Technological Feasibility Report

11/06/2017

Team: Nimbus Technology

Sponsor: Daniel Boros, IBM
Faculty Mentor: Austin Sanders

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

Table of Contents

1 Introduction

1.1 Team

We are Nimbus Technology, our team members are Itreau Bigsby, Matthew Cocchi, Richard “Riley” Deen, and Benjamin George, and we are working with our mentor, NAU graduate student Austin Sanders.

Our project is sponsored by Daniel Boros, a senior software developer in the Spectrum Protect Server Development department of IBM. While vendors such as Amazon Web Services offer cloud data storage, they tend not to offer ways to simplify and streamline the management of cloud-stored data. What Spectrum Protect offers is a suite of tools that make cloud data storage not only easier, but cheaper. These tools come under a license deal, in which a client pays IBM Spectrum Protect for a license to some or all of the storage management services. One of the provisions of this license deal is that IBM will only cover a set amount of data for a client, which is a core motivation of our project that we discuss in section 1.2 of this document.

A critical piece of Spectrum Protect’s data storage management suite is what they refer to as incremental “forever” backups: when data is for the first time backed up by a client, all of it is backed up. From then on, each successive backup only stores the data that changed since the previous backup, cutting the cost of data storage by a significant amount, while still retaining all data.

Similarly, Spectrum Protect further reduces storage costs via deduplication, wherein all but one instance of duplicated data is removed, and the removed instances of that data are replaced with pointers to the one remaining instance. This can further cut down on space used by a client, without actually losing any data, resulting in the client having significantly reduced data storage costs. With these and other services, Spectrum Protect enables clients to have data storage that is both cheap and reliable.

1.2 Problem Statement

As part of the license deal IBM offers to their client for Spectrum Protect’s services, there are limits on how long data will be retained, according to the level of service they are provided by Spectrum Protect. At a certain threshold of either outdating or out-versioning, data becomes “expired,” meaning IBM is no longer responsible for managing the data. Once data is expired, its storage and management costs IBM, without providing any benefit, since it is not covered by the license deal with the client.

Clearly, the data needs to be removed from cloud storage, so that IBM is not responsible for managing it. However, Spectrum Protect does not yet have an automated system to handle the cleaning of expired data and reclamation of storage space occupied by expired data, meaning that handling such reclamation is currently a laborious process for Spectrum Protect.

Additionally, Spectrum Protect currently faces the issue that much of their software is implemented in a monolithic design, meaning that there is a great deal of dependence between their software components. The impedes Spectrum Protect's ability to update their software, as well as offer scalable services, since an update to a single component likely means developing and deploying an updated version of the entire software suite. To alleviate this problem, they wish to step away from the monolithic design and towards a microservice model.

Nimbus Technology aims to offer to IBM a microservice that will automate the culling of expired data and reclamation of according storage space. This microservice will take as input a JSON file representing the layout of a container and, based on information therein, will make a decision on whether to reformat the respective container. If the decision is made to reformat, the container is pulled from AWS, reformatted, and pushed back to AWS. In either event, the JSON file representing the container is stored in a database for future use.

1.3 Document Outline

The purpose of this document is to consider the main technological hurdles entailed in our project and to analyze some of the options that are available to overcome those hurdles. We begin by listing and describing the challenges we expect to face, in terms of what capabilities our finished project will need to have. Afterwards, we examine the choices in technologies available to ensure those capabilities and evaluate the choices based on how effectively they address our needs. Finally, we assess the possibility of using all of our chosen technologies in a single, comprehensive solution to the problem our project is trying to solve.

2 Technological Challenges

In this section, we consider the high level challenges we expect to face in the design and implementation of our project. We keep a high level view here so as not to get bogged down in details and instead focus on a bigger picture of the workflow of our project.

