)

SME Subject Matter Expert [6](#), [7](#), [44](#)

SRS Software Requirement Specification [7](#), [8](#), [12](#)

TDH Trigger Data Header [20](#), [38](#)

TDT Trigger Data Trailer [20](#)

TMP Template Metaprogramming [14](#), [15](#)

UL User Logic [10](#), [41](#)

UML Unified Modeling Language [11](#)

3 Preface

This bachelor report has been produced to fulfill the requirements of the 20 ECTS course **I7BAC-01 Bachelorprojekt**, in the software technology major, a Bachelor of Engineering program at the Department of Electrical and Computer Engineering at Aarhus University.

Some background knowledge in software engineering is assumed of the reader so that some of the used terms are considered known and not further explained.

Through an internship in the spring of 2022, I established contact with staff at [CERN ALICE Inner Tracking System \(ITS\)](#), and the prospect of future collaboration was discussed. After going through a selection process to receive a contract at [CERN](#), I received an offer for a project titled «Verification of the ALICE Inner Tracking System readout chain», the development of [fastPASTA](#) was proposed as the target for exploring the impact of using Rust in a development process, and as the subject of my bachelor project.

The project has been supervised by Jesper Michael Kristensen from the Department of Electrical and Computer Engineering.

I want to express my gratitude to my [CERN](#) supervisor Ola Slettevoll Grøttvik, for supervising me, especially in the technical aspects of the project. I would also like to thank Jesper Michael Kristensen for his help supervising my project from the side of Aarhus University.

4 Introduction

This report details the exploratory project of developing the command-line utility nicknamed [fastPASTA](#) using the Rust programming language and ecosystem. Part of the preparation involved studying Rust literature [2–10].

The [ALICE ITS](#) team proposed a tool to be designed for engineers and physicists in the [ALICE](#) community and at [CERN](#) for inspecting, filtering, and verifying the protocol correctness of the [raw data](#) that is produced by the [ALICE](#) detector and its readout system during physics experiments in the [Large Hadron Collider \(LHC\)](#). In this project, [fastPASTA](#) is developed to include verification of the [ALICE](#) subdetector [ITS](#)'s readout protocol but with the intent to support the implementation of verification of additional [ALICE](#) subsystems. As of this writing, the [ITS](#) consists of the [ITS2](#) installed during the [LS2](#) but will hereafter be referred to as [ITS](#).

[CERN](#) is an intergovernmental organization that operates the largest particle physics laboratory in the world [11]. [CERN](#) features the [LHC](#), the largest particle accelerator in the world, which features four large detectors, one of which is [ALICE](#). [ALICE](#)'s central detector is the [ITS](#), a silicon pixel detector system with a long and complex chain of readout subsystems. Maintaining and improving the stability of the [ITS](#) is a large and challenging task.

4.1 Problem domain

The [ALICE](#) detector is dedicated to the heavy-ion physics in the [LHC](#) during [Pb-Pb collisions](#), see Figure 1. The detector is 26 meters long, 16 meters in height and width, weighs 10000 tons, and is located 100 meters underground. It is a [hermetic detector](#), also known as an «onion detector,» as the structure of layers of subdetectors resembles an onion.

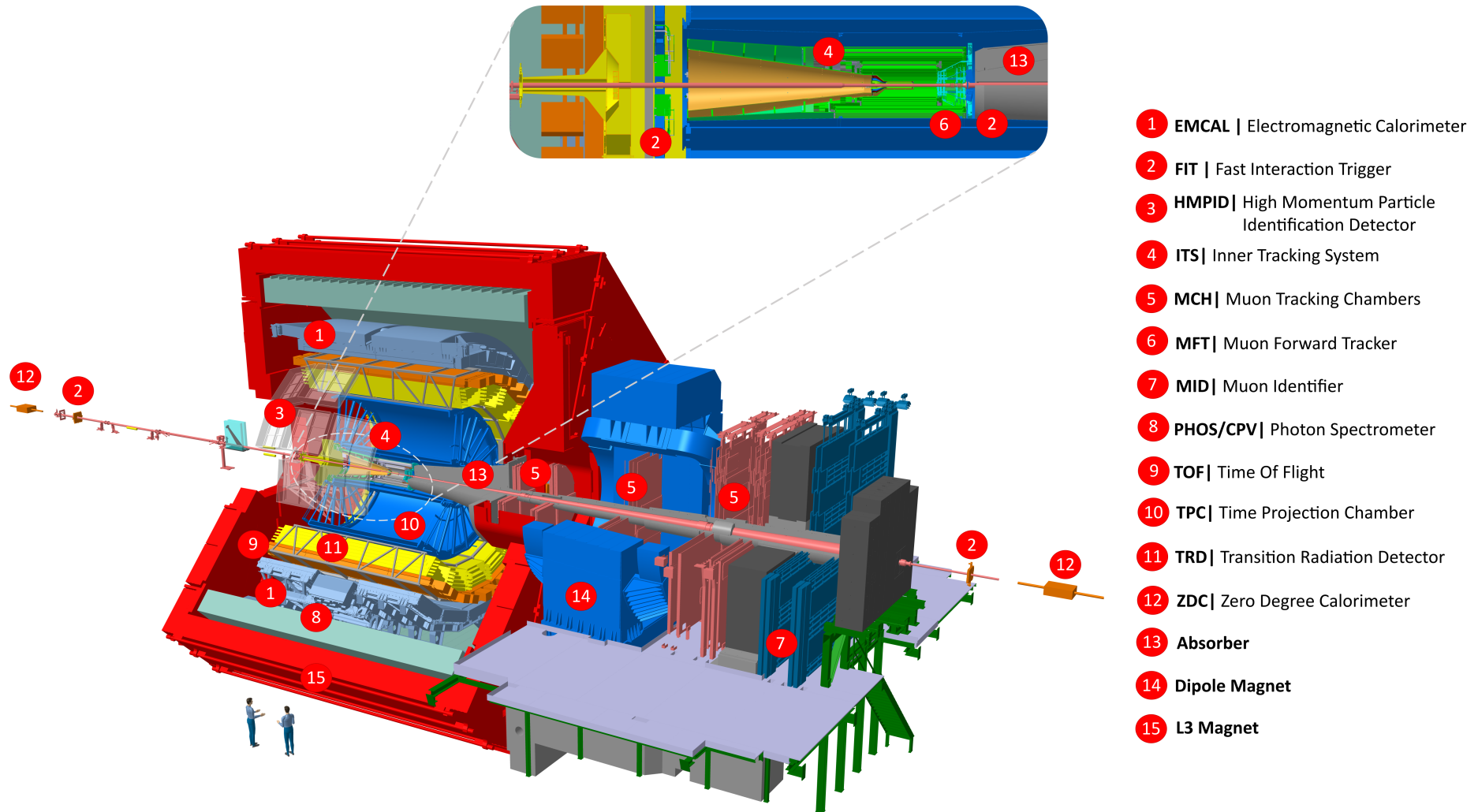


Figure 1: 3D model of the ALICE detector. Notice label 4, the inner-most subdetector *ITS* built around the collision point, and how subdetectors are arranged in concentric layers around it [12, p. 12].

Each subdetector has custom-built sensors with each their own data protocol(s), describing the semantics of the data they collect each time a particle collision occurs. Still, all subdetectors share a common point in the [Data Acquisition system \(DAQ\)](#), the [Common Readout Unit \(CRU\)](#). In the [CRU](#), the [raw data](#) is wrapped in the [CRU](#)'s custom protocol before being transmitted to the [First Level Processor \(FLP\)](#), the next of many links in the readout chain.

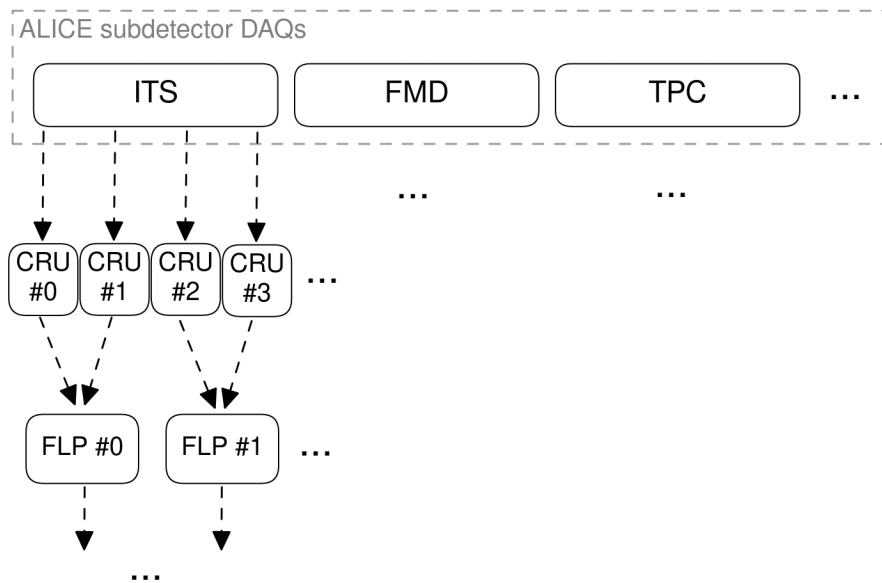


Figure 2: *Sketch of the composition of the [ALICE](#) readout chain. Each subdetector has a separate way of reading out data from their sensors, but from the [CRU](#) link, all data is packaged in common protocols.*

The concept of the [ALICE](#) readout chain is illustrated in Figure 2. Each unit of a link in the readout chain processes data from multiple units in the link before it. The [ITS](#) subdetector's [DAQ](#) is illustrated in Figure 3. In the [ITS](#)' [DAQ](#), data from a particle collision is first read from approximately 24000 [ALPIDE](#) pixel sensors via 192 [Readout Units \(RUs\)](#), those 192 [RUs](#) transmit repackaged data through 3 [Gigabit Transceiver \(GBT\)](#) links to [CRUs](#). Each [CRU](#) receives data from 5-12 [RUs](#)¹ before transmitting repackaged data to one [FLP](#) per 2 [CRUs](#).

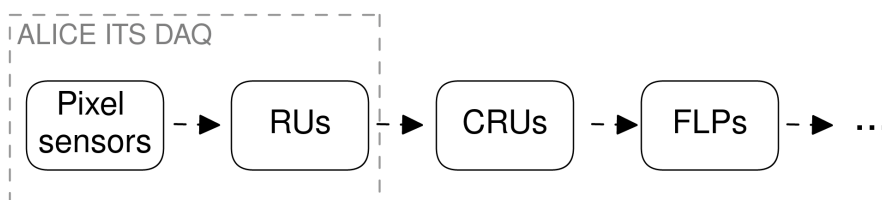


Figure 3: *Simplified sketch of the [ITS](#) readout chain.*

¹The number of [RUs](#) connected to a [CRU](#) depends on how much data they transmit, which is mainly determined by how close the pixel sensors that the [RUs](#) read from are to the collision point.

Each subdetectors [DAQ](#) operates in a highly irradiated and magnetic environment, which causes high rates of [single event upsets](#), and substantial wear and tear on the electronics. One consequence of the hostile environment is a high failure rate among all the subsystems; for instance, the [Mean Time Between Failure \(MTBF\)](#) on [RUs](#) is approximately 19 hours, while experiments often run for weeks. Various techniques are used to mitigate the effects of radiation on digital electronics, such as using radiation hardened [Field-Programmable Gate Arrays \(FPGAs\)](#) and digital designs modeled to be more resistant to radiation.

Each subdetector is continuously upgraded and maintained by a team of engineers and physicists. They share a large part of the [DAQ](#), but each subdetector is a large and complex system, so coordination is required, especially when issues arise. The [CRU](#) is where all subdetectors' [DAQs](#) converge, meaning the [CRU](#) has to interoperate with various custom protocols, and this is where issues typically appear. The [CRUs](#) are not deployed in a highly irradiated environment, but all the systems it receives data from are, and tracking down errors first uncovered at the [CRU](#) is a long and involved process.

4.2 Problem statement

Problem 1

Engineers from different teams are regularly tasked with fixing detected patterns of errors in the [raw data](#) output or providing hard evidence that the subsystem of their concern is not the source of the errors.

There is a need for a user-friendly and performant command-line utility to analyze the [raw data](#) output, report errors in the data, and allow for viewing the data in a human-readable format.

Background

When a pattern of errors has been identified in the [raw data](#), a [Jira ticket](#) is created where all the maintainers of the subsystems that potentially are the source of the issue are tagged. Maintainers are typically not part of the same team. The workflow creates a blame game where maintainers start reviewing each other's code and pointing out potential sources of errors. Issues typically go months before a fix is deployed.

Problem 2

Software at [CERN](#) is hard to reuse, and there are few good options if performance and a fast development process are requirements.

Members of ALICE have requested an assessment of the Rust programming language and

its ecosystem's potential for streamlined development of reusable and performant software tools.

Background

New software tools are continuously developed at [CERN](#), reusing older software if possible. Producing maintainable and performant software is challenging, and the significant employee turnover rate at [CERN](#) makes it that much harder.

4.3 System description

It is a clear expectation from the [Subject Matter Experts \(SMEs\)](#) that maintain the [ALICE](#) readout system that the developed system [fastPASTA](#) is a *command-line utility*, usable from a terminal such as Bash [\[13\]](#) with the same flexibility as other standard tools like `grep` [\[14\]](#) or `less` [\[15\]](#) in terms of compatibility with Unix pipelines [\[16\]](#). The system will be developed and released as a finished product.

