CNOME

A Project Report

Submitted by

Atharva Mutsaddi 112103015

in partial fulfilment for the award of the degree

of

B.Tech (Computer Engineering)

Under the guidance of

Dr.Tanuja Pattanshetti

COEP Technological University, Pune

DEPARTMENT OF COMPUTER ENGINEERING AND INFORMATION TECHNOLOGY,

COEP Technological University

May, 2011

DEPARTMENT OF COMPUTER ENGINEERING AND INFORMATION TECHNOLOGY,

COEP Technological University

CERTIFICATE

Certified that this project, titled ”CNOME” has been successfully completed by

Atharva Mutsaddi 112103015

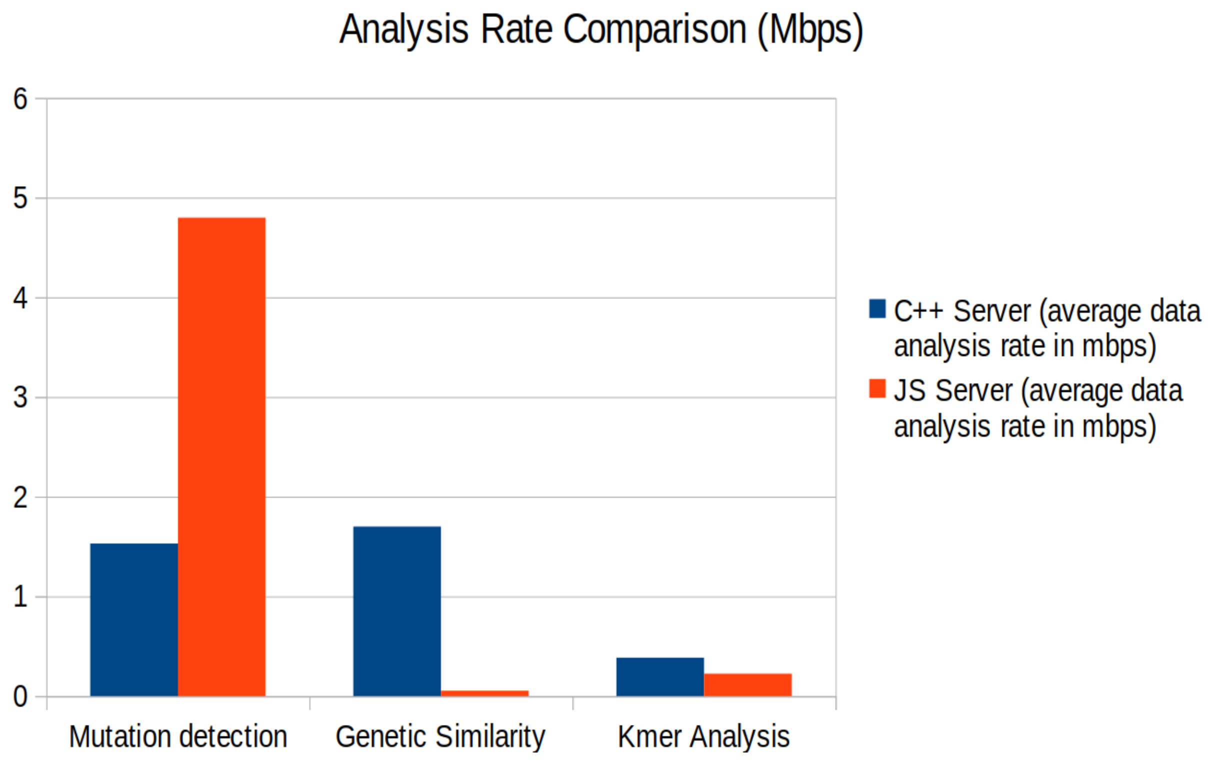
and is approved for the partial fulfilment of the requirements for the degree of “B.Tech. Computer Engineering”.

SIGNATURE	SIGNATURE
Dr. T. R. Pattanshetti	**Dr P. K. Deshmukh**
Project Guide	**Head**
Department of Computer Engineering	**Department of Computer Engineering**
and Information Technology,	**and Information Technology,**
COEP Technological University,	**COEP Technological Univesity,**
Shivajinagar, Pune - 5.	**Shivajinagar, Pune - 5.**

Abstract

CNOME is a high-performance Analytics platform designed for the rapid analysis of genetic sequences. It offers capabilities such as detecting genetic mutations within input samples, conducting efficient KMer Analysis, and determining similarity between genetic sequences. Powered by optimized algorithms written in C++, CNOME operates on a multi-threaded backend, leveraging low-level language features to minimize computational latency.

The platform's backend is built on a C++ HTTPS web server, facilitating the transmission of analytics data using REST principles. This data is then presented to users through a modern frontend developed with React and TypeScript. CNOME aims to significantly accelerate genetic data analysis compared to traditional methods by harnessing the computational efficiency of C++ in a multi-threaded environment.



Contents

Chapter 1

Introduction

1.1 Introduction

In the field of genetics and bioinformatics, the analysis of genetic sequences plays a pivotal role in understanding various biological phenomena, such as disease susceptibility, evolutionary relationships, and genetic diversity. With advancements in sequencing technologies, there is a growing demand for efficient and high-performance analytics platforms capable of processing vast amounts of genetic data with minimal latency.

1.2 Aim of the Project

The primary objective of this project is to develop CNOME, a low-latency Analytics platform tailored specifically for the analysis of genetic sequences. CNOME aims to address the need for rapid and accurate detection of genetic mutations, conduct comprehensive KMer Analysis, and determine sequence similarity efficiently.

1.3 Methodology

The methodology employed in this project involves the utilization of optimized algorithms implemented in C++ to perform various genetic sequence analysis tasks. Leveraging multi-threading capabilities, CNOME operates on a backend powered by a C++ HTTPS web server, facilitating seamless data transmission using REST principles. The frontend, developed with React and TypeScript, provides users with a modern interface for accessing and visualizing analytical results.

1.4 Significance of the Work

The significance of CNOME lies in its ability to accelerate genetic data analysis, offering researchers and clinicians a powerful tool for deciphering complex genetic information swiftly and accurately. By harnessing the computational efficiency of C++ and multi-threading, CNOME aims to streamline genetic analysis workflows, thereby enhancing research productivity and facilitating advancements in various domains such as precision medicine, evolutionary biology, and genetic epidemiology.

Chapter 2

Presentation of the Problem

2.1 Overview

In this chapter, we present a detailed analysis of the problem statement addressed by the CNOME project. The chapter begins with an overview of the challenges associated with genetic sequence analysis, highlighting the need for a low-latency analytics platform. Subsequently, the specific problems targeted by CNOME, including the detection of genetic mutations, KMer Analysis, and sequence similarity computation, are discussed in depth.

2.2 Challenges in Genetic Sequence Analysis

The analysis of genetic sequences poses several challenges due to the sheer volume of data generated by modern sequencing technologies. Traditional approaches often suffer from high computational latency and inefficiency, hindering timely insights into genetic information. Furthermore, the complexity of genetic data, including variations in sequence lengths and the presence of mutations, exacerbates the computational burden.

