Requirements Document

12/08/2017

Version: 2.0

Team: Nimbus Technology
Sponsor: Daniel Boros

Faculty Mentor: Austin Sanders

Team Members:
Itreau Bigsby
Matthew Cocchi
Richard “Riley” Deen
Benjamin George

Accepted as baseline requirements for the project:

Team Lead Signature	Date	Client Signature	Date

Table of Contents

1) Introduction 2

2) Problem Statement 4

3) Solution Vision 5

4) Project Requirements 7

5) Potential Risks 12

6) Project Plan 13

7) Conclusion 15

8) Glossary 17

1 Introduction

Consider any business, from a local family-owned “mom ‘n pop” shop, up to the mega-corporations such as Walmart and Ikea. These businesses require data storage for various reasons such as customer transaction history, inventory stock, market performance, and more. In the case of smaller businesses, their need for storage may range from just a few gigabytes up to one or two terabytes. For the massive corporations, they could need several petabytes--millions of gigabytes.

In the past, all of that data would have been stored in company-owned servers that were purchased, maintained, and operated by the business itself. However, this required investing heavily into owning and operating those servers, a cost that can now be forgone thanks to the recent advent of cloud technology. In cloud systems, services such as data storage are offered by companies like Amazon Web Services and Microsoft, and those services are run on servers owned by those companies. Clients then pay to simply purchase, rent, or license those services without having to pay for the ownership, maintenance, and operation of servers.

This solution isn’t perfect, though: these cloud services often come with little to assist the clients paying for them, resulting in management of these cloud services still being an issue. Someone at the client business would need to have the technical knowledge to interact with AWS--another cost the business would have to account for. Additionally, as the amount of data storage needed by business continues to ramp up, so does the cost of that data storage. Per Figure 1 below, an example corporation using 6 petabytes of data storage on Amazon Web Services (AWS) incurs a monthly cost of $144,000.

Cost of Data Storage on AWS			
Storage Plan	Price	Businesses Storage	Total Cost per Month
First 50 TB / month	$0.026 per GB	50 TB	$1,300
Next 450 TB / month	$0.025 per GB	200 TB	$5,000
Over 500 TB / month	$0.024 per GB	6 PB	$144,000

Figure 1: AWS Storage Costs

Cost of Cloud Data Interactions on AWS	
PUT, COPY, or POST Requests	$0.01 per 1,000 requests
GET and all other Requests	$0.01 per 10,000 requests

Figure 2: AWS Request Costs

Our sponsor, Daniel Boros at IBM, works on solutions to these problems. As a senior software developer in IBM's Spectrum Protect Server Development, Mr. Boros helps develop key tools and services that IBM provides to its clients. When a client is already paying for data storage and other services from a cloud vendor such as AWS that client can license services from Spectrum Protect to make it easier to manage their cloud stored data.

It is important here to take a moment to discuss how Spectrum Protect interacts with AWS to cover a client's data storage. When a client pays for storage on AWS they pay a simple fee according to Figure 1 based on the amount of storage they are using, and that storage on AWS takes the form of "objects," which are simply pieces of data of variable size. When that client pays for Spectrum Protect to manage their data, Spectrum Protect interacts with the data using "containers," which hold a variable amount of the client's data, up to a maximum of 1 gigabyte.

Each Spectrum Protect container maps to exactly one AWS object, and vice versa, meaning that moving from one system to the other is fairly straightforward. Thus an example user with 50 terabytes of storage on AWS would have their data split into *at least* 50,000 containers by Spectrum Protect. Considering the prices shown in Figure 2, and the scale of storage needs of Spectrum Protect's clients, making requests for thousands upon thousands of AWS objects can become quite expensive--a metric we consider more in depth in the Domain Level Requirements section of this document.

One of the most important services Spectrum Protect offers is the ability to reduce the amount of data storage a client needs, which is achieved in multiple ways. For example, Spectrum Protect uses "incremental backups," wherein a client that is backing up data to the cloud only sends the data that has changed since the last backup. Thus all of the necessary data is stored, without copying old, unupdated data and wasting space.

