AWSS

DNAs substring matching

We make biology easier and faster

Danilo Modica • Guglielmo Cassini • Alessandro Capici • Domenico Ragusa

June 4, 2022

Contents

1 Introduction

The purpose of this document is to provide a comprehensive description of the application, outlining its various use cases, behavior in certain situations, requirements met, constraints, and possible future developments.

1.1 Product purpose

The web application is designed in order to provide a service for computing matches between two DNA strings. It can be used by a general user who will have the possibility to perform the calculation by connecting directly from his own device to the service website. At that point he will be able to insert two DNA strings and start the calculation. Receiving an email notification afterwards, he can eventually download the result from the same website.

2 Requirements

2.1 Functional requirements

This section will outline the main functional requirements of the application, i.e., the features that must be implemented to meet the application specification.

2.1.1 User requirements

ID	**RF01**
Name	**New computation**
Definition	Customers launching the web application will have the ability to perform the calculation, given two strings of DNA, of a substring with the longest common base sequence
Affected by	Contents of the web application homepage
Actors	Customer

ID	**RF02**
Name	**FAQ & Contacts**
Definition	The site will make available to the customer a page dedicated to FAQ section, in which there are answers to frequently asked questions that may arise on the use of the services offered, also will provide a section for contact in case, due to problems, there is a need for use
Affected by	Contents of the web application homepage
Actors	Customer

2.1.2 System requirements

ID	**RF03**
Name	**Storage of results**
Definition	The results that are produced should be stored for future reference as well and must be identified by a unique code
Affected by	Type of the storage chosen and unique identifier generation system
Actors	System

2.2 Non-functional requirements

This section will set out the main non-functional requirements of the application, i.e., the constraints and rules through which the web application will be created.

2.2.1 Product requirements

ID	**RNF01**
Name	**Intuitivity**
Definition	The application should be intuitive, and its mode of use will be guessed by the customer within a minute of the access
Motivation	A customer might lose patience and thus give up on making a reservation should he or she interface with a web application that is unintuitive and therefore complicated to use for inexperienced users.

ID	**RNF02**
Name	**Portability**
Definition	The system will have to allow online use through devices clients with different operating systems
Motivation	Ensure access to the widest catchment area.

ID	**RNF03**
Name	**Continuity**
Definition	The system will be able to operate with an uptime of 99%.
Motivation	Ensure permanent service for clients

ID	**RNF04**
Name	**Automatic deletion of results**
Definition	A daily check will allow input and/or output files to be automatically deleted when they may be considered obsolete
Motivation	To avoid wasting resources

2.2.2 External requirements

ID	**RNF05**
Name	**Customer privacy**
Definition	The system will not store any information regarding the client

ID	**RNF06**
Name	**Legislative constraints**
Definition	The application must comply with legislative constraints regarding the protection of sensitive data

3 System models

This section will list and illustrate the models that formed the basis of the development of the AWSS web application by showing the scenarios in which a given actor interacts with the system.

3.1 Use cases

There are two use cases on which the application is based. These, illustrated in the figure 1 and described in the tables 1 and 2 are:

- **UC1D**: Computing the substring
- **UC1B**: On-demand download of previously calculated substrings

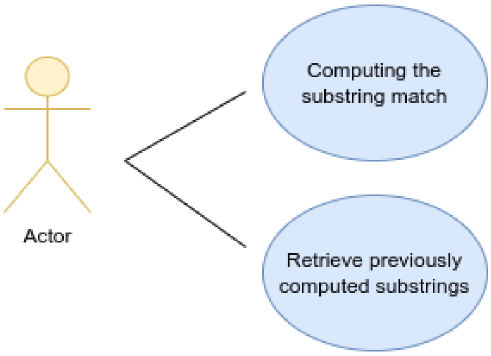

Actor
Computing the substring match
Retrieve previously computed substrings

Figure 1: Use cases UML diagram

Code	**UC1D**
Name	**Computing the substring**
Goal	Starting the calculation of a common substring between two DNA strings
Primary actor	Customer
Stakeholders and interests	**Customer**: wants to be able to calculate a DNA substring quickly and accurately by entering the DNA strings to be compared. He would like to be notified as soon as the execution of the algorithm is complete.
Preconditions	The customer must have a stable internet connection.
Main success scenario	• The client enters the application to begin processing. • The client enters the two starting DNA strings. • The customer specifies the email to which to send notification when finished. • The system loads the files and starts the processing.
Special requirements	• GUI text must be visible from at least three feet away. • The steps to follow to begin processing should be simple and intuitive.
Occurence frequency	High, almost continuous

Table 1: Description of use case UC1D

Code	**UC1B**
Name	**Retrieve previously computed substrings**
Goal	On-demand download of previously calculated substrings
Primary actor	Customer
Preconditions	A customer has successfully performed a processing.
Description	The customer accesses the system and has the ability to download any substring they have previously calculated within the 365-day time limit.