2.1 Web Application Front-End

- The front-end serves as a portal for an IBM client to see the analytics of our service
- This portal will serve as the container for our data visualization mechanics
- The front-end must be able to read from a database of stored container layout files, to display information on owned containers to IBM's clients
- The front-end should be focused on modularity and reusable components in order to better adhere to the microservice model

2.2 Data Visualization

- We must implement a clean interface that elegantly displays non-standard, heterogeneous data (e.g. storage layout of a container)
- The project will have an interactive interface that can be expanded and collapsed to display varying degrees of information

2.3 Program Back-End

- The project will center around a back-end implementation capable of effective multithreading, to service multiple containers concurrently
- The back-end must be able to write to and read from a database of stored container layout files

2.4 Database

- The project requires a database that is compatible with the front-end and back-end of our project, be it through official drivers or community-written drivers
- The database must be capable of safe concurrent queries so as not to bottleneck our multithreaded back-end

These high level requirements effectively outline a big picture view of what our project will need to be capable of when finished. Each of these points needs to be adequately addressed with a choice in technology, which we discuss in section 3 of this document.

3 Technology Analysis

We now consider some of the technology options available to us, and weigh those options against the needs of our project. For each domain of our project, we give a brief description of each option and rank them all in terms of how well they do or do not address each need of the respective domain.

This rank comes in the form of a number from 1 to 3, a 1 indicating that the option poorly addresses the need (if at all), and a 3 indicating that the option excellently addresses the need. Based on this rank, a final choice is made as to which technology we will use for the given domain, and an explanation will be given for our rationale behind the choice. Finally, we briefly explain how we can prove the feasibility of our choice with a technological demo (to be performed at a later date).

3.1 Web Application Front-End

Our project requires that we provide analytics as to how our tool is servicing client's AWS storage. A web based front-end is required in order to retrieve that data and format it in a way that our data visualization tool can display it appropriately (see section 3.2 regarding data visualization). The highest priority goal in our front-end implementation is modularity. With IBM/Spectrum Protect moving away from a monolithic design towards microservices, we have an excellent opportunity to help set the groundwork for microservices becoming the standard at IBM. Of course, this means we need to adhere to the microservice model.

In that pursuit, we are focusing our front-end around modularized, reusable components. Reusability is a core concept of the microservice model, since it greatly eases the process of creating a new application that adheres to the standards of previously developed applications. To ensure that our project can be used by IBM as a jumping point into microservices, we are first and foremost looking for modularity and reusability in a web framework.

Minor concerns we have for this domain of the project are speed and efficiency, as well as the stability and community or official support for a given framework.

Available Options

AngularJS:

The first and most obvious possibility for a web framework is AngularJS, developed by Google, and known for its strong adherence to Model-View-Controller (MVC) software pattern. The MVC model lends itself to a two-way binding between back-end data and front-end UI, meaning that when data is updated in the back-end (in our case, a database) it is immediately reflected in the front-end web application. This makes AngularJS a strong option when real-time updates are required. However, it is highly unlikely that any of IBM's clients would happen to be viewing our front-end at the same time as our back-end is servicing their container. Thus, the two-way binding of AngularJS is not a meaningful advantage for our project.

AngularJS also provides considerable support for modularity, with modules being a fundamental design piece of the framework. This ensures that we can, with relative ease, develop our front-end as a system of modules that IBM can reuse and extend in the future.

Perhaps AngularJS's greatest strength over its competitors here, though, is its popularity: AngularJS is incredibly popular in enterprise applications, providing a large community that supports it, meaning it will be stable and reliable.

ReactJS:

ReactJS is an offshoot of AngularJS, primarily developed by Facebook, which aims to expand on the modularity of AngularJS. ReactJS capitalizes on Javascript's capability for object-orientation, turning modules into classes that can be reused and extended, and even exported for use in other projects. This makes ReactJS highly ideal for satisfying the modularity requirements to help IBM get off the ground with microservices.

ReactJS, being newer and slightly less popular, will have less community support than AngularJS, but it more than makes up for this with its considerable gains in speed, being as much as ten times faster than AngularJS across the standard web browsers.

It is also worth noting that ReactJS has one-way binding, meaning updated back-end data will not be immediately reflected in the front-end representation. However, for the same reason AngularJS's two-way binding is an irrelevant strength for us, so too is ReactJS's one-way binding an irrelevant weakness.