5 Requirements

This chapter presents the result of requirement elicitation, and requirements are prioritized. See Section 6.2.2 for details on the structure of the requirement elicitation process. The stakeholders of [fastPASTA](#) are [SMEs](#) within the [ALICE](#) team; this is reflected in the domain-specific and technical nature of the functional requirements in particular. Interviews with stakeholders were conducted as part of the requirement elicitation process, and each requirement reflect direct statements from a stakeholder about which features would bring value to them.

The requirements are divided into functional and non-functional categories, with the non-functional requirements further subdivided into five categories, as seen below.

- Functional
- Non-functional
 - Interface
 - Performance
 - Design constraints
 - Non-functional attributes
 - Preliminary schedule

5.1 Functional requirements

The requirement description summarizes what a stakeholder wants to achieve. An example of 3 functional requirements can be seen in table 1 below. Further technical details are provided in the [Software Requirement Specification \(SRS\)](#) [17] and on the *issues* page of the Gitlab repository [18].

Table 1: *Example of functional requirements.*

Requirement description (user can..)
Generate a human-readable overview of Raw Data Headers (RDHs)
Validate that the CRU Data Packet (CDP) payload is following the ITS payload protocol
Parse ITS payload protocol from a calibration run

5.2 Non-functional requirements

The non-functional requirements are divided into five categories: *Interface requirements*, *performance requirements*, *design constraints*, *non-functional attributes*, and *preliminary schedule*.

An example of these requirements can be seen below as a list of interface and performance requirements. For the full description of non-functional requirements, see the [SRS \[17\]](#).

5.2.1 Interface requirements

Users interact with [fastPASTA](#) through the command line, with the associated requirements:

- Accepts as input a [raw data](#) file or stream via *standard input* structured according to the [ITS](#) data format [19], and the «RDH-CRU»-section in *ALICE RUN 3 Raw Data Header* [20].
- Adhere to CLI design guidelines [21].
- [fastPASTA](#) is cross-platform compatible. Supporting Windows 10/11, and [Red Hat Enterprise Linux \(RHEL\)](#) distributions.

5.2.2 Performance requirements

In the interest of producing a performant [fastPASTA](#), but with the caveat that specifying exact performance metrics is infeasible, the following heuristics instead serve to communicate performance requirements:

- Process 1 GB 20 times faster than *decode.py*.²
- Processing 1 GB is convenient on a laptop.

5.3 Delimitation

A [Minimum Viable Product \(MVP\)](#) is described in requirements prioritization. The [MVP](#) is a product version that incorporates as few features as possible for fast development while still providing value for users and allowing for early user feedback to guide the continuous development efforts. This [MVP](#) would allow a user to use [fastPASTA](#) through a [Command-Line Interface \(CLI\)](#) to identify and locate errors in the [CRU](#) protocol, filter out data from a specific [GBT](#) link in [raw data](#) that contains a mix of readout data from multiple [GBT](#) links, and save it to a file and do verification on the filtered data.

²*decode.py* is a legacy tool used internally for performing several basic protocol checks on ITS Readout data.

5.3.1 Prioritization of requirements

A list of the functional requirements and their prioritization can be seen in Table 2. The phrase *can parse [data description]* indicates a requirement that the described data is supported by all features the product implements.

Each requirement is prefixed with a 1-5 priority, e.g., highest priority requirements are prefixed with **P1**. This prioritization follows the *MoSCoW method* [22] but expands the *Must have*-category into priority 1 & 2, where priority 1 is most urgently needed and, therefore, should be incorporated in the first version of the product (the **MVP**), and priority 2 must be included in the released product but is not strictly needed for early users to get value from the product, and start providing feedback. The prioritization in relation to the *MoSCoW method* should be understood as follows:

1. **MVP**
2. Must have
3. Should have
4. Could have
5. Won't have

Table 2: *Functional requirements. The requirements are numbered with their priority and the order of appearance, separated by punctuation.*

#	Requirement description (user can..)
MVP	
1.1	Parse a CDP (RDH + payload)
1.2	Parse a Heart Beat Frame (HBF) with multiple CDPs
1.3	Parse a CDP payload formatted in both User Logic (UL) flavor 0 and 1
1.4	Validate that the data from each GBT link (RU output/ CRU input) is in a HBF pattern
1.5	Filter data from individual GBT links
1.6	Validate all individual 80-bit GBT words with sanity checks , e.g. that all GBT words have a valid identifier field
1.7	Validate CDPs with a sanity check establishing that the CDP 's structure is coherent as per the CRU protocol
Must have	
2.1	Parse raw data that includes data from an arbitrary number of different GBT links
2.2	Generate a human-readable overview of RDHs
Should have	
3.1	Validate that the CDP payload is following the ITS payload protocol
3.2	Parse ITS payload protocol from a calibration run
3.3	Generate a human-readable overview of HBFs
3.4	See a summary of detected attributes in the parsed data, including: RDH version, trigger type, data format, number of HBFs , and RDHs
3.5	See a summary for parsed ITS payloads, of which layers and staves the data originated from
Could have	
4.1	Filter data from individual ALPIDE chips
4.2	Do advanced ITS protocol checking in split HBFs (multiple CDPs)
Won't have	
5.1	Validate ALPIDE payload protocol
5.2	Convert UL flavor 0 to flavor 1, and vice versa
5.3	Validate diagnostic flags in an HBF , in the case where event data has been dropped, observed in the ALPIDE protocol by an unexpected skip in the pixel address

6 Process & methodologies

This chapter explains processes and methodologies used in the project.

6.1 Methodologies

This section details the methodologies used in developing [fastPASTA](#).

6.1.1 Unified modeling language

[Unified Modeling Language \(UML\)](#) is a general-purpose modeling language used to visualize a system's design. [UML](#) is used as a visual aid to give an overview of the control flow of the system, abstracting away implementation details. [UML](#) is also used to visualize [Finite State Machines \(FSMs\)](#) and document the design implementation's structure. [UML](#) versions include strict rules about notation such that there is a 1 to 1 correspondence between diagrams and implementations; that is not how [UML](#) is used in this project.

6.1.2 Semantic versioning

Semantic versioning is a versioning scheme that describes how to assign a version to software in the form of X.Y.Z. [\[23\]](#). Semantic versioning is used to assign a version with each release of [fastPASTA](#).

6.2 Process

This section contains descriptions of which processes were applied during this project.

6.2.1 Planning

It was important for the [ALICE](#) team that a product was delivered as soon as possible and that it provided value to ongoing debugging efforts. Because of the urgency, the process overhead was required to be kept as small as possible; this led to the agreement on two weeks of dedicated requirement elicitation, followed by four weeks to deliver a usable [MVP](#) and then to develop new features based on prioritization. Acceptance testing was planned to be carried out at least two weeks before the project report hand-in deadline, with the remaining weeks dedicated to report writing.

6.2.2 Requirement elicitation

The requirement elicitation process involved interviews with users about how they approach debugging [raw data](#) from the ALICE readout system with their current tools and where

these tools are insufficient. It also involved brainstorming sessions with users about which features could provide value in debugging.

The product of interviews and brainstorming sessions was a [SRS](#) document. The method for prioritization of requirements was chosen in terms of what would provide value right now taking precedence over requirements that would provide value in the long term, and an agreement was made to work towards a [MVP](#) that could be deployed, which would also allow fast user-feedback, for the ongoing development.

Requirements were reevaluated weekly during meetings with my [CERN](#) supervisor, who is also a user of [fastPASTA](#); this often included changes in the prioritization of requirements and user feedback on current features and new feature requests. New feature requests and bug reports were made through Gitlab issues [\[24\]](#). GitLab milestones [\[25\]](#) were used to group issues into a release goal, such as completing the [MVP](#).

6.2.3 Git & Gitlab

Gitlab is a software package used to develop and operate software. Git is a distributed version control system; Git was used with a Gitlab instance hosted by [CERN](#). Git branching was used to add features to the codebase by developing on a Git branch dedicated to a feature and then opening a Gitlab *merge request* to merge the feature branch with the main branch. Git tagging was used to annotate releases and their versions. Pushing a Git tag to the main branch also triggered a Gitlab pipeline to build a release version of [fastPASTA](#) which was made available for download via Gitlabs release pages [\[26\]](#). Additionally, each version was published on *crates.io* [\[27\]](#), and documentation was published on *docs.rs* [\[28\]](#).

6.2.3.1 Continuous integration

[Continuous Integration \(CI\)](#) is the process of automatically building and testing code as it is committed to a code base. Gitlab's [CI](#) pipelines were used to protect the codebase from regression. A merge request had to pass a pipeline that included checks on code formatting, security auditing of dependencies, code adherence to best practices, unit tests, test coverage, leak & address sanitizing, and the presence of documentation before it could be merged with the main branch.

7 Technical analysis

This chapter contains an analysis of choices regarding programming language and software architecture for implementing [fastPASTA](#).

7.1 Choice of technology

The [ALICE](#) team requested an assessment of Rust's impact on the development cycle; therefore, a comparison has to be made with the alternatives available for developing a tool like [fastPASTA](#). In this section, the impact of the choice of programming language for [fastPASTA](#) is evaluated concerning non-functional requirements and ease of development.

Performance is the most critical aspect of [fastPASTA](#), so the choice of programming language is also vital. First, the language should be statically typed for compile-time optimizations and minimal runtime overhead. Many compiled languages also provide robustness through type safety and static analysis. For performance-critical software, looking at which systems programming languages are available is obvious. In the interest of a fast development cycle, it should be evaluated how difficult incorporating and maintaining external dependencies is, as well as managing the development toolchain. As [fastPASTA](#) is intended to be utilized and expanded for years to come, it is also essential to choose a language that is not niche and preferably one that is likely to remain popular in the foreseeable future. These requirements can be summarized as follows:

- High Performance
- Statically typed
- Robust through static analysis
- Easy to include external dependencies
- Popular
- Long-term viable

Three popular and prominent candidates are considered here, namely C++, Rust, and Go. See [Figure 4](#) for one measure of the popularity over time of these three languages.

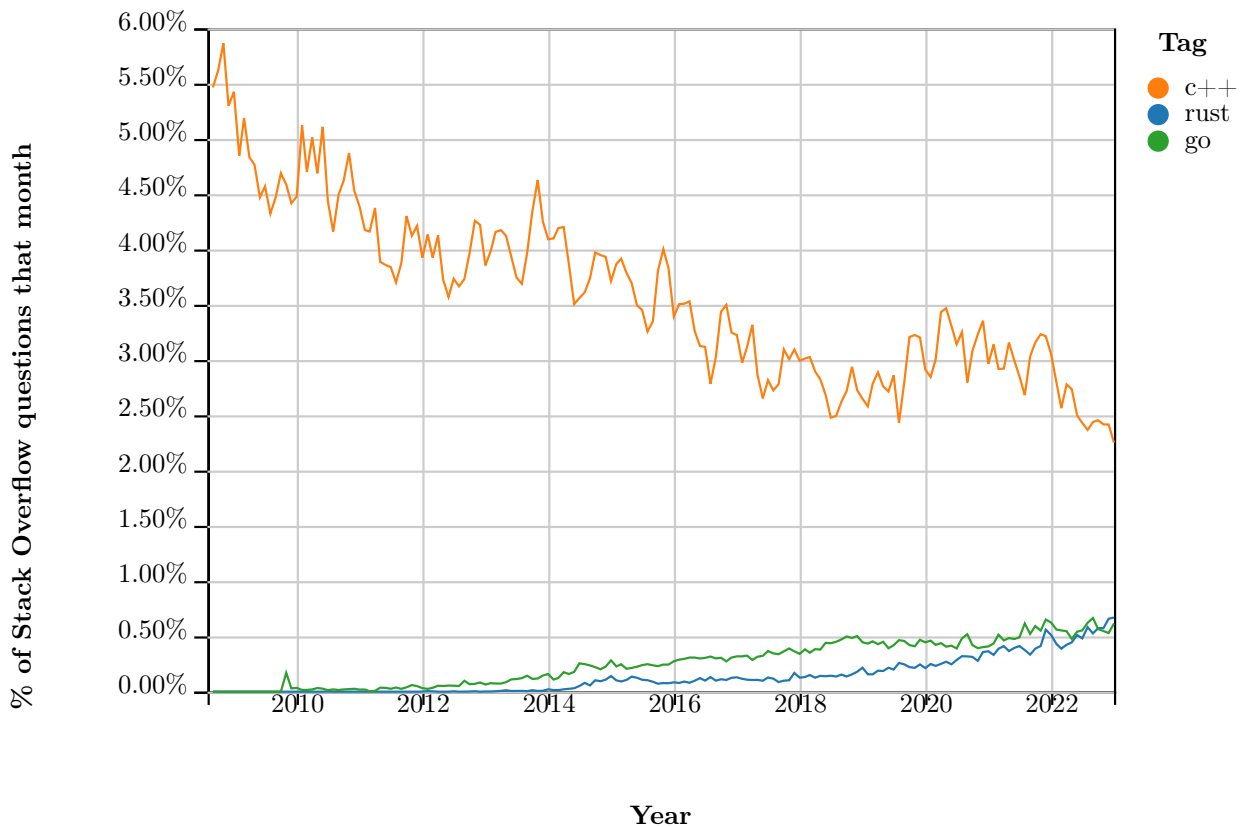


Figure 4: *Percentage of questions asked on stackoverflow.com about Rust, Go or C++ since 2008 [29].*

7.1.1 C++

Advantages

C++ is a widely used language in high-performance computing, real-time systems, and other contexts where performance is critical. C++ is a widely adopted language; thus, a future developer likely already has some background knowledge. C++ offers plenty of flexibility with extensive support for [Object-Oriented Programming \(OOP\)](#) but also supports many popular functional programming idioms, such as *type polymorphism*, *pattern matching*, *higher-order functions*, and more. Through generics and [Template Metaprogramming \(TMP\)](#), there is also potential for high reusability, but at no cost to performance, thanks to static polymorphism. A mature standard library and the high-quality open-source Boost library [30] offer many highly-optimized implementations of various algorithms, data structures, and more. C++ also benefits from mature and modern compilers, such as Clang and GCC, which are fast and efficient in producing optimized machine code.