Table 2: Description of use case UC1B

3.2 Domain model

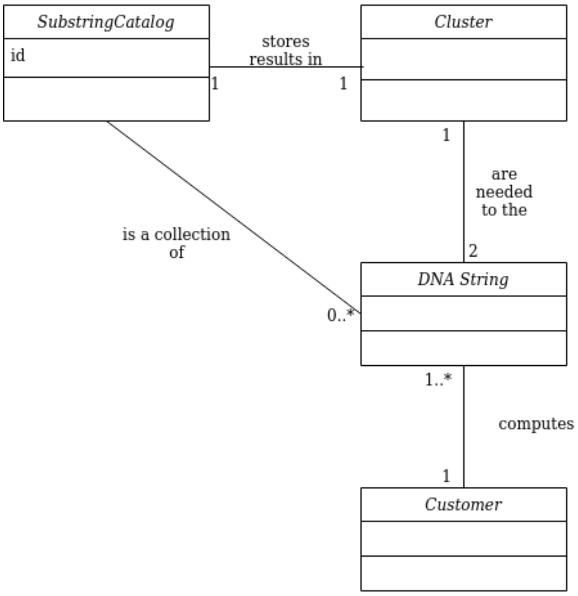


Figure 2: Domain model

4 Architecture

The system is based on the **Infrastructure as a Service** model and, in particular, two different layers can be distinguished within the architecture:

- **Presentation layer**
- **Data processing and management layer**

The management and processing data layer are implemented with the Amazon AWS cloud infrastructure whose architecture is shown in figure 3

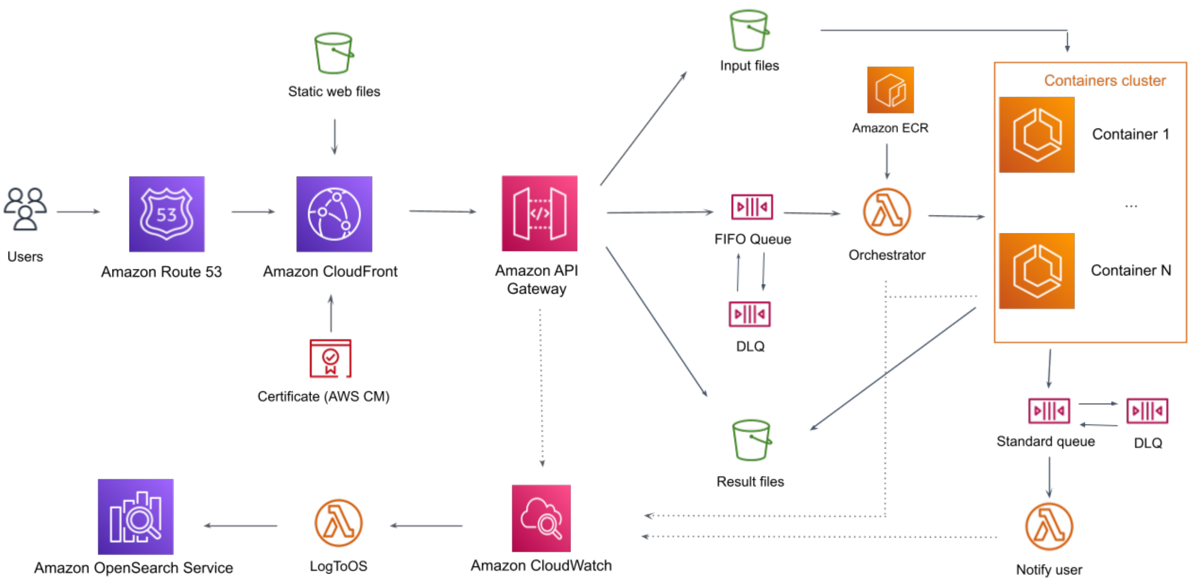


Figure 3: Amazon AWS Architecture

4.1 Amazon Route 53

To easily reach the web content, with an easy-to-remember name, a domain was registered on *Freenom.com* (having no credits available for direct registration within Route53). Then a *hosted zone* was built on the Route53 in order to store DNS records. The domain was then made to point to nameservers in the hosted zone. To avoid having to change the domain nameservers in the registrar website in case of infrastructure destruction and reconstruction, part of the hosted zone was created manually and not implemented through IaaC (hosted zone itself and SOA, certificate CNAME and NS records). What was instead written in Terraform is the code to add A and AAAA records pointing to CloudFront DNS and the CNAME alias for hostname www.
As explained later, there are also CNAME records in the hosted zone for validation and automatic renewal of SSL certificates.

In order to perform an availability check, a *health check* has been created. This executes each 30 seconds HTTP requests and in case of 5 consecutive failures triggers a Cloudwatch alert with notification on the developers' email through the SNS topic *R53-healthcheck*.

4.2 Amazon CloudFront and Certificate

Amazon CloudFront is a Content Distribution Network service. AWSS makes use of Cloudfront so that web content can be replicated in different locations and be accessed more quickly from different parts of the world, to do this the *best performance* option is used as well as the cache mechanism.

Instead, to ensure the security of the content, access to the origin bucket is strictly restricted to Cloudfront itself via *Origin Access Identities* and *Origin Shield*. There is also an SSL certificate so as to enable encryption via TLS 1.2.

Regarding certificates, one was required for both the main domain *awss-cloud.ga* and the sub domain www. It is automatically renewed due to the presence of CNAME records in the Route53 zone, as said above. In any case, should there be any problems, there is a Cloudwatch alert that warns if they are approaching the expiration date.