VueJS:

VueJS offers great speed in conjunction with being extremely lightweight. While AngularJS and ReactJS are relatively complex to set up, using VueJS is as simple as doing an import, and it works in scripts embedded in HTML files. This, combined with the fact that it is as much as twice as fast as ReactJS, makes it a strong option when efficiency is a concern.

VueJS offers considerable modularity, similar to ReactJS in that it takes advantage of Javascript's potential for object-orientation, but remains slightly weaker because modules cannot be exported and reused elsewhere. While VueJS would be strong for single, lightweight pages, it is not fully viable in a microservice model.

It is also worth considering that VueJS is a relative newcomer in Javascript frameworks, having had its first official release only a year ago. This means community support will be quite low, creating a potentially steep learning curve for our team.

Chosen Approach

The table below shows a comparison of how well each of the three options addresses our needs for a web application front-end: modularity, speed, and support.

	Modularity	Speed	Support
AngularJS	1	1	3
ReactJS	3	2	2
VueJS	2	3	1

Figure 1

Considering this breakdown, we have determined that **ReactJS** will be the best option for us to employ in our web application front-end. It offers incredible modularity, the primary focus of our front-end, with a negligible cost to speed (actually a gain in speed, compared to the next best modular option) and a minor loss in support, which we do not expect will drastically impact our project.

Proof of Feasibility

Proving the feasibility and effectiveness of ReactJS will be straightforward. We will create a simple module and reuse it within the same file without redefining it, proving that we have the modularity we need for our project. Then we will attempt to export the module to another file, again without redefining it, to prove that our modules can be easily reused and extended by IBM once our project is done.

3.2 Data Visualization

When an IBM client views the front-end of our application, they should be able to view an interactive interface that displays information about their containers, particularly statistics and analytics about how our program has changed their containers. This information may include metrics such as a comparison between storage space used before and after our program serviced their container, how much data has been compressed, how long it took to cull data and reclaim space, and how much money the client has been saved.

To ensure we are capable of displaying that information in an elegant form, we will require a library or framework to assist in graphically visualizing data. Our primary objective here is convenience: being able to take the data representing the layout of a container from a JSON file, and perform as little work as necessary to present it in a neat, graphical format.

Our next most important goal with data visualization is having some degree of interactivity. While it would be totally viable to make the page static and non-interactive, this would lead to a clunky design that would likely be user unfriendly. Interactivity offers us the ability to change the degree of detail shown to the user.

A less important, but still desirable, goal for data visualization is a shallow learning curve: having resources available to help us quickly get up to speed on using the framework or library in question would help immensely when under the time constraints of a capstone project.

Available Options

ChartJS:

The greatest strength of ChartJS is its simplicity. It requires no other frameworks or build tools, instead requiring only an additional HTML tag and some Javascript code. It is very lightweight for this reason, and could easily be learned by any member of the group.

However, the result of this is that ChartJS is a very rudimentary graphical library. It uses the “canvas” style of graphics, which means that a graphical object is statically rendered once; that is, ChartJS offers no interactivity, since it cannot redraw graphical elements on the fly.

The ease of using ChartJS is further outweighed by the fact that it does not accept JSON, instead having predefined methods to set the attributes of a graphical object before it is “drawn.” This creates considerable overhead for our team, making ChartJS undesirable.

HighchartsJS:

HighchartsJS takes the strengths of ChartJS and expands on them, while also reducing or removing some of the weaknesses most relevant to our project. Most importantly, HighchartJS can directly take JSON as input, which will cut out a significant portion of overhead in our data visualization.

HighchartsJS also offers interactivity via dynamic objects, enabling us to update previously drawn graphics on the fly. This would be particularly useful for expanding graphics to show or hide extra data when a client clicks an according button.

However, in exchange for this extra power, HighchartsJS would carry a slight learning curve due to its more complex scripting requirements.

D3JS (Data-Driven Documents):

D3JS further builds on the strengths of HighchartJS and ChartJS. Like HighchartsJS, D3 is able to take in JSON directly, greatly alleviating some of the overhead we would be facing otherwise.

Where D3 vastly improves on HighchartsJS is in its capability for interactivity. D3 allows the manipulation of individual elements of graphical objects--for example, adding an additional slice to a pie chart, or changing the sizes of some slices. This allows us to very neatly expand and collapse graphics according to the level of information the user wants to see.