Disadvantages

The cost of C++ is a complex process of setting up a toolchain for compiling and linking source code. A lack of a mature package manager for C++ makes it difficult to reuse

any open-source libraries that offer the functionality one might need and even harder to keep external dependencies up to date. There is a high complexity and readability cost to using [TMP](#), a feature rarely used in industry software. C++ also has a long list of known issues [\[31\]](#) that are hard to track, such as various functionality in the standard library that is inefficient (e.g., `std::regex`) but is kept there in the name of backward compatibility. The community is split between different C++ standards, and the language's popularity is dwindling (see [Figure 4](#)) as viable competing languages appear. In November 2022, the [National Security Agency \(NSA\)](#) published a report advising organizations to «consider making a strategic shift from programming languages that provide little or no inherent memory protection, such as C/C++, to a memory-safe language when possible.» [\[32\]](#) and named Rust and Go among memory-safe alternatives, which puts into question the long-term viability of C++ for new projects.

7.1.2 Rust

Advantages

Rust is a fast-growing programming language, and in 2022 it was voted the most-loved programming language for the 7th year in a row [\[33\]](#); later the same year, Rust officially became the third language supported in the Linux kernel, joining C and Assembly [\[34\]](#). Rust supports both [OOP](#) and functional programming styles but lends itself more to functional style programming, e.g., does not support inheritance. By rejecting some of the performance-costly features of [OOP](#), Rust has common cases where it produces more efficient code than the C++ equivalent [\[35; 3, pp. 21–22\]](#). Microsoft is rewriting core libraries in Rust and mandating that new projects use Rust rather than C++ [\[36\]](#). Rust standardizes many aspects of its use; there is a single package manager, which also facilitates the use of its toolchain for compiling, testing, and benchmarking source code, as well as enforcing *idiomatic Rust* through a formatting plugin so that a single coding style is used across a given codebase [\[2, pp. 511–512\]](#). There is a single website where all open-source Rust libraries are published and shared [\[27\]](#). The Rust compiler contains the *borrow checker* fixture that enforces strict rules for shared memory and ownership at compile-time, enabling further optimizations [\[3, pp. 10–11\]](#) while mitigating much of the headaches of memory management and concurrent programming inherent to languages like C++.

Disadvantages

Rust's standard library is far from as mature as C++, and features are less stable than C++, which means that care must be taken when choosing to include external dependencies in a project and that some features that the C++ standard library has, are not yet stable in Rust's standard library. The Rust compiler is known to be slow and has a long way to go before it is as mature as the C++ compilers are. Rust's compiler is also

far from producing as efficient machine code as Clang or GCC; this is well-known to the community [37], and the gap will likely close over time.

7.1.3 Go

Advantages

Go is a statically typed, procedural language built for concurrency and scalability. It is easy to learn and has high performance and speedy compilation times. Docker and Kubernetes are examples of where Go has been applied with significant success.

Disadvantages

Go is often compared to Rust, and in that respect, it has been criticized for the lack of a streamlined toolchain and package manager, the difficulty of documenting libraries, and up until recently [38], its lack of support for generics [39] which is still limited. Go is a garbage-collected language, which has the advantage of simplifying memory management but takes a big hit on performance when compared to manual memory-managed languages such as Rust and C++, in 2020 the company *Discord* shared their experience of rewriting critical software from Go to Rust out of necessity [40].

7.1.4 Conclusion

Based on the analysis in the previous sections, each language is evaluated in the table below on its ability to fulfill the requirements listed at the beginning of Section 7.1. Color-coded in green/yellow/red, signifying a clear, somewhat, or missing ability to meet a requirement.

Table 3: Development requirements and the ability for C++, Rust, and Go to fulfill them. Symbol- and color-coded signifying a **clear ✓**, **somewhat -**, or **missing ✗** ability to meet a requirement.

Requirement	C++	Rust	Go
High Performance	✓	✓	✓
Statically typed	✓	✓	✓
Robust through static analysis	-	✓	-
Easy to include external dependencies	✗	✓	✗
Popular	✓	✓	✓
Long-term viable	-	✓	✓

7.2 Software architecture choices

It is a priority that [fastPASTA](#) is developed to support reusability (see **Problem 2** in Section 4.2). Still, the development overhead must also be limited to support the fast delivery of a usable product.

A monolithic architecture has low overhead but high coupling challenges reusability. A micro-services architecture is highly reusable but at the cost of high operational complexity and development overhead. The modular monolith [41] is chosen as the architecture for [fastPASTA](#) as a compromise between the two architectures mentioned above. A modular monolith enables low coupling similar to a micro-services architecture by dividing code into logical modules that expose a clearly defined interface but avoid the significant development overhead of developing multiple independent systems.

7.2.1 Defining logical modules

There are three fundamental operations that [fastPASTA](#) has to support:

1. Read input data
2. Analyze data
3. Write data out

These operations can be considered a producer-consumer problem or as filters in a pipeline; they operate independently, but each filter depends on data produced by the filter before it. The filters can be subdivided into smaller filters, which can be swapped in and out depending on the operation the user wants to execute, such as *viewing* or *validating* data.

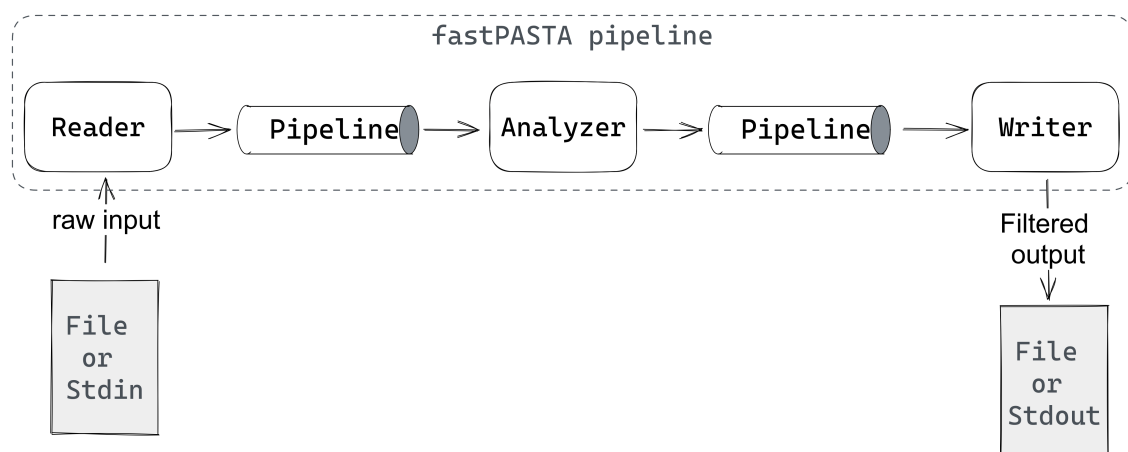


Figure 5: *Simplified sketch of the [fastPASTA](#) pipeline. The Analyzer module is a substitute for either data verification or view generation.*

Besides data processing, [fastPASTA](#) also has to facilitate an interface for the user to configure the operations they want to execute and a way for these configurations to affect

all the filters in the desired manner. These goals can be met by a module containing a configuration and defining how the user sets the configuration, which can then be passed to filters that adjust their operations based on the configuration.

The pipeline should be decoupled from the specific protocols to support adding new protocols to be validated in the pipeline. The data words of a protocol should be defined in a dedicated module. In another module, verification steps can be implemented that accept a specific protocol word, perform verification, and log informative error messages when validation fails.

Finally, to fulfill requirement 3.4 (see table 2), a module is needed to collect attributes of the parsed data and communicate it to the user.

8 ALICE ITS readout protocol

The [ALICE ITS](#) readout protocol is detailed in this chapter, giving an overview of the structure without documenting the numerous edge cases described by each subprotocol. Section [4.1](#) explains the physical readout system in more detail.

8.1 CRU protocol

Recall from Section [4.1](#) that the readout system of each subdetector in [ALICE](#) converges in the [CRU](#). The [CRU](#) protocol describes a [CDP](#), the packets the [CRU](#) receives from a subdetector's readout system. A [CDP](#) contains one [RDH](#) and a payload (can be empty). The [RDH](#) is a 64-byte sized header³ with the minimal required information to process data from a subdetector. [CDPs](#) are grouped into [HBFs](#), and a [HBF](#) consists of two or more [CDPs](#), Figure [6](#) illustrates a [HBF](#).

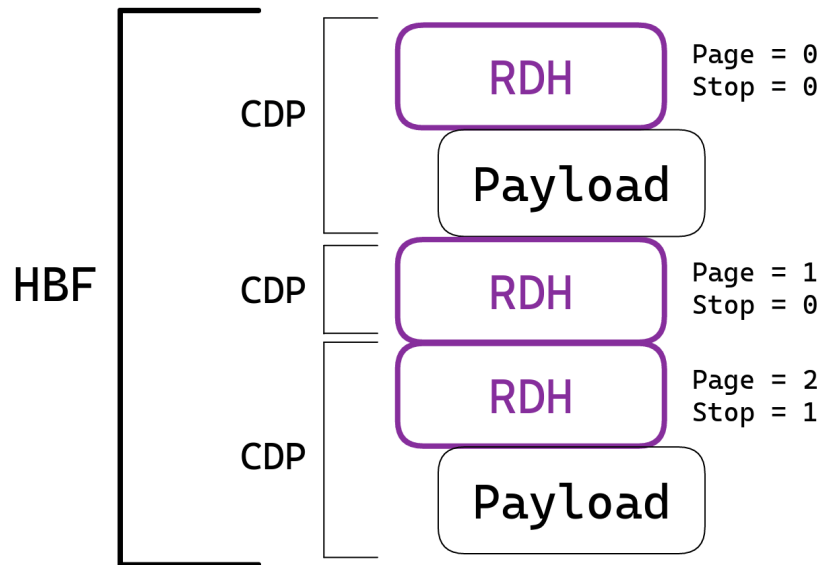


Figure 6: Sketch of a [HBF](#) containing three [CDPs](#). Notice that payloads can be empty, which is the case with the 2nd [CDP](#) here.

The [CDPs](#) in a [HBF](#) are characterized by the first [CDP](#) containing a [RDH](#) where the stop- and page field is zero, followed by zero or more [CDPs](#) with [RDHs](#) with non-zero page field, and the last [CDP](#) having a [RDH](#) with a stop field set to 1, indicating the end of a [HBF](#). The [RDH](#) includes fields that describe, among other things, which subdetector the payload is from, the data format, the packet count, and whether the following payload is closing a packet. The packet-closing [CDP](#) payload typically contains diagnostic data from the subdetector system.

³The [RDH](#) varies in size depending on the implementation details of the subdetector. Still, once it is transmitted from the [CRU](#) to the [FLP](#), it is always 64 bytes, and [fastPASTA](#) works on data received at this level.

8.2 ITS data protocol

Within a [HBF](#), the [ITS](#) data protocol describes the payload from the [ITS](#). The [ITS](#) data protocol describes 4 *status words* identified through their ID field. The status words contain information about the configuration of the [ITS](#), diagnostic data, and metadata.

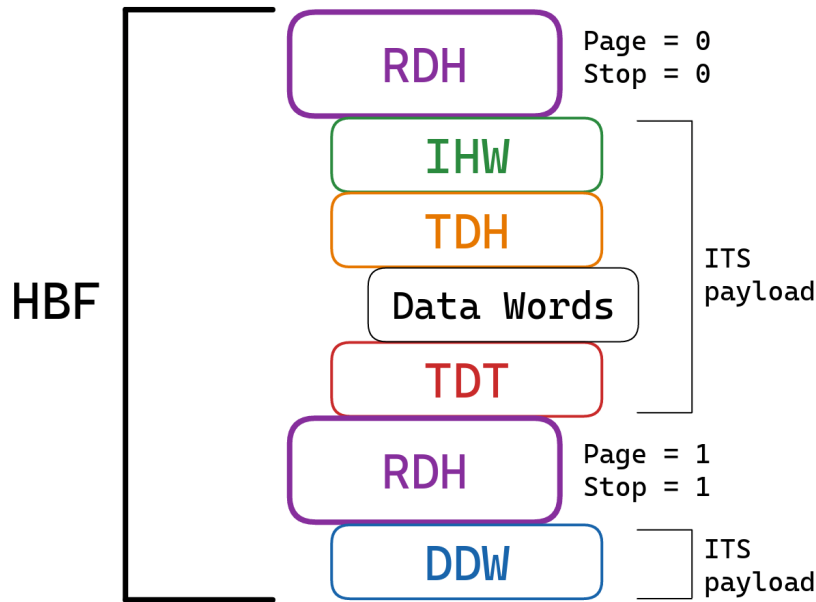


Figure 7: Sketch of a [HBF](#) containing two [CDPs](#) with [ITS](#) payloads.

Figure 7 illustrates an example of a [HBF](#) with two [CDPs](#) with [ITS](#) payloads. The first status word in a [HBF](#) is the [ITS](#) Header Word ([IHW](#)) which contains information about the [ITS](#) configuration (which parts of the [ITS](#) are eligible for readout). The second status word is the [Trigger Data Header](#) ([TDH](#)), followed by the [Trigger Data Trailer](#) ([TDT](#)), which indicates the start and end of a *readout frame* consisting of *data words* containing sensor data. Data words contain sensor data from the [ALPIDE](#) pixel sensors in the [ITS](#). Unlike status words, data words do not have a fixed ID field. Instead, their ID corresponds to a location in the [ITS](#). The last [CDP](#) of a [HBF](#) contains the status word [Diagnostic Data Word](#) ([DDW](#)), which includes diagnostic data about the [ITS](#).