4.3 Amazon S3 Buckets

An S3 Bucket is a cloud storage resource available in AWS. It is simply a container of objects, similar to OS file folders. The architecture contains three buckets:

- **www.awss**
  This is a special S3 bucket in the architecture, in fact it hosts our static website, i.e. it contains all the website files (HTML, JavaScript, CSS, etc.). It exposes website files by using the bucket entry-point, but we do not directly use it since CloudFront and Route53 are configured, as explained in the previous sections.
  In fact, the bucket and its objects are not public, but files are accessible only by passing through CloudFront thanks to the configured Origin Access Identity (OAI). Furthermore, a CORS configuration policy is enabled. The website tree structure is the following:
  - *index.html*, the homepage of the website
  - *error.html*, the 404 error page, used when an user tries to access a non-existing page/file in the website.
  - *assets/*, the folder containing all files which are not HTML, organized in folders respectively: .js, .css, .png
- **awss-input-files**
  This bucket contains all the input DNA strings, in couples, uploaded by the users through the website homepage. It uses an accelerated endpoint for fast transfers (transfer acceleration property). All its objects are not public and the only way to upload an object into the bucket is to pass through the API Gateway and a generated signed URL, as explained in the following paragraph.

  To avoid keeping files in the bucket that would not be used in the future, a lifecycle rule expiration was added so that after 7 days of their creation automatically deletes the files from the bucket.
- **awss-result-files**
  This bucket contains all the result DNA matches, stored by the containers after performing the computation, as explained in the next sections. It uses an accelerated endpoint for fast transfers. All its objects are not public and the only way to download a file, again, is to pass through the API Gateway and a generated signed URL.
  As with the input bucket, a *lifecycle rule* was added here so as to save on storage on files that would be used only once or twice while still maintaining the availability of them. In this case, the *north-pole* policy says that after 30 days of their addition to the bucket the files are moved to *Standard-IA* class, after 60 days to *Glacier Flexible Retrieval* class, and finally after 365 days deleted from the bucket.

4.4 Amazon API Gateway

API Gateway generally takes all API requests from a client, determines which services are needed, and combines them into a unified, seamless experience for the user. The gateway acts as protector, enforcing security, scalability and high availability. In our architecture it has two purposes:

1. **Allows uploads/downloads to/from an S3 bucket**
   For this operations, the API Gateway URL must be followed by */bucket/filename*, where **bucket** will be the name of the S3 bucket where to upload/download a file and **filename** the name of the file.
   Amazon API Gateway allows to directly upload/download files to/from an S3 bucket only if that file's size is lower than 10MB.
   To avoid this problem, a signed URL is used: as shown in figure 4, when a user wants to upload a file to the S3 bucket, it firstly calls the API Gateway through an HTTP request (POST) with the objective of selecting the type of operation to perform (upload). The API Gateway forwards the request to an AWS Lambda,

written in Python, that generates an S3 signed URL using the Boto3 library and sends it back to the gateway. Finally, API gateway sends the signed URL back to the client that can use to directly upload a file to the S3 bucket. Same process applies to the download operation, with the only difference of using GET request instead of the POST one.

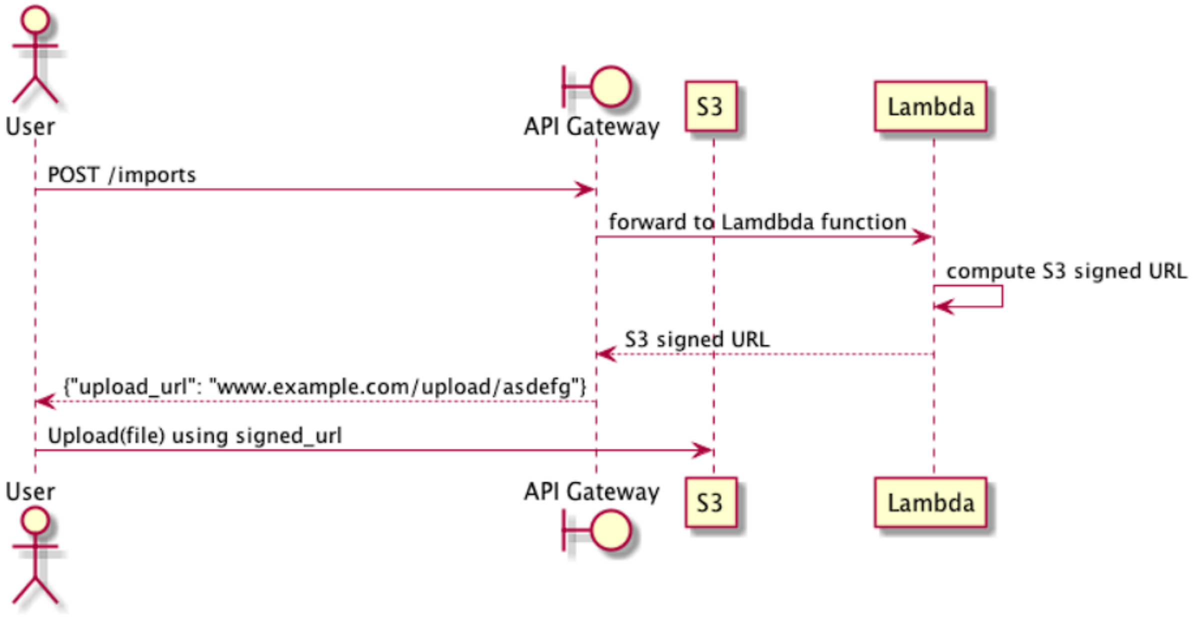

User
API Gateway
S3
Lambda
POST /imports
forward to Lamdbda function
compute S3 signed URL
S3 signed URL
{"upload_url": "www.example.com/upload/asdefg"}
Upload(file) using signed_url
User
API Gateway
S3
Lambda

Figure 4: Generate S3 signed URLs with Lambda through API Gateway

2. **Allow storing messages into an SQS queue**
   For this operation, the API Gateway URL must be followed by */sqs*.
   Simply performing a POST request to the API Gateway will store a message into the SQS FIFO queue. The message must be specified into the body of the HTTP request and it must satisfy the following JSON format:
   {"path1": "", "path2": "", "email": "", "size1": "", "size2": ""},
   as will be explained in the SQS section.

