Enrique Martínez Martínez

A serverless platform for metabolomics data analysis

Final Degree Project

directed by Dr. Pedro García López

Computer Engineering

Tarragona
September 2022

Resum
El camp de la recerca està en constant evolució, amb noves tecnologies i processos sent desenvolupats. Això resulta en un increment dels requisits d'emmagatzematge i computació, i els laboratoris necessiten actualitzar les seves solucions cares i estacions de treball. Amb la computació al núvol, els investigadors poden pujar dades des de qualsevol lloc directament al núvol, on després poden ser analitzats, reduint costs de *hardware* i només pagant per aquells recursos que realment utilitzen. Tot i així, una gran quantitat de programari està desenvolupat amb la màquina d'escriptori en ment i no és apte per ser executat al núvol. Això habitualment significa que l'aplicació s'ha d'empaquetar en un contenidor, a més de dissenyar i implementar una arquitectura apropiada, per exemple una basada en principis *serverless.* A més a més, per tal que l'aplicació sigui usable sense necessitar eines basades en línia de comandes, cal desenvolupar un *frontend* (per exemple un basat en tecnologies web).

En aquest projecte, es migrarà una aplicació d'aquestes característiques desenvolupada en R i C++.

Resumen
El campo de la investigación está en constante evolución, con nuevas tecnologías y procesos siendo desarrollados. Esto resulta en un incremeneto de los requisitos de almacenamiento y computación, y los laboratorios necesitan actualizar sus caras soluciones y estaciones de trabajo. Con la computación en la nube, los investigadores pueden subir datos desde cualquier lugar directamente a la nube, donde luego pueden ser analizados, reduciendo costes de *hardware* y sólo pagando por aquellos recursos que utilizan. Sin embargo, una gran cantidad de *software* está escrito con la máquina de escritorio tradicional en mente y no es apto para ejecutar en la nube. Esto habitualmente significa que la aplicación debe ser empaquetada en un contenedor, además de diseñar e implementar una arquitectura apropiada, por ejemplo una basada en principios *serverless.* Además, para que la aplicación sea usable sin requerir herramientas de línea de comandos, se debe desarrollar un *frontend* (por ejemplo, uno basado en tecnologías web).

En este proyecto, se migrará una aplicación de estas características desarrollada en R y C++.

Abstract
Research is constantly evolving, with new technologies and processes being developed. This also results in the requirements for storage and compute resources increasing, and research labs need to upgrade expensive storage solutions and workstations. With cloud computing,

researchers anywhere can ingest these large datasets directly into the cloud and analyze them there, lowering hardware expenses and only paying for the resources they use. However, a lot of software is written with the desktop in mind and unsuitable to run as-is on the cloud. This usually means that the application will need to be containerized, and some work needs to be done to design and implement an appropiate architecture, for example one based on serverless principles. Furthermore, in order for the application to be usable without the need for command line tools, a frontend must be developed (for example, a web-based one).

In this project, we'll be migrating such an application written in R and C++.

Contents

1 Introduction **1**
1.1 Project goals 1

2 Background **2**
2.1 The application 2
2.2 Cloud computing 3
2.3 AWS Batch 4
2.4 The imzML file format 5
2.5 Storage 6

3 Proposed architecture **7**
3.1 Building the code on the cloud 8

4 Changes performed to the original code **9**
4.1 Reading and writing to S3 9
4.2 Converting multithreaded code to run on FaaS 12
4.2.1 ThreadingMsiProc 12
4.2.2 Running a job: calling R code from Python (and vice-versa) 13
4.2.3 Invoking Lambda functions and reading .ibd in parallel 15
4.2.4 Adapting the processing functions 16

5 Performance evaluation **21**

6 Developing the frontend **24**
6.1 About AWS Amplify 25
6.2 Login page 26
6.3 File manager page 26
6.4 Submit job page 28
6.5 Job Metadata API 31
6.5.1 Creating the GraphQL Schema 31
6.5.2 Adding a IAM-authorized mutation with a custom resolver 32
6.6 My jobs page 35
6.7 Job status page 35

7 Conclusion **36**

List of Figures

1 Scheme of imzML structure. [1] 6
2 Proposed architecture 8
3 Step 1: enqueuing blocks. 11
4 Step 2: reading from the current block. 11
5 Execution time comparison. 22
6 Frontend architecture. 24
7 The login page. 26
8 The file manager page. 27
9 Submit job: first step. 28
10 Submit job: second step (not all parameters are shown). 29
11 Submit job: third step. 29
12 Process flow of a unit resolver (left) and a pipeline resolver (right). [2] 33
13 My jobs page. 35
14 Job status page. 36

Code Index

1 Common API for reading and writing files. 10
2 WorkerMsiProc header file. 12
3 Running a job in an isolated Python process. 13
4 Defining a function that can be called from R. 14
5 JobRunner 15
6 MergeTree struct with the macro added. 16
7 RedisSave and RedisMerge for CommonMassAxis. 17
8 RedisSave for MTInternalRef. 18
9 RedisMerge for MTInternalRef. 18
10 Storing the offsets. 20
11 Payload sent to the job submission lambda. 30
12 Job type in the schema. 31
13 adminCreateJob mutation and input type. 32
14 Resolver mapping template (request). 34

List of Tables

1 Experiment results broken down by step. 23

1 Introduction

In this final degree project we will be migrating a bioinformatics application to the cloud. The project consists of two parts: a backend and a frontend. The backend part will focus on the changes to the package's code and designing a cloud architecture that adapts to the demand and reduces costs when the service is not in use.

The frontend part will focus on the cloud services used and backend integration. The proposed solution will target minimising costs, while delivering a service that can scale to a large number of users.

This project has been a collaboration between CloudLab (URV) and members of the MIL@b (URV) research groups.

1.1 Project goals

The goals for this project are:

- Study and optimize a bioinformatics application for its migration to a cloud platform.
  - Identify the project's requirements.
  - Design an appropiate architecture for processing large files.
  - Adapt the code to use cloud object storage instead of a local filesystem, while maintaining support for both options.
  - Abstract filesystem operations with a common interface for object storage and local filesystems.
  - Parallelize data input (and output when possible).
  - Use object storage features (byte-range reads) to support reading parts of an object and parallelize reads.
  - Use object storage features (multipart uploads) to parallelize writes to a single object, while keeping format-specific metadata coherent to the data written.
  - Adapt multithreaded code to take full advantage of cloud services (serverless functions) using a cloud-agnostic toolkit (Lithops), keeping support for running the code locally. This will have to be done in a case-by-case basis, although some techniques can be reused.
  - As part of the abovementioned adaptations, an alternative to shared memory will be required.
  - Minimize changes to the original code where possible.
  - Run the infrastructure on ARM-based CPUs for a better price/performance based on available cloud offerings.

    - Run the builds on the cloud, producing a multi-architecture container image.

- Develop an easy to use web frontend with a robust user authentication system.
    - In order for the frontend to easily scale and avoid fixed server costs, we will use serverless resources and other task-specific services when possible (for example, user account management and hosting).
    - The frontend will have file management capabilities, including uploads and downloads. A file permission model will be enforced to ensure users can't access unauthorized files.
    - The frontend will be used to submit jobs to be executed by the backend also developed in this project.

2 Background

2.1 The application

Mass spectrometry is an analytical technique that determines the molecular mass of compounds while also identifying and quantifying the compounds under analysis [3]. Mass spectrometry imaging (MSI) is a technique that uses mass spectrometry to visualize and localize analytes in the sample [4]. There is a wide range of uses, since it can measure many kinds of analytes such as peptides, proteins, metabolites or chemical compounds in a large variety of samples (like cells or tissues) [5]. For example, it can be used in pharmacokinetic studies, biomarker discovery and more. A biomarker, put simply, is something that can be objectively measured that is a sign of a normal or pathogenic process [6].

MSI experiments produce large amounts of data that need to be processed. Such tools are available from instrument manufacturers, but they're often very expensive and closed-source, limiting its usefulness [7]. In contrast, the rMSI2 package developed by Pere Ràfols is open-source and licensed under a GPLv3 license. This is not the only available open-source package, but a differentiating point is that it provides a GUI that lets the user browse for files on their computer, tune the processing settings and also visualize the data.

Since it is an R package, scientists can use other R packages if needed to further analyze or plot the data. The R ecosystem is widely used by scientists, data analysts, statisticians and more. There are lots of useful open-source packages for a wide range of uses, like making plots with ggplot2 or Plotly, web applications with Shiny and implementations of most statistical functions or other algorithms you would need.

Like other high-performance R packages, rMSI2 is at its core written in C++ and it uses threads to speed up the processing and take full advantage of the user's CPU. The Rcpp package is used to map R types to C++. Another feature of rMSI2 is its optimized memory model. MS images use a large amount of memory, as the rMSI2 paper describes:

> MS images use a large amount of memory since they contain a large collection of spectra. The memory needed could range from hundreds of megabytes to several tens of gigabytes, depending on the m/z resolution and the total number of acquired pixels. [7]

For this reason, rMSI2 only loads parts of the data as it needs them, allowing it to process larger-than-memory datasets.

2.2 Cloud computing

Organizations of every type, size, and industry are using the cloud for a wide variety of cases, such as big data analytics, data backup and customer-facing web applications [8]. With cloud computing, instead of using their own servers and other infrastructure (known as on-premises), they rent them from a cloud service provider, like Amazon Web Services, Microsoft Azure, Google Cloud or IBM Cloud. This lets companies save on upfront costs such as hardware, and also ongoing maintenance costs (backups, patches, hardware maintenance, etc.).

The cloud service provider (CSP) we will be using in this project is Amazon Web Services (AWS). Although each CSP will have a different name for these services, some basic ones available on any of them are:

- **Virtual machines (EC2)**. Used to run virtual machines, also called "instances". There are a variety of instance types to choose from, depending on the application's requirements. There are general-purpose instances that provide a great balance of CPU and Memory, compute-optimized instances for tasks that heavily lean on CPU and don't need as much memory, memory-optimized for the opposite case, and other specific purpose instances featuring GPUs for machine learning, media transcoding or other workloads that can benefit from GPU acceleration.
- **Object storage (S3)**. Service for storing files of any size (objects). Objects are stored in a bucket. The main benefits of object storage are its scalability: there's no need to worry about performance even with a large amount of requests; and pricing model, which is based on the size of the object and the time it stays stored (once you delete an object, you're not charged for it anymore). In addition, there's different tiers with cheaper pricing for infrequently accessed objects, suitable for data archival. There are also no ingress charges, however there are egress charges for transfers outside the bucket's region. There is also support for storing custom metadata with objects. Since the service is accessed over HTTP, objects can be accessed with a web browser (public objects or presigned URLs).
- **Permissions (IAM)**. Used to specify who or what can access services and resources. For example, you might define a role that lets your virtual machine read and write to

object storage. You should always try to apply the principle of least privilege: only grant the minimal privileges required for the task.

- **Serverless functions (Lambda)**. Also known as Function as a Service (FaaS). Run code (functions) without worrying about the underlying infrastructure. You can choose the memory limit available per function (more memory is more expensive), higher memory configurations can use more vCPU.
- **Virtual Private Cloud (VPC)**. Used to create virtual networks for interconnecting AWS resources.