Another way Spectrum Protect reduces data storage needs is via "deduplication," in which a chunk of data that exists in multiple places in a client's storage is removed in all but one instance; in the duplicate instanced, the data is replaced with a pointer to the one remaining instance. In this way, the client's total amount of stored data is reduced--in some cases by as much as 90%--without actually losing any of the client's data. Through these methods and others, Spectrum Protect's services can reduce the cost of the example 6 petabytes of data storage from $144,000 per month to as little as $14,400 per month. This is a savings of nearly $130,000 per month, and over $1,500,000 per year, making Spectrum Protect's work quite valuable to clients in need of cloud data storage.

In this document we detail a problem that Mr. Boros is facing in managing cloud stored data, and discuss our solution to the problem. This includes a full breakdown of the requirements of our project, as well as risks we may face throughout the project and our plan for development.

2 Problem Statement

When a client business purchases the licence from IBM for Spectrum Protect services, they agree to a deal which includes a stipulation about the maximum amount of data that Spectrum Protect will cover. When that maximum is exceeded by the client, Spectrum Protect issues a warning to that client that they are exceeding their maximum, and the client must begin manually trimming their own data.

Before removing any data, the storage is first inspected to identify “chunks” (contiguous pieces of data, ranging in size from about 50 kilobytes to 4 megabytes) that can be marked as “expired.” There are a number of factors that could lead to data being considered expired, such as data being simply too old or being several versions outdated. But the primary way by which a chunk is identified as expired is if it is no longer being referenced by any other data, and thus is not required to keep the integrity of the rest of the data. Once all of the appropriate chunks of data have been flagged as expired, data removal can begin.

The process for data removal and storage reclamation is depicted in Figure 3 below. There are three steps to reclaiming data storage: 1) identifying the expired chunks of data, 2) removing those chunks of data from the container, and 3) reformatting the container so that there is no empty space between non-expired chunks. The first two steps are straightforward, but the necessity of the third step may not be overly obvious. If the expired data is removed from the container, but the container is not re-formatted, the extra space where expired chunks once were goes to waste, clogging up IBM’s storage containers which causes data to take up more containers than necessary, driving up AWS interaction costs.

Figure 3: Storage Reclamation Process

Currently this process is handled manually, in a laborious manner. An IBM client must check by hand for expired chunks within a container, remove those expired chunks, move up all of the non-expired data, and push the new container to AWS--and this process is repeated for each container, with hundred or even thousands of containers for a single client. This is a painstaking process that wastes the time of an IBM client, and makes Spectrum Protect’s services unnecessarily difficult to use. Additionally, Spectrum Protect currently has no frontend for this process. Instead all of the reclamation occurs via command line, making an already cumbersome process that much more difficult. We at Nimbus Technology are here to offer a solution to these problems.

To give a clearer idea of the exact problems Mr. Boros and Spectrum protect are facing, we have assembled a list of issues that need to be addressed:

- IBM clients are responsible for removing their own expired data, potentially making Spectrum Protect less desirable than competitors that handle such services.
- Data removal is a manual process, creating a time (and money) sink for IBM clients.
- There is no front end to display statistics concerning the removal of data and storage savings earned through that removal.

3 Solution Vision

To solve Mr. Boros' problem, we are proposing an automated microservice to handle the reclamation of data storage from cloud storage containers and show analytics about the effectiveness and reclamation history of the program. This will consist of a backend program that handles the data reclamation and surrounding choices, a database to store application data, and a web page to display graphics with useful analytics. Figure 4 below depicts a high-level view of the planned architecture of our project.