Throttling is enabled for any methods of the API Gateway, i.e. the gateway can only accept a certain **rate** (maximum number of requests that can occur within one second) and a certain **burst** (the maximum number of concurrent requests that API gateway will serve at any given point). Our gateway is configured with a rate of *50 req/s* and a burst of *500 requests*.
The API Gateway is also protected by the Web Application Firewall, that helps protecting the gateway by filtering and monitoring HTTP traffic between the gateway and the Internet. It consists of a set of Web Access Control Lists (ACLs), i.e. a set of rules, both managed by the Cloud provider AWS and the user. Our WAF is made of four rules:

- *IP Limiter*, created by us, to limit the maximum number of requests coming from the same source IP address to 100 requests every 5 minutes.
- *KnownBadInputsRuleSet*, managed by AWS, to block request patterns that are known to be invalid and are associated with exploitation or discovery of vulnerabilities.
- *AnonymousIpList*, managed by AWS, to block requests from services that allow obfuscation of viewer identity, e.g. VPN, TOR, proxies, etc.
- *AmazonIpReputationList*, managed by AWS, it is a group of rules that are based on Amazon threat intelligence, to block sources associated with bots or other threats.

4.5 Amazon SQS Queues

Amazon SQS is a fully managed message queuing service that enables you to decouple and scale micro-services, distributed systems, and server-less applications. It allows to send and receive messages between software components at any volume, without losing messages or requiring other services to be available. The architecture is made of two main queues, each one has its relative *Dead Letter Queue* (DLQ):

- **inputMsgQueue.fifo** FIFO Queue, contains messages that will be processed by the orchestrator AWS Lambda to run ECS cluster/containers. The message must be a JSON formatted message with the following parameters:
  - *path1*, bucket path of the first DNA file to use for the LCS algorithm.
  - *path2*, bucket path of the second DNA file to use for the LCS algorithm.
  - *email*, the email of the user, to be notified once the algorithm execution has been terminated.
  - *size1*, the size in Bytes of the first DNA file, to allow the selection of correct task definition, as will be described in the ECS section..
  - *size2*, the size in Bytes of the second DNA file.

  FIFO queuing is made so that the jobs that arrive first also have a chance to finish first. In case a message cannot be processed by the lambda executing ECS tasks, for more than two times, the message ends up in the DLQ *inputMsgQueue_DLQ.fifo*.

- **sendMail** Standard Queue, contains messages that will be processed by the sendMail AWS Lambda, to notify users via email. The message, which contains information to correctly notify the user in case of success or failure of his job, must be a JSON formatted message with the following parameters:
  - *mail*, the email of the user
  - *job_id*, the ID of the job, i.e. an identifier of the file containing the result of the LCS computation.
  - *message_type*, specifies whether the mail will contain a *success* message (**1**) or an *error* message (**0**).
  - *error_msg*, the error message, specified only if message_type is 0.

  Similar to the fifo queue, the same mechanism is also used here for DLQ, sending unprocessable messages there in the case of more than two failed processing attempts.

For both DLQs, in case there are more than 3 incoming messages in 45 minutes, the Cloudwatch alarm *fifoDLQ_alarm* or *sendMailDLQ_alarm* are triggered which in turn sends a notification to the SNS topic, *Default_CloudWatch_Alarms_Topic* by sending an email notification to the developers.

4.6 AWS Lambda

An AWS Lambda is a compute service that allows to run code, in several supported programming languages, without provisioning or managing servers. Our AWS infrastructure makes use of four lambdas:

- **urlSigner**, written in Python, allows to generate signed URLs (valid for five minutes) to upload or download objects from S3 buckets, thanks to the Python *Boto3* library. It is an event source mapping lambda, i.e. it reads from an event source and invokes an handler function in the lambda.
  This lambda is automatically triggered by the API Gateway whenever its correspondent API is called and it looks for the bucket name and filename passed through the event parameters. Furthermore, according to the type of HTTP request (GET/POST), which has been used by the user, it decides the type of operation for the S3 bucket objects (download/upload). If no error are encountered, it returns back to the API Gateway a JSON response containing the signed URL.

- **sendMail**, written in Python, allows to send emails to the users, using our AWSS Gmail account, thanks to the Python *smtplib* library. It is an event source mapping lambda automatically triggered by the SQS service whenever a new message is added to the *sendMail* queue. In this way the message delivered to the sendMail contains useful information to notify the user about his/her job requests.

- **runEcsTask**, written in Python, allows to run tasks on the ECS Fargate cluster, thanks to the Python *Boto3* library. It is an event source mapping lambda which is automatically triggered by the SQS service whenever a new message is added to the *inputMsg* queue.
  The handler function simply retrieves messages from the queue, checks the sizes of the files to be computed with LCS algorithm and decides which task definitions use for running a container on the cluster.

- **LogsToElasticSearch** is the classic lambda function that is created by AWS when an OpenSearch subscription to a Cloudwatch log group is generated from the console. In this case the code, in NodeJs, was reused by adding it to the architecture's IaaC.