2.3 AWS Batch

Batch is a service that plans, schedules and executes jobs running on AWS compute resources. Jobs aren't immediately started, they're placed on a queue and the scheduler decides when to run them (for example, based on the priority, submission order, dependencies on other jobs, etc.).

- **Job definition**. Describes the requirements of the job, for example what service it will run on (Fargate, which runs containers or EC2), what command to execute, the scheduling priority, resource requirements (CPU, RAM, GPU...), whether it should be automatically retried if it fails, and a timeout. These are just some of the parameters that can be specified. Additionally, even though a job definition is required when submitting a job, a lot of the parameters can be overriden if needed (for example, submit a job that has a higher priority than the default for that job definition).
- **Compute environment**. Defines the compute resources that can be used, for example, for a EC2 compute envinroment we can manually pick the instance types that can be used, the AMI (Amazon Machine Image, the OS Image), VPC and subnets (), minimum and maximum vCPUs. These limits let us choose if we want to keep the envinroment warm even when there is no workload or scale all the way down to avoid being billed for unused resources, and limit how many.
- **Job queue**. Jobs are submitted to a job queue where they reside until they can be scheduled to run in a compute environment [9]. A job queue has one or more compute environments associated, ordered by preference. Queues also have a priority that determines how Batch places jobs in the compute environments when there is more than one queue associated to them.

2.4 The imzML file format

The imzML format was developed to exchange mass spectrometry imaging data in a flexible and efficient way between different instruments and data analysis software, without relying on proprietary formats [10]. It uses two separate files: an XML file for the metadata (with the .imzML extension) and a binary (with the .ibd extension) file for the mass spectral data. As a note, the XML part tries to be as close as possible to the mzML format, this is one of the design goals so it can be easily interchangeable with mzML.

The imzML file can contain the following data (this is not an exhaustive list):

- UUID: A Universally Unique Identifier (as described in RFC 4122). The UUID in the imzML and the ibd file must match (in the ibd file, the UUID is the first 16 bytes).
- Parameters of the instrument used and other experiment metadata.
- MD5 or SHA1 of the ibd file.
- Binary data type: describes how values are stored in the binary file. The imzML binary format allows signed integers (8 bit, 16 bit or 32 bit) or floating point (IEEE 754), using either 32 bit single precision or 64 bit double precision. Can be specified for each data array separately.
- Binary mode: continuous or processed. With continuous data, the m/z array is stored only once in the binary file.
- Metadata about the pixels in the image and offsets to locate the data stored in the .ibd file.

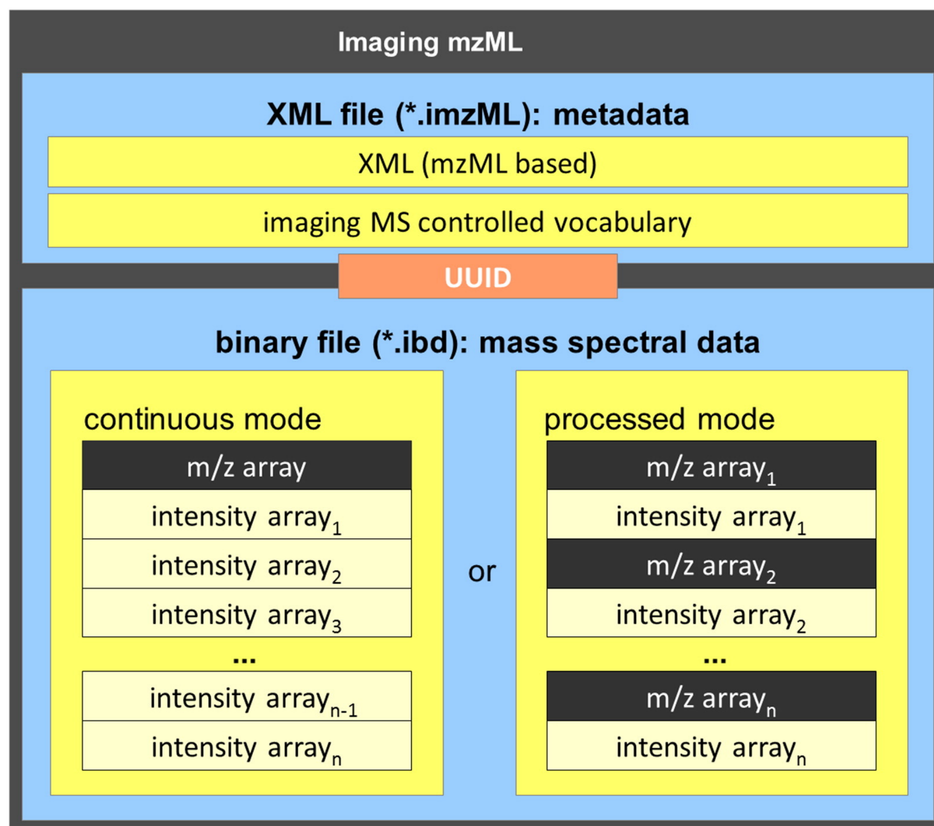

Figure 1: Scheme of imzML structure. [1]

In summary the imzML format is very flexible, it allows describing the details about how an experiment was carried out and it is possible to locate and load spectra from the ibd file without reading the whole file sequentially.

2.5 Storage

AWS has different offerings for storing data, each suited to different use cases.

- **Elastic Block Storage**: It serves the same purpose as a hard drive does in a regular computer. EBS volumes are used as boot volumes, but an instance can have more than one volume attached (for example, a boot volume and a data volume). A volume can only be attached to one instance at a time.
- **Elastic File System**: EFS works like Network Attached Storage systems, so an EFS volume can be mounted on more than one instance at a time. It also grows and shrinks automatically as files are added or removed.
- **Simple Storage Service (S3)**: See 2.2 Cloud computing.

- **Instance store**: Some EC2 instances provide storage that is physically attached to the host computer [11]. Depending on the instance, this might be one or more HDDs, SSDs or NVMe SSDs. It is usually used for low-latency access to temporary data, like caches or scratch data. Another use case is data that is replicated across a fleet of EC2 instances. It is important to mention that the data is lost if the instance is stopped or hibernated (but not if rebooted).

For our use case, Amazon S3 is the best choice. Users will upload their files directly to S3 using a web browser, and the application can stream the files to avoid needless copies.

3 Proposed architecture

As mentioned, S3 will be used for storage. S3 is a distributed system, so we can benefit from parallelizing reads and writes when possible. This is in contrast to the current paradigm, which is single-threaded I/O. Lambda workers will be used to run some of the processing steps, and Redis is used to store any results that need to be combined with others (reduce), as well as storing file offsets when workers write a .ibd file using multipart upload.

The instances are spun up by AWS Batch to run the task. Batch manages scaling up and down according to the demand, this lets us scale as the number of jobs to run increases. They run a custom AMI with Redis and RabbitMQ preinstalled and ready to be used, based on the ECS-optimized Amazon Linux 2 AMI. The reason to use the ECS-optimized AMI as a base is because it has the ECS agent preinstalled, which is necessary for an AMI to be used with Batch since the latter uses ECS to run containerized workloads.

We'll be using ARM instances based on Amazon's Graviton processors. The reason is that they offer a better price/performance ratio compared to their x86 counterparts, as can be seen in different software benchmarks [12] [13]. Lambda is 20% cheaper given the same configuration when compared to x86 [14], but it is not available in all regions yet.

A diagram of the proposed architecture is provided below:

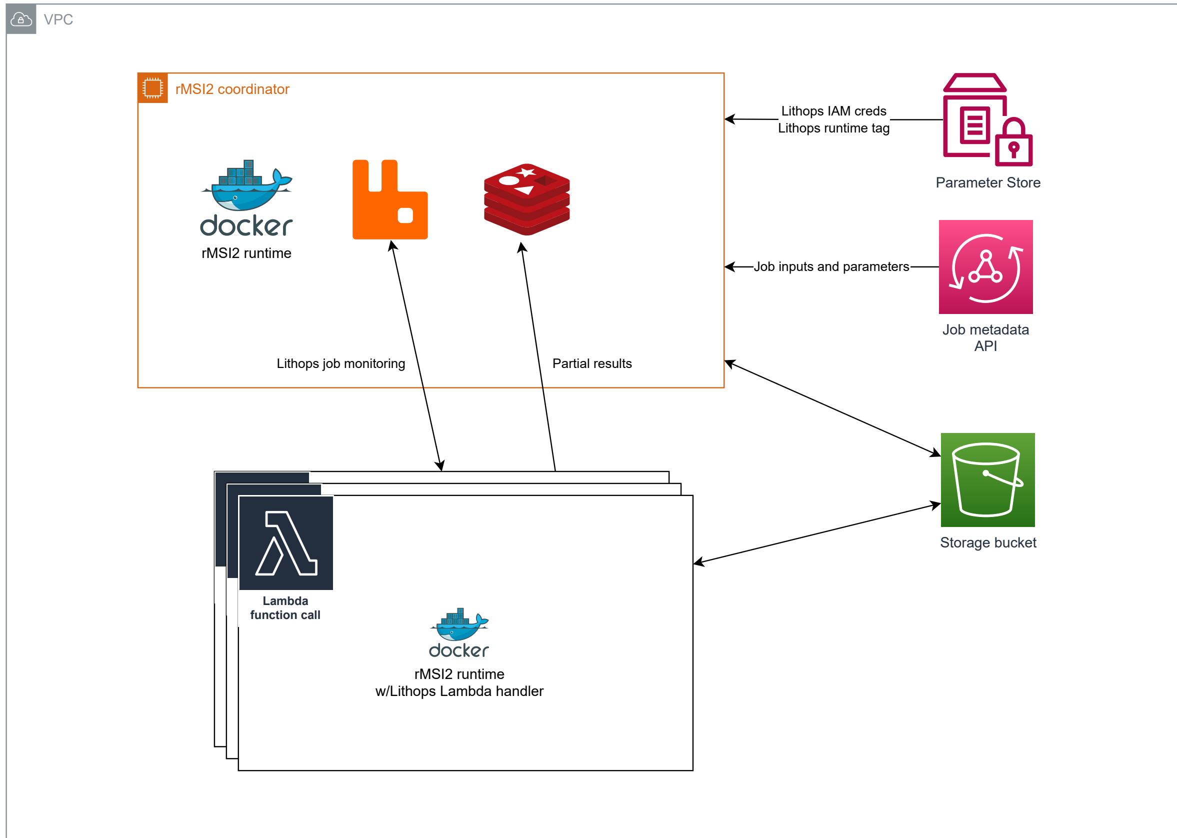

Figure 2: Proposed architecture

3.1 Building the code on the cloud

Since containers are architecture-specific, we need to build them for the platform we're targeting (in this case, arm64). However, most desktop and laptop computers we use are x86-based. Although it is possible to build a Docker image for another platform using `docker buildx`, it is not very convenient since it uses QEMU [15] which results in a very slow build time, as it has to run non-native code (unlike a cross compiler which would run natively but generate code for another platform).