2.3 Objectives of CNOME

The primary objective of CNOME is to overcome the challenges associated with genetic sequence analysis by providing a low-latency analytics platform. Specifically, CNOME aims to achieve rapid and accurate detection of genetic mutations, conduct comprehensive KMer Analysis, and facilitate efficient computation of sequence similarity. By addressing these objectives, CNOME seeks to streamline genetic analysis workflows and enhance the productivity of researchers and clinicians in various domains.

2.4 Backend Architecture

CNOME employs a backend architecture implemented in C++, leveraging the httplib library for creating an HTTPs web server. This backend architecture facilitates seamless communication between the client and server components of CNOME, enabling efficient data transmission and processing.

2.5 Optimized Data Structures

To enhance performance and efficiency, CNOME utilizes optimized data structures implemented in C++. These data structures are specifically tailored to handle genetic sequence data, enabling fast and scalable processing of genetic information.

2.6 Multi-threading

Multi-threading is a key component of CNOME's methodology, allowing concurrent execution of tasks to maximize resource utilization and minimize latency. By harnessing the power of multi-threading, CNOME achieves parallel processing of genetic sequences, significantly improving the overall speed and efficiency of analysis.

Chapter 3

General Outline of the Solution

3.1 Overview

The solution to the problem addressed by CNOME is based on a comprehensive approach that encompasses various stages of genetic sequence analysis. This section provides a general outline of the steps involved in CNOME's solution methodology.

3.2 Data Ingestion and Preprocessing

The solution begins with the ingestion of genetic sequence data, which is preprocessed to remove noise and ensure data integrity. This preprocessing step involves cleaning and standardizing the input data to facilitate downstream analysis.

3.3 Mutation Detection

CNOME employs optimized algorithms to detect genetic mutations within input sequences. By leveraging efficient search techniques and data structures, CNOME accurately identifies mutations and flags regions of interest for further analysis.

3.4 KMer Analysis

KMer Analysis is performed using specialized algorithms designed to extract and analyze short subsequences (KMers) within genetic sequences. CNOME calculates KMer frequencies and distributions to identify patterns and motifs indicative of biological significance.

3.5 Sequence Similarity Computation

The final stage of CNOME's solution involves computing the similarity between genetic sequences. Utilizing advanced similarity metrics and algorithms, CNOME quantifies the degree of similarity between sequences, enabling researchers to infer evolutionary relationships and functional similarities.

Chapter 4

Algorithm Explanation

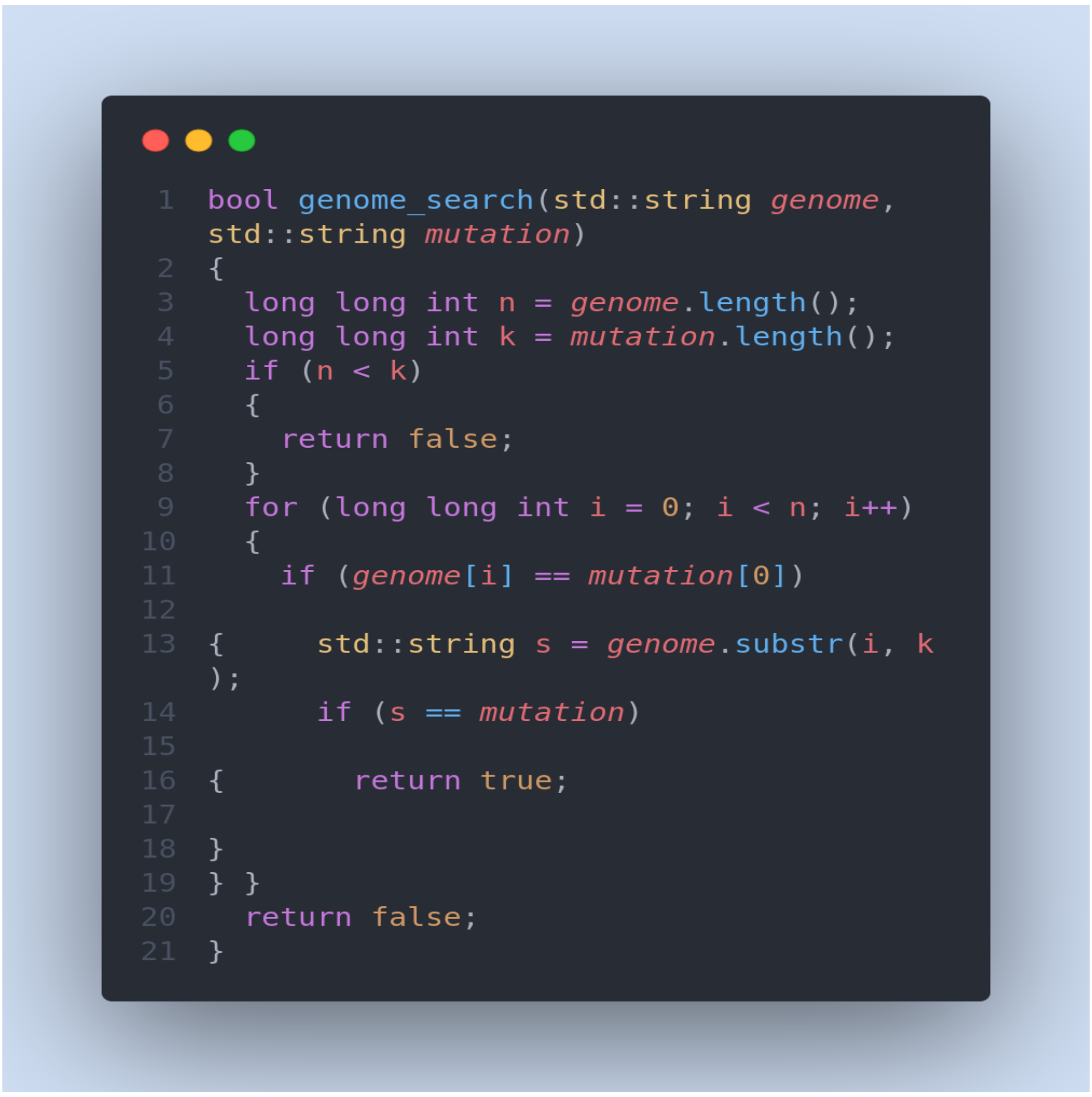
In this chapter, we provide detailed explanations of the algorithms used in the CNOME project for genetic sequence analysis. Each algorithm is accompanied by a description of its approach and methodology.

4.1 Mutation Detection

The `genome_search` function is responsible for detecting mutations within a given genetic sequence. It iterates through the sequence and compares subsequences of length k with known mutation patterns. If a match is found, the function returns `true`; otherwise, it returns `false`.

4.2 KMer Analysis

The `KMer_analysis` function performs KMer analysis on a given genetic sequence. It iterates through the sequence and extracts all k-length subsequences (KMers), calculating their frequencies. Subsequently, it filters out KMers with low frequencies, retaining only those that occur frequently enough to be considered significant. This algorithm provides insights into the distribution of short subsequences within the sequence, which can aid in



```cpp
bool genome_search(std::string genome,
std::string mutation)
{
  long long int n = genome.length();
  long long int k = mutation.length();
  if (n < k)
  {
    return false;
  }
  for (long long int i = 0; i < n; i++)
  {
    if (genome[i] == mutation[0])

{     std::string s = genome.substr(i, k
);
      if (s == mutation)

{       return true;

}
} }
  return false;
}
```

Figure 4.1: Code implementation of the genome search algorithm.