This lambda is also event source which means that in the case of an event, it triggers and sends the Cloudwatch logs to Opensearch to be aggregated. In this case the trigger event is the generation of a message in a log stream of the log groups to which the lambda is subscribed.
Thanks to the lambda function, logs can be added and seen on OpenSearch in real time.
In case the lambda incurs in errors, a Cloudwatch alert is triggered by exceeding the threshold of 5 errors in 15 minutes and developers are notified by email.

4.7 Amazon ECR

Amazon Elastic Container Registry (Amazon ECR) is a repository for saving images that can be used to instantiate containers, through user interface on the AWS console or through code executed from other AWS components like a Lambda.
In our architecture we have one repository, called *lcs* for the container image, that will be described in the following subsection.

Thanks to the CICD pipeline a new image is generated each time the code is modified, in order to avoid an accumulation of useless images and thus a waste of money, a *lifecycle policy* has been associated with the repository which will go and delete the *untagged* images if more than two in number, in chronological order.

4.8 Containers with Docker image

In our architecture the components related to high computing have to be launched dynamically to satisfy the user demand, therefore we choose to use docker images, that are faster than EC2 instances to boot.
Our docker image is base on Ubuntu and it is setup to download latest libraries and to contain the main parallelized LCS algorithm.
After being created, it executes a Python script, called *run.py* that will:

1. Read from the *inputMsg* queue a message related to the process, and delete it from the queue.
2. Download from the *awss-input-files* S3 bucket the two input files.
3. Execute the compiled C algorithm.
4. Upload the result of the algorithm into the *awss-result-files* S3 bucket.
5. Add to the *sendMail* queue the message to signal that the process has been completed (successfully or not).

After executing correctly these steps, the container will automatically stop.
To perform all these steps, the Python scripts uses the Boto3 library to interact with the AWS components of the architecture.

4.9 Amazon ECS

Amazon Elastic Container Service (ECS) is a container orchestration service that helps to develop, handle e resize containerized applications. Our architecture uses a single Fargate cluster in which our images will be executed as tasks. Every time a message is sent into the input FIFO queue, the orchestrator lambda function (*runEcsTasks*) is triggered, and it uses a task definition, which changes the allocated computational power according to the workload, to instantiate a new container, pulling the image from the ECR and running the container that will execute the longest common sub-sequence algorithm.

4.10 OpenSearch and Cloudwatch

To keep track of the logs and eventually analyze them easily, Cloudwatch and OpenSearch are made use of. In particular, several services such as lambda functions, containers, and the API Gateway have their own log group on Cloudwatch in which they write log messages within log streams.
This categorization of logs combined with the difficulty in performing detailed searches led to the use of OpenSearch as a log aggregator (pattern *Log Aggregation*).

What is done is that in the log groups of interest, a subscription to OpenSearch is set, via the *LogsToElasticsearch* lambda (explained in Section 4.6), and this causes the log messages from those log groups to be sent in real time to the Opensearch cluster for easy analysis via the dashboard.

The OpenSearch domain, *awss-logs*, is designed to ensure performance and availability so it consists of a 2 availability zones with 1 data nodes of type *t3.small.search* for each zone. Each data node has 10GB of EBS General Purpose (SSD) storage type. The choice about the number of nodes and the type of machines was made taking into account the current user traffic and the fact that only the errors of the AWS services in use are logged through the filter pattern options of the Cloudwatch subscriptions, in case more extensive logging is planned nothing would prohibit changing the settings by adding more instances, even specialized ones and another availability zone. It would also be possible to add dedicated master nodes to increase the stability of the domain but for the current workload the cost/benefit ratio would not be favorable.

For the initial setup, the OpenSearch API via *curl* are used. However, since it is not yet documented and therefore impossible, to proceed with initial index creation go to *Stack Management* → *Index Patterns* → *Create index pattern* and entering in the first step and *@timestamp* in the second. To see the logs instead, simply go to the *Discover* page from the dashboard overview.