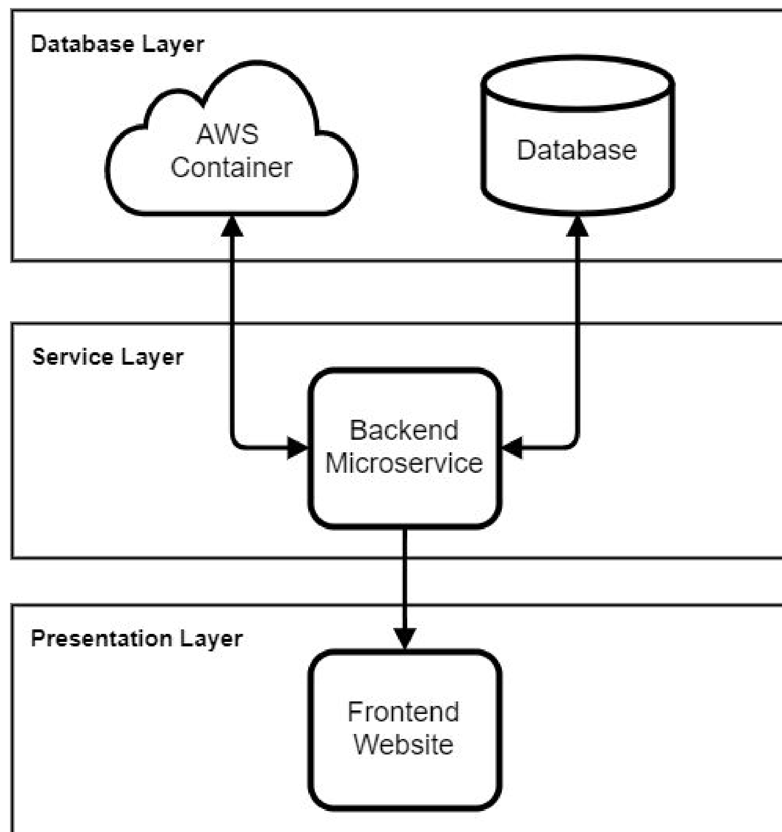

Figure 4: Project Architecture

A breakdown of the key functionality of our project is as follows:

- Read data in from a CSV (Comma Separated Values) or JSON (JavaScript Object Notation) file representing the contents of a container
- Make a choice based on that data on whether or not to reclaim storage from the corresponding container
- If necessary, pull the container from the cloud, reclaim space and re-format the container
- Merge containers where possible and ideal to save space
- Write the container back to AWS
- Store application data in the form of JSON in a database, for future use
- Present analytics based on application data in frontend webpage

This functionality can be encapsulated in three layers, as depicted in Figure 4:

- **Backend:** A backend program that will analyze containers and remove expired chunks of data. This program will need to be able to service up to thousands of containers concurrently with little degradation of performance and no risk of race conditions. This will be implemented in Golang for two reasons: 1) the language offers simple and scalable multi-threading capabilities, and 2) Golang is also used in many other IBM products, allowing our project to more seamlessly integrate with their work.

- **Front end:** A web application that will allow IBM employees to view analytics regarding client data, including metrics such as the amount of data saved per container, a total amount of data saved across all containers, the number of containers currently being services, and more. This will implemented using React.js and D3. React was chosen due to its high modularity and support, again making it easier for IBM to integrate our project into their work. D3 was chosen due to its position as the industry standard for data visualization, with its ability to dynamically create and populate a variety of graphics, which we can put to use by offering different levels of detail in our analytics display.

- **Database:** A database that will store solution metrics and client container information that won't be available from AWS. We will be using MongoDB to store our application data, since its document-oriented nature allows us to store JSON files with little to no work formatting the data.

This architectural layout was chosen to adhere to the practices of the REST (REpresentational State Transfer) software design philosophy which states, among other things, that software ought to consist of layers that only talk to their adjacent layers. For example, in our design the frontend web page is not able to directly read from the database or AWS; instead it must request data from the backend program, which then fetches that data from the database layer. This principle is also reflected in the arrows in Figure 4, which depict the directional flow of data between the layers of our architecture.

4 Project Requirements

To give a better breakdown of exactly how we will implement all of the functionality as described in our solution overview and solve Mr. Boros' problem in cloud storage, we consider here all of the requirements of implementing our solution. This starts with domain level requirements, which describe how IBM's and Spectrum Protect's needs will influence the development of our project and what we will focus on in our software. With the domain level requirements in mind, we describe the functional, software level ways in which we will address the needs of our sponsor. Finally, we consider some extra constraints placed on our project by our sponsor and the domain in which they work.

Domain Level Requirements:

Here we describe the core principles of IBM's business model and software design, and explain how those principles necessitate certain requirements for our project.

1. **Reliability:** IBM is in the business of protecting clients' data. If they should fail, even in the slightest, it may drive clients away towards competitor businesses. With that in mind, our program must not, under any circumstances, result in inappropriate operations on clients' data. This breaks down into the following:

    1.1. The program should never crash.
    1.2. The program must be able to detect and handle inappropriate input.
    1.3. If the program encounters a problem while servicing a container, the problem must not interfere with servicing other containers.
    1.4. The program should log processing information for IBM employees.