With our source code stored on AWS CodeCommit, Amazon's hosted Git service, we can run our builds on the cloud with AWS CodeBuild, this allows us to use ARM and x86 instances to compile for each architecture.

Build steps are specified in a `buildspec.yml` file. With the "batch builds" feature we can define more than one build and how they run. We define two builds for the rMSI2 runtime, one per architecture. Another build task that depends on the previous two builds creates a Docker manifest list containing the SHA256 hash for each platform's image, tags it with the first seven characters of the commit's hash and pushes it to ECR (Amazon's Docker Hub equivalent). This is used in multi-architecture Docker images so that `docker pull` can choose

the appropiate variant for the architecture and OS the host is running.

Once the image is built and pushed to ECR, the AWS CLI can be used to retag it as `latest`, so that newly enqueued jobs will run using that image. The image used for the Lambda workers is stored on another ECR repository and built at the same time from the same commit, with Lithops' Lambda handler code added to it. To change the runtime to be used by Lithops, set the corresponding parameter in Parameter store to the image tag.

4 Changes performed to the original code

4.1 Reading and writing to S3

Since it is developed with traditional filesystems in mind, rMSI uses `std::fstream` to perform file I/O. However, as we mentioned earlier, we want to be able to read and write to S3 directly. The first attempts were to use a FUSE-based solution, like s3fs-fuse or goofys. This allowed mounting the S3 bucket as a directory, and the application could transparently access it like any other path in the filesystem. However, the shortcomings were far too great. The performance was bad (especially for writing), even when tuning the available parameters. Also, there is currently no way to use FUSE on Lambda, since our code runs in a container and there is no way to enable the `SYS_ADMIN` capability or access the device file `/dev/fuse`. This limitation also applies to AWS Fargate, which at the time of writing only supports adding the `SYS_PTRACE` kernel capability [16]. It was clear that another approach would be necessary.

The approach taken was to integrate the AWS SDK for C++, and perform the requests to S3 ourselves. In order to minimize the changes required to the read/write logic used by rMSI, we need a similar API to the one it is already using (functions like `open`, `read`, `write`, `seek` and `tell`).

The following code shows this API, with a few functions specific to rMSI and uploads.

```cpp
class FileLike
{
public:
    virtual ~FileLike() {};
    virtual void Open(const char *filename, std::ios_base::openmode mode) = 0;
    virtual bool IsOpen() = 0;
    virtual void Close() = 0;
    virtual void Read(char *buff, std::streamsize count) = 0;
    virtual void Write(const char *buff, std::streamsize count) = 0;
    virtual void Seekg(std::streampos pos) = 0;
    virtual std::streampos Tellg() = 0;
    virtual void Seekp(std::streampos pos) = 0;
    virtual std::streampos Tellp() = 0;
    virtual bool IsEof() = 0;
    virtual bool IsFail() = 0;
    virtual bool IsBad() = 0;
    virtual void FinishUpload(std::string etagKey) = 0;
    virtual void ReadUUID(char *uuid) = 0;
    virtual void WriteUUID(const char *uuid) = 0;
};
```

Code 1: Common API for reading and writing files.

When developing the implementation, the following goals were set:

- **Streaming data without writing to disk**: One of the goals of this implementation is to not need to write any data to disk, to avoid bottlenecks.

- **Reading parts of the file/seeking**: We want to be able to skip to an offset in the file and read from there, which will allow workers to download only the data they need. This will be achieved implementing the `seek()` function, keeping track of the current file offset and making byte-range requets to S3.

- **Buffering**: When loading data, rMSI calls `read()` often and loads a small amount of data each time. Without buffering, there would be a lot of overhead due to the amount of HTTP requests that would be needed. Instead, we'll make a large request and keep the data in a buffer, so subsequent requests will read the data from the buffer.

The implementation of S3 reads works in the following way. The file is logically split in blocks of a fixed size. A read request will fetch the whole block at the current reading offset, and it will also start transferring the next blocks asynchronously (prefetching). Subsequent reads will keep consuming the block that was previously fetched. Once there is no data left in that block, it will be discarded and replaced by the next one, an operation that may block or not depending on whether the prefetch has finished yet. As an example, if the block size is configured to 16MB and we keep three blocks in the queue, a read for the first byte in the file will fetch the following blocks:

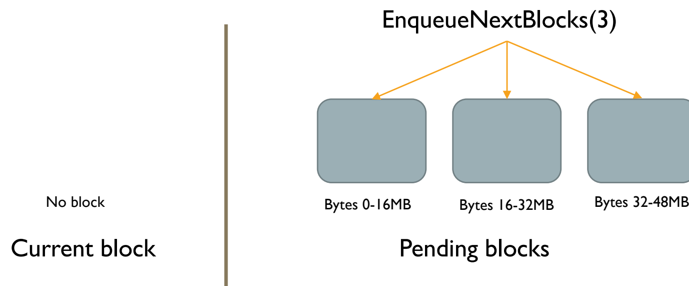

Figure 3: Step 1: enqueuing blocks.

The `EnqueueNextBlocks` call will also start the data transfer. This is a private function and callers don't need to worry about these details. By spreading requests across more than one connection we can get better performance from S3, as described in the Performance Guidelines for Amazon S3 [17].

To use a block for reading, `PopBlock` (a private function) is used. When called it dequeues the next block, if the download has not finished yet this call will block until the data is ready.

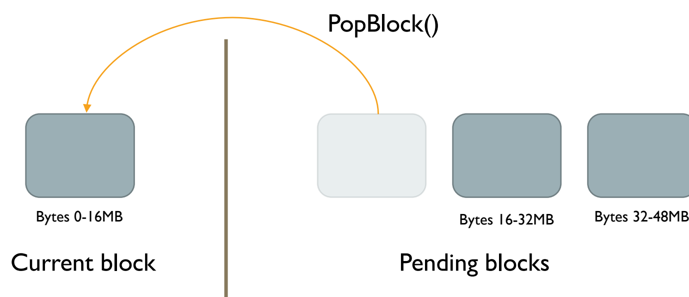

Figure 4: Step 2: reading from the current block.

As mentioned, `seek()` is also supported. Since in most scenarios rMSI only seeks forward, the `seek()` implementation favours this case. More precisely, the following situations can occur when seeking:

- **`read()` has not been called yet**: The current reading offset will be set, but no data transfer will begin.
- **Seeking within the current block**: Will set the next byte to be read by seeking in the underlying `Aws::IOStream`.
- **Seeking past the current block**: The current block will be discarded. If a matching block has already been fetched, it will become the current block.
- **Seeking to an offset before the current block**: All blocks (current and pending) will be discarded and the current reading offset will be set. No data transfer will begin. Any transfers will be aborted and all data will be discarded.

4.2 Converting multithreaded code to run on FaaS

Another change made to the rMSI code is the ability to run some of its processing functions on FaaS. More specifically, this is achieved using Lithops and its Lambda backend but the original code must also be adapted. Furthermore, there are some special cases like writing a ibd file in parallel.

Note that this section will not be an in-depth review of the code changes. It is intended to be an overview of the challenges faced and techniques used.

4.2.1 ThreadingMsiProc

`ThreadingMsiProc` is the base class for all multithreaded functions in rMSI. It is in charge of loading data from disk, spawning threads and writing to disk (in the cases that need it). This class has been replaced by an abstraction that can either call the original methods, or a custom implementation meant to be running in a separate process that will run the processing function for a single datacube, then store the results in Redis. Below is the header file for it, called `WorkerMsiProc`.

```
// Provides an API to multithreaded/distributed functions
class WorkerMsiProc
{
  public:
    // Similar constructor to ThreadingMsiProc
    WorkerMsiProc(Rcpp::List rMSIObj_list, int numberOfThreads, double memoryPerThreadMB,
        Rcpp::NumericVector commonMassAxis, DataCubeIOMode storeDataModeimzml = DataCubeIOMode::DATA_READ,
        bool redisSave = false, Rcpp::StringVector uuid = Rcpp::StringVector(),
        Rcpp::String outputImzMLPath = "", Rcpp::StringVector outputImzMLfnames = Rcpp::StringVector());
    ~WorkerMsiProc();

  protected:
    virtual void ProcessingFunction(int threadSlot);
    virtual void RedisSave(std::string key);
    virtual void RedisMerge(std::string key);

    // API calls
    void runMSIProcessingCpp();
    int GetNumberOfThreadsDouble();
    Rcpp::NumericVector GetMassAxis();
    int GetNumPixels();
    CrMSIDataCubeIO::DataCube* GetCubeForSlot(int threadSlot);
    CrMSIDataCubeIO* GetIoObj();

  private:
    MsiProcStrategy* delegate;
    bool doRedisSave;
};
```

Code 2: WorkerMsiProc header file.

The original class's member variables have been replaced by function calls (`GetMassAxis`,

`GetNumPixels`, ...) to allow greater flexibility for the implementation. The `delegate` member variable holds a pointer to the implementation in use.

4.2.2 Running a job: calling R code from Python (and vice-versa)

To bridge the gap between R and Python, we will be using the rpy2 Python package. It loads R's shared library and abstracts its C API, although there is also a low-level interface closer to the C API. Since the R session runs embedded in the Python process, a segmentation fault in R will crash our Python process (it's the same process). Because we're dealing with C++ code on the R side (via the Rcpp package), it is important to launch a new process for running rpy2 using `multiprocessing.Process`, while taking care to set the start method to spawn instead of fork, so the child process has a fresh Python interpreter and no resources are shared between the two. The following code shows how to run the job in a separate process:

```
def run_job_isolated(job_id, job_input, output_path):
    jr = JobRunner(job_id)
    for file in job_input:
        jr.appendImzML(file)
    jr.setOutputPath(output_path)
    jr.start()

if __name__ == '__main__':
    # must only be done once
    multiprocessing.set_start_method('spawn')

    # collect the id, input and output path (for example, from a queue)

    # start the job
    p = multiprocessing.Process(target=run_job_isolated, args=(job_id, job_input, output_path))
    p.start()
    p.join()

    # p.exitcode can now be checked to determine success
```

Code 3: Running a job in an isolated Python process.

Note that this isolation is only meant to let us recover from a job crashing the process, it does not add any security from exploits like a specially crafted input taking advantage of a buffer overflow in order to run arbitrary code.

Once that's done, we can begin using the rpy2 package. Since our code resides in a package, we can use `importr` to import it. Any functions our package exports will be available in the object this call returns.

As is mentioned in the rpy2 documentation [18], data in R is mostly represented by vectors, even when looking like scalars. This means we'll need to remember that a scalar is represented as a vector of length 1.

Calling Python code from R while using rpy2 is a bit trickier. First, a Python function

must be defined and decorated using `@ri.rternalize`. The symbol can then be exposed to the global R environment. Below is an example on how to do this:

```
@ri.rternalize
def py_hello_world():
    print("Hello from Python")

if __name__ == '__main__':
        # Expose the function in R
        rpy2.robjects.globalenv['pyHelloWorld'] = py_hello_world
        # Call our R code, which can then call our Python function when needed
        my_package = importr('mypackage')
        my_package.SomeFunction()
```

Code 4: Defining a function that can be called from R.