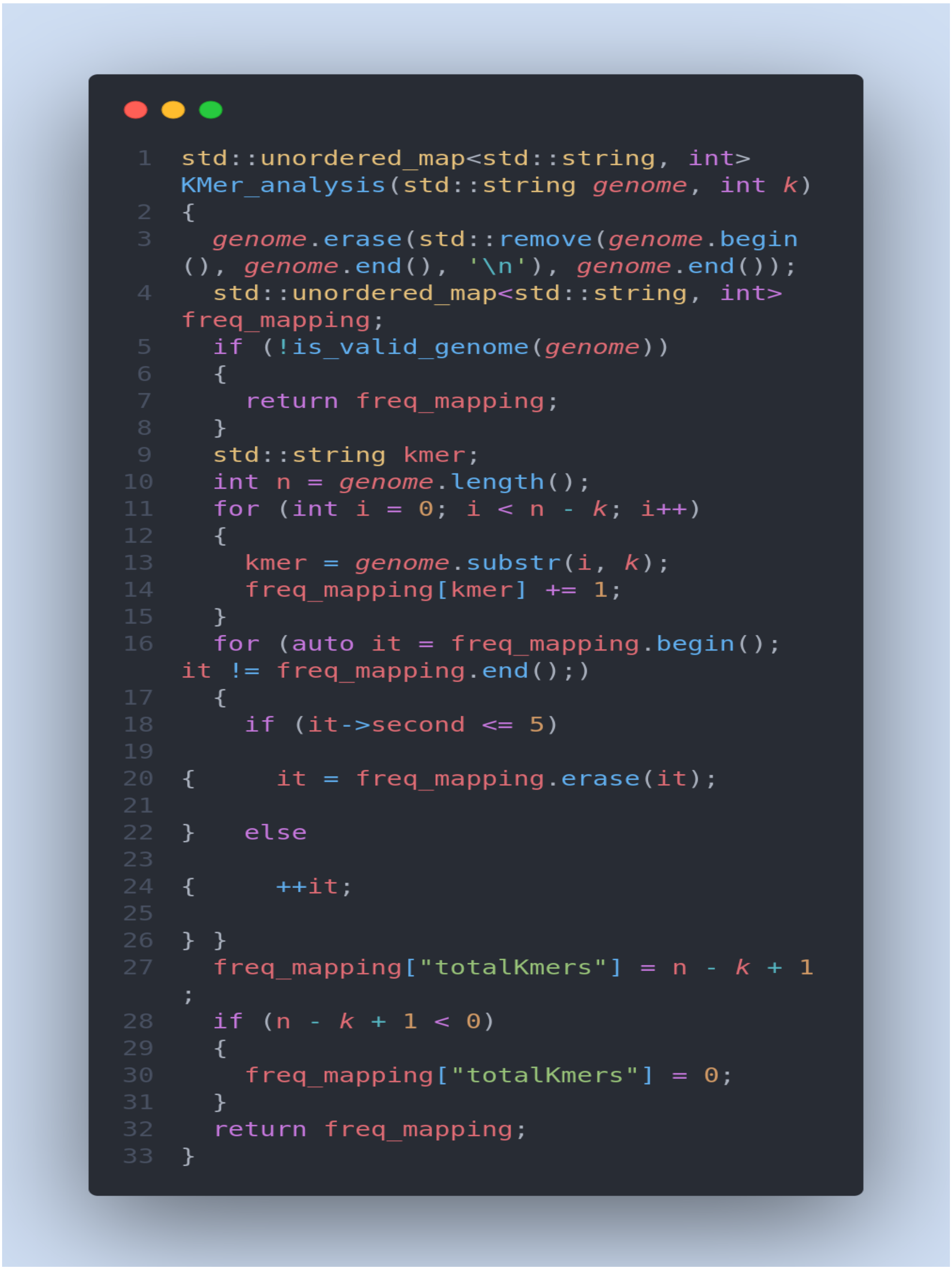
identifying patterns and motifs.

4.3 Sequence Similarity Computation

The `calculateNgramSimilarity`, `jaccard_index`, and `sequence_identity` functions compute the similarity between two genetic sequences using different metrics. `calculateNgramSimilarity` calculates the similarity based on the frequency of n-grams (subsequences of length n) shared between the sequences. `jaccard_index` computes the Jaccard Index, which measures the overlap of k-mers between the sequences. `sequence_identity` calculates the percentage of matching positions between the sequences. Each of these algorithms provides a different perspective on sequence similarity, allowing researchers to assess genetic relationships from multiple angles. It is important to note that these algorithms are run using 3 different threads, each parallelly computing their own scores to optimize the analysis. Utilizes the unordered set and map data structures.