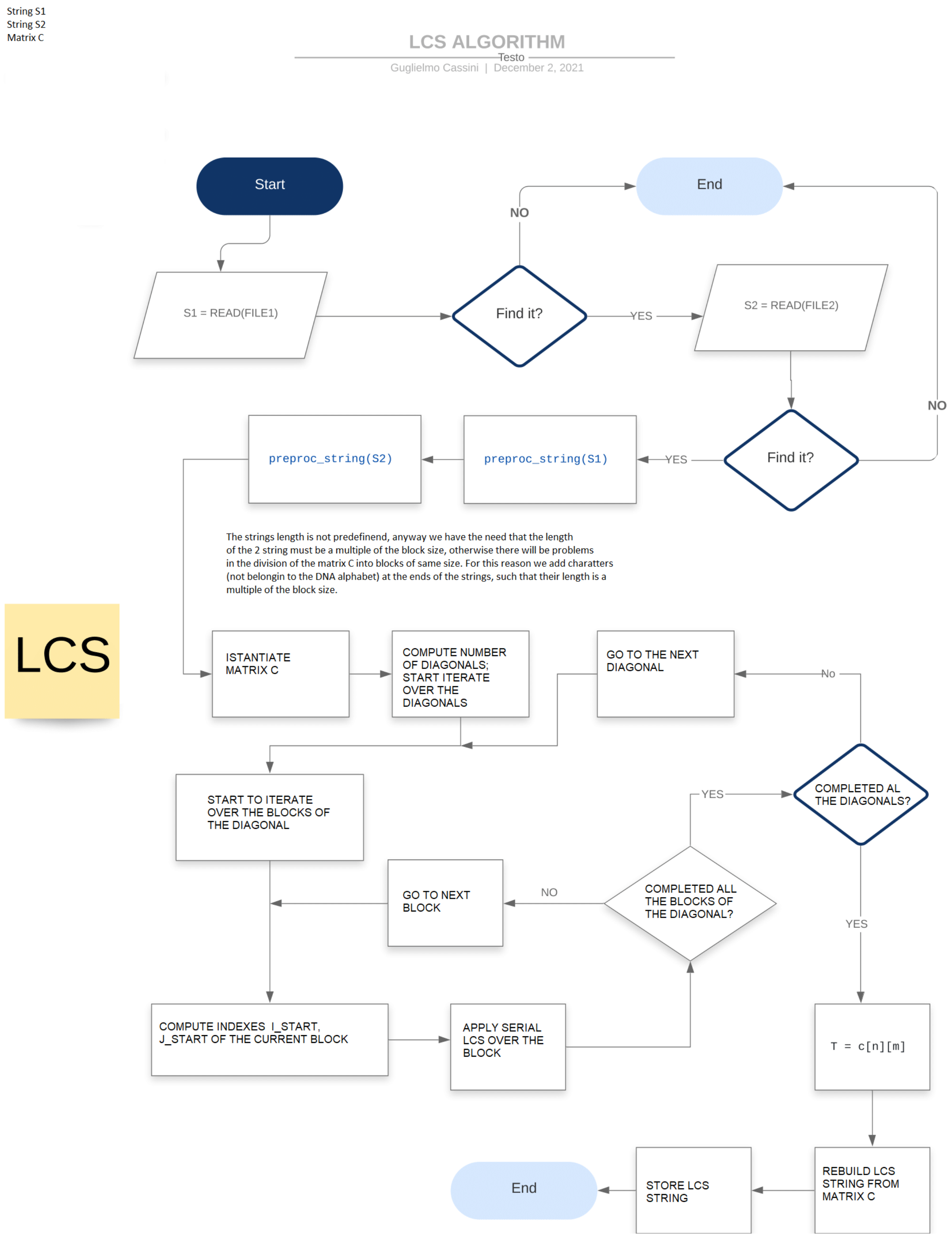
The basic software is *OpenSearch 1.2* and in case of updates, automatic scheduled maintenance is designed on Sundays at 9 a.m. for a duration of 3 hours.

Finally, in order to access the dashboard, public access with fine-grained control has been enabled, so to access the logs it is necessary to go to the dashboard, via the appropriate link, and login with the credentials choosen when building the cloud infrastructure.

Because of the associated costs, it was not deemed essential to enable and ingests access logs for Route53, Cloudfront, S3 bucket, and CloudTrail, which, in case, can be done with the same mechanism, eventually even increasing the power of the OpenSearch domain, explained above.

5 Core LCS Algorithm

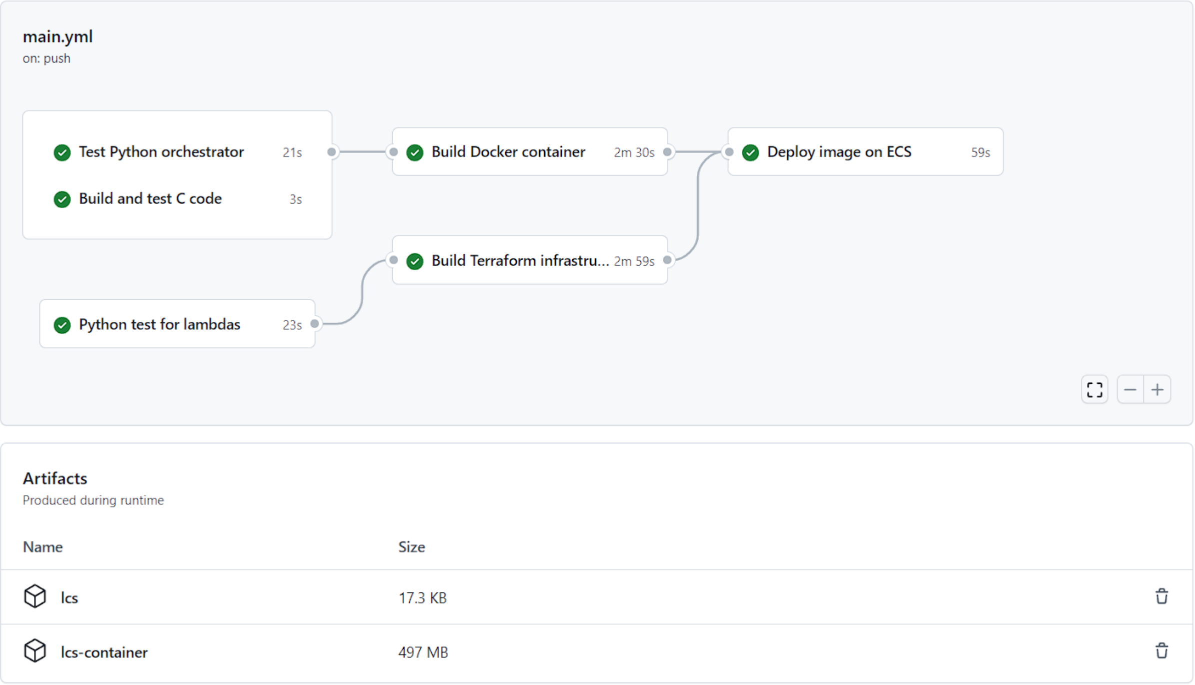
The longest common sub-sequence is a valid criteria to establish how much 2 or more strings of characters differ from each other. This criteria takes into account which are the characters appearing in the string and also their order of appearance.
This algorithm is often used in genomics and proteomics applications, because it allows to measure the similarity between two proteins or two genomics sequences. Its usage is shown in the paper: *Sequence alignment using Longest Common Sub-sequence algorithm*[1]
The algorithm can be easily defined in recursive way, but the algorithm is developed with dynamic programming techniques to speedup its recursive version reducing the number of computations. In its dynamic programming version, the algorithm needs to build a NxM matrix, where N is the length of the first string and M the length of the second string. The algorithm itself is shown in the flowchart in Figure 5 In our docker image we use a parallelization of this algorithm using OpenMP, that exploits the division in blocks of the matrix to enhance the code performances. The practical results was that the speedup grows from 1.9 to 5 using a single machine with 2 up to 24 cores. [2].
Unfortunately, due to the dynamic memory allocation via the malloc function and the inability to allocate more than 8GB of memory contiguously, so the algorithm can only process a maximum of about 2,070,000,000 DNA bases in total per execution. The solution to this problem could be implemented in the future as reported in the chapter *Future improvements*.