It is worth noting, though, that of these three options, D3JS is by far the most difficult to learn, with a complex syntax used for manipulating individual elements of graphical objects.

Chosen Approach

The table below shows how well each of the three options addresses our needs for data visualization: (lack of) formatting overhead, interactivity, and ease of learning.

	Little Overhead	Interactivity	Ease of Learning
ChartJS	1	1	3
HighchartsJS	3	2	2
D3JS	3	3	1

Figure 2

With this analysis done, we believe that the best choice for the data visualization aspect of our project is **D3JS**. D3 offers, most importantly, the ability to directly take in JSON, meaning very little overhead will be required when passing the files to our visualization module. While HighchartsJS also takes in JSON directly, it offers less interactivity, which will be a desirable quality of our front-end. Notably, D3JS has the steepest learning curve of the options we reviewed, but we are confident that with proper time investment we will be able to use D3 more than well enough to meet the needs of our project.

Proof of Feasibility

To prove that D3JS will work for our project, we will take a sample JSON file and pass it to a D3 script which will be set to graphically display some metrics about the data in the file. This will demonstrate both that we can use D3 for data visualization, and that we can use JSON directly with D3 to do so.

3.3 Program Back-End

As with any back-end implementation, our main focus here will be safety and graceful error handling. Seeing as our project will be used to handle real data that IBM's clients are paying for, a simple off-by-one error could incur great cost by deleting data the client should have been able to keep.

Our second concern, once safety is assured, is multithreading capability. The intention with our microservice is to have a single instance running, that is able to service hundreds, if not thousands, of containers at a time. This requires multithreading, and a language that makes multithreading simple and elegant would be greatly desirable.

Our final consideration for the back-end of our project is the libraries and frameworks available to support our work. The more libraries and frameworks we can make use of, the less minutiae we may have to worry about, allowing us to focus on the more important implementation details of our project.

Available Options

Python:

Python's greatest advantage is safety: with dynamic typing, programs are significantly less likely to outright fail, instead allowing for conditional checks that enable us to gracefully shut the program down in the event of an error. Additionally, Python has features that prevent inappropriate memory access, protecting memory that should not be affected by a given operation.

However, Python provides a very weak implementation of multithreading, since the language was not designed with multithreading in mind. Under the hood, Python actually implements concurrent threads as processes, resulting in dramatically reduced performance.

Regarding support from libraries and frameworks, Python has been a highly popular language for 15-20 years now, and there is a wide pool of pre-made software we can use. Tapping into that pool would help us take care of details we would prefer not to focus on, which could dramatically speed up development time for our team.

C:

In terms of safety, C is lacking: strict typing enforcement can create issues when data is being passed around to and from multiple places, off-by-one errors often cause segmentation faults, and pointers can easily create messy memory accessions. To be fair, this can be remedied by simply writing better code, but extra safety is still preferred.

For multithreading, C offers two common implementations: pthreads and OpenMP. Pthreads give a great deal of control to the developer and allow for many forms of synchronization control, while also requiring more work from the developer. OpenMP, on the other hand, offers clean and elegant solutions to multithreading, especially when concerning non-data-dependent tasks that can be run concurrently with no need for synchronization--which is certainly the case for our project. However, this still requires imported libraries and clever code design. Either way, C is fantastic for multithreading.

Concerning the availability of libraries, while Python has been popular for 15-20 years, C has been a mainstay programming language for 40 years, and there are certainly many tools at our disposal for C.

Golang:

Like most modern languages, Golang is quite safe, most notably in comparison to C because of its garbage collection feature. This ensures that manual management of memory is not such a problem, allowing developers to focus on the logic of their program instead.

Since Golang was designed with multithreading in mind, it implements threads as primitives, greatly simplifying multithreading. Most notably, Golang threads handle synchronization under the hood, and consume orders of magnitude less memory than in C. The result is that multithreading in Golang is easy, safe, and cheap.

Library support is the area where Golang will be weakest. The language's first official release was in 2009, and it has been growing slowly since. There isn't yet a great deal of libraries and frameworks available in Golang, meaning we will likely have to get into the finer grain details we would otherwise wish to avoid, if we chose Golang.