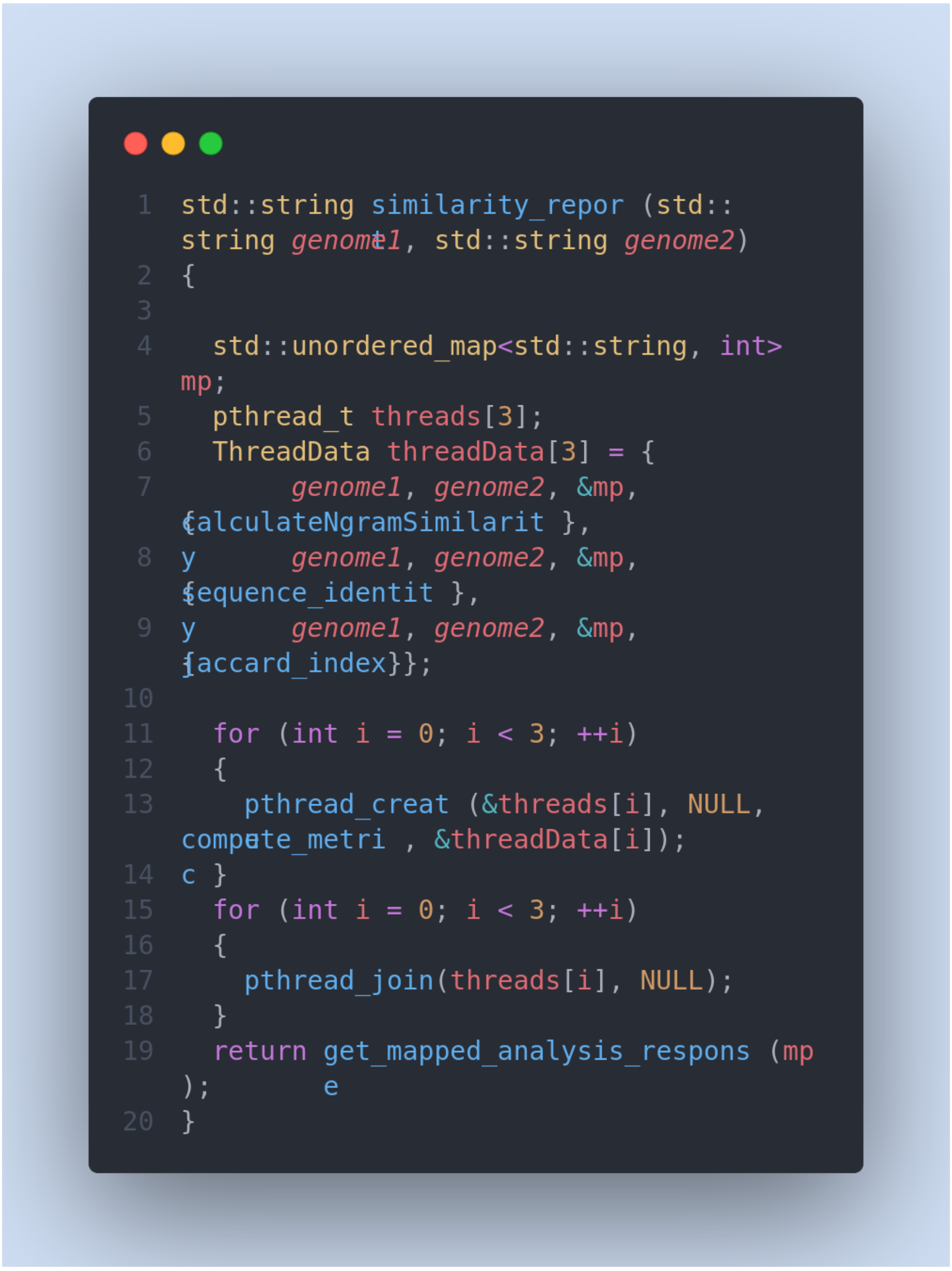
4.4 Methodology Overview

The algorithms described above collectively form the backbone of CNOME's genetic sequence analysis capabilities. By combining mutation detection, KMer analysis, and sequence similarity computation, CNOME provides a comprehensive toolkit for researchers and clinicians to analyze genetic data efficiently and accurately.



```cpp
std::unordered_map<std::string, int> KMer_analysis(std::string genome, int k)
{
  genome.erase(std::remove(genome.begin(), genome.end(), '\n'), genome.end());
  std::unordered_map<std::string, int> freq_mapping;
  if (!is_valid_genome(genome))
  {
    return freq_mapping;
  }
  std::string kmer;
  int n = genome.length();
  for (int i = 0; i < n - k; i++)
  {
    kmer = genome.substr(i, k);
    freq_mapping[kmer] += 1;
  }
  for (auto it = freq_mapping.begin(); it != freq_mapping.end();)
  {
    if (it->second <= 5)

{      it = freq_mapping.erase(it);

}    else

{      ++it;

} }
  freq_mapping["totalKmers"] = n - k + 1;
  if (n - k + 1 < 0)
  {
    freq_mapping["totalKmers"] = 0;
  }
  return freq_mapping;
}
```

Figure 4.2: Code implementation of the KMER analysis algorithm.



```cpp
std::string similarity_report(std::string genome1, std::string genome2)
{

  std::unordered_map<std::string, int> mp;
  pthread_t threads[3];
  ThreadData threadData[3] = {
        genome1, genome2, &mp, calculateNgramSimilarity},
        genome1, genome2, &mp, sequence_identity},
        genome1, genome2, &mp, jaccard_index}};

  for (int i = 0; i < 3; ++i)
  {
    pthread_create(&threads[i], NULL, compute_metric, &threadData[i]);
  }
  for (int i = 0; i < 3; ++i)
  {
    pthread_join(threads[i], NULL);
  }
  return get_mapped_analysis_response(mp);
}
```

Figure 4.3: Code implementation of the Similarity calculation algorithm.