9 Software architecture

Based on the analysis detailed in Section 7.2, the chosen software architecture of [fastPASTA](#) is a *modular monolith* structured as a tree of modules and submodules. The top modules and short descriptions can be seen in Figure 8.

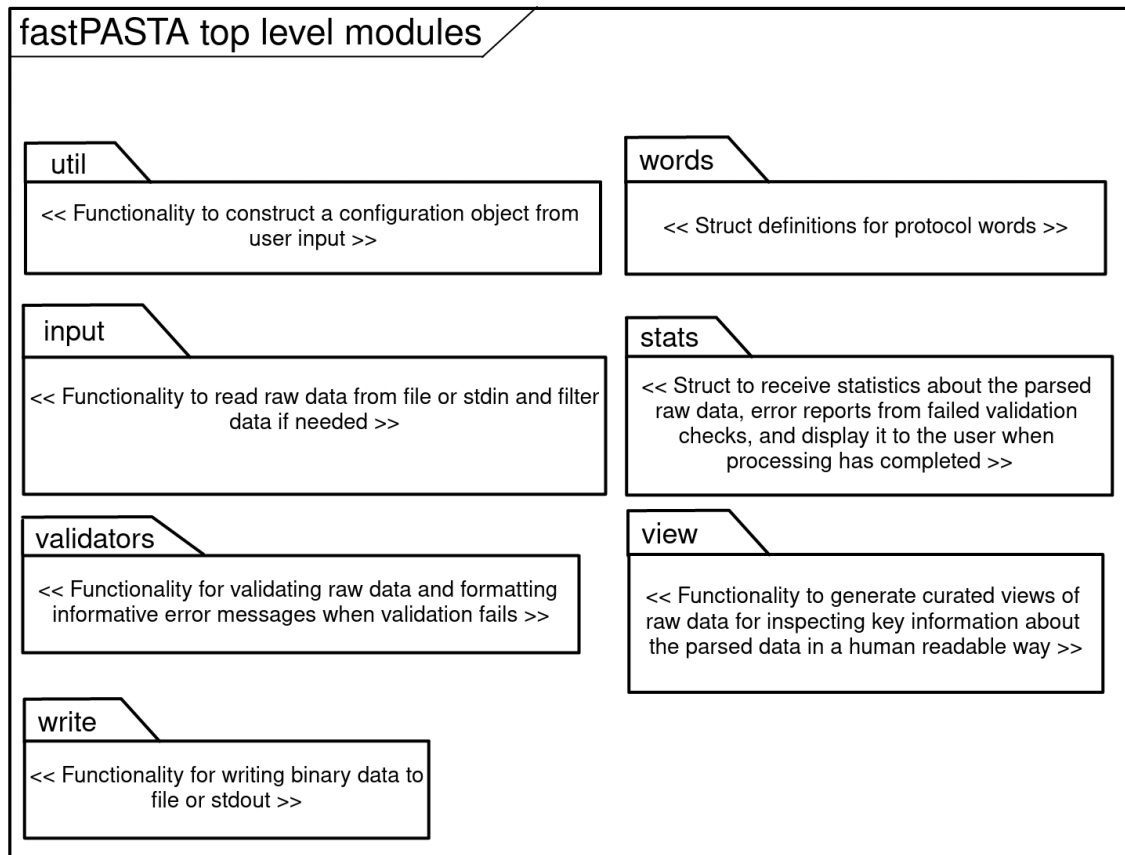


Figure 8: [fastPASTA](#) top level modules. The data processing pipeline starts with the *input* module reading in data, then passing it to submodules of either the *validators* or the *view* module. If the user is *filtering* data then the *write* module is used to write the data to a file or to *stdout*.

The more technical responsibility of each of the top modules is detailed in Figure 8, but can be described in more broad terms like this:

- **util** Parse and validate user input to set a configuration.
- **words** Define protocols that describe the [raw data](#).
- **input** Read [raw data](#).
- **write** Write [raw data](#).
- **stats** Collect information about the [raw data](#) and present it to the user.
- **validators** Define rules for and perform verification of [raw data](#).

- **view** Present [raw data](#) in a human-readable way.

10 Software design & implementation

In this chapter, the software design and implementation are described and justified. [fastPASTA](#) has two main modes of use: *validation* and *view mode*. Examples in this chapter use *validation mode*, but these modes take up the same spot in the pipeline and are interchangeable for all intents and purposes.

10.1 Structure & module interdependence

This section describes how relationships between modules are implemented to achieve low coupling and how it facilitates parallelism.

10.1.1 Inversion of control

The top-level modules of [fastPASTA](#) (see Chapter 9 for an introduction to the top-level modules) are loosely coupled using the dependency inversion principle to only have an interdependence through Rust *Traits* or *enums*. Traits are like interfaces but can be used without relying on polymorphism and dynamic dispatch⁴, making them highly performant compared to interfaces or abstract classes in other languages, typically associated with runtime overhead. Figure 9 below shows a more detailed view of the top modules, including some of their submodules, trait, and struct declarations that are important to understand how the top-level modules interface each other.

⁴Dynamic dispatch is the act of selecting which polymorphic operation to execute at runtime, as opposed to static dispatch where the operation is determined at compile-time.

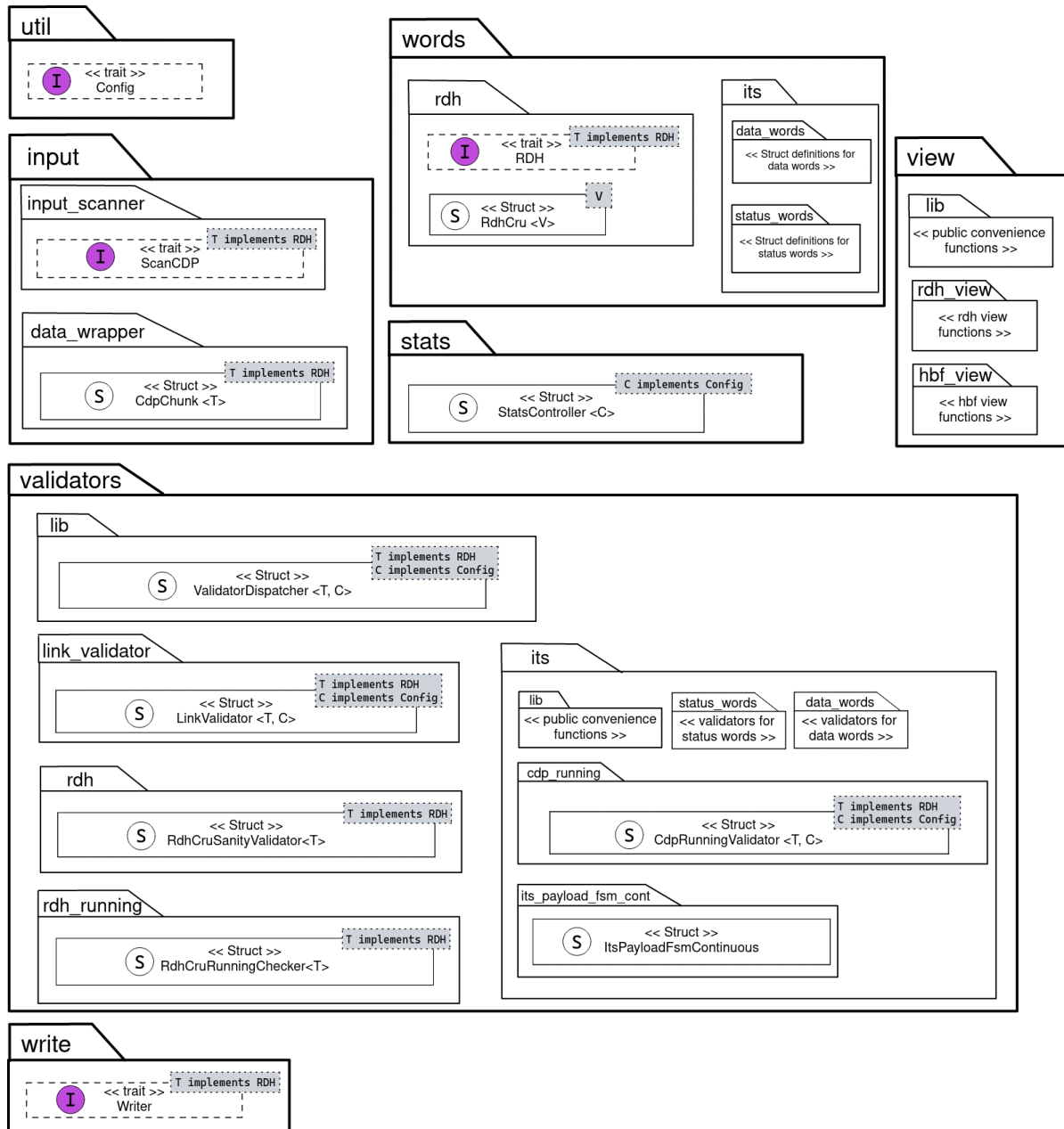


Figure 9: *fastPASTA* top-level modules with selected submodules, structs, and traits. Purple circles with an **I** indicate a trait that is similar to an interface or pure abstract class in *OOP* languages. Circles with an **S** indicate a struct. The frequent use of generics promotes reusability.

An example of the loose coupling via traits is that all data processing modules (validators, input, view, and write) depend on the *RDH* trait by requiring their generic parameter *T* to implement it. This dependence is analogous to the existing *ALICE* readout system, as even to begin to understand and navigate the *raw data*, it is necessary to decode the *RDH* and examine the value of fields such as *RDH* version, memory size, trigger type, and more. The *RDH* currently used in the *ALICE* readout system is version 7, and when version 8 eventually comes around, supporting the new protocol is as simple as writing a struct definition for *RDH* version 8 and then implementing the *RDH* trait;

this is how [fastPASTA](#) currently supports versions 6 and 7 and chooses a verification strategy at runtime depending on whether the first [RDH](#)'s version field value is 6 or 7.

10.1.2 Interface segregation

The `Config` trait contains all the possible settings that a user can configure when using [fastPASTA](#), but all these settings are not relevant in all modules configured by the settings. To follow the *interface segregation principle* by not having modules depend on traits with functions they do not use, the `Config` trait is a *super trait* containing multiple traits that group together related settings. As an example, the `input_scanner` module does not depend on the `Config`; it needs only to know if a filter option is set (such as filtering by [GBT](#) link) and if the [CDP](#) payload should be skipped (in the case where only [RDHs](#) are analyzed), these two options are covered by the `Filter` and `InputOutput` traits that the `Config` super trait requires, and therefore the `input_scanner` depends on those two traits only.

10.1.3 Inter-process communication

As discussed in Section 7.2, [fastPASTA](#) is modeled after a pipeline that lends itself to a high degree of parallelism. Defining the pipes between producers and consumers is trivial in the case of passing data from the `input` module to the `validators` module, as it is a single producer and a single consumer. This case is handled with *message passing* by using a *channel* and defining a message to be passed through the channel. The message `CdpChunk` is defined in the `data_wrapper` submodule; `CdpChunk` is produced from reading input data in the `input` module and, e.g., passing it to the `validators` module, which receives the messages in an *event loop*. This case is illustrated in Figure 10.

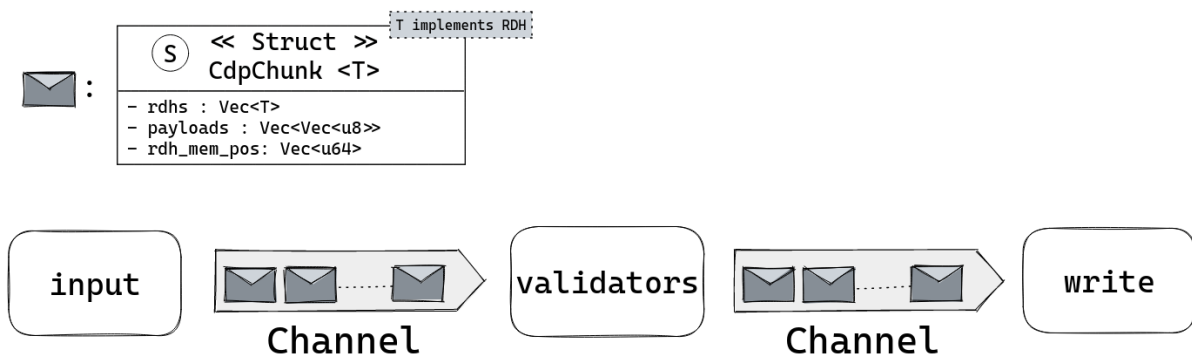


Figure 10: [fastPASTA](#) message passing between the `input`-, `validators`-, and `write`-modules. The message `CdpChunk` is defined in the `data_wrapper`-submodule and is a thin wrapper around 3 vectors. The `CdpChunk` struct defines convenience methods similar to a regular vector, and also implements the *Iterator* pattern to make it easy to iterate over the contents, and makes it compatible with functional algorithms e.g. *filter* and *map*.

In the case of the *stats* module, however, it is not as trivial, considering this is a *multiple producers/single consumer* problem. The messages could be anything from an error message to communicating that an [RDH](#) was observed in the input data and, therefore, the [RDH](#) counter should be incremented. Again, the problem can be solved efficiently and elegantly with message passing, but using an enum as the message, see [Figure 11](#).