2. **Performance:** Software development at IBM involves a great deal of tuning for performance and efficiency, both in runtime and in memory usage. Any software used by IBM to service clients and their data must be reasonably fast and not become unresponsive under heavy loads. In line with this, our project should do the following:

    2.1. The program should service multiple containers concurrently.
        2.1.1. The program will need a way to prevent data writing conflicts.
    2.2. The program should be able to service thousands of storage containers with negligible performance degradation.
        2.2.1. The program should have a way to prevent heavy workload.
        2.2.2. The program will need a way to manage the memory used across the hundreds of containers being actively serviced.

3. **Cost Effectiveness:** While IBM is first and foremost concerned with protecting their clients' data, they are also heavily invested in being cost effective so that their clients' costs are reduced, making IBM a competitive cloud service provider. For us this means:

    3.1. The program should remove expired data only when optimal.
        3.1.1. The program should only cull expired data if the storage cost outweighs the interaction cost.
            3.1.1.1. The program should calculate cost of AWS interactions.
            3.1.1.2. The program should calculate cost of storing expired AWS data.
        3.1.2. The program should hold on to information about sub-optimal containers, to be used in future reclamation decisions.
    3.2. The program should merge small storage containers to reduce interaction costs.

4. **Metrics Reporting:** A key piece of functionality that Mr. Boros is looking for is the ability to report metrics and analytics on the data and containers that are being and have been serviced. This breaks down into the following requirements:

    4.1. IBM faculty need a graphical interface to view program metrics.
        4.1.1. Frontend should be able to show which containers are being serviced at any given time.
        4.1.2. Frontend should be able to show how much expired data has been reclaimed over time.
            4.1.2.1. Frontend should be able to show reclamation history of an individual container over time
            4.1.2.2. Frontend should be able to show aggregate reclamation history of all containers over time.
        4.1.3. Frontend should be able to show how much money has been saved through reclamation over time.
            4.1.3.1. Frontend should be able to show monetary savings for an individual container over time.
            4.1.3.2. Frontend should be able to show aggregate monetary savings for all containers over time.
    4.2. Frontend metrics display should allow for a variety of data and levels of detail.
        4.2.1. Frontend should have dynamic graphics to populate with information of varying metrics and levels of detail.
    4.3. Frontend should be able to display information based on a given timeframe.
        4.3.1. Frontend graphics should be configurable based on timeframe.

5. **Modularity:** A software design principle that Mr. Boros wants this software to adhere to is modularity; different functionality should be handled by different modules, and those modules should be bound to each other as little as possible. For us this means:

    5.1. Software needs to be composed of reusable parts.
    5.2. Each module needs to be functionally independent from the rest.
    5.3. Application modules need to be able to communicate with each other using a standardized communication protocol.

Functional Requirements:

With the domain level requirements listed and explained, we now discuss how those requirements can be addressed in our software. This includes use of particular languages and libraries, certain data structures, and communication protocols.

Requirement Description	Domain Requirement(s)
Backend program will implement exception statements in Golang in order to handle any input.	1.1 1.2
Microservice will use a custom REST API error code library in order to process any errors encountered.	1.1 1.2
Backend program will place threads with exceptions in an error queue data structure.	1.1. 1.3
Backend program will create .txt log files daily that will relay information regarding data reclamation and errors encountered (if any).	1.4
Backend program will use S3 API calls to retrieve client data from and send client data to AWS.	2.1
Backend program will have a module that performs cost/benefit analysis of S3 GET/PUT requests versus storage cost of holding on to expired data.	2.1.1 2.1.1.1 2.1.1.2
Backend program will store, in a database, information on the containers that do not meet the cost/benefit requirements for storage reclamation.	2.1 2.1.2 2.2
Backend program will attempt to merge small containers when possible.	2.2
Backend will work on little data per thread at any one time.	2.2.2