Chapter 5

UML Diagrams

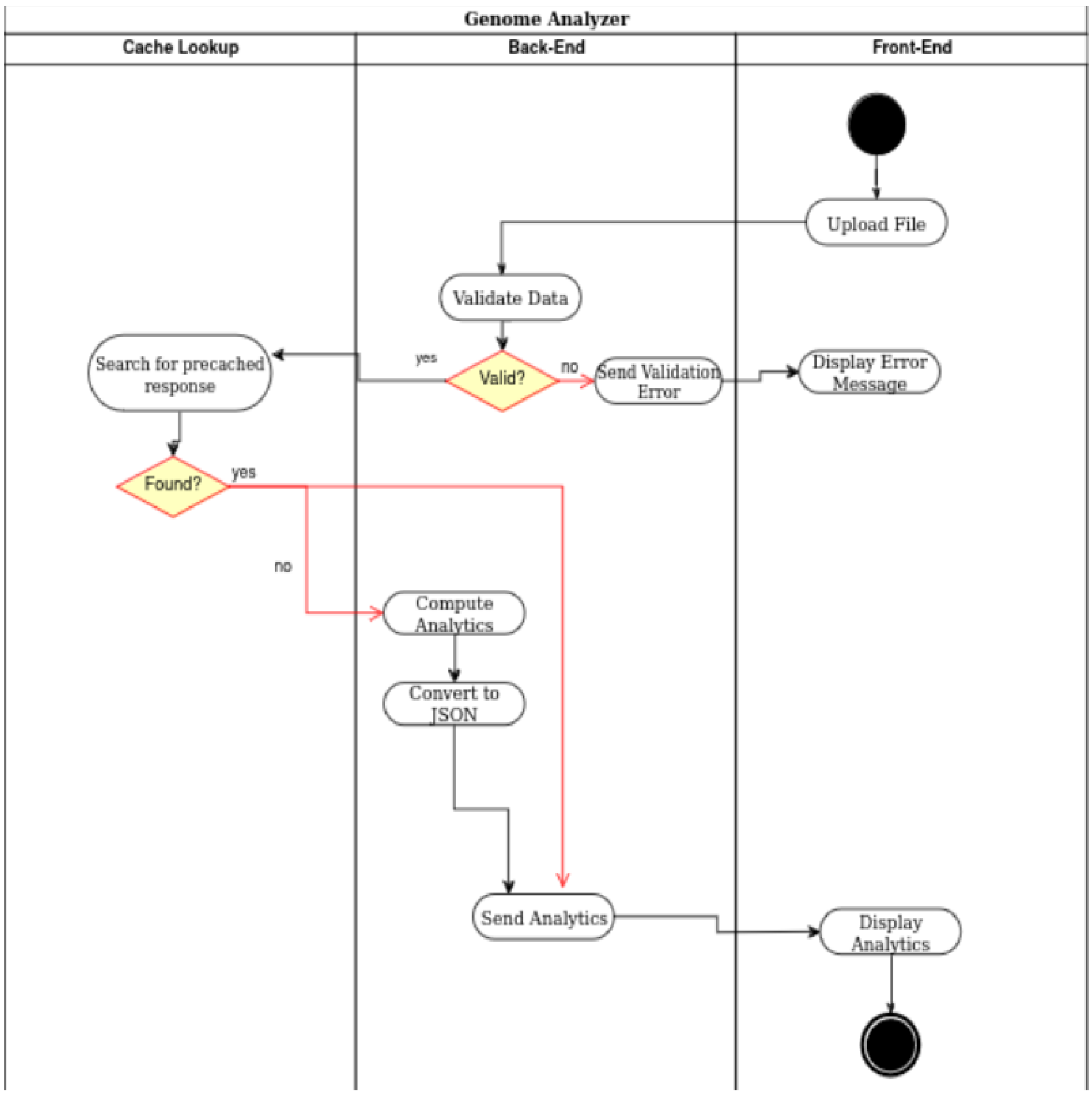


Figure 5.1: Swimlane diagram of the Project

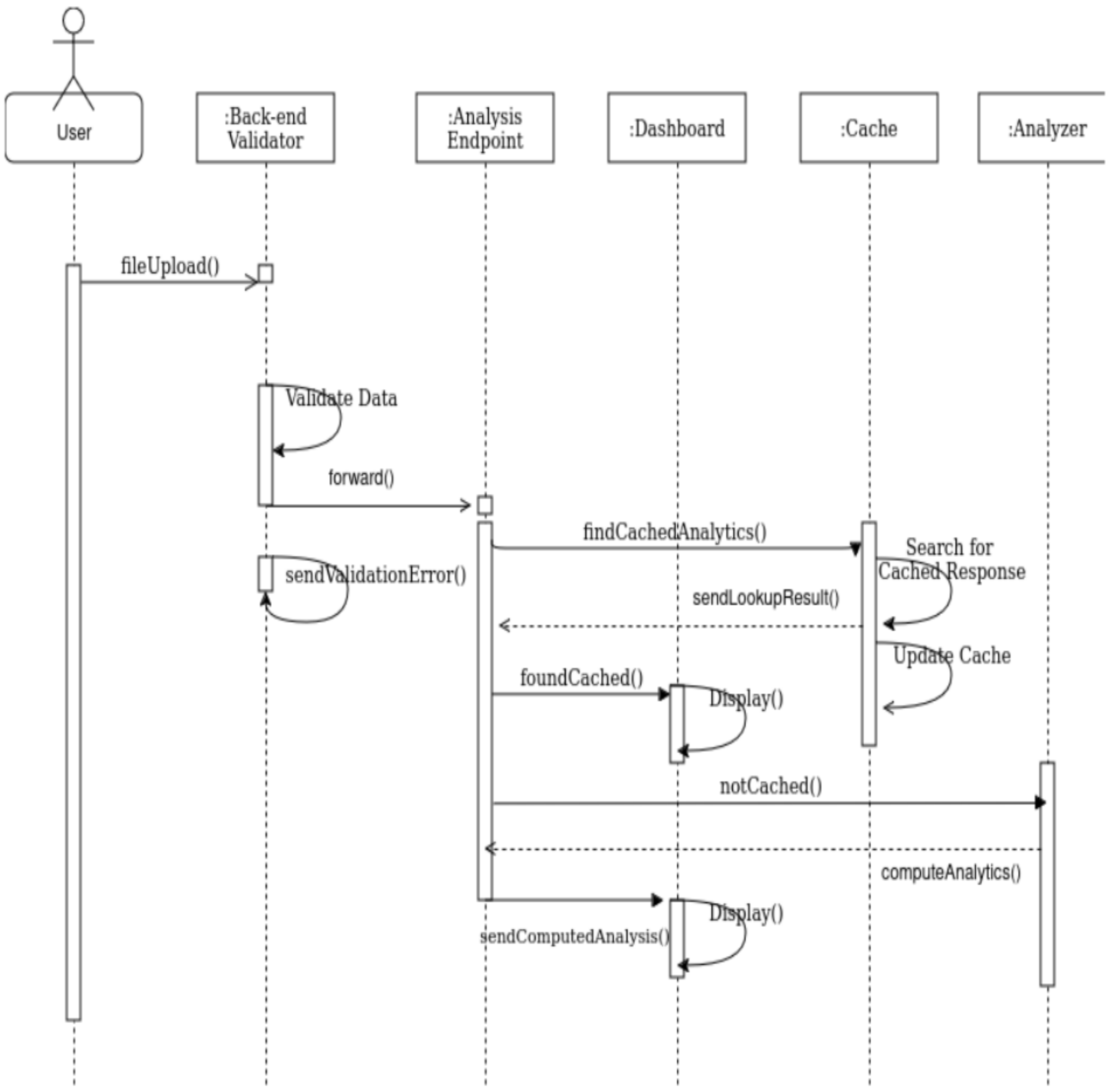


Figure 5.2: Sequence diagram of the Project

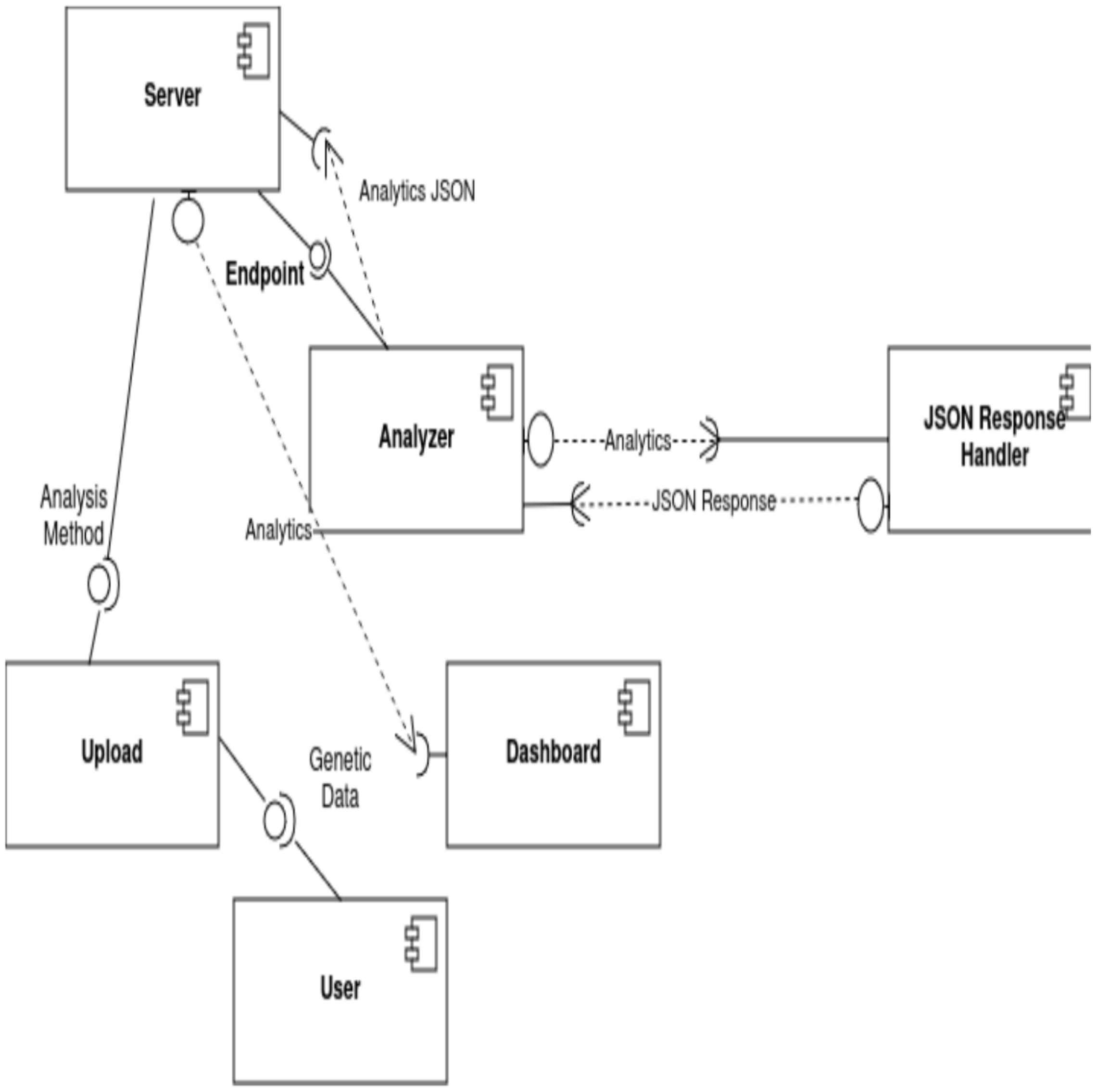


Figure 5.3: Component diagram of the Project

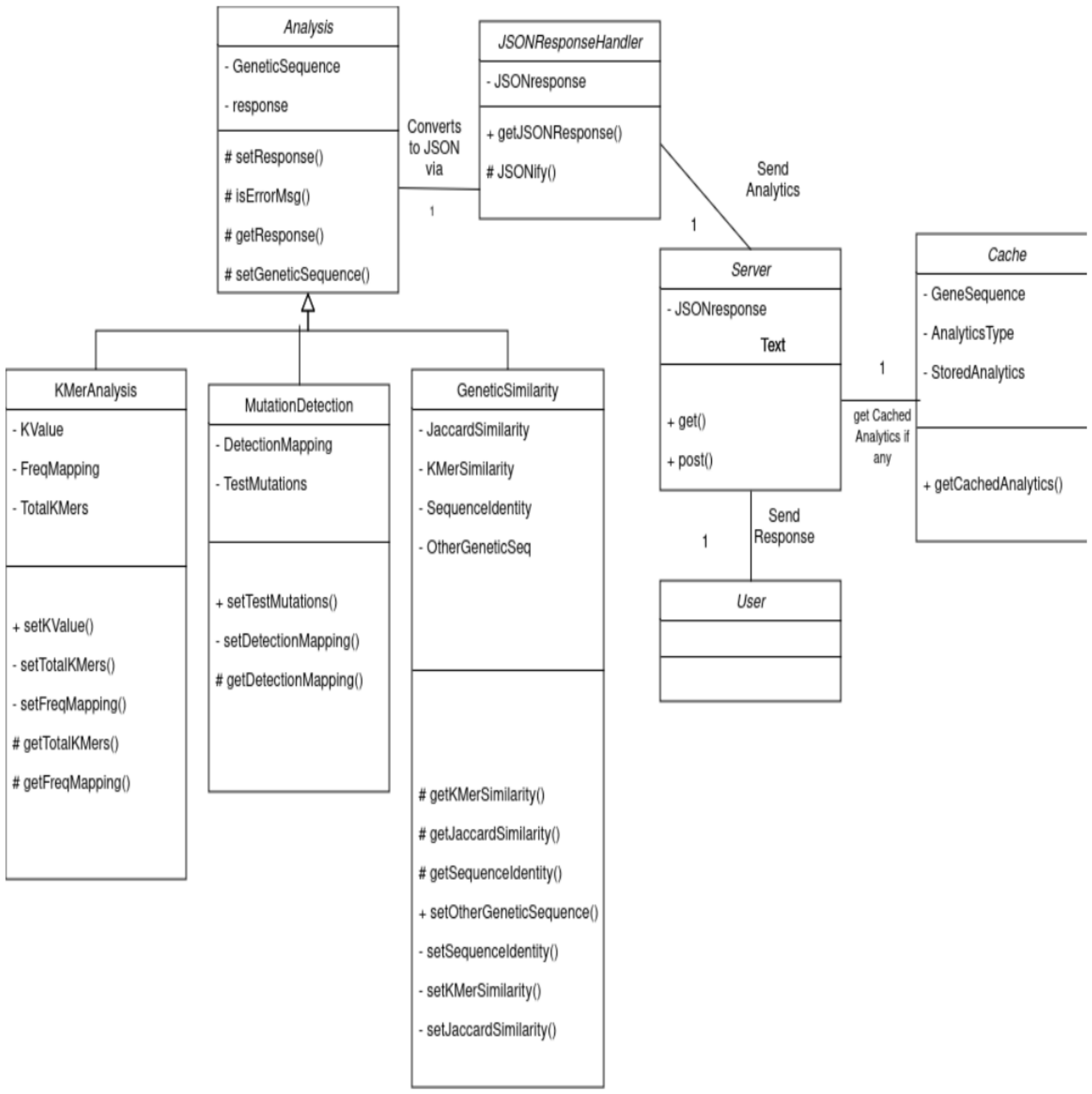


Figure 5.4: Class diagram of the Project

Chapter 6

Results and Conclusions

This chapter summarizes the key findings of the project and discusses the performance implications of using C++ and Javascript backends for genome analysis tasks.

6.1 Performance Analysis

The project investigated the performance of C++ and Javascript backends for three core functionalities: mutation detection, genome similarity analysis, and KMer analysis. Here's a breakdown of the observed performance trends:

Mutation Detection: Javascript exhibited faster execution times compared to C++. This can be attributed to the interpreted nature of Javascript, avoiding compilation overhead, and automatic memory management through garbage collection. These factors streamline string manipulation tasks often involved in mutation detection. Genome Similarity Analysis and KMer Analysis: C++ outperformed Javascript significantly for these tasks. C++ offers fine-grained control over memory, enabling optimizations like vectorization, which are crucial for computationally intensive sequence analysis algorithms. Additionally, mature C++ libraries specifically designed for bioinformatics leverage hardware-specific optimizations, leading to substantial performance gains over Javascript implementations.

6.2 Discussion

The choice of backend technology for genome analysis workflows should be guided by the specific task at hand. Javascript's ease of use and interpreted nature make it a potentially suitable option for simpler operations like mutation detection, where rapid development cycles and quick turnaround times are prioritized.

However, for computationally intensive tasks like genome similarity analysis and KMer analysis, C++ reigns supreme. Its control over memory, optimization opportunities, and established bioinformatics libraries empower it to handle these tasks significantly faster than Javascript.