String S1
String S2
Matrix C
LCS ALGORITHM
Testo
Guglielmo Cassini | December 2, 2021
Start
End
NO
S1 = READ(FILE1)
Find it?
YES
S2 = READ(FILE2)
NO
preproc_string(S2)
preproc_string(S1)
YES
Find it?
LCS
ISTANTIATE MATRIX C
COMPUTE NUMBER OF DIAGONALS; START ITERATE OVER THE DIAGONALS
GO TO THE NEXT DIAGONAL
No
START TO ITERATE OVER THE BLOCKS OF THE DIAGONAL
YES
COMPLETED AL THE DIAGONALS?
GO TO NEXT BLOCK
NO
COMPLETED ALL THE BLOCKS OF THE DIAGONAL?
YES
COMPUTE INDEXES I_START, J_START OF THE CURRENT BLOCK
APPLY SERIAL LCS OVER THE BLOCK
T = c[n][m]
End
STORE LCS STRING
REBUILD LCS STRING FROM MATRIX C

The strings length is not predefinend, anyway we have the need that the length of the 2 string must be a multiple of the block size, otherwise there will be problems in the division of the matrix C into blocks of same size. For this reason we add charatters (not belongin to the DNA alphabet) at the ends of the strings, such that their length is a multiple of the block size.

Figure 5: Flowchart of the LCS Algorithm

6 CI/CD


main.yml
on: push
Test Python orchestrator 21s
Build and test C code 3s
Build Docker container 2m 30s
Deploy image on ECS 59s
Build Terraform infrastru... 2m 59s
Python test for lambdas 23s

Artifacts
Produced during runtime

Name	Size
lcs	17.3 KB
lcs-container	497 MB

Figure 6: GitHub Actions Workflow

The pipeline for Continuous Integration / Continuous Delivery was built to have two branches that end with updating the Docker image in the ECR repository.

- In the first branch it starts by running the tests for the Python script that is responsible for checking files with DNA strings and running the LCS algorithm, in particulare *flake8* for lint and *Pytest* for the actual tests. In parallel, another job is responsible for the compilation of the executable, made via *Makefile*, and for testing via a simple bash script. If both of these operations are successful then the Docker image can be built in the next step.

- The second branch, on the other hand, takes care of updating the cloud infrastructure, first by linting the Python codes for lambdas and then testing them via the Pytest library. If the job is successful it proceeds to check the Terraform code via *fmt* command and then to update the infrastructure.

To save time these two branches are run in parallel and converge to the deployment of the Docker image on the ECR repository, *Deploy image on ECS* step, (this step in fact needs the infrastructure and Docker image both updated).

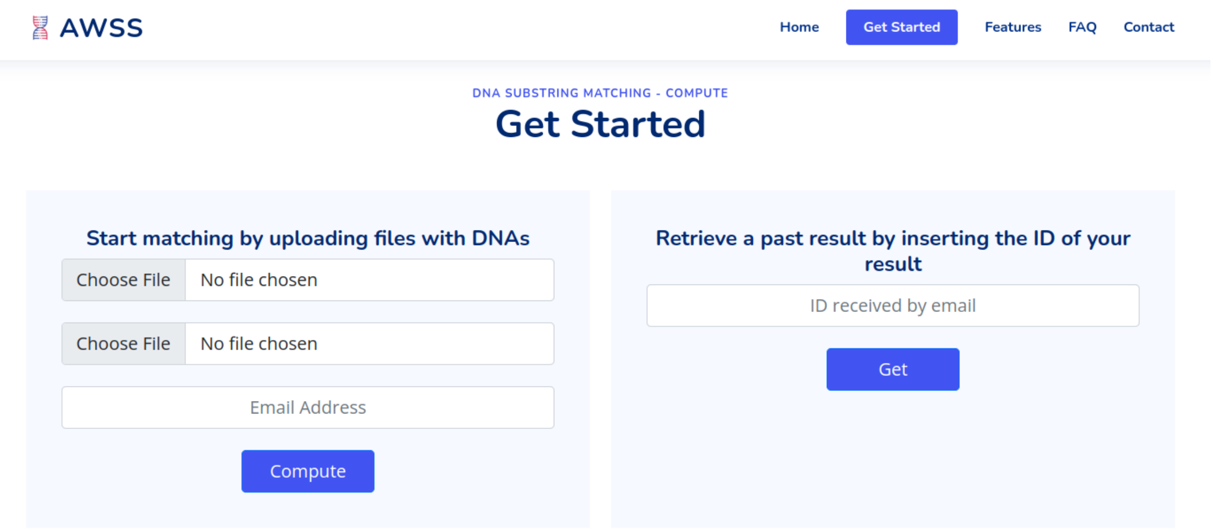
To pass intermediate results from one job to another artifacts are created, specifically the *lcs* executable, in the *Build and test C code* job, to be used in the creation of the Docker image, and *lcs-container* which is the Docker image created in this step to be used later in the deployment.