With those building blocks, we can begin writing the code that manages job execution. The `JobRunner` class exposes functions for adding the input files, setting the output directory and processing parameters from a JSON string. You can find an abridged version of the class below.

```python
class JobRunner:
    def __init__(self, job_id, redis_uri):
        self.job_id = job_id
        self.__redis_uri = redis_uri
        self.__rmsi = importr('rMSI2')
        self.__rmsi.CRedisInitialize(self.__redis_uri)

        @ri.rternalize
        def pyCommonMassAxis(*args):
            return rmsiproc.pyCommonMassAxis(self.__rmsi, self.job_id, self.__redis_uri, *args)

        # keep the functions as instance vars otherwise the GC collects them
        self.__common_mass_axis = pyCommonMassAxis
        rpy2.robjects.globalenv['pyCommonMassAxis'] = self.__common_mass_axis
        # [other functions...]

        self.__desc = self.__rmsi.ImzMLDataDescription()

    def appendImzML(self, s3_uri):
        rpy2.robjects.baseenv['$'](self.__desc, 'appendImzMLDataPath')(s3_uri)

    def setOutputPath(self, s3_uri):
        rpy2.robjects.baseenv['$'](self.__desc, 'setOutputPath')(s3_uri)

    def start(self, paramsStr):
        self.__params = self.__rmsi.ProcessingParameters()
        paramsObj = json.loads(paramsStr)
        if not isinstance(paramsObj, dict):
            raise TypeError('JSON payload must be a dictionary')

        # [code omitted...]

        self.__rmsi.ProcessImages(self.__params, self.__desc)
```

Code 5: JobRunner

There are two things to highlight about this class: First, it allows us to choose which files are processed, which parameters are used and where outputs are stored from Python code. Second, it exposes a few Python functions to the R environment that we will use to call Lithops. Once the `start` function is called, control is handed over to the R code.

4.2.3 Invoking Lambda functions and reading .ibd in parallel

In the previous section we mentioned exposing functions that will call back to Python code. The Python side will generally perform these steps:

1. Determine the number of functions to invoke. This is done using `CrMSIDataCubeIO::getNumberOfCubes`.
2. Upload the common function arguments to S3. This was first implemented using Lithops's `extra_args` argument, but due to the way it is implemented the S3 object

(aggdata.pickle) grew too big in size. Instead, common arguments are pickled and uploaded, and only the key is passed to extra_args.

3. Lambda functions are invoked using FunctionExecutor.map. Each worker/mapper downloads and unpickles the object containing the common arguments, sets the datacube index it is in charge of and the IP/port of the Redis server, and then runs the corresponding rMSI2 function.

4. Once all workers/mappers have finished, the reduce step is executed.

4.2.4 Adapting the processing functions

This section will explain the changes needed to convert the multithreaded code to run on Lambda.

4.2.4.1 Common mass axis

This was the first function converted, once S3 reads and WorkerMsiProc were implemented. First, the base class was changed from ThreadingMsiProc to WorkerMsiProc. One of the things needed after inheriting from this class is changing the way datacubes are accessed, so cubes[threadSlot] must be replaced with GetCubeForSlot(threadSlot).

Next, we make the MergeTree struct serializable with libnop by adding the NOP_STRUCTURE macro. This allows us to store this struct in Redis as a string.

```
1  typedef struct CommonMassMergeTree
2  {
3      unsigned int level;
4      bool bMerge;
5      std::vector<double> mass;
6      std::vector<double> bins;
7      NOP_STRUCTURE(CommonMassMergeTree, level, bMerge, mass, bins);
8  } MergeTree;
```

Code 6: MergeTree struct with the macro added.

Then, we implement RedisSave and RedisMerge. The former will store the worker's output to Redis, and the latter takes all those outputs and reduces them.

```cpp
void MTCommonMass::RedisSave(std::string key)
{
    auto& redis = RedisClient::GetInstance();
    using Writer = nop::StreamWriter<std::stringstream>;
    nop::Serializer<Writer> serializer;
    serializer.Write(globalMergedSpc);
    serializer.Write(bNoNeed2Resample);
    redis.rpush(key, serializer.writer().stream().str());
}

void MTCommonMass::RedisMerge(std::string key)
{
    auto& redis = RedisClient::GetInstance();
    using Reader = nop::StreamReader<std::stringstream>;

    while (true)
    {
        auto elem = redis.lpop(key);
        if (!elem) break;
        nop::Deserializer<Reader> deserializer{elem.value()};
        MergeTree mt;
        bool noNeed2Resample;
        deserializer.Read(&mt);
        deserializer.Read(&noNeed2Resample);
        globalMergedSpc = MergeMassAxis(globalMergedSpc, mt);
        bNoNeed2Resample &= noNeed2Resample;
    }
}
```

Code 7: RedisSave and RedisMerge for CommonMassAxis.

This completes the changes needed on the C++ side. We must also implement the Python side as described in 4.2.3 Invoking Lambda functions and reading .ibd in parallel.

4.2.4.2 Internal reference spectrum

This step tries to find a maximum. After the usual steps of switching from `ThreadingMsiProc` to `WorkerMsiProc`, we have to implement `RedisSave` and `RedisMerge`. In this case, we'll be using Redis' sorted sets, to store the values with a score. Once all workers have finished we'll retrieve the value with the highest score. Insertion to a Redis sorted set has $\mathcal{O}(\log n)$ time complexity, the same goes for `zpopmin` and `zpopmax`.

To add the element to the set, we'll use `zadd` in the following way:

```
void MTInternalRef::RedisSave(std::string key)
{
    auto& redis = RedisClient::GetInstance();
    using Writer = nop::StreamWriter<std::stringstream>;
    nop::Serializer<Writer> serializer;
    serializer.Write(maxCorGlobal);
    redis.zadd(key, serializer.writer().stream().str(), maxCorGlobal.cor);
}
```

Code 8: RedisSave for MTInternalRef.

Then, to get the element with the highest score, we'll use `zpopmax`. After that we can just delete the set.

```
void MTInternalRef::RedisMerge(std::string key)
{
  auto& redis = RedisClient::GetInstance();
  using Reader = nop::StreamReader<std::stringstream>;

  auto elem = redis.zpopmax(key);
  if (!elem)
  {
    throw std::runtime_error("MTInternalRef::RedisMerge: Could not read element from Redis\n");
  }
  auto value = elem.value();
  nop::Deserializer<Reader> deserializer{value.first};
  deserializer.Read(&maxCorGlobal);
  redis.del(key);
}
```

Code 9: RedisMerge for MTInternalRef.

4.2.4.3 Preprocessing and writing .ibd in parallel

One of the most interesting aspects implemented, is the support for writing `.ibd` data in parallel using a S3 multipart upload, used during the preprocessing step. This works because each worker's output can be concatenated into a single file. The basic idea is as follows:

First, a multipart upload is created for each of the output files using the `CreateMultipartUpload` action. Each worker stores its data in a buffer, and once it finishes, it uploads it as a part providing the upload id, key and part number. Since each worker uploads a single part, the part numbers will go from 1 to `num_workers`. The data will end up concatenated following the part number. There are some things to take into account:

- Multipart upload limitations: the most relevant is that each part (except the last) must be at least 5 MB.
- The UUID must only be written once, in the first 16 bytes of the `.ibd` file. This means only the worker writing part 1 has to write it.
- When writing data in continuous mode, the m/z array is written only once, but the offsets are referenced in each pixel. Workers will store `mzOffset = 0, mzLength = UINT_MAX` to let the coordinator know it has to replace those values with the ones corresponding to the m/z array, which will be written only by the worker writing part 1.

Each worker stores the ETag returned by S3 and pixel offsets relative to the start of their part to a Redis server. The code that stores the offsets to Redis is provided below. It serializes the `std::vector` using libnop and uses the `set` command to store it on the Redis server.

```cpp
void ImzMLBinWrite::redisSaveOffsets(std::string key)
{
    std::vector<DistributedImzML::PixelOffset> offs;
    offs.resize(sequentialWriteIndex_MzData);

    for (unsigned int i = 0; i < sequentialWriteIndex_MzData; i++)
    {
        offs[i].mzLength = Offsets[i].mzLength;
        offs[i].mzOffset = Offsets[i].mzOffset;
        offs[i].intLength = Offsets[i].intLength;
        offs[i].intOffset = Offsets[i].intOffset;
    }

    auto& redis = RedisClient::GetInstance();
    using Writer = nop::StreamWriter<std::stringstream>;
    nop::Serializer<Writer> serializer;

    serializer.Write(offs);
    serializer.Write((std::uint64_t)ibdFile->Tellp());
    redis.set(key, serializer.writer().stream().str());
}
```

Code 10: Storing the offsets.

The coordinator calls the S3 API to complete the multipart uploads, submitting the ETag of each part. It then builds the corresponding `.imzML` files, using the relative pixel offsets the workers stored, and uploads them to S3.

4.2.4.4 Peak picking

Like the preprocessing step, this step needs to write an imzML (with its corresponding ibd) for each of the input files. However, in this step it wasn't possible to use a multipart upload since there's no guarantee that each worker generates at least 5 MB of data.

- **Send the output to the coordinator, which will upload the data to S3**. This is a great option as long as the data isn't too big, otherwise the coordinator will run out of memory.
- **Upload each part as a separate S3 object**. Then, a coordinator reads all this data and uploads it as a single object to S3. It's similar to the first option but better suited for bigger files, which may not fit in memory.
- **Upload each part as a separate S3 object, but avoid having to reassemble the file**. This can potentially be a great option, but it requires writing the logic to fetch the corresponding object according to the file offset that is being read. For example, if the file is split in two parts, reads for the first half of the file go to one object and reads for the second half go to another.

The remaining challenge is what to do when the user wants to download the file. Of course, if they're running a custom program, we have full access to the filesystem and we can just append bytes as we stream the file from the cloud. But for a web-based application it can be a bit more difficult. One option would be to have a server that streams the data from object storage (i.e., it reassembles the file on the fly), with the extra costs, maintenance and scalability concerns. The other option is to rely on the available APIs on the browser. The File System Access API [19] can be used to read or write files to the user's disk. The user is shown a file picker to select which file to write or a directory to store the file, depending on the use case. After that, the website has a readable or writable stream, again depending on the scenario. This would let us handle the download ourselves, streaming the data from the cloud and writing it directly to the user's disk. The downside of this API is the browser support. Currently, it is supported on Chrome and other Chromium-based browsers, but not on Firefox or Safari. A more widely supported alternative would be Blob, but most likely it will pose problems with large files.

Currently, the first option has been implemented.

5 Performance evaluation

In this section, we will test the performance by processing different datasets of various sizes and characteristics. The datasets used are as follows:

- Dataset 1: 1.8 GB imzML, continuous mode
- Dataset 2: 21.6 GB imzML, continuous mode
- Dataset 3: 34.6 GB imzML, continuous mode
- Dataset 4: 6.7 GB imzML, processed mode
- Dataset 5: 20.2 GB imzML with region of interest file, continuous mode
- Dataset 6: 7.6 GB imzML split in three files, continuous mode

We will test the following scenarios:

- c6g.4xlarge (AWS Graviton2, 16 vCPU, 32 GB RAM) running the multithreaded implementation.
- c6g.2xlage (AWS Graviton2, 8 vCPU, 16 GB RAM) running as a coordinator using our FaaS implementation. The number of Lambda calls will vary depending on the dataset.