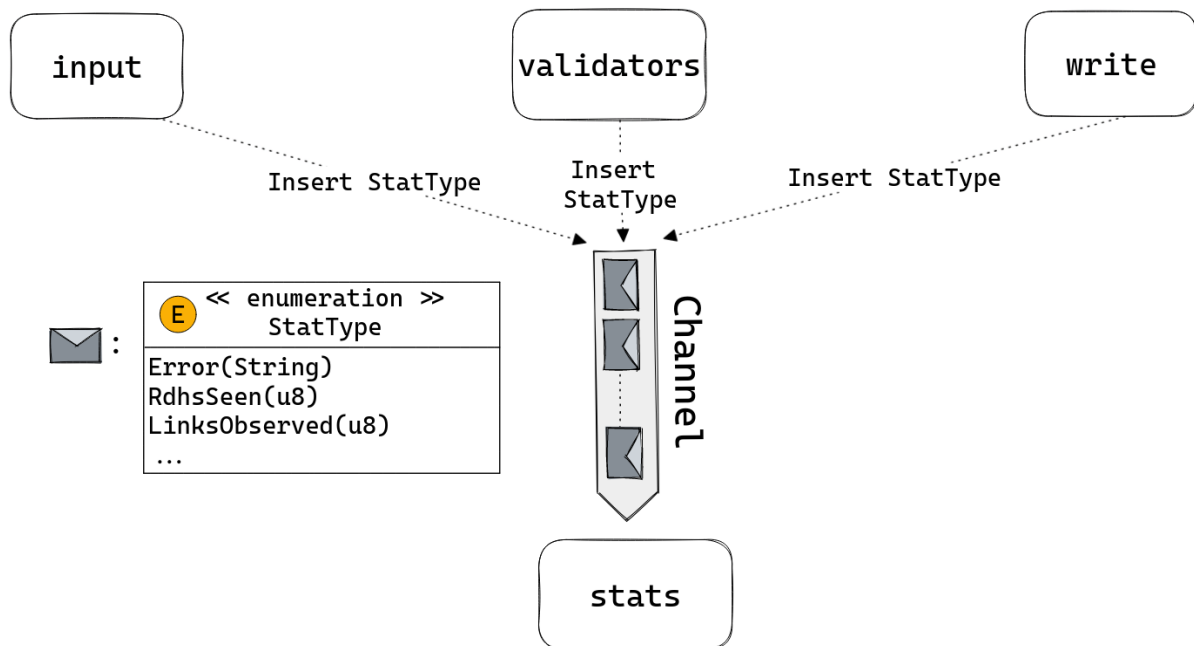


Figure 11: *fastPASTA* message passing from modules to the *stats*-module. Messages are enums with varying content, such as *String* or *u8*, extracted on the receiving end to record the statistic.

A *StatType*-enum has been implemented in the *stats* module, describing all the statistics that can be reported from other modules in a thread-safe manner. The *StatsController* in the *stats* module then receives *StatType* messages through a channel in an event loop and utilizes pattern matching to handle each message appropriately. The only coupling there is to the *stats* module is an enum. If a feature is wanted that records a new type of statistic, it is added to the enum, which will force the developer to update the event loop of the *StatsController* to handle the new type of message before *fastPASTA* compiles again, thus preventing adding a new type of statistic and forgetting to implement handling of the new type.

10.2 Parallelism implementation

[Figure 12](#) illustrates how the initialization steps are implemented, configuring the different steps in the pipeline and setting up channels between them before data processing starts. These steps are performed when a user invokes *fastPASTA* through the command line.

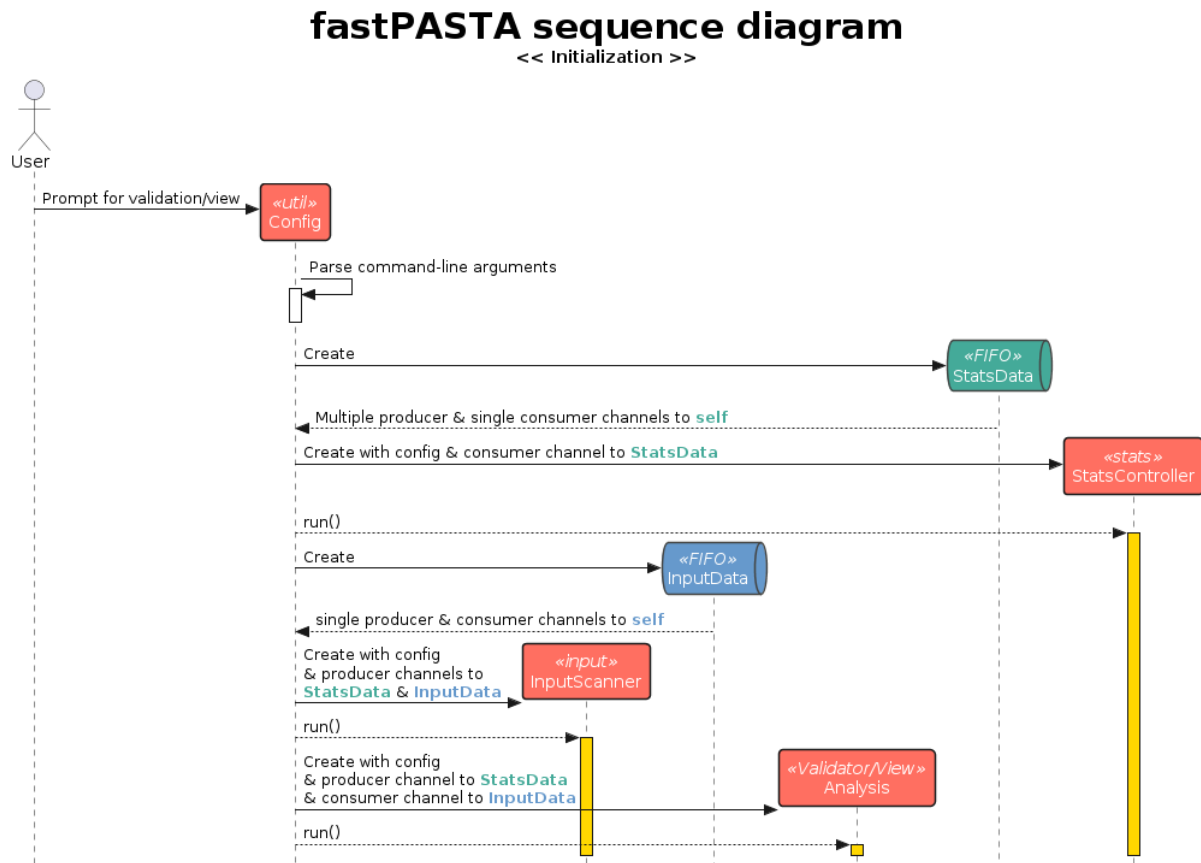


Figure 12: *fastPASTA* initialization is configured by the command-line arguments the user supplies. The initial steps involve passing the config object around and setting up communication channels between the structs that manage each step in the pipeline.

All the relationships between structs in different modules must be established when *fastPASTA* is executed. Each lifeline depicted in Figure 12 runs in a separate thread. It starts with a user invoking *fastPASTA*, and a config object is generated from parsing the command-line arguments. When the config object is created, its contents guide the rest of the initialization process.

After the setup has concluded, data processing starts. The overview of the data processing pipeline is depicted in Figure 13.

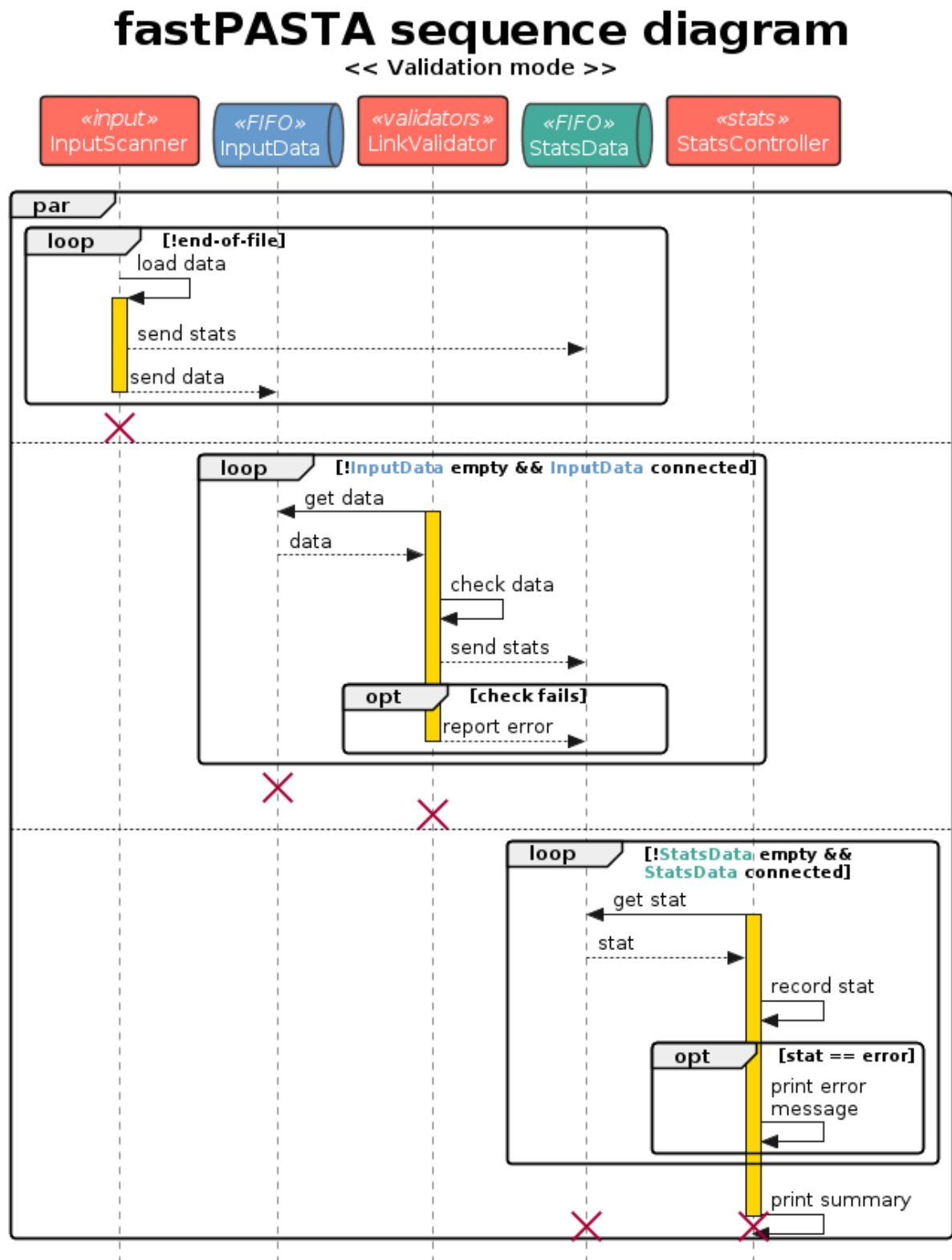


Figure 13: Data processing pipeline in *fastPASTA* validation mode. The *InputScanner* reads *raw data* and inserts it in the *InputData*-channel as *CdpChunk* messages. The *LinkValidator* receives the messages and performs data validation. The *StatController* receives *StatType* messages from each other step in the pipeline and processes them, and prints a summary when all data has been processed.

The data processing starts with the `InputScanner` reading input data and packing it into messages in the form of `CdpChunk` structs before inserting them in the channel `InputData`. The `LinkValidator` receives `CdpChunk` in its event loop and performs validation checks. The `StatsController` receives `StatType` messages from the previously mentioned structs and handles them in an event loop. When all data has been processed, each channel is closed on the sender end, which leads the `StatController` to break out of its event loop and print a summary of data processing before `fastPASTA` exits.

Requirement 2.1 is that `fastPASTA` must parse `raw data` from an arbitrary number of `GBT` links; the difficulty comes when `raw data` has data from multiple `GBT` links intertwined and the fact that each `GBT` link maintains a separate state of the protocols. The `LinkValidator` is implemented to correspond to one `GBT` link. A new thread is spawned at runtime with an instance of a `LinkValidator` running each time a new link is observed in the input `raw data`. The `ValidatorDispatcher` manages the communication to each `LinkValidator` and dispatches `CDPs` based on the `GBT` link in the `RDHs`. This implementation tackles the issue of mixed stateful `raw data` by tracking the protocol in each `LinkValidator` but also has the advantage of performance when each `LinkValidator` runs in parallel.

10.3 Tracking ITS payload protocol

To implement parsing of `ITS`-specific data and perform validation of the `ITS` payload, mentioned in requirements 1.1-4, 3.1, 3.3, 3.5, and 4.1-2, a state machine was implemented to track the protocol. The state machine diagram for the `ITS` payload protocol is illustrated in Figure 14.

Each `LinkValidator` has an instance of the state machine used to determine if each word identified in the input data's payload is valid according to the protocol. In addition to seeing if a word appears in a valid position, checks are performed on the field values of each word to see if the fields are valid in the current state.