Chosen Approach

The table below shows how well each of the three options addresses our needs for back-end implementation: graceful error handling, elegant multithreading, and support from libraries.

	Error Handling	Multithreading	Libraries
Python	3	1	2
C	1	2	3
Golang	2	3	1

Figure 3

Considering the above table, we believe **Golang** to be our best option. It provides the cleanest support for multithreading, which will be the largest hurdle for our back-end. While Python does offer slicker error-handling, it comes at the cost of essentially any multithreading capability. C provides fairly competitive multithreading, but with such weak and ungraceful error handling, we feel safer using Golang.

Proof of Feasibility

To prove that Golang is our best option for back-end implementation, we will create a program that performs some simple, but non-trivial, task with multiple threads. Possibilities are a sorting algorithm such as mergesort, summing an array, or something similar.

3.4 Database

Our program will take as input a JSON document detailing the layout of a container of data owned by one of IBM's clients. Based on the data the program reads about the container, it will either decide to wait until reclaiming space is more cost-efficient, or to perform space reclamation immediately. In the former case, the JSON file is stored locally; in the latter case, the container is cleaned of expired data and reformatted to reclaim space, then the JSON file is changed to match the newly updated container, and the JSON file is store locally. In either case we need a database to store the files, so that we can perform queries for stored files.

Seeing as the database is only being used to store the JSON files, and has no other functional requirements, our most important goal for the database we use is that it should require very little formatting overhead for storing the JSON. If the JSON can be stored directly, it will be an immense advantage for the given database.

In keeping with the multithreaded nature of our back-end, the database needs to be capable of hundreds, potentially thousands, of safe, concurrent read and write requests. If we do not account for concurrent requests, threads could easily bottleneck each other and drastically reduce the performance of our back-end.

Finally, to ensure that our project is a viable microservice, we require at least moderate scalability--being able to store, and effectively manage, thousands of JSON files without becoming bogged down.

Available Options

MySQL:

MySQL is a tabular database in which the tables consist of fields (columns) that describe the data, and entries (rows) which are the data themselves. This model doesn't very well fit our needs, since the JSON data has little in the way of fields or attributes, instead being a hierarchical structure representing the data in a container. This means using MySQL will incur significant overhead when storing data after looking at and/or servicing a container.

MySQL provides little in the way of innate support for concurrent requests; while it can handle concurrent requests, at large enough numbers, the database engine will slow down measurably. While we don't expect to frequently hit such large volumes of concurrent requests, it is still a constraint to consider.

Similarly, MySQL provides minimal support for scalability of storage, making the management of large volumes of data (i.e. thousands of JSON files) slow and laborious. This results in MySQL being an especially poor choice with regard to creating a sustainable microservice.

PostgreSQL:

PostgreSQL provides excellent tools for storing and handling JSON documents, with inherent syntax specifically for indexing into JSON files. This gives PostgreSQL a great advantage in that we will have to perform little, if any, extra work in formatting our JSON data to be used in a PostgreSQL database.

Where PostgreSQL truly shines, though, is its support for concurrent queries. Using a mechanism known as “multiversion concurrency control”, PostgreSQL protects the database by offering the querying agent a snapshot of the database to work with. If the agent was writing, the agent returns an updated snapshot, which is sent to the concurrency control unit, which then safely integrates the updated values into the database. The result is a system which maintains the safety of the database, while obsoleting the use of mutexes, leading to a database which can handle large volumes of concurrent queries not only safely, but quickly.

Unfortunately, PostgreSQL offers little in the way of scalability, since it does not perform any clever tricks for memory management such as partitioning. This would greatly hinder our ability to adhere to the microservice model, making PostgreSQL a questionable choice long-term.

MongoDB:

Like PostgreSQL, MongoDB is an excellent choice for systems using JSON. MongoDB’s native document type is (slightly modified) JSON, and it also provides dynamic indexing, allowing for retrieval of only those documents that contain particular sought fields. As a result, MongoDB would eliminate a great deal of overhead otherwise posed by MySQL.