The performance will be tested by running all the steps that have been optimized to run on Lambda, which is up to the peak picking (there are more steps in the pipeline).

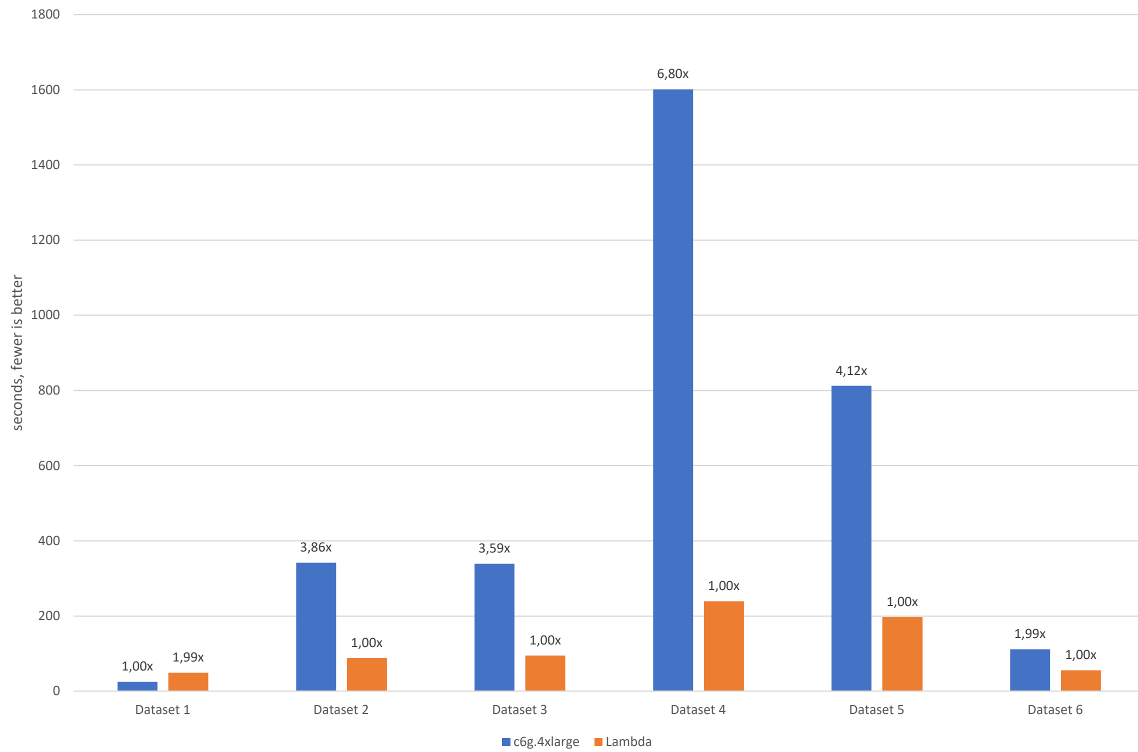

Figure 5: Execution time comparison.

As we can see, the performance does indeed improve with our FaaS approach, since we're able to run the map phase completely in parallel, including parallel writes for the preprocessing step. Although the reduce phases do slow us down a bit in some datasets, we still manage to run the same steps in a sixth of the time when compared to the original version. A table with the results is provided below:

Dataset 1	c6g.4xlarge	Lambda	Speedup
CommonMassAxis	53,3	10,7	4,98
NormalizationsAndMeans	49,91	15,3	3,26
OverallAverageSpectrum	46,33	15,63	2,96
InternalReferenceSpectrum	47,2	15,9	2,97
PreProcessing	122,02	23	5,31
PeakPicking	20,5	13,97	1,47
Dataset 2	c6g.4xlarge	Lambda	Speedup
CommonMassAxis	57	10,4	5,48
NormalizationsAndMeans	40	14,35	2,79
OverallAverageSpectrum	35,44	13,14	2,7
InternalReferenceSpectrum	29,76	14,48	2,06
PreProcessing	68,55	24,21	2,83
PeakPicking	111,1	12,07	9,20
Dataset 3	c6g.4xlarge	Lambda	Speedup
CommonMassAxis	5,7	5,3	1,08
NormalizationsAndMeans	4,41	7,87	0,56
OverallAverageSpectrum	3,57	8,3	0,430
InternalReferenceSpectrum	3,76	8,57	0,44
PreProcessing	6,47	13,44	0,48
PeakPicking	0,85	5,86	0,15
Dataset 4	c6g.4xlarge	Lambda	Speedup
CommonMassAxis	22,1	9,9	2,23
NormalizationsAndMeans	103,88	38,7	2,68
OverallAverageSpectrum	104,1	39,17	2,66
InternalReferenceSpectrum	103,85	28,4	3,66
PreProcessing	927,55	88,89	10,43
PeakPicking	339,89	34,03	10
Dataset 5	c6g.4xlarge	Lambda	Speedup
CommonMassAxis	0	0	N/A
NormalizationsAndMeans	249,33	38,9	6,41
OverallAverageSpectrum	179,82	41,01	4,38
InternalReferenceSpectrum	179,31	45,86	3,91
PreProcessing	203,56	56,55	3,6
PeakPicking	184,09	14,98	12,29
Dataset 6	c6g.4xlarge	Lambda	Speedup
CommonMassAxis	0	0	N/A
NormalizationsAndMeans	23,41	11,09	2,11
OverallAverageSpectrum	16,82	9,5	1,77
InternalReferenceSpectrum	15,63	8,57	1,82
PreProcessing	50,54	17	2,97
PeakPicking	5,29	9,87	0,54

Table 1: Experiment results broken down by step.

6 Developing the frontend

The frontend has been developed using modern web technologies. It uses the Vue.js 3 framework and the PrimeVue UI library. Hosting and AWS integrations are powered by the AWS Amplify service and its client library. The following services are used:

- Amplify Hosting
- Cognito: User authentication
- S3: Storage
- API Gateway: API calls (currently just job submission)
- AppSync: GraphQL API that manages job metadata

The diagram below shows how these services relate to each other and their role. Arrows are meant to show the flow of data (the arrowhead points where data is written or sent to).

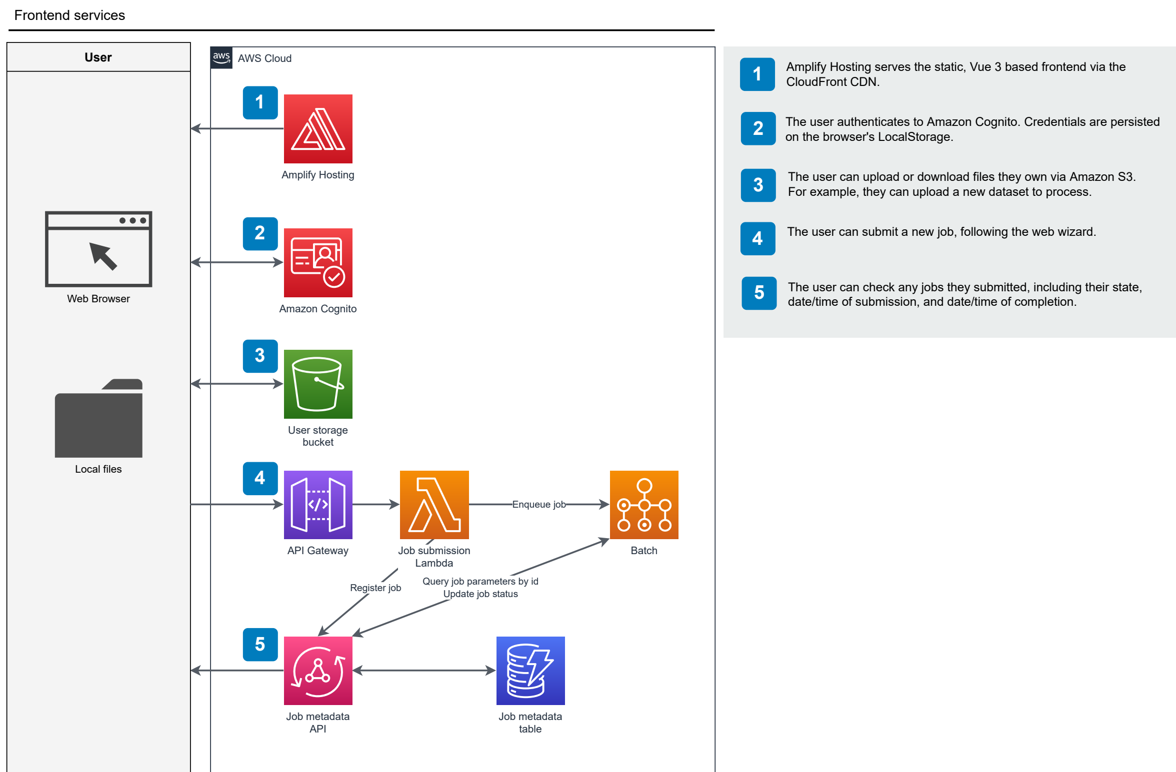

Figure 6: Frontend architecture.

6.1 About AWS Amplify

Amplify is a set of services and tools by AWS that work together to easily develop full-stack web and mobile applications. More specifically, these tools and services are:

- **Amplify CLI**. Used to configure the application's backend. Services can be added by using the command `amplify add` followed by the category, for example `amplify add auth`. A text-based wizard will appear to configure any relevant aspects. This saves the backend configuration to a folder in the project, which is then uploaded to the cloud and applied.

  This means Amplify uses infrastructure-as-code (IaC) under the hood to manage our cloud resources. With IaC, instead of logging into a console and creating resources manually, resources are described in a configuration file. In the case of CloudFormation, these are called "templates" and are in the JSON or YAML format. Other tools like HashiCorp's Terraform use a domain-specific language (in this case, the HashiCorp Configuration Language or HCL). But other IaC tools don't use configuration files, instead they use code written in a programming language like TypeScript, Go or Python with specific libraries. Amazon's Cloud Development Kit (AWS CDK) and Pulumi are in the latter category. Amplify supports adding custom resources using either the CDK or a CloudFormation template.

- **Amplify Library for Android, iOS and web (JavaScript)**. Used to interact with AWS services like Storage (S3), Analytics, Publish/Subscribe, Push Notifications, your backend's APIs and more.

- **Amplify Hosting**. Used to host web applications. Files are served from Amazon's CloudFront CDN with hundreds of points of presence globally, so users will experience fast loading times no matter where they are. It supports CI/CD, connect a git branch and it will deploy any changes on pushing. Other features include custom domains and pull request previews (deploys every PR to a unique URL to preview changes).

- **Amplify UI Components**. Prebuilt UI components for authentication, maps and location search, powered by AWS services.

- **Amplify Studio**. Can be used to manage some resources, like users and content without needing access to the AWS console. It can also convert Figma designs to React code, with support for binding data sources to those components. Moreover, it can be used to visually build a backend. All in all, it seems it's Amazon's take on a low-code development platform with management capabilities built-in.