In the case of Github, since the repository is public, there is no limit to the space used by artifacts nor even a limit to the execution time of the pipeline.
To ensure security, *Dependabot* has been enabled, which is responsible for checking dependencies in the yml code and suggesting any updates to them in pull requests repository section.

7 Web interface

The web interface, based on Bootstrap, was developed on a single page to improve usability and efficiency in pursuing the user's goal.
In particular, it consists of four sections:

- **Get started**
  In the first part of the section it is possible to select the two files with the DNA strings to load for processing and to specify the email to which send a notification in case of success/failure in the computation. In the second part it is possible to insert the unique code associated to a result, previously sent by email, in order to download the result of the associated computation.
  In both cases, if the user enters invalid inputs, an error is shown inviting the user to correct the data entered. An error message is also displayed if the request cannot proceed due to an internal error.



Start matching by uploading files with DNAs

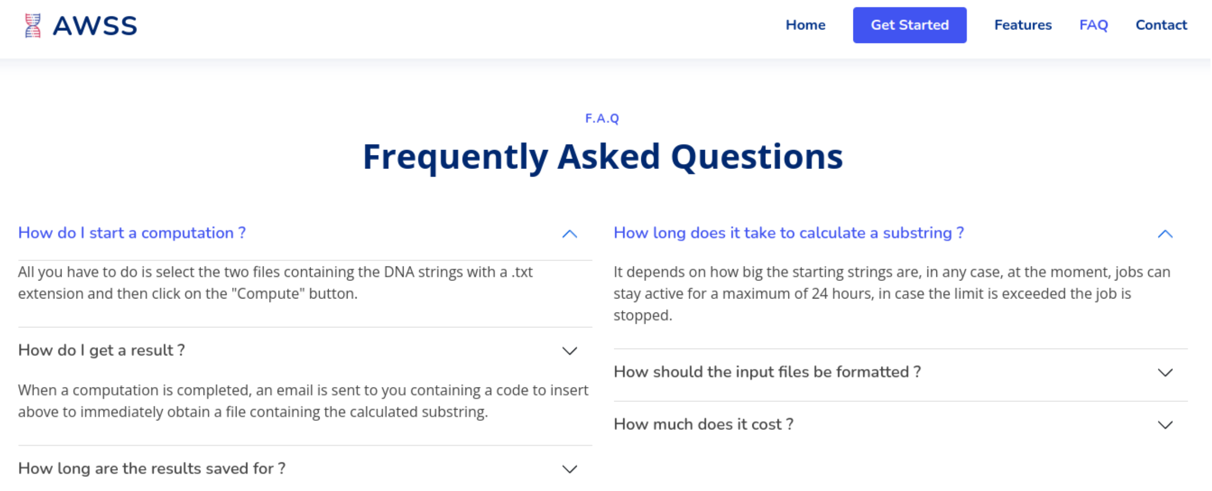
No file chosen

No file chosen

Retrieve a past result by inserting the ID of your result

Figure 7: Main section of the interface for interacting with the app

- **Features**
  This section simply highlights the main features of AWSS webapp.
- **FAQ**
  This is the section used to clarify the most common doubts of the user with the questions that are often asked.



Frequently Asked Questions

How do I start a computation ?

All you have to do is select the two files containing the DNA strings with a .txt extension and then click on the "Compute" button.

How do I get a result ?

When a computation is completed, an email is sent to you containing a code to insert above to immediately obtain a file containing the calculated substring.

How long are the results saved for ?

How long does it take to calculate a substring ?

It depends on how big the starting strings are, in any case, at the moment, jobs can stay active for a maximum of 24 hours, in case the limit is exceeded the job is stopped.

How should the input files be formatted ?

How much does it cost ?

Figure 8: FAQ section of the interface

- **Contacts**
  The contact section contains the main means of contact between the developers and the user in case of problems or to give feedback.

In case of a 404 error, the site displays a user-friendly page notifying it and prompting the user to return to the homepage.

8 Future improvements

Possible future developments of the application include:

- Design of a system for authentication so that the user can easily find all the processes in execution and completed in a restricted area with its results.
- Implementation of a system for counting the resources spent in computation and related payment system for authenticated users.
- Implement multipart download/upload so that data upload/download is more resilient to network issues and faster due to parallelization.
- Modify the algorithm to become distributed so that it can distribute the workload over multiple machines, speeding up data processing and allowing the ability to handle larger workloads without problems due to memory allocation.

9 License

10 References

[1] Commonlounge, *Sequence alignment using Longest Common Subsequence*

[2] G. Cassini, I. Schimperna, *Substring matching with applications to proteomics and genomics*