ITS payload FSM (continuous mode)

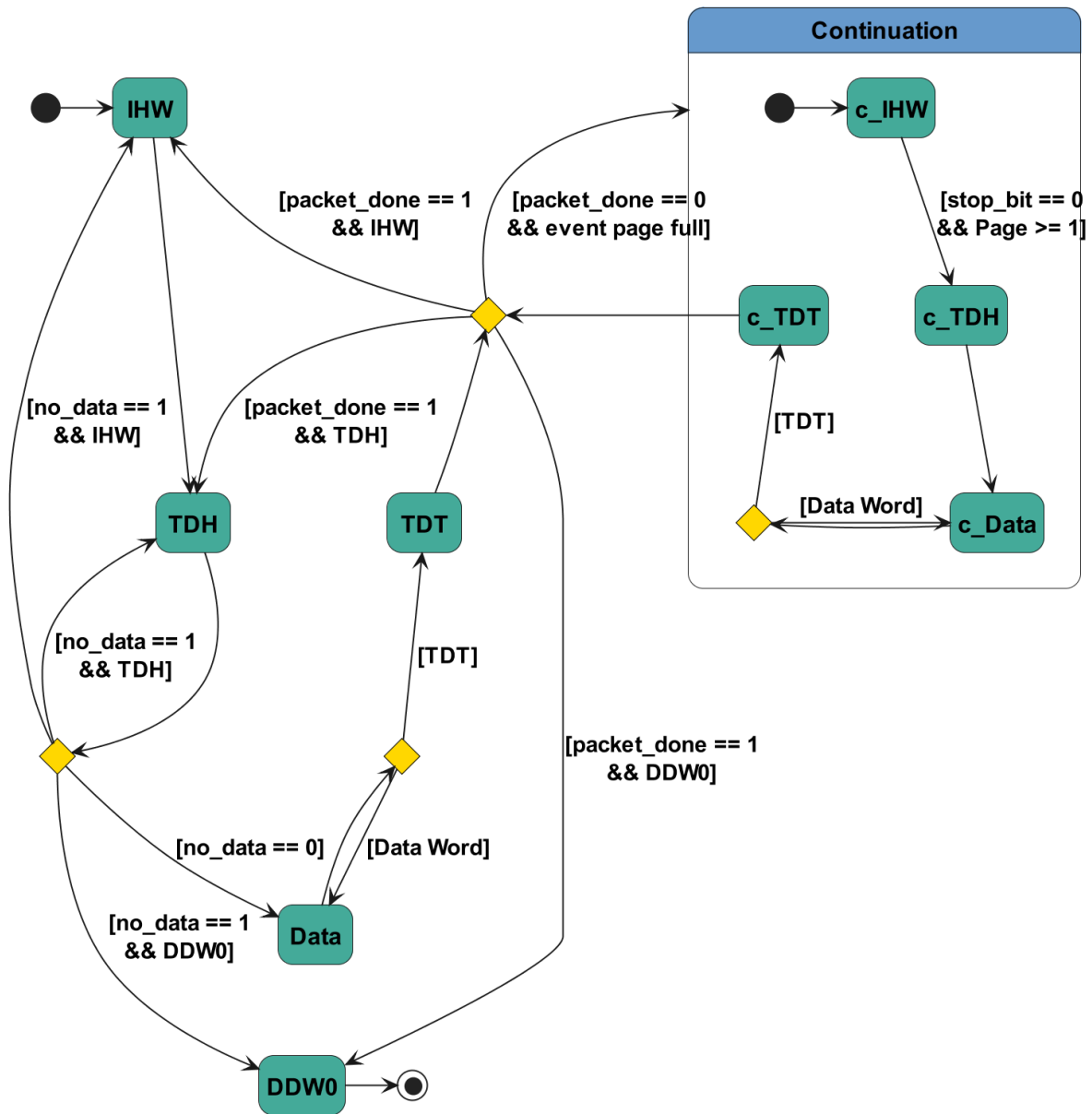


Figure 14: *FSM diagram of the ITS payload words. Most states have several valid next states, and in that case, the transitions are determined from a value in the previous state and the ID of the word in the next state. The ITS payload FSM is depicted in the continuous mode but also has a triggered mode which has never been used and is not expected to see use at this point.*

10.4 Performance

In this section, design and implementation choices related to fulfilling performance requirements are detailed, and the approach to performance and optimization is described with an example case of choosing between implementations of serialization and deserialization.

The statistics-driven micro-benchmarking library *criterion* [42] was used to help make

informed decisions about optimization and performance when evaluating and choosing between implementations.

10.4.1 Reading input data

The ceiling for [fastPASTA](#)'s performance is the speed at which disk read operations can be performed. In the most extreme case, memory mapping can be used to avoid ever copying data into the memory a user space application like [fastPASTA](#) occupies. Memory mapping is only for files and cannot be used when reading from stdin. Because of this constraint, a *buffered reader* is used as the entry point for reading input data. Using a buffered reader significantly reduces the number of system calls for read operations by reading larger chunks specified by the buffer size in the buffered reader. This approach is also compatible with Unix pipelines. A buffered writer was used to limit write operations when writing to disk or stdout.

10.4.2 Serialization & deserialization

Once the [raw data](#) has been loaded into memory, it has to be deserialized into *structs* that make it convenient to work with, as simply working with the [raw data](#) as it is, is inflexible and infeasible for the complicated protocols and nesting of protocols in the [ALICE](#) readout data. If [fastPASTA](#)'s filtering feature is used (requirement 1.5), the data must be serialized again and written to disk or stdout. Serialization and deserialization performance is another potential ceiling for performance, sure to be performed millions of times when even a GB of data is parsed. Therefore different approaches were considered and benchmarked before settling on a solution.

Manual implementation vs. *binrw*

binrw [43] is an open-source library that implements binary serialization and deserialization using *procedural macros* [44]. *Binrw* was an appealing choice as these macros can be written on top of structs, decorating the structs and auto-implementing binary serialization and deserialization.

A proof-of-concept was produced using *binrw* macros to evaluate its feasibility by decorating the [RDH](#) struct definition. 100 [Mebibytes \(MiBs\)](#) of [raw data](#), contains about 70000 [RDHs](#). Each [RDH](#) takes up 64 bytes, translating to about 4 [MiBs](#) or 4% of the data. This number will vary depending on the size of the payload. 100 [MiBs](#) is a tiny amount of data for the [ALICE](#) readout system where several terabytes of data are produced every second, containing billions of [RDHs](#). Executing serialization or deserialization of 70000 [RDHs](#) should be done in milliseconds.

A manual implementation was used to compare to *binrw*. The manual implementation of deserialization is tedious, as it requires loading raw bytes into struct fields one-by-one

and specifying endianness for multi-byte fields. Hence, the *binrw* solution is a very ergonomic alternative. In terms of serializing the structs, the solution was to represent them as *packed structs*, meaning they are memory aligned with no padding, and then use a raw pointer to write them as raw bytes to disk or stdout.

A plot of the execution time of deserializing 1000, 10000, and 50000 RDHs with either the manual or the *binrw* implementation can be seen in Figure 15.

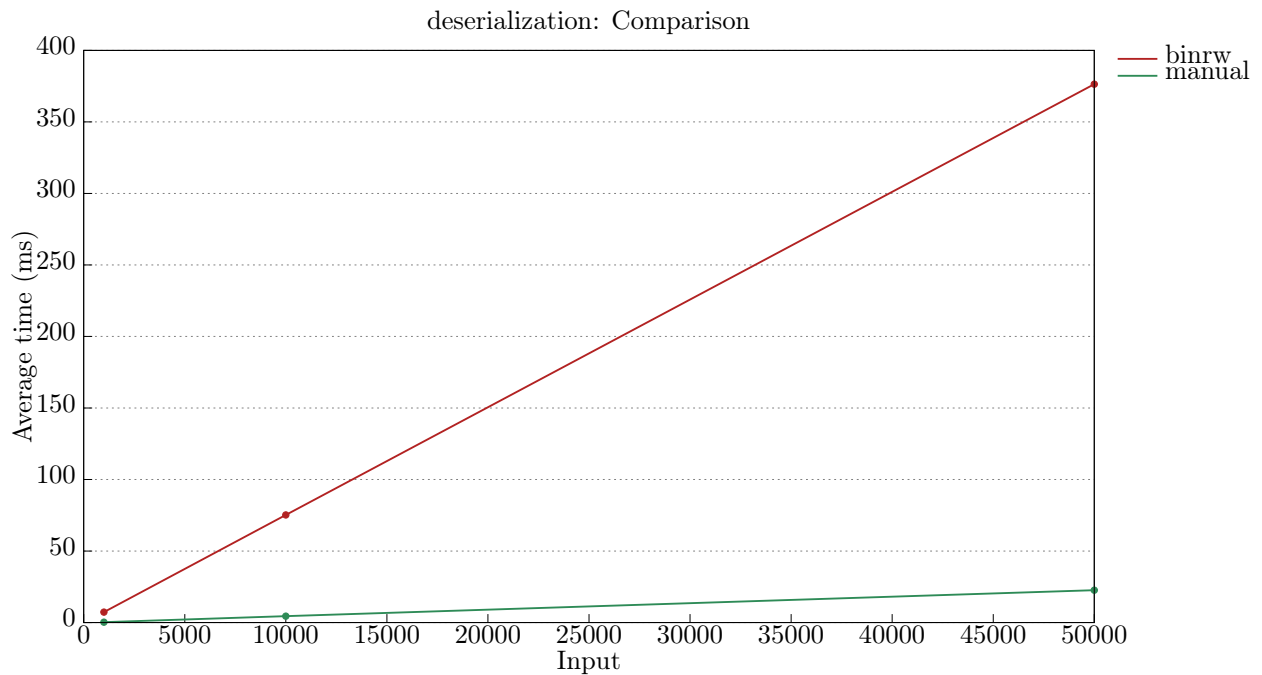


Figure 15: Comparison of scaling of RDH implementations of deserialization, between a manual and a *binrw* implementation through a derive macro. The input denotes the number of RDHs that are deserialized.

The author of *binrw* boasts high performance, but in this case, it takes almost half a second to deserialize 50000 RDHs, compared to ≈ 25 ms for the manual implementation.

The comparison for serializing 50000 RDHs can be seen in Figure 15, where the Probability Density Functions (PDFs) of execution time over 100 iterations are shown for each implementation.

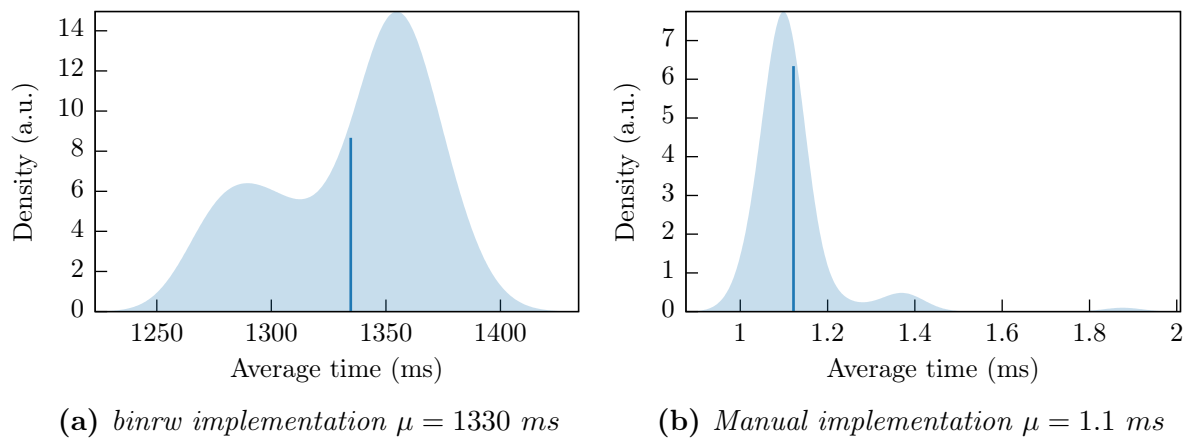


Figure 16: *PDF for the benchmarks of the manual and binrw implementation of the binary serialization of an RDH. Showing the timing distribution of 100 iterations of serializing 50000 RDHs.*

In serialization operations, *binrw* is three orders of magnitude slower than the manual solution.

Based on performance requirements and benchmarking results, the manual approach was chosen for all serialization and deserialization in [fastPASTA](#).

11 Testing

[fastPASTA](#) is a tool for validating other software and digital electronics; therefore, it is crucial that a user can trust error messages. Quality assurance and comprehensive documentation were prioritized early in development to increase quality and trust in [fastPASTA](#)'s correctness. This chapter details how testing of [fastPASTA](#) was carried out, including code documentation, coverage, benchmarking, and acceptance testing.

11.1 Unit testing & code documentation

Each module and sub-module of [fastPASTA](#) contains a dedicated submodule named `tests`, which includes unit tests for the parent module. Some modules also have unit tests as *doc tests*, a feature of Rust where documentation is written as code comments, and these comments can include code, such as unit tests, that is run the same way other unit tests are run. See Listing 1 for an example of how documentation and unit tests are written simultaneously, using the `CdpChunk`'s `with_capacity`-method and asserting that it has a length equal to 0 even though the capacity is 10. All source code documentation can be found on [fastPASTA](#)'s *docs.rs* page [45].

```
/// Construct a new, empty `CdpChunk<T: RDH>` with at least the
/// specified capacity.
///
/// The chunk will be able to hold at least `capacity` elements without
/// reallocating.
///
/// # Examples
/// ```
/// # use fastpasta::input::data_wrapper::CdpChunk;
/// # use fastpasta::words::rdh_cru::{RdhCRU, V7};
/// let mut chunk = CdpChunk::<RdhCRU<V7>>::with_capacity(10);
/// assert!(chunk.len() == 0);
/// ```
pub fn with_capacity(capacity: usize) -> Self {
    Self {
        rdhs: Vec::with_capacity(capacity),
        payloads: Vec::with_capacity(capacity),
        rdh_mem_pos: Vec::with_capacity(capacity),
    }
}
```

Listing 1: *with_capacity* is documented with code comments, and an example is described, which also acts as a unit test.