Amplify also supports multiple environments (e.g. production and preproduction). Each environment has its own set of resources, so for example users or files aren't shared between

environments.

6.2 Login page

The login page uses the Amplify UI "Authenticator" component to provide a sign-in and sign-up form with minimal code required. It also handles persisting the credentials to the browser's LocalStorage, validating the email address with a code sent during sign-up, MFA (multi-factor authentication), password reset flow and social login providers if configured (Google, Facebook, Apple, etc.). You can find a screenshot of the result below:

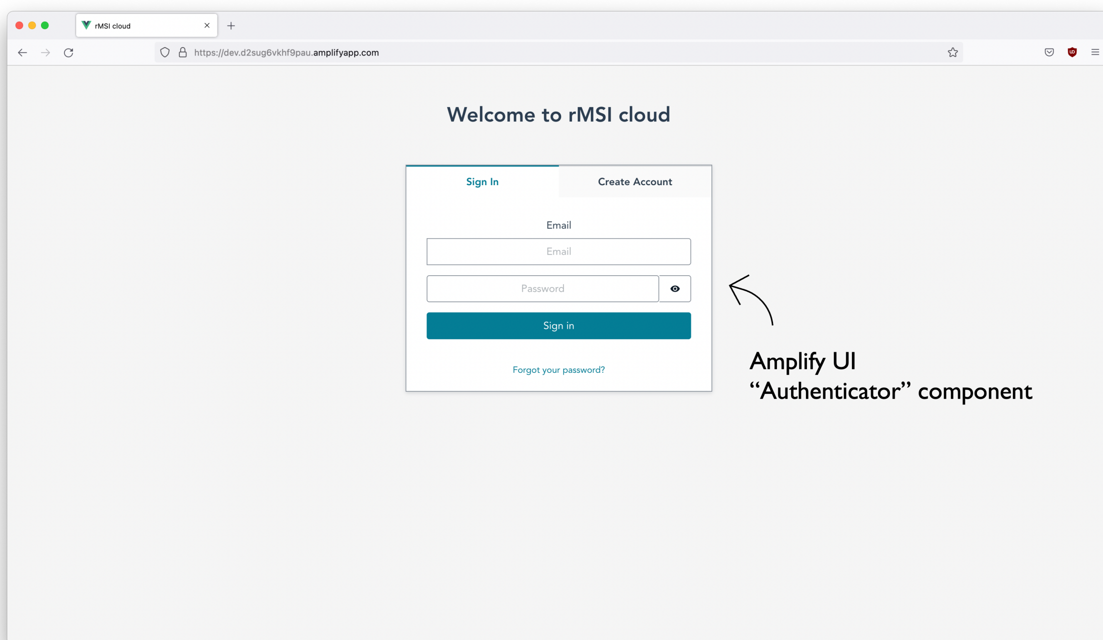

Figure 7: The login page.

In the future, this will be replaced with a Single Sign-On page shared with other projects of the platform.

6.3 File manager page

The file manager page lets users upload and download files to use as input for submitting a job. The files are stored in S3 and the Amplify Storage module is used to manage them. Amplify Storage supports a mechanism called "File access levels" which provides a simple model of file permissions. The access levels are as follows:

- Public. Anyone can read or write any file with this access level.
- Protected. Only the owner (file creator) can write to the file, but anyone can read it.

- Private. Only accessible by the owner.

These permissions are enforced by an IAM policy that Amplify creates. They're applied based on the file's prefix in object storage. Public files will be stored under the `public/` prefix, while protected and private files will be stored under the `protected/user-identity-id/` and `private/user-identity-id/` prefixes respectively, where `user-identity-id` is the user's id in the Cognito Identity Pool (which is not the same as the id in the User Pool).

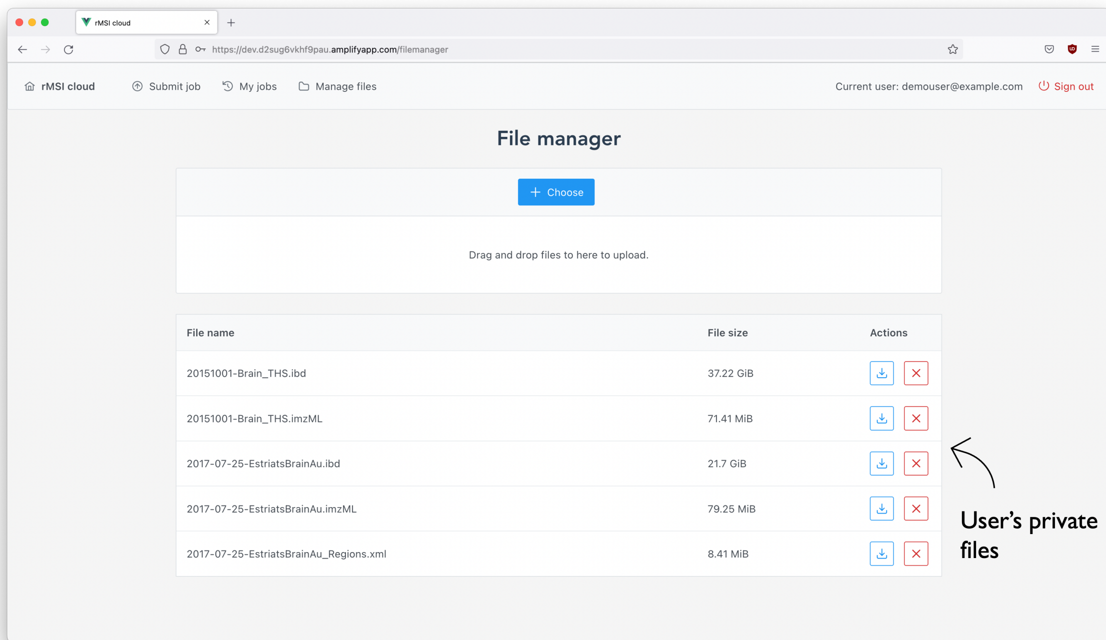

Figure 8: The file manager page.

The file manager does not currently support folders as the user interface would be more complex to design and develop. Furthermore, only one file can be uploaded at a time for now. Uploads are handled by the Amplify client library, which uses AJAX to upload the file as chunks in a multipart upload.

6.4 Submit job page

The submit job page lets users choose input files and parameters to execute the task. First, the user is shown the files they've uploaded via the file manager. The user is then supposed to select the .imzML files.

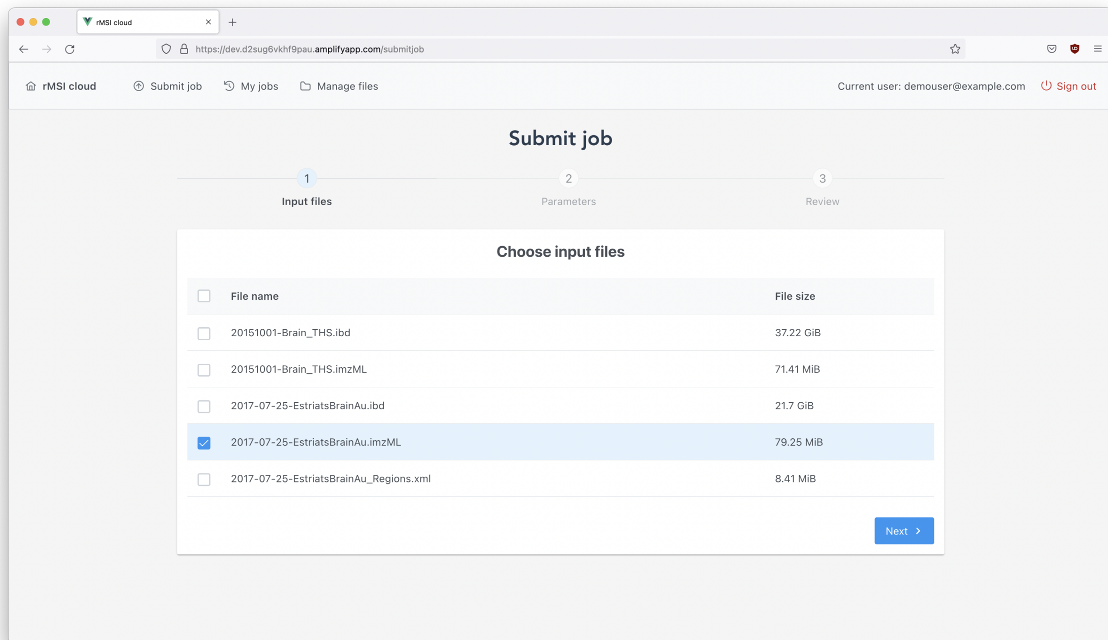

Figure 9: Submit job: first step.

Afterwards, the user can change the parameters if needed.

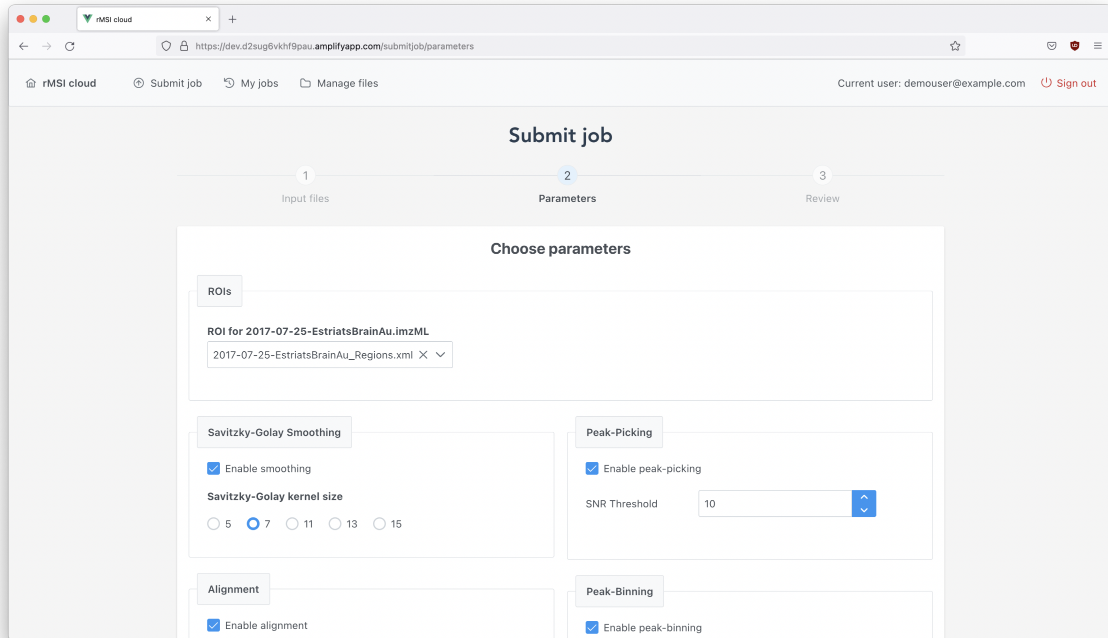

Figure 10: Submit job: second step (not all parameters are shown).

Finally, the user can click the submit button, which will submit the request via API Gateway.