MongoDB does have the capability for concurrent requests, but doesn’t implement any clever system such as PostgreSQL’s MVCC, meaning that MongoDB will be slower, and potentially less safe, when concerning large volumes of concurrent queries. Fortunately, we do not expect to see large enough volumes to cause issues, but it is worth considering.

MongoDB’s greatest strength is its scalability. MongoDB inherently and automatically provides horizontal scaling by “sharding” the stored data. Sharding data is the process of partitioning it into groups that can be specifically queried, as opposed to querying the entire collection of documents. Because of this, MongoDB is highly desirable in adhering to the microservice model.

Chosen Approach

The table on the next page shows how well each of the three options addresses our needs for database implementation: little formatting overhead, capability for concurrent queries, scalability of storage.

	Little Overhead	Concurrent Requests	Scalability
MySQL	1	1	2
PostgreSQL	3	3	1
MongoDB	3	2	3

Figure 4

With these factors considered, we believe **MongoDB** is our strongest option. It offers the same reduction in data formatting overhead that PostgreSQL does, while also providing much stronger adherence to the microservice model via horizontal scaling of data storage. This does come at the cost of ability to handle concurrent requests, but we expect the volume of concurrent requests our database will need to handle will be low enough to not cause issues.

Proof of Feasibility

To prove the feasibility of MongoDB, we will create a sample database and, using a Golang program, will create a sizeable volume of threads that will each ping the database with a JSON object to store. This will demonstrate both the lack of overhead necessary in storing JSON, as well as MongoDB's ability to handle concurrent queries. This won't show data storage scalability, admittedly, but without thousands of JSON files to write and read from, scalability is difficult to demonstrate at this point in time.

4 Technology Integration

With each of the technology components outlined and discussed, we now have to consider how we will bring all of the components together in a cohesive unit that will adequately address all of the needs of our project. This primarily means we need to establish which components are connected, and how they are connected.

In particular, we are designing our project in a RESTful manner. In RESTful systems, it is typical to break up the system into layers that represent different functionality. For our project, these layers are as follows:

- **Database Layer:** This layer is comprised of all of the data storage services and mechanisms.
- **Service Layer:** This layer will perform data transformations and communicate data between the front-end (presentation layer) and back-end (database layer).
- **Presentation Layer:** This layer is the front-end application that the user will interact with, which will present to the user any data communicated to it from the database layer by the service layer.

This architecture is depicted below. Note that the arrows represent transfer of data between components.