The code from Listing 1 is included when the documentation is compiled; the result is displayed in Figure 17.

```
[1] pub fn with_capacity(capacity: usize) -> Self
```

[source](#)

Construct a new, empty `CdpChunk<T: RDH>` with at least the specified capacity.

The chunk will be able to hold at least `capacity` elements without reallocating.

Examples

```
let mut chunk = CdpChunk::<RdhCRU<V7>>::with_capacity(10);
assert!(chunk.len() == 0);
```

Figure 17: Documentation of the `CdpChunk`'s `with_capacity` method generated from the code in Listing 1.

11.2 Integration testing

The large modules, with many moving parts, contain a *lib*-module with a *tests* submodule with integration tests. In those tests, several parent modules' submodules are tested with black-box testing. In the integration tests, dependency injection was frequently used, e.g., using a mock config that implemented the `Config-trait` to test behavior influenced by the state of the configuration object.

11.3 System tests

Black box testing of the total `fastPASTA` system was performed using a bash script to input command-line arguments and launch `fastPASTA` in a process. The output was parsed with `grep`, and the match count was checked against the expected match count. Below is an example of such a test.

```
# Array with a file, command to run, and regex to match with.
test_bad_its_payload_errors_detected=(
    "1_hbf_bad_its_payload.raw check all its"
    # Check the invalid 2nd IHW is detected
    "error - 0x50: \[E..\].*ID"
    1
    # Check the error that there is data from lane 8 but the IHW
    # does not have lane 8 set as active
    "error - 0x70: \[E..\].*lane 8.*IHW"
    1
)
```

Listing 2: Example of a system test. the first entry of the array contains the file and command to run, followed by one or more Regular Expressions and the expected match count.

A function runs each test written as in Listing 2, iterating through the array as seen in Listing 3 below.

```
# Run all the tests in a test array
function do_tests {
    # Get the test array by name
    declare -n test_arr=$1
    # The first element of the array is the test case (cmd)
    test_case=${test_arr[0]}
    # Run all the tests in the array
    for ((i=1; i<${#test_arr[@]}; i+=2)); do
        pattern=${test_arr[$i]}
        cond=${test_arr[$((i+1))]}
        run_test $1 "$test_case" "$pattern" "$cond"
    done
}
```

Listing 3: Bash function that takes a test array and iterates through it. Calls the `run_test`-function for each test case in the test array.

For each regular expression and expected match count, the `run_test`-function is called; see Listing 4 below.

```
function run_test {
    test_var_name=$1; test_case=$2; pattern=$3; cond=$4;
    # Run the test, redirecting stderr to stdout,
    # and skip the first two lines of output (cargo info prints)
    test_out=$(eval ${cmd_prefix}${test_case} 2>&1 | tail -n +3 )
    # Count the number of matches
    matches=$(echo "${test_out}" | egrep -i -c "${pattern}")
    if (( "${matches}" == "${cond}" ));
    then
        echo -e "Test passed"
    else
        echo -e "Test failed"
        # Add the test info to the failed tests arrays
        failed_tests+=("${test}")
        failed_patterns+=("${pattern}")
        failed_expected_matches+=("${cond}")
        failed_matches+=("${matches}")
        failed_output+=("${test_out}")
    fi
}
```

Listing 4: Bash function that takes a test with a regular expression and an expected match count. The test is run and checked if the regular expression matches the test output the expected amount of times.

If the regular expression pattern does not match the test output the expected amount of times, the test output and the information about the test case are saved. When the entire test suite has been executed, if any test fails, the failed test(s) are printed to the terminal, and the process exits with error code 1, which would fail a GitLab [CI](#) job if that is where the tests are run. If all tests pass, the process exits with error code 0.

The Rust library `assert_cmd` [\[46\]](#) was also used to write system tests in native Rust similarly.

11.4 Continuous integration & regression testing

A Gitlab pipeline that included jobs that ran unit, integration, and system tests was set up. The main branch would be protected from merging the new changes if any tests failed. The pipeline also included jobs for leak and address sanitizing to detect runtime memory leaks and memory errors. No memory leaks or errors were detected throughout development.

Any time a bug was found, a Gitlab issue was opened, and the bug resolution process then started by adding a minimal reproducible example to the system testing suite and a link to the Gitlab issue. The next step was resolving the bug and passing the test before adding the new changes to the code base. This approach ensured that bugs would not reappear in later changes. Once issues were solved, code could not be added to reintroduce a bug resolved earlier as it would fail the system/regression test suite.

11.5 Code coverage

Code coverage was collected to get an overview of how much code is exercised during tests, and a report was generated to facilitate inspecting which part of the code base was not covered by tests. The tools to collect code coverage of Rust source code are not yet mature, and the options are limited; therefore, only branch coverage was used. Instrumentation-based coverage data was generated via the [LLVM](#) [\[47\]](#) backend of the Rust compiler [\[48\]](#).

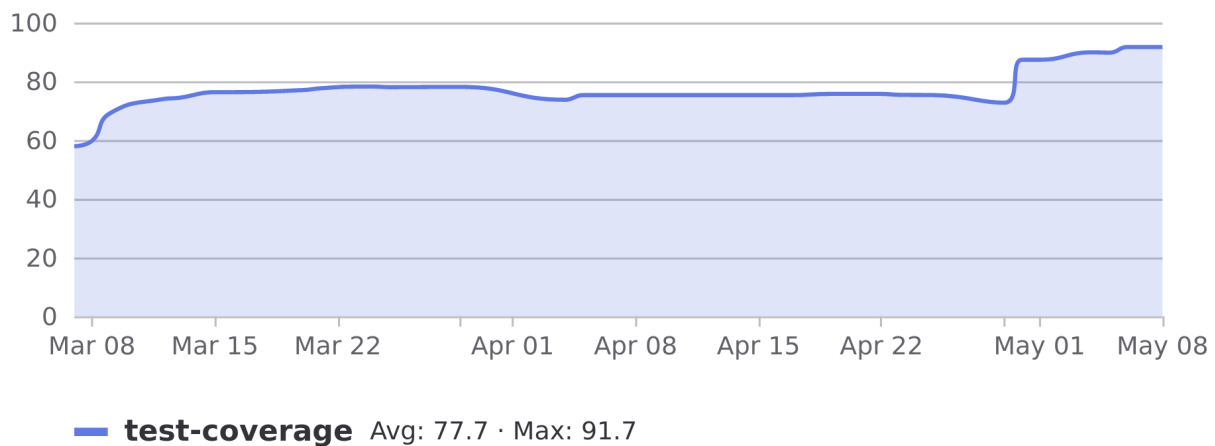


Figure 18: Code coverage history retrieved from the *fastPASTA* GitLab repository, starting from the date code coverage was added. The coverage includes unit tests, integration tests, and system tests. Development concluded at 91.66% code coverage of 11200 lines of code.

See Figure 18 for the code coverage history throughout development. The coverage data was collected into an HTML report with the tool *grcov* [49] and was integrated with GitLab to show code coverage in a badge on the repository and automatically list code coverage changes in merge requests, including showing which lines were not covered when inspecting the changes made in a merge request.

11.6 Acceptance testing

An acceptance test was made for each requirement to determine if *fastPASTA* fulfilled the requirement. The complete acceptance testing document can be found in the appendix [50]. The acceptance test included a collection of files with *raw data* for the tests. These test files were edited manually, e.g., to introduce an error in the ID field of a *TDH*, to test if *fastPASTA* correctly identified the error in the correct memory location. An example of an acceptance test for the functional requirement 1.4 can be seen in table 4 below.

Table 4: *Test of requirement 1.4*

Requirement under test	1.4 Validate that the data from each GBT link (RU output/ CRU input) is in a HBF pattern		
Precondition	fastPASTA version 1.0.5 is installed. Current Working Directory (CWD) contains the test files.		
Action	Expected outcome	Actual outcome	Assessment (OK/FAIL)
In a terminal run: <code>fastpasta</code> <code>err_not_hbf.raw</code> <code>check all</code>	Visual test: An error message indicates a mismatch between 2 RDHs ' pages_counter field. A summary includes an error count of 1 and a RDH count of 2.	An error message indicates a mismatch between the pages_counter field of two RDHs . A summary with the expected information is displayed.	OK

Acceptance tests were also made for non-functional requirements. An example of a test of the non-functional interface requirement 1.8 can be seen in table 5 below.

Table 5: *Test of interface requirement 1.8*

Requirement under test	1.8 Interface Accepts as input a raw data file or stream via standard input structured according to the ITS data format [19] and the “RDH-CRU”-section in ALICE RUN 3 Raw Data Header [20].		
Precondition	fastPASTA version 1.0.5 is installed. CWD contains the test files. The Unix utility <code>cat</code> is installed.		
Action	Expected outcome	Actual outcome	Assessment (OK/FAIL)
In a terminal run: <code>fastpasta</code> <code>10_rdh.raw</code> <code>check sanity</code> <code>its</code>	Visual test: A summary that includes a RDH count of 10, detected RDH version of 7, data format of 0, and link observed is 8.	A summary with the expected information is displayed.	OK
In a terminal run: <code>cat 10_rdh.raw</code> <code> fastpasta</code> <code>check sanity</code> <code>its</code>	Visual test: A summary that includes a RDH count of 10, detected RDH version of 7, data format of 0, and link observed is 8.	A summary with the expected information is displayed.	OK

11.7 System benchmarking

The command-line benchmarking tool *hyperfine* [51] was used to evaluate the speed at which [fastPASTA](#) processes data and performs verification tasks. It was also used to benchmark tools used earlier at [ALICE ITS](#) to perform similar verification tasks, which the [ALICE ITS](#) community seeks to replace with [fastPASTA](#). The benchmarking results are in Chapter [12](#).

12 Results

The project results are presented in terms of [fastPASTA](#) ability to fulfill requirements according to the acceptance tests. Aarhus University supervisor Jesper Michael Kristensen completed the acceptance testing steps, with [CERN](#) supervisor Ola Slettevoll Grøttvik confirming each result. Each requirement and the result of the acceptance test are shown in a table. See table 6 for functional requirements and table 7 for non-functional requirements. Table 8 shows the benchmarking results for [fastPASTA](#) and two similar tools.

Table 6: *Acceptance testing results of the functional requirements.*

#	Functional requirement	Result (OK/FAIL)
1.1	Parse a CDP (RDH + payload)	OK
1.2	Parse a HBF with multiple CDPs	OK
1.3	Parse a CDP payload formatted in both UL flavor 0 and 1	OK
1.4	Validate that the data from each GBT link (RU output/ CRU input) is in a HBF pattern	OK
1.5	Filter data from individual GBT links	OK
1.6	Validate all individual 80-bit GBT words with sanity checks , e.g. that all GBT words have a valid identifier field	OK
1.7	Validate CDPs with a sanity check establishing that the CDP 's structure is coherent as per the CRU protocol	OK
2.1	Parse raw data that includes data from an arbitrary number of different GBT links	OK
2.2	Generate a human-readable overview of RDHs	OK
3.1	Validate that the CDP payload is following the ITS payload protocol	OK
3.2	Parse ITS payload protocol from a calibration run	OK
3.3	Generate a human-readable overview of HBFs	OK
3.4	See a summary of detected attributes in the parsed data, including: RDH version, trigger type, data format, number of HBFs , and RDHs	OK
3.5	See a summary for parsed ITS payloads, of which layers and staves the data originated from	OK

Table 7: *Acceptance testing results of the non-functional requirements.*

#	Non-functional Requirement	Result (OK/FAIL)
Interface		
1.8	Accept as input a raw data file or stream via standard input structured according to the ITS data format [19], and the “RDH-CRU”-section in ALICE RUN 3 Raw Data Header [20].	OK
2.3	Adhere to CLI design guidelines [21].	OK
4.3	fastPASTA is cross-platform compatible. Supporting Windows 10 & 11, and RHEL distributions.	OK
Performance		
1.9	Process 1 GB 20 times faster than <i>decode.py</i>	OK
1.10	Processing 1 GB is convenient on a laptop	OK
Attributes		
1.11	Include comprehensive documentation of implemented validation checks	OK
Schedule		
1.12	Deliver a MVP within 1 month	OK

Table 8: Benchmarking comparison between *fastPASTA* and two similar protocol verification tools *decode.py* and *rawdata-parser*.

Tool	Command	Mean $\pm \sigma$ [s]	Min [s]	Max [s]
<i>Verifying all RDHs of 260 MB file with data from 1 GBT link</i>				
<i>fastPASTA</i>	<code>fastpasta input.raw check all</code>	0.039 ± 0.001	0.037	0.043
<i>rawdata-parser</i>	<code>rawdata-parser --skip- packet-counter-checks input.raw</code>	0.381 ± 0.012	0.356	0.438
<i>decode.py</i>	<code>python3 decode.py -i 20522 -f input.raw -skip_data</code>	13.674 ± 0.386	13.610	14.499
<i>Verifying all RDHs in 2 GB file with data from 12 different GBT links</i>				
<i>fastPASTA</i>	<code>fastpasta input.raw check all</code>	0.459 ± 0.018	0.411	0.535
<i>rawdata-parser</i>	<code>rawdata-parser --skip- packet-counter-checks input.raw</code>	3.080 ± 0.047	3.005	3.178
<i>decode.py</i>	Verifying multiple links simultaneously is not supported	N/A	N/A	N/A
<i>Verifying all RDHs and payloads in 260 MB file with data from 1 GBT link</i>				
<i>fastPASTA</i>	<code>fastpasta input.raw check all</code>	0.106 ± 0.002	0.103	0.111
<i>rawdata-parser</i>	Verifying payloads is not supported	N/A	N/A	N/A
<i>decode.py</i>	<code>python3 decode.py -i 20522 -f input.raw</code>	55.903 ± 0.571	54.561	56.837

Table 8 compares benchmarking results between *fastPASTA*, *decode.py*, a command-line utility implemented in Python, and *rawdata-parser* implemented in Racket (and pre-compiled). The comparisons should be taken with a grain of salt, as *fastPASTA* performs significantly more checks on the data than *decode.py* and *rawdata-parser* do, and the exact measurements will vary depending on the machine where the benchmarking is performed.