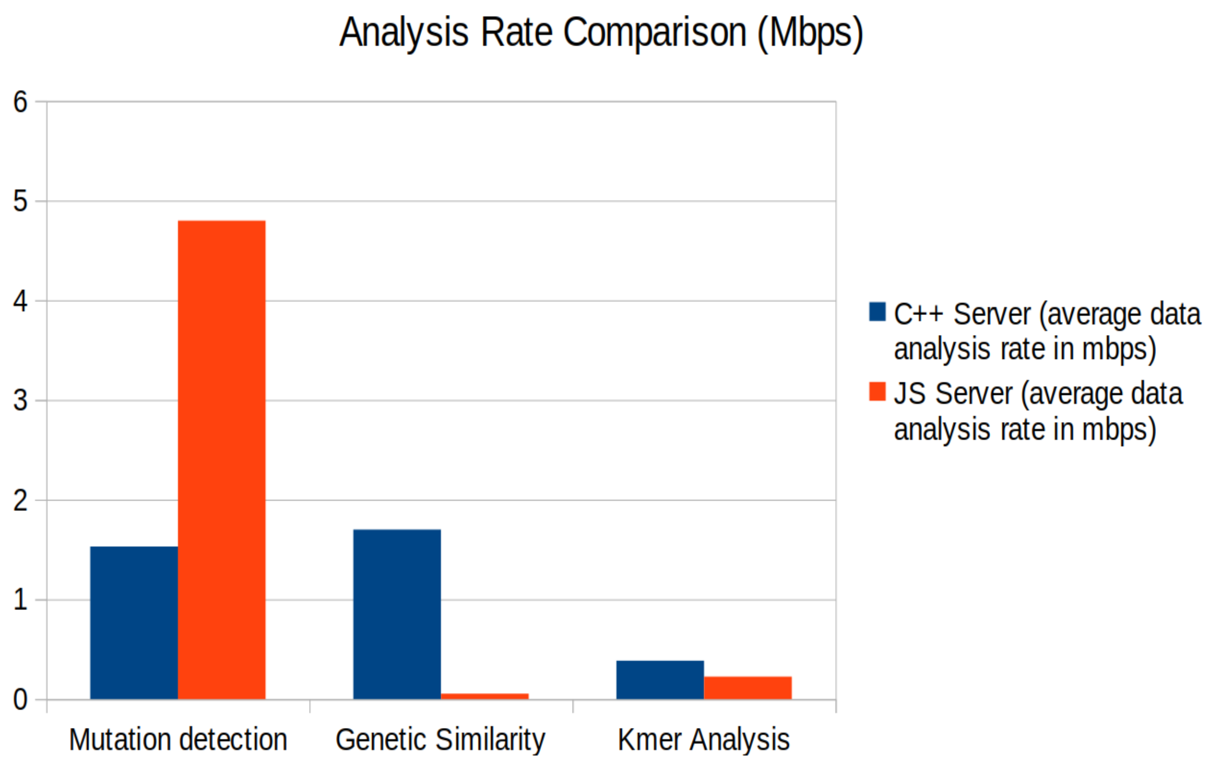
In conclusion, this project demonstrates the importance of considering both the programming language and available libraries when building high-performance genome analysis pipelines. While Javascript offers advantages for rapid prototyping and simpler tasks, C++ remains the language of choice for computationally intensive bioinformatics workflows.

6.3 Performance Data

Tables showing the response times observed for different file sizes and functionalities. The table shows the file size in Megabytes (Mb), the response time for the C++ backend in seconds, and the response time for the Javascript backend in seconds.

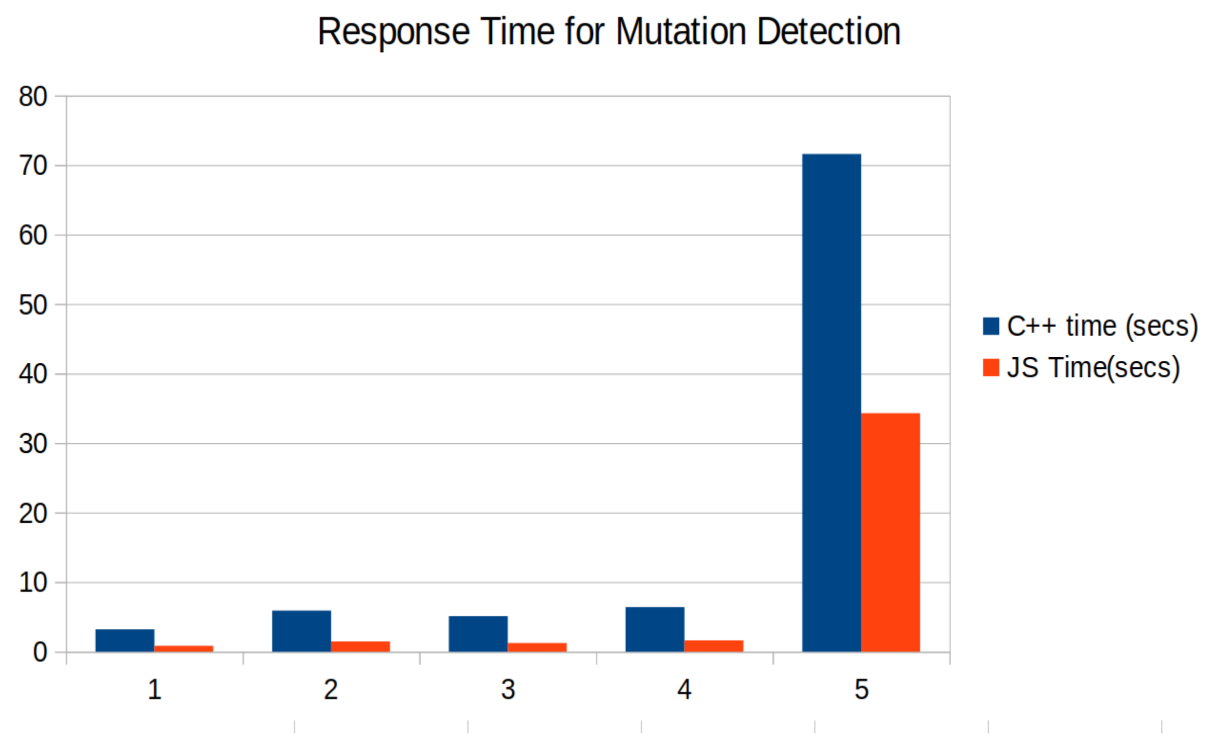
Average Rate Comparison

Analytics Method	C++ Server (average data analysis rate in mbps)	JS Server (average data analysis rate in mbps)
Mutation detection	1.53	4.8
Genetic Similarity	1.7	0.054
KMer Analysis	0.388	0.205



Response Times for Mutation Detection

Test case size (MB)	C++ response time (secs)	JS response time (secs)
6	3.2	0.84
15	5.9	1.46
18	6.4	1.62
103	71.6	34.3
10.2	5.1	1.24