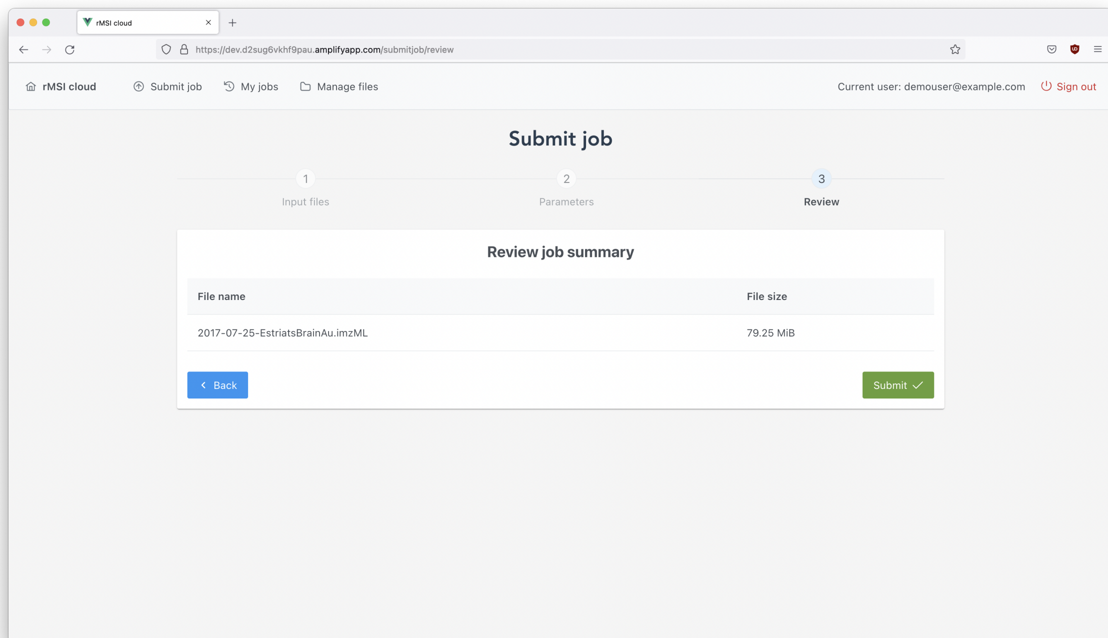

Figure 11: Submit job: third step.

The Amplify client library will sign the request with the user's secret access key (AWS Signature Version 4). You can see an example of a request below:

```
{
    "input": [
        {
            "key": "2017-07-25-EstriatsBrainAu.imzML",
            "roi": "2017-07-25-EstriatsBrainAu_Regions.xml"
        }
    ],
    "params": {
        "preprocessing": {
            "merge": true,
            "smoothing": {
                "enable": true,
                "kernelSize": 7
            },
            "alignment": {
                "enable": true,
                "bilinear": false,
                "iterations": 1,
                "maxShiftppm": 200,
                "refLow": 0.1,
                "refMid": 0.5,
                "refHigh": 0.9,
                "overSampling": 2,
                "winSizeRelative": 0.6
            },
            "peakpicking": {
                "enable": true,
                "SNR": 5,
                "WinSize": 20,
                "overSampling": 10
            },
            "peakbinning": {
                "enable": true,
                "tolerance": 6,
                "tolerance_in_ppm": false,
                "binFilter": 0.05
            }
        }
    },
    "useBatch": true
}
```

Code 11: Payload sent to the job submission lambda.

This request is processed by a Lambda function written in NodeJS, which performs the following steps:

1. Obtain the identity of the requesting user, which is passed by API Gateway in the `requestContext`.

2. Build the list of input files, using the identity ID to format the full path to the files in S3.

3. Register a new job in the Job Metadata API (explained later). This returns the newly created job's ID.

4. Submit the job to SQS or Batch, according to the value of `useBatch`.

6.5 Job Metadata API

The Job Metadata API is a GraphQL API powered by AWS AppSync and DynamoDB. It manages the following information:

- Job ID (a UUID).
- Job status (enqueued, running, success, failed).
- Input file(s).
- Parameters.
- Owner. This is the user's id in the User Pool ("sub", also known as Subject). This field is used for authorization, so that users can only list and obtain information about jobs they submitted.

There are a few reasons for choosing AppSync and DynamoDB. First, they can easily be integrated with Amplify. They are fully managed by AWS, so there's no need to worry about servers or scalability. The pricing model is per-request (AppSync and DynamoDB) and there are no recurring costs, other than the storage used by the DynamoDB table. Also, AppSync does not have cold starts, unless you use a Lambda resolver.

6.5.1 Creating the GraphQL Schema

Let's dive a bit deeper into AppSync and DynamoDB. As explained, the key points of this setup is that both of these services are completely serverless, there's no need to manage anything and they're pay-as-you-go. Amplify simplifies common use cases for CRUD APIs. To get started, we first need to create our GraphQL schema. Let's begin with our Job type:

```
1 type Job @model @auth(rules: [{ allow: owner, operations: [read] }]) {
2   id: ID!
3   status: String!
4   input: [String!]!
5   params: String
6 }
```

Code 12: Job type in the schema.

The @model directive creates a DynamoDB table for this object, in addition to automatically configuring CRUDL (create, read, update, delete, list) queries and mutations [20].

The @auth directive sets up authorization rules: in this case, we're only allowing the owner read access (this includes permission to list jobs they own too). This directive supports other authorization providers, like IAM or API Keys, but in this case we're using the Cognito user pool configured for this project. Note that the user doesn't have the create permission: the job submission Lambda is the one that will create it on behalf of the user.

6.5.2 Adding a IAM-authorized mutation with a custom resolver

As explained, the job submission Lambda adds the job to the database via this API. To do that, we must first add a new mutation to our schema and define its input type. Note the owner field, which defines the owner and the @aws_iam directive.

```
1  input AdminCreateJobInput {
2    id: ID
3    status: String!
4    input: [String!]!
5    params: String
6    owner: String!
7  }
8
9  type Mutation {
10   adminCreateJob(input: AdminCreateJobInput!): Job @aws_iam
11 }
```

Code 13: adminCreateJob mutation and input type.

There is some extra setup needed, like setting AWS_IAM as an additional authorization mode (which can be done via the Amplify CLI), writing the resolver mapping template, adding the necessary values in CustomResources.json and granting our function the permissions to execute the mutation (this can also be done via the Amplify CLI). This section will only focus on the resolver mapping template.

Resolver mapping templates are written in Apache Velocity Template Language (VTL). They take the request as input and output a JSON document containing the instructions for the resolver, which is the "connector" to the data source (e.g., DynamoDB to read/write to a table or Lambda to call a function). There are two types of resolvers in AppSync:

- Unit Resolvers: These consist of a request template (GraphQL request → data source request) and a response template (data source response → GraphQL response that fits the schema).

- Pipeline Resolvers: These are more complex. They contain one or more functions that are performed in sequential order, each including a request and response template.

These functions allow writing common logic that is reused across resolvers, or performing operations that require multiple data source calls. Pipeline resolvers also have a template that is executed before the first function and a template that is executed after the last function.

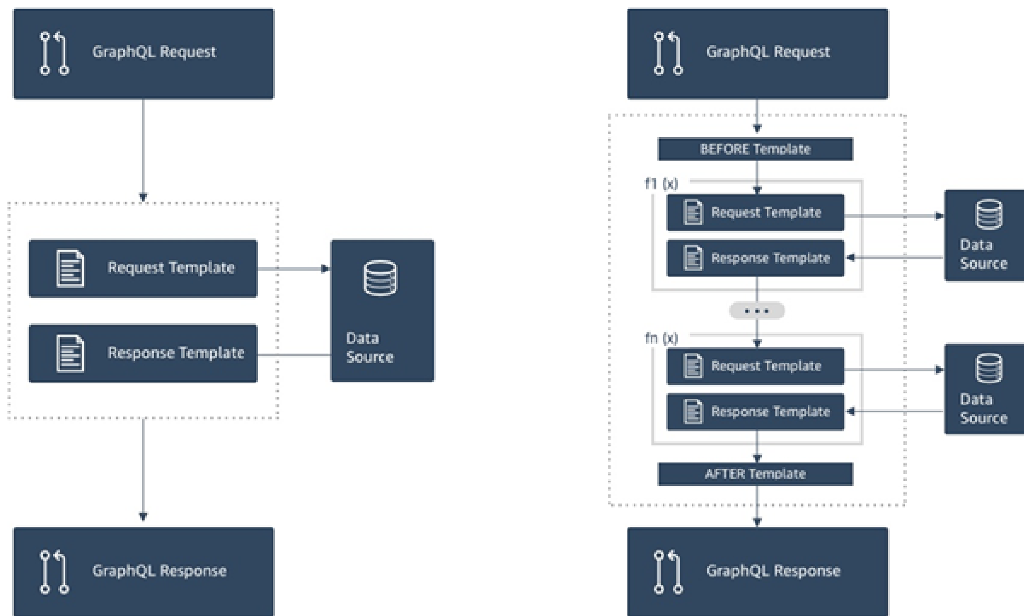

Figure 12: Process flow of a unit resolver (left) and a pipeline resolver (right). [2]

In this case, we'll be using a Unit Resolver. The request template will:

- Generate the new job ID and get the current date/time to set as the creation and last update date.
- Check the caller's Amazon Resource Name (ARN) as an extra authorization check, although it should not be necessary if policies are correctly set.
- Get the input received from Lambda and merge it with the values in the first step.
- Prepare the DynamoDB PutItem request with the new item's data.

```
## Generate Job ID and set createdAt, updatedAt to the current date
$util.qr($ctx.stash.put("defaultValues", $util.defaultIfNull($ctx.stash.defaultValues, {})))
#set( $createdAt = $util.time.nowISO8601() )
$util.qr($ctx.stash.defaultValues.put("id", $util.autoId()))
$util.qr($ctx.stash.defaultValues.put("createdAt", $createdAt))
$util.qr($ctx.stash.defaultValues.put("updatedAt", $createdAt))

## Extra check
#if( !$ctx.identity.userArn.contains("submitJob") )
    $util.unauthorized()
#end

## Merge defaultValues and the input
#set( $mergedValues = $util.defaultIfNull($ctx.stash.defaultValues, {}) )
$util.qr($mergedValues.putAll($util.defaultIfNull($ctx.args.input, {})))

## Prepare the request for DynamoDB
$util.qr($mergedValues.put("__typename", "Job"))
#set( $PutObject = {
    "version": "2018-05-29",
    "operation": "PutItem",
    "attributeValues":   $util.dynamodb.toMapValues($mergedValues)
} )
#set( $Key = {
    "id":   $util.dynamodb.toDynamoDB($mergedValues.id)
} )
$util.qr($PutObject.put("key", $Key))
$util.toJson($PutObject)
```

Code 14: Resolver mapping template (request).

6.6 My jobs page

This page lets users check what they have previously submitted. This queries the Job Metadata API described previously. Dates are displayed in relative format so they're easier to understand at a glance, but it is possible to hover the cursor over the text to show the exact time. Converting the dates to relative is done with a library called `luxon`.