Backend program will use Golang threads to handle reclamation requests concurrently.	3.1 3.2
Backend program will use Golang multi-threading data protection mechanisms such as mutexes to safeguard against writing conflicts.	3.3
React.js frontend will feed data to D3.js components to display graphics regarding useful metrics.	4.1
Frontend will include a way to expand aggregate views of metrics to show those metrics for individual containers.	4.1.2.1 4.1.2.2 4.1.3.1 4.1.3.2 4.2
Frontend will include a mechanism to adjust the timeframe over which to display metrics, which will reload the according D3 graphics.	4.3.1
React.js frontend will use HTTPS request message to request data from service layer.	5.2 5.3
Backend program will use HTTPS response messages when accepting requests from React.js frontend.	5.2 5.3
Backend program will use HTTPS request messages to write container data to and read container data from AWS.	5.2 5.3
Backend and frontend will consist of modules that separately handle independent functionality.	5.1

Performance Requirements:

Here we consider some benchmark goals for the performance of our software. This entails front end load times, front end usability and clicks per action, backend data servicing speed, and backend memory consumption. This breaks down as follows:

- Usability
    - Changing the front end graphs should take at most four clicks, one for each changed criterion for displayed data (individual container vs aggregate, timeframe start, timeframe end, and choice to show monetary savings).
- Runtime
    - A container holding at most 1 gigabyte of data should be serviced and formatted within 1 second. (Note that this is only the time for servicing the data, not retrieving it from AWS or rewriting it to AWS, since network transmission speeds will vary and are out of our control.)
    - A new graph should be rendered and displayed within 1 second.
- Memory
    - Each thread should use no more than 5 megabytes of memory; 4 megabytes for the largest possible chunk size, and 1 additional megabyte for other data.

Environmental Constraints:

Finally, having explained how our software implementation will meet the requirements of our sponsor, we have to consider some limitations imposed by our sponsor which limit and determine what our software can (or must) actually do.

1. The core computational work of the software will rely on a JSON file.
2. The software must be able to accept JSON or CSV file types.
    2.1. The software must be able to convert CSV to JSON.
    2.2. The software must not accept any other file type.
        2.2.1. The software must reject any other file type without error.
3. The software must be able to communicate with Amazon Web Services.
    3.1. The software must account for AWS authentication.
        3.1.1. AWS requests will require verifying credentials.

5 Potential Risks

Any software project faces risks, and must account for those risks with mitigation strategies. Our project is no different, and here we consider the risks that could affect our rise out of our project. For each risk, we lay out a mitigation strategy which covers how we can effectively reduce the likelihood of such risk occurring.

S3 API Changes

If AWS should change the S3 API with which their cloud storage is accessed and interacted with, all of the requests our software makes will no longer work. This would absolutely break our program and would, at best, cause our API requests to fail. At worst, our program could end up modifying unintended data on AWS, violating IBM's primary focus of reliability.

This risk is highly unlikely to occur, considering all of the software out in the world that is reliant on the S3 API being as it is. A change to the API would break not only our software, but that of many other products and projects--something Amazon would not want to cause. In fact, even the largest changes to the S3 API within the past decade have been backwards compatible, meaning a change that breaks our program is almost guaranteed not to occur.

If an API change should occur, there's not much we could do to prevent it from breaking our program. If possible, we will attempt a request and abort execution if it fails. However, we can take steps to ensure our program can be easily fixed in the event of an API change, such as keeping S3 API requests in as few places as possible in our software. For example, we could have one function for AWS reading and one function for AWS writing; if the API should change, only the code inside those functions would need to be updated to make the program work again.

AWS Cost Changes

Similar to the possibility of the S3 API changing, it is possible that the price of storage on and requests to AWS may change. If this occurs, it could cause our program to make sub-optimal decisions on whether or not to reclaim storage from a container. This is because the cost/benefit analysis module we will implement would be relying on old prices and performing calculations based on those prices.

The likelihood of this occurring is moderate to high, since AWS prices tend to change after a year or two. It is very well possible that the prices used in our cost/benefit analysis module at the time of deployment will be invalidated within a year. However, the cost of such an event wouldn't be terribly large, on the order of tens of dollars (according to typical S3 price changes and the cost of storage listed by AWS) at most for even the largest of Spectrum Protect's clients.

Fortunately, our design already remedies this problem: by having the calculations on AWS prices performed in a separate module, we ensure that the numbers used to perform those calculations will only have to be updated in one place for our program to continue working as intended and making the appropriate decisions on whether to reclaim storage. Further, we could develop a separate module with the capability of reading from the web page that lists S3 prices, and using those values instead of anything hardcoded.