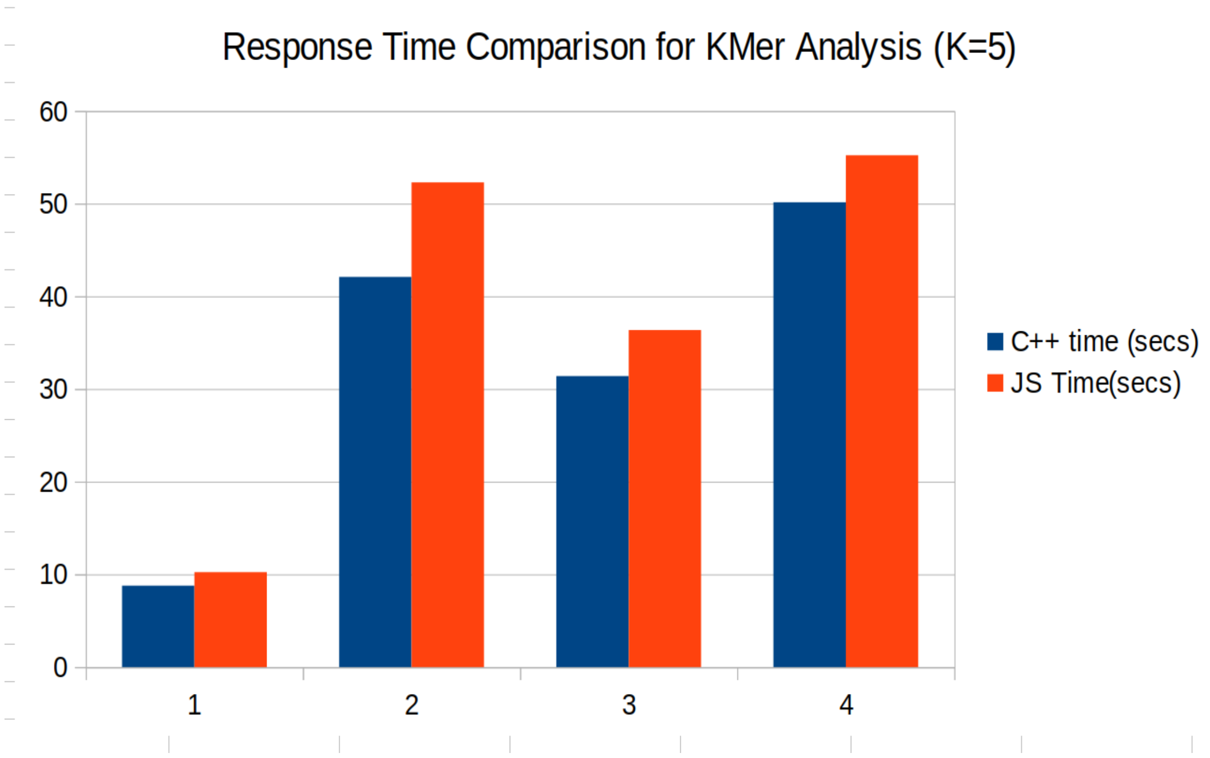


Response Time Comparison for KMer Analysis (K=5)

Test case size (MB)	C++ response time (secs)	JS response time (secs)
6	8.78	10.24
15	42.1	52.3
12	31.4	38.37
18	50.15	55.24

Response Time Comparison for Genetic Similarity Analysis

Test case size (MB)	C++ response time (secs)	JS response time (secs)
5	5.1	80.42
20	11.01	370.4
18	10.4	328.37
26	13.15	485.24



6.4 Future Work and Optimization Opportunities

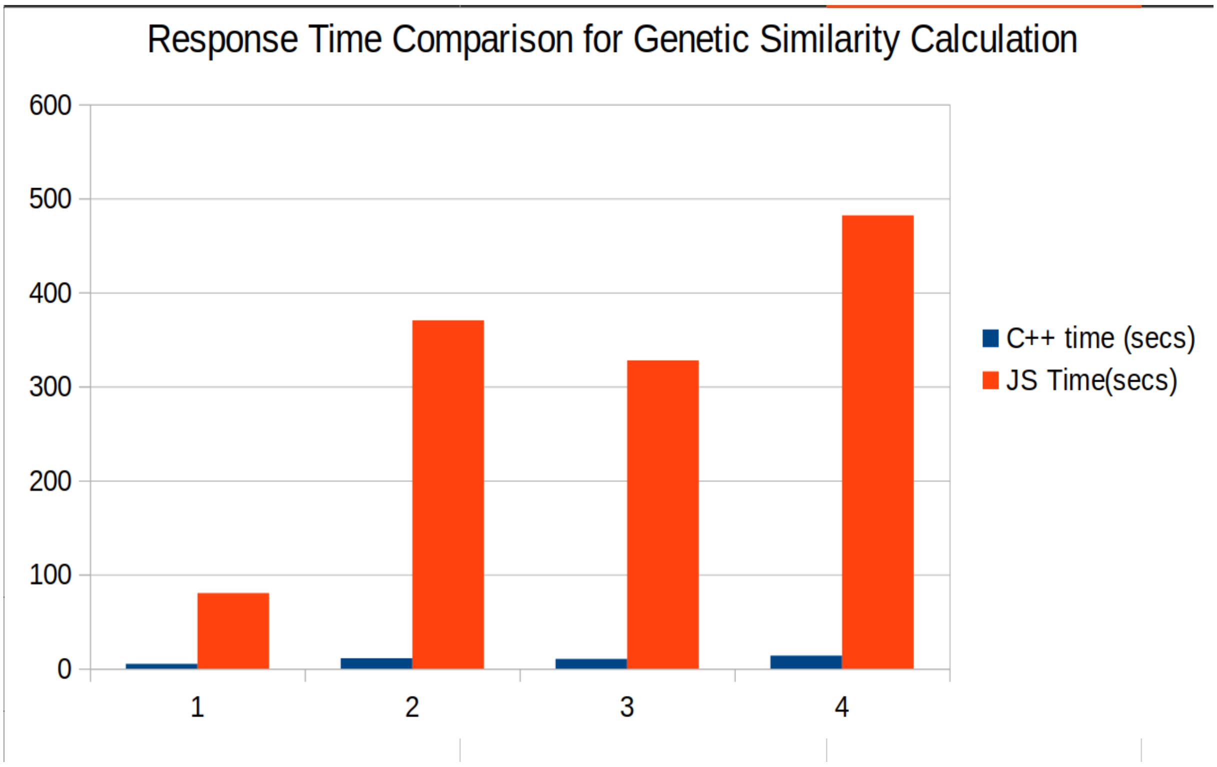
While the previous sections discussed optimizations achievable within the current framework, this section explores promising avenues for further performance enhancements in future iterations of the C++ backend. Due to time constraints or the need for deeper exploration, these optimizations might not be immediately implementable but hold significant potential.

1. Leveraging Language Interoperability

Investigating frameworks or techniques for interoperability between C++ and languages like Python or R can be highly beneficial. This approach allows exploiting the strengths of each language:

C++: Handles computationally intensive bioinformatics tasks within the backend due to its efficiency and fine-grained control.

Python/R: Maintains the user interface, scripting functionalities, and data visualization aspects, benefiting from their well-developed ecosystems in these areas.



Frameworks like Boost.Python or Rcpp can facilitate communication between C++ and these languages, enabling you to construct a hybrid backend that leverages the strengths of both worlds.

2. Exploring Alternative Programming Models

Delving into alternative programming models like functional programming or data parallelism using libraries like HPX or Kokkos presents an exciting opportunity. These models offer advantages such as:

Automatic vectorization: Improves performance by taking advantage of modern CPU instruction sets for parallel computations. Efficient memory management: Reduces overhead associated with manual memory management in C++.

While adopting these models might require a steeper learning curve compared to traditional C++ programming, the potential performance gains for specific bioinformatics algorithms can be significant.

3. Researching Emerging Hardware Trends

Staying abreast of advancements in hardware specifically designed for bioinformatics or emerging architectures like neuromorphic computing is crucial for future optimization strategies. These technologies might offer substantial performance improvements for specific genome analysis tasks in the coming years.

4. Integration with Cloud Platforms

Investigating how to integrate the C++ backend with cloud platforms like Google Cloud Platform (GCP) or Amazon Web Services (AWS) can be a game-changer for large-scale analysis. These platforms offer cloud-based infrastructure with high-performance computing resources like GPUs and TPUs.