13 Discussion

Meeting with [SMEs](#) of the [ALICE](#) team regularly has been crucial, as understanding the binary protocols is challenging, and implementing stateful verification requires intimate knowledge of how the [ALICE](#) detector and the [LHC](#) experiment are designed. Understanding the protocols and how to perform verification had to be done continuously alongside development in a divide-and-conquer approach.

In Chapter [12](#), it is shown that the requirements for the [MVP](#) are fulfilled, and the [MVP](#) was delivered within one month. All *Must Have*- and *Should Have*-requirements were fulfilled, and one *Could Have* interface requirement was also fulfilled.

The final product, [fastPASTA](#), allows a member of [ALICE](#) to perform various tasks on the [raw data](#) from the [ALICE](#) detector's readout system, including verifying the protocol correctness of [RDHs](#) in the [CRU](#) protocol, agnostic to the payload protocol, but also allows them to specify [ITS](#) as the payload protocol to enable more rigorous verification of the [RDHs](#). It is made possible to verify the [ITS](#) data protocol with [sanity checks](#) or additional stateful checks, filter data from specific [GBT](#) links present in the [raw data](#), and display a human-readable overview of the [raw data](#), either at the [CRU](#) protocol or the [ITS](#) data protocol level.

13.1 Impact of Rust

The high performance of [fastPASTA](#) was made possible using Rust, a statically typed, compiled language, letting benchmarking guide development and always utilizing generics instead of dynamic polymorphism. The performance of [fastPASTA](#) surpassed both tools it is replacing (see Section [11.7](#)). Unsurprisingly, [fastPASTA](#) is over 500 times faster than the Python tool *decode.py*. A dynamic language rarely beats a compiled language like Rust in a benchmark. More surprising is the approximate factor 10 improvement compared to *rawdata-parser*, a tool implemented in Racket and compiled with Raco [\[52\]](#).

Its modular pipeline architecture facilitates the reusability of [fastPASTA](#). The developer seeking to extend [fastPASTA](#) with more features will need a solid fundamental understanding of systems programming and a basic understanding of Rust.

The barrier to entry for contributing to a project written in Rust is significant. The language is complex and forces the developer to decide how to deal with every detail. It impacts the speed of development when the developer has to decide everything from how to handle the value of a 32-bit unsigned integer not fitting into a 32-bit signed integer when type-casting (even if it is known that it will always fit in this use case) to how to handle data races in parallel code. Rust is very verbose; everything has to be explicitly handled, so project size in lines of code grows quickly ([fastPASTA](#) has over 11000 lines of code). The upside is very robust code. The only bugs encountered in [fastPASTA](#) were logic errors.

Taking advantage of Rust's ecosystem is incredibly easy. Searching for open-source libraries that fit a use case and adding them to the project takes just a few minutes. [fastPASTA](#) has 20 external dependencies, and at no point did any conflicts or issues of linking arise. Customizing build profiles and distributing [fastPASTA](#) was equally simple.

14 Conclusion

A usable [fastPASTA](#) has been delivered to the [ALICE](#) community that tackles issues described in the problem statement. A performant tool that can aid [ALICE](#) members in debugging the [ALICE](#) readout system and verifying the correctness of the [raw data](#). [fastPASTA](#)'s modular architecture makes it reusable and easy to add features. Rust's high degree of safety and ease of using its toolchain and ecosystem further simplifies the reuse of and contribution to [fastPASTA](#). The test suite, [CI](#) implementation, and comprehensive documentation make it maintainable and protect the code base from regression.

Consulting [ALICE](#) members weekly to get feedback on [fastPASTA](#) boosted the iterative development. Studying Rust literature before development and continuously consulting it was crucial to maintain momentum and make informed decisions about design and implementation.

15 Future development

As part of the delimitation of this project, several requirements were excluded, and these would instead be part of potential future development. The excluded requirements include verification of the [ALPIDE](#) protocol and consistency checks across the [CRU](#) and [ALPIDE](#) protocol. For these checks to be implemented, it is necessary first to implement an [ALPIDE](#) protocol decoder.

Only the [ITS](#) subdetector readout system is currently supported for advanced verification beyond [RDHs](#). A team of another subdetector might wish to implement verification of their custom protocols, as well as the human-readable views of the protocols, similar to what is implemented for [ITS](#).

[ITS](#) is set for a major upgrade during [LS3](#) 2026-2029, including the replacement of all [ALPIDE](#) sensors, and with that, the [ALPIDE](#) protocol is also replaced. Supporting the changes to protocols and setting up a framework to quickly adapt [fastPASTA](#) to newer versions without cluttering the [CLI](#) would be desirable, such as coupling the version of [fastPASTA](#) with version changes of the [ALICE](#) readout system so that an older version of [fastPASTA](#) can be used for verifying/inspecting old data without having the most recent version of [fastPASTA](#) burdened by having to be fully backward compatible.

References

- [1] Marc König / *fastPASTA* · *GitLab*. URL: <https://gitlab.cern.ch/mkonig/fastpasta> (visited on 16/05/2023).
- [2] Steve Klabnik and Carol Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, Dec. 2019.
- [3] Jon Gjengset. *Rust for Rustaceans*. No Starch Press, Nov. 2021.
- [4] *The Rustonomicon*. URL: <https://doc.rust-lang.org/nomicon/intro.html> (visited on 19/05/2023).
- [5] *Rust Design Patterns*. URL: <https://rust-unofficial.github.io/patterns/intro.html> (visited on 19/05/2023).
- [6] *Rust By Example*. URL: <https://doc.rust-lang.org/rust-by-example/index.html> (visited on 19/05/2023).
- [7] *The Cargo Book*. URL: <https://doc.rust-lang.org/cargo/index.html> (visited on 19/05/2023).
- [8] *The rustc book*. URL: <https://doc.rust-lang.org/rustc/index.html> (visited on 17/05/2023).
- [9] *The rustdoc book*. URL: <https://doc.rust-lang.org/rustdoc/index.html> (visited on 19/05/2023).
- [10] *Command Line Applications in Rust*. URL: <https://rust-cli.github.io/book/index.html> (visited on 19/05/2023).
- [11] wikipedia. *CERN*. URL: <https://en.wikipedia.org/wiki/CERN> (visited on 26/05/2023).
- [12] ALICE Collaboration. *ALICE upgrades during the LHC Long Shutdown 2*. 2023. arXiv: [2302.01238](https://arxiv.org/abs/2302.01238) [[physics.ins-det](https://arxiv.org/abs/2302.01238)].
- [13] *Bash (Unix shell)* - *Wikipedia*. URL: [https://en.wikipedia.org/wiki/Bash_\(Unix_shell\)](https://en.wikipedia.org/wiki/Bash_(Unix_shell)) (visited on 28/04/2023).
- [14] *grep(1)* - *Linux manual page*. URL: <https://www.man7.org/linux/man-pages/man1/grep.1.html> (visited on 28/04/2023).
- [15] *less(1)* - *Linux manual page*. URL: <https://man7.org/linux/man-pages/man1/less.1.html> (visited on 28/04/2023).
- [16] *Pipeline (Unix)* - *Wikipedia*. URL: [https://en.wikipedia.org/wiki/Pipeline_\(Unix\)](https://en.wikipedia.org/wiki/Pipeline_(Unix)) (visited on 28/04/2023).
- [17] Marc Beck König. *Appendix A - fastPASTA Software Requirement Specification Document*. Tech. rep.

- [18] *Issues · Marc Konig / fastPASTA · GitLab*. URL: <https://gitlab.cern.ch/mkonig/fastpasta/-/issues> (visited on 28/04/2023).
- [19] ALICE ITS WP10. «ITS Data Format». Unpublished internal CERN document. URL: https://gitlab.cern.ch/alice-its-wp10-firmware/RU_mainFPGA/-/wikis/ITS%20Data%20Format (visited on 16/02/2023).
- [20] ALICE O2 Group. *ALICE Run 3 Raw Data Header*. Unpublished internal CERN document. URL: <https://gitlab.cern.ch/AliceO2Group/wp6-doc/-/blob/master/rdh/RDHv7.md> (visited on 16/02/2023).
- [21] Aanand Prasad et al. *Command Line Interface Guidelines*. URL: <https://clig.dev/>.
- [22] *MoSCoW method - Wikipedia*. URL: https://en.wikipedia.org/wiki/MoSCoW_method (visited on 16/02/2023).
- [23] Tom Preston-Werner. *Semantic Versioning 2.0.0 / Semantic Versioning*. URL: <https://semver.org/> (visited on 18/04/2023).
- [24] *Issues / GitLab*. URL: <https://docs.gitlab.com/ee/user/project/issues/> (visited on 19/04/2023).
- [25] *Milestones / GitLab*. URL: <https://docs.gitlab.com/ee/user/project/milestones/> (visited on 19/04/2023).
- [26] *Releases / GitLab*. URL: <https://docs.gitlab.com/ee/user/project/releases/index.html> (visited on 18/04/2023).
- [27] *crates.io: Rust Package Registry*. URL: <https://crates.io/> (visited on 28/04/2023).
- [28] *Docs.rs*. URL: <https://docs.rs/> (visited on 19/05/2023).
- [29] *Stack Overflow Trends*. URL: <https://insights.stackoverflow.com/trends/?tags=rust%2Cc%2B%2B%2Cgo> (visited on 05/03/2023).
- [30] *Boost C++ Libraries*. URL: <https://www.boost.org/> (visited on 28/04/2023).
- [31] C++ Standard Committee. *C++ Standard Core Language Active Issues, Revision 110*. URL: https://open-std.org/JTC1/SC22/WG21/docs/cwg_active.html.
- [32] National Security Agency. *Software Memory Safety*. Tech. rep. URL: https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF (visited on 26/05/2023).
- [33] Stack Overflow. *2022 Developer Survey*. URL: <https://survey.stackoverflow.co/2022/#technology> (visited on 26/05/2023).
- [34] Sergio De Simone. *Linux 6.1 Officially Adds Support for Rust in the Kernel*. URL: <https://www.infoq.com/news/2022/12/linux-6-1-rust/> (visited on 05/03/2023).
- [35] Bei Chu. *Two things that Rust does better than C++*. URL: <https://getdozer.io/blog/rust-cpp-move-and-dispatch/> (visited on 26/05/2023).

- [36] Thomas Claburn. *Microsoft is rewriting core Windows libraries in Rust* • *The Register*. URL: https://www.theregister.com/2023/04/27/microsoft_windows_rust/ (visited on 29/04/2023).
- [37] Patrick Walton. *Are we stack-efficient yet?* URL: <https://arewestackefficientyet.com/> (visited on 26/05/2023).
- [38] *Go 1.18 is released! - The Go Programming Language*. URL: <https://go.dev/blog/go1.18> (visited on 05/03/2023).
- [39] *Go Developer Survey 2020 Results - The Go Programming Language*. URL: <https://go.dev/blog/survey2020-results#:~:text=Among%20the%206%25%20of%20respondents%20who%20said%20Go%20lacks%20language%20features%20they%20need%2C%2088%25%20selected%20generics%20as%20a%20critical%20missing%20feature.> (visited on 05/03/2023).
- [40] Jesse Howard. *Why Discord is switching from Go to Rust*. URL: <https://discord.com/blog/why-discord-is-switching-from-go-to-rust> (visited on 28/04/2023).
- [41] Priyank Gupta. *Understanding the modular monolith and its ideal use cases / TechTarget*. URL: <https://www.techtarget.com/searchapparchitecture/tip/Understanding-the-modular-monolith-and-its-ideal-use-cases> (visited on 28/04/2023).
- [42] *Criterion.rs - Criterion.rs Documentation*. URL: <https://bheisler.github.io/criterion.rs/book/index.html> (visited on 09/05/2023).
- [43] *binrw - crates.io: Rust Package Registry*. URL: <https://crates.io/crates/binrw> (visited on 09/05/2023).
- [44] *Procedural Macros - The Rust Reference*. URL: <https://doc.rust-lang.org/reference/procedural-macros.html> (visited on 09/05/2023).
- [45] *fastpasta - Rust*. URL: <https://docs.rs/fastpasta/1.0.5/fastpasta/index.html> (visited on 19/05/2023).
- [46] *assert_cmd - crates.io: Rust Package Registry*. URL: https://crates.io/crates/assert_cmd (visited on 10/05/2023).
- [47] *The LLVM Compiler Infrastructure Project*. URL: <https://llvm.org/> (visited on 17/05/2023).
- [48] *Instrumentation-based Code Coverage - The rustc book*. URL: <https://doc.rust-lang.org/rustc/instrument-coverage.html> (visited on 17/05/2023).
- [49] *mozilla/grcov: Rust tool to collect and aggregate code coverage data for multiple source files*. URL: <https://github.com/mozilla/grcov#example-how-to-generate-source-based-coverage-for-a-rust-project> (visited on 17/05/2023).
- [50] Marc Beck König. *Appendix B - fastPASTA Acceptance testing*. Tech. rep.
- [51] *sharkdp/hyperfine: A command-line benchmarking tool*. URL: <https://github.com/sharkdp/hyperfine> (visited on 17/05/2023).

- [52] *3 raco distribute: Sharing Stand-Alone Executables*. URL: <https://docs.racket-lang.org/raco/exe-dist.html> (visited on 19/05/2023).