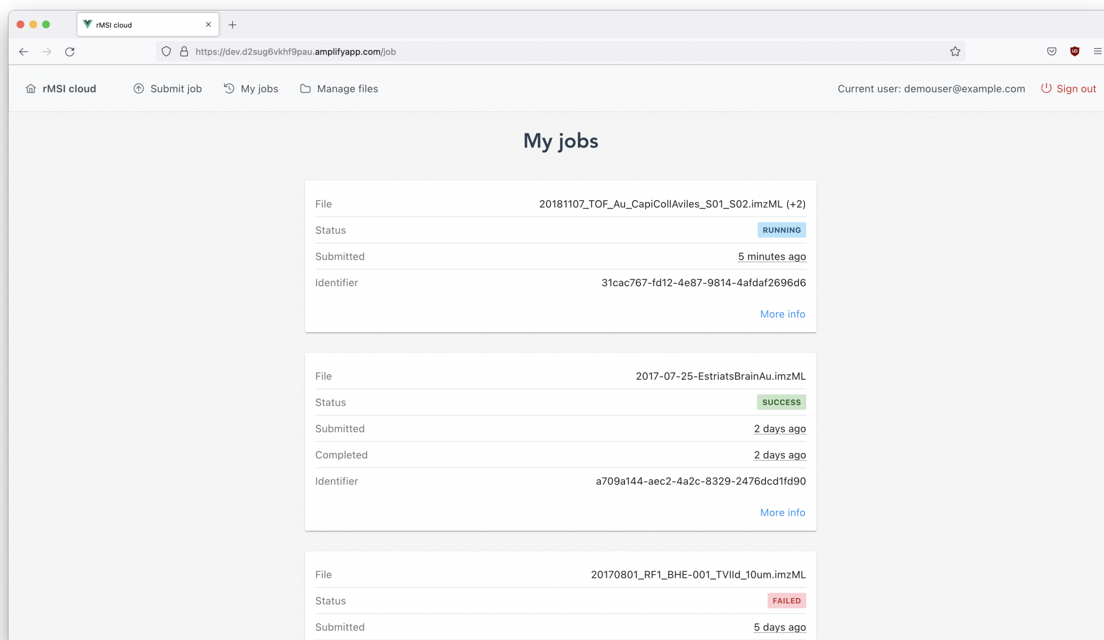

Figure 13: My jobs page.

6.7 Job status page

This page is displayed after submitting a job or clicking "More info" on the previous page. Here the users can download the outputs from S3. These are stored under the prefix `private/user-identity-id/output/job-id`, which is why the Job Metadata API does not need to keep track of outputs since we can just list the objects from S3.

Clicking the Download button generates a temporary URL to download the file from S3, and then the browser starts the download.

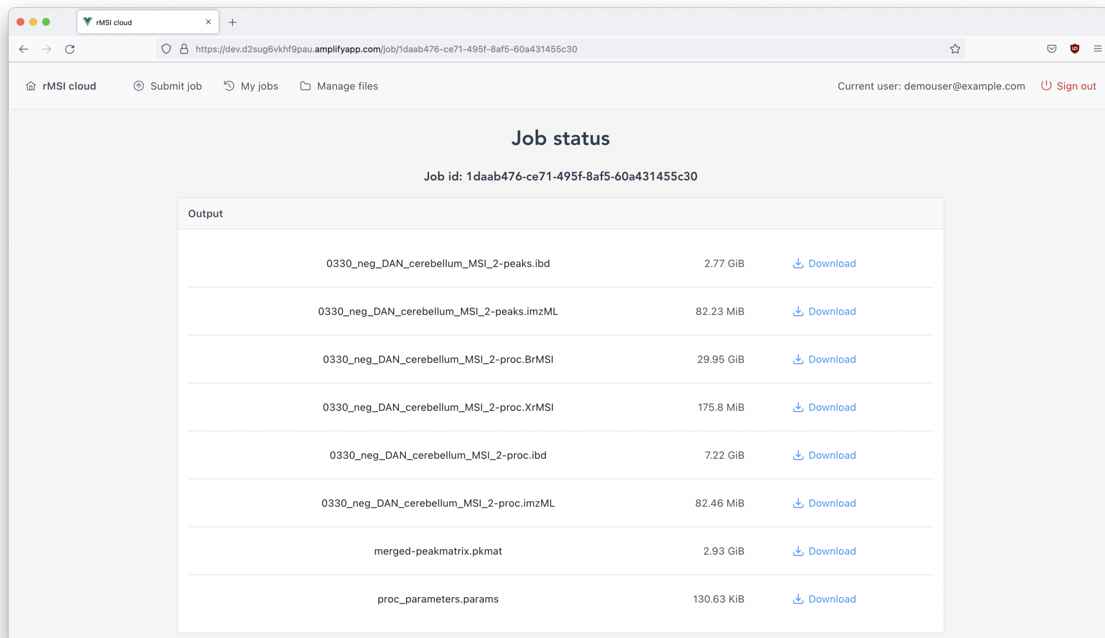

Figure 14: Job status page.

7 Conclusion

In this final degree project, I have presented the architecture, techniques used and some of the optimizations applied to the codebase. I want to highlight the parallelization of input, using byte-range reads that the Lambda workers use, and especially writing the binary (ibd) data in parallel, then writing the corresponding imzML file. This required learning about the data format used. I also want to emphasize the complexity of modifying the original code, since it is written in C++ so there is manual memory management, ownership semantics and other pain points when working with it. Before even beginning with the project, I had to familiarise myself with the codebase and with the external SDKs I was going to use, like the AWS SDK. Then, I had to solve other issues like integrating R and Python code, for which I found a workaround to call Python code from R. Finally, one of the most challenging parts of this project was adapting the functions on a case-by-case basis, minimizing code changes (when possible) and supporting the reduce step using an alternative to shared memory. In doing so, some patches were required to other underlying code. As an example, I even had to patch the XML library used to support the same API used for reading and writing to S3.

As for the frontend, I decided to go with a static site since it provides the best scalability, while also adopting a current-generation framework and tools (Vue.js and the node package ecosystem). Then, I had to research AWS services that could be used to host the site and provide the required functionality, like authentication. This led me to finding out

about Amplify, and reading its documentation, how it works under the hood and determining whether there could be limitations we could hit in the future. Given that Amplify supports describing custom resources with either CloudFormation templates or the CDK, and overriding Amplify-generated ones, it seemed like a good fit. Also, updating the frontend is as easy as pushing the code, as Amplify will run the build and serve the newest version. Another point I want to mention is that I had never used most of these services, so I had to learn about schemas, the available directives and resolver mapping templates, among others. It is important as I mentioned to know how things work under the hood, since as an example, I needed to understand how the Amplify-generated AppSync resolvers managed authorization (the owner field) so my Lambda function could create jobs and make them accessible to the owner.

With all this in mind, I can say the goals described in 1.1 Project goals have been met. This final degree project has required me to familiarise myself with a lot of different technologies and cloud services, as well as some of the metabolomics concepts, processes involved and file formats. I'm grateful for the great opportunity to learn about real-world problems involving metabolomics, while also being able to expand my knowledge on cloud computing.

References

[1] Thorsten Schramm, Zoë Hester, Ivo Klinkert, Jean-Pierre Both, Ron M.A. Heeren, Alain Brunelle, Olivier Laprévote, Nicolas Desbenoit, Marie-France Robbe, Markus Stoeckli, Bernhard Spengler, and Andreas Römpp. imzml — a common data format for the flexible exchange and processing of mass spectrometry imaging data. *Journal of Proteomics*, 75(16):5106–5110, 2012. Special Issue: Imaging Mass Spectrometry: A User's Guide to a New Technique for Biological and Biomedical Research.

[2] Amazon Web Services. Aws appsync developer guide: Resolver mapping template overview. `https://docs.aws.amazon.com/appsync/latest/devguide/resolver-mapping-template-reference-overview.html`, 2022.

[3] Eurofins Biomnis. Mass spectrometry: do we need it? `https://www.eurofins-biomnis.com/wp-content/uploads/2015/09/30-Focus-Mass-spectrometry-UK.pdf`, 2022.

[4] Euro-Bioimaging. Msi: Making pictures out of mass-spectrometry data. `https://www.eurobioimaging.eu/news/msi-making-pictures-out-of-mass-spectrometry-data/`, 2022.

[5] Galaxy Training Network. Mass spectrometry imaging: Examining the spatial distribution of analytes. `https://training.galaxyproject.org/training-material/topics/metabolomics/tutorials/msi-analyte-distribution/tutorial.html`, 2022.

[6] Valentina Puntmann. How-to guide on biomarkers: Biomarker definitions, validation and applications with examples from cardiovascular disease. *Postgraduate medical journal*, 85:538–45, 10 2009.

[7] Pere Ràfols, Sònia Torres, Noelia Ramírez, Esteban del Castillo, Oscar Yanes, Jesús Brezmes, and Xavier Correig. rMSI: an R package for MS imaging data handling and visualization. *Bioinformatics*, 33(15):2427–2428, 03 2017.

[8] Amazon Web Services. What is cloud computing. `https://aws.amazon.com/what-is-cloud-computing/`, 2022.

[9] Amazon Web Services. Aws batch user guide: Job queues. `https://docs.aws.amazon.com/batch/latest/userguide/job_queues.html`, 2022.

[10] Michael Hamacher, M. Eisenacher, Christian Stephan, A. Römpp, T. Schramm, Zoë Hester, I. Klinkert, J.-P Both, Ron Heeren, Markus Stoeckli, and B. Spengler. Imzml: Imaging mass spectrometry markup language: A common data format for mass spectrometry imaging. *Data Mining in Proteomics*, 01 2011.

[11] Amazon Web Services. Amazon ec2 instance store. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/InstanceStorage.html, 2022.

[12] Michał Chojnowski. Aws graviton2: Arm brings better price-performance than intel. https://www.scylladb.com/2021/09/16/aws-graviton2-arm-brings-better-price-performance-than-intel/, 2022.

[13] Travis Downs. Evaluating graviton 2 for data-intensive applications: An arm vs intel comparison. https://redpanda.com/blog/aws-graviton-2-arm-vs-x86-comparison, 2022.

[14] Amazon Web Services. Aws lambda pricing. https://aws.amazon.com/lambda/pricing, 2022.

[15] Docker. Building multi-platform images. https://docs.docker.com/build/buildx/multiplatform-images, 2022.

[16] Amazon Web Services. Amazon elastic container service api reference: Kernel-capabilities. https://docs.aws.amazon.com/AmazonECS/latest/APIReference/API_KernelCapabilities.html, 2022.

[17] Amazon Web Services. Performance guidelines for amazon s3. https://docs.aws.amazon.com/AmazonS3/latest/userguide/optimizing-performance-guidelines.html, 2022.

[18] Laurent Gautier and rpy2 contributors. Introduction to rpy2. https://rpy2.github.io/doc/latest/html/introduction.html, 2022.

[19] Peter LePage and Thomas Steiner. The file system access api: simplifying access to local files. https://web.dev/file-system-access, 2022.

[20] Amazon Web Services. Data modeling. https://docs.amplify.aws/cli/graphql/data-modeling/, 2022.