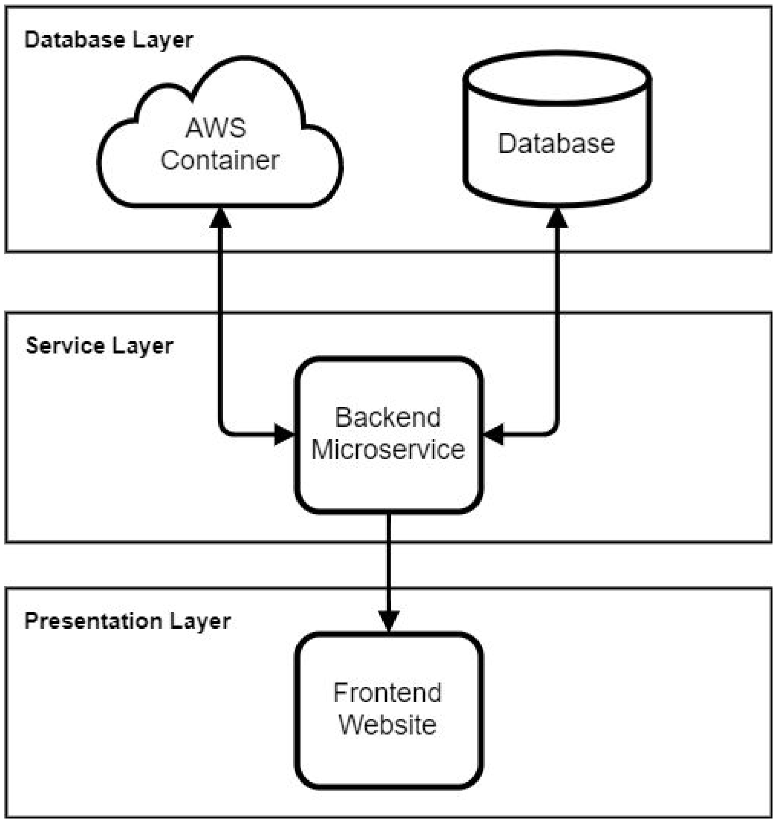

Figure 5

As shown in the graphic, there is data flow occurring between all three layers, but it is not always bi-directional. It may be useful to consider this flow of data similarly to the five layer model of networks: each layer only needs to know and be responsible for anything occurring at its layer, and nothing else. This ensures as little dependency and coupling between the layers as possible.

4.1 Database Layer to Service Layer

Our database layer consists of the AWS containers, where IBM clients' data is stored, and our own MongoDB database which will store JSON documents representing the clients' containers.

Both of these components will need to talk to the back-end program, written in Golang, which will be responsible for analyzing input JSON documents as well as handling AWS containers. When the Golang program is finished with a container, it will feed a JSON document back to the MongoDB database. In order to make these connections, two challenges must be considered: modularity and authentication.

We require a modular design so that our microservice will be capable of communicating with AWS and with our MongoDB database no matter where they are (physically or logically). For AWS, this is as simple as HTTP requests to the Amazon Web Services API. For MongoDB, we will need to have configurable database paths to ensure that, regardless of where the database is, we can access it.

Authentication requires that a user have credentials set up for a given system, and the correct credentials are used when attempting to interact with that system. Fortunately, both AWS and MongoDB innately handle authentication; all we need to do is supply the components with correct credentials.

4.2 Service Layer to Presentation Layer

The presentation layer is made up of both the front-end web portal and data visualization components discussed earlier in this report. Both of these will be fed data from the service layer about the contents of a client’s container, as well as previously discussed useful metrics. Note that the presentation layer is not sending any data to the service layer: IBM clients will not be able to directly invoke our microservice, and instead are only shown data about their container.

There are no major hurdles in the transition between presentation layer and service layer; the front-end will simply make HTTP GET requests to the service layer in order to retrieve data. Fortunately, ReactJS has many full-stack libraries for communicating with Golang to facilitate this. Likewise, the service layer will receive those requests, fetch the appropriate data from the database layer, and send it to the presentation layer.

The only requirement here is authentication: we need to make sure that the data being sent to the user is intelligible only by that user. However, Javascript and Go both have libraries for hashing data and other security measures, so we do not expect this will be a problem.

5 Conclusion

At IBM, Spectrum Protect offers a variety of licenses for cloud data storage management tools to their clients. Part of the contract surrounding these licensed tools is a provision about IBM not managing “expired” data--expired meaning severely outdated or out-versioned. But Spectrum Protect does not yet have an automated means by which to remove their clients’ expired data. Providing that automated means is the goal of Nimbus Technology.

In that effort, we are using this document to detail the technological hurdles we expect to face when designing and implementing this project, and analyze the options available to us to overcome those hurdles. After considering those options and weighing them against the needs of our project, we have made decisions about the core technologies we will use, shown below:

Technological Problem	**Technology Solution**	**Confidence**
Front-end web application	ReactJS	4
Data analytics and visualization	D3JS	3
Back-end implementation	Golang	4
Document Database	MongoDB	5

Figure 6

Note that the confidence values given in Figure 6 are on a scale from 1 to 5, 1 being absolutely unconfident, and 5 being absolutely confident. We currently have a middling level of confidence for D3JS due to the steep learning curve it carries, and the toolchain surrounding its use. However, with enough time invested, we are sure we will be able to effectively use D3JS.

Having done extensive research into each technological domain of our project, and compared several options for each domain, we are quite confident that we have chosen the best technologies to meet the needs of our project. We strongly believe that we can successfully implement a microservice that will both handle culling expired data from IBM clients' containers, as well as help IBM move away from a monolithic design and towards microservices.

Given all of this, we at Nimbus Technology are fully confident that we can solve the problem IBM has tasked us with, and we are ready to move forward in the design process.