Malformed Input

As with any program that takes input, there is a possibility our software will be given input that doesn't conform to the standards we expect. If that input is not handled appropriately, our software may experience serious defects such as deleting incorrect data from containers, crashing while servicing containers, or other serious effects. Such outcomes could be disastrous when dealing with sensitive data for which IBM's clients are paying for protection.

The likelihood of this risk occurring is relatively low, since our software will only be used by Spectrum Protect, who will be sure to only to provide files of a valid type. Improper file type is still a possible risk, though, and must be mitigated.

File input errors is no new issue and can be handled with rigorous input testing and graceful error handling. Rather than checking for any possible bad input, we will check for input that conforms to the exact standards we expect, and abort execution as soon as the given input deviates from those standards.

6 Project Plan

Before discussing our plan for the phases of development of this project, we first review the work we have completed thus far. All of this work comprises the process of laying the groundwork for our project and ensuring that we are amply prepared when we begin development. The work we have completed so far is displayed in the Gantt chart below.

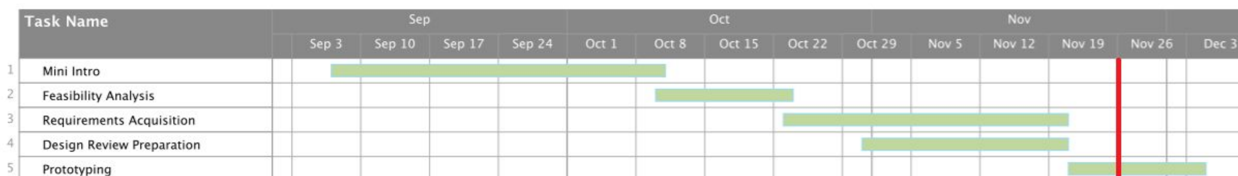

Figure 5: Project Initiation Schedule

Figure 5 above shows the phases of preparation that we undertook for this project. In early September we first met as a team and established the standards by which our team operates. We also had our first meeting with Mr. Boros in which we began to learn about the problems Spectrum Protect is having, and what we can do to solve those problems.

This carried into our Feasibility Analysis phase, in which we broke the project down into four areas of technology: front end, data visualization, database, and backend. Based on the project's needs in each of those areas, we examined potential technologies and resources available to us to satisfying the needs of each technological area. Finally we made a choice as to which tools we would use and how we could prove the feasibility of each of those tools.

After performing this analysis of the feasibility of our project, we began a formalizing the requirements of our project during the Requirements Acquisition phase. While we had already been discussing the needs of the project, it wasn't until this phase that we began asking the particular questions to Mr. Boros that would make clear the exact nature of our project and what it must accomplish. This process was only recently finished, with our team creating this Requirements Document based on the information given to us by Mr. Boros.

The Requirements Acquisition phase ran largely in tandem with a Design Review phase, in which our team created and gave a presentation that covered the problem Mr. Boros is facing, how we intend to solve that problem, and what we have done towards that goal.

We are now in the Prototyping phase, in which we are creating a demoable piece of software that shows the viability of each of the technologies decided upon in our Feasibility Analysis phase. This software will be presented in a live demo to our team mentor Austin Sanders.

Now we discuss our plan for the development of this project, which is split into four key phases: backend development, frontend development, testing, and release. This is shown in the chart below.

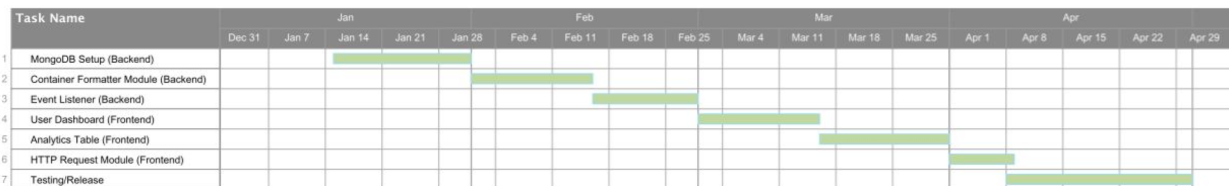

Figure 6: Project Development Schedule

As shown in Figure 6, we are starting development with the backend since it is the core of our project and where most of the work will occur. Our work on the backend will also determine what information is available for us to display in the frontend, for which development will follow after our backend is finished. Once our frontend is complete, we give ourselves about three weeks to conduct testing, particularly integration testing to ensure all of our modules and components are working together with no error. Finally, we will move into the release stage, which will consist of finalizing our capstone work and presenting our project. A more thorough breakdown is as follows:

- Backend: developing the Service Layer, as seen in Figure 4.
    - Establish the ability of our backend to communicate with MongoDB and AWS.
    - Create a module for listening to requests to invoke the backend program.
        - Create a module for converting CSV to JSON.
    - Create a module for storing JSON documents in the MongoDB database.
    - Create a module for making HTTP requests and sending HTTP responses.
    - Create a module for making S3 requests.
        - Implement ability to make S3 GET requests.
        - Implement ability to make S3 PUT requests.
    - Create a module for performing cost/benefit analysis of uploading versus storage.
    - Create the core module responsible for data reclamation.
    - Create a module for merging small containers.

- Frontend: developing the Presentation Layer, as seen in Figure 4.
    - Determine what metrics information is available from the backend.
    - Create mockups and prototype front end layout until a desirable layout is found.
    - Create React.js components according to layout design.
    - Decide which D3.js graphs are most fitting to display our analytics information.
    - Implement ability to create those graphs.
    - Add ability to dynamically recreate and repopulate graphs based on different selection of analytics.
    - Add ability to display analytics based on the timeframe they describe.

- Testing: test that modules all work together with no error.
    - Test that frontend and backend can communicate all data necessary, particularly via HTTP requests.

- Release: finalize capstone experience and present our project.

7 Conclusion

Cloud technology is an ever-growing industry as the world continues to move away from first party-owned servers and towards a model of rented and managed cloud services. Our sponsor, Daniel Boros at IBM, works on competitive products in the cloud technology industry, where his work and that of his coworkers saves businesses thousands, if not millions of dollars per year--while ensuring the protection and easy management of those businesses' data.

Our sponsor has run into a problem with an in-house process: removing expired data from a client business' cloud storage. Currently, IBM clients manually remove their expired data, a process is that is a laborious waste of their time.

We at Nimbus Technology aim to offer Mr. Boros an alternative: an automated microservice that will handle the removal of expired data from cloud storage, with little input or intervention from IBM employees necessary. Additionally, we will provide important metrics such as total expired data removed and money saved thereby.

To that end, in this document we have explained the design plan for our project: a RESTful microservice which will consist of three layers that run independently of each other and communicate with each other through a standardized protocol. With this design established, we examined the requirements of implementing such a system, given our sponsor's need for qualities such as reliability and cost-effectiveness. This provides a concrete understanding of how we can proceed into the development of our project.

While there are external risks our project faces, the risks have little likelihood of occurring or little impact if they do occur. Even so, we can easily mitigate those risks through software practices such as compartmentalization and decoupling, ensuring that if a risk event should occur, it will have little impact on our software, and that impact can easily be addressed.

Having said all of this, we are equal parts prepared and excited to get started with the development of this project. We are confident we can provide the solution that Mr. Boros is looking for, and do so well within the timeframe given to us.

8 Glossary

- API: The way in which software modules communicate with each other.
- AWS: Amazon Web Services, a provider of cloud services, and the source of data storage with which our software will be interacting.
- Chunk: An atomic unit of storage within IBM containers; variable size, from 50 kilobytes to 4 megabytes.
- Container: A construct used by IBM when dealing with data storage; holds a maximum of one gigabyte of data; maps to AWS objects in a one-to-one relationship.
- Expired: A chunk of data in an IBM container which has been identified as low value, either because it is too old or it is no longer referenced by any other chunks.
- Object: An atomic unit of storage on AWS; variable size, up to 1 gigabyte per the relationship with IBM containers; one API request works on one object.
- Petabyte: 1 million gigabytes.
- Reclamation: The process of reclaiming space in a container by removing expired chunks from the container and reformatting it to get rid of gaps between valid chunks.
- REST: A software design philosophy that is becoming increasingly popular; suggests, among other things, a modular software design with low cohesion between modules.
- S3: The portion of AWS that deals with cloud data storage.
- Spectrum Protect: Spectrum Protect Server Development, the department within IBM responsible for creating cloud management services and products.