University of Nevada, Reno

Microservice-Based System for Environmental Science Software Applications

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science
in Computer Science and Engineering

by

Vinh Dac Le

Dr. Sergiu M. Dascalu, Thesis Advisor
Dr. Frederick C. Harris, Jr., Thesis Advisor

August, 2018

THE GRADUATE SCHOOL

We recommend that the thesis
prepared under our supervision by

VINH DAC LE

Entitled

Microservice-Based System for Environmental Science Software Applications

be accepted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

Sergiu Dascalu, Advisor

Frederick C. Harris Jr., Co-advisor

Yantao Shen, Graduate School Representative

David W. Zeh, Ph.D., Dean, Graduate School

August, 2018

Abstract

Often times as an environmental research project grows, technical aspects such as system scalability, data exposure, and third-party application support are overlooked. This is largely due to researchers not possessing the necessary resources and technical knowledge to implement a distributed system that supports that growth. This thesis presents a system, the Microservice-based Envirosensing Support Applications (MESA), stylized off a microservice-based architecture that provides a scalable environment and data infrastructure solutions for the NSF-funded Solar Energy-Water-Environment Nexus Project. MESA can be broken into three major parts. The first part showcases the overall system flow and the suite of microservices developed to provide complex software solutions and infrastructure. The second part details the service discovery, which plays the role of overseer and tester for the other microservices. Finally, the third part gives a deeper look into the application support that this system actively provides.

Dedication

To my entire family, both at home and school, for the immense support in pushing me this far and to the CI Team who have been the best group of people I have ever had the privilege of working with.

Acknowledgments

I would like to thank my advisers, Dr. Sergiu M. Dascalu and Dr. Frederick C. Harris Jr., and my committee member Dr. Yantao Shen for their time and suggestions.

This material is based in part upon work supported by the National Science Foundation under grant number(s) IIA-1301726. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the last ten years, there has been a substantial push for autonomous high volume data collection in the earth sciences. This push was driven by the need to answer crucial questions that are affecting current and future generations; global warming, water management, and energy. However, with each step forward and advancements made in an environmental project, technical knowledge finds its way as burdens for researchers and scales up to levels beyond the scope of the Earth sciences.

1.1 Problem Statement

When it comes to environmental research projects, one of the most difficult subject to broach is the handling of data once it has been gathered. A common approach to data storage in the earth sciences is spinning up a monolithic system consisting of one or more databases, interwoven code, and a small suite of sensors streaming in data at specific intervals. This approach has its merits in smaller projects when the data gathered is not overly immense and still easily manageable by technicians. However, as data begins to accumulate and more sensors are deployed as the project begins to grow, new problems emerge.

The querying of data becomes increasingly more difficult due the size of accumulated data entries. Additionally, once the data amount reaches a point where it becomes viable for meaningful analytics or quality checks, the lack of a scalable infrastructure forces researchers to seek aid from expensive third-party applications.

Finally, as the monolithic system meets more usage from researchers, more failures will occur and each time they do, the entire system will be forced to reset.

Addressing these concerns naturally suggest the reconstruction of the existing system, but this would cause an enormous investment in time and development. However, the greatest risk of all would be the suspension of research within an environmental project. With that in mind, if the system in question contains vital functionality to continuing research, could environmental research projects afford the time to shut the system down to rebuild and calibrate it?

1.2 Solution

Usually, the choice to restructure an existing monolithic system into a more modular and scalable setup requires a significant amount of time and money. This is due to the sheer amount of technical knowledge and time needed to unravel and rebuild the system. This approach is especially not viable for smaller and growing environmental projects. To address these, and many more problems, this thesis presents the Microservice-based Envirosensing Support Applications (MESA), a distributed support system based off of the Microservice Architecture style that is tailored for the use in environmental research projects. MESA utilizes a suite of independent services, dubbed "microservices", that provides infrastructure and complex software solutions to support an existing system, all without having to tear down an existing monolith.

Through the Microservice Architecture style, MESA is designed to be scalable and can provide the development platform that would enable technicians or other developers within a project to easily create their own tools. MESA's microservices are all independent of one another and the original system, so should a functionality fail, the entire system would retain operational status. In fact, MESA is designed so that should a service fail suddenly, an identical one would be be easily brought back up with minimal effort. Finally, MESA provides a means for technicians to monitor the system and apply autonomous tests through a Consul service discovery hub.

In order to ensure that the MESA system best meets the functional requirements

set down by stakeholders, the features of this system were compared against other existing microservice architectures. Specifically, the systems that were referenced were: OceanTEA, the Public Environmental Information System (PEIS), and the DIMMER Smart City Platform. These systems were chosen because, like MESA, each of the microservice architectures were designed around the purpose of being an applications platform for their respective projects.

Through this feature comparison, it was established that MESA provided more functionality than both OceanTEA and the DIMMER Smart City Platform. However, MESA was shown to be slightly deficient when compared to the Public Environmental Information System due to the fact that PEIS has support for Machine Learning and High Performance Computing solutions. We address these deficiencies by indicating that these are key areas of future work.

The rest of this thesis is structured as follows: a detailed background into the NRDC, the original monolithic system, an overview of microservice architectures, and the libraries used are presented in Chapter 2. Chapter 3 goes into depth over related works inside and outside the Earth sciences. Chapter 4 provides a deeper look at the software solution in the form of specifications and design. Chapter 5 showcases the implementation of the microservices alongside the application support it provides with other affiliated projects. Chapter 6 discusses how MESA compares with similar systems. Finally, the paper wraps up in Chapters 7 with a discussion of the future work and the conclusion.

Chapter 2

Background

In this chapter, the thesis starts off in Sections 2.1 and 2.2 with a brief overview on web technologies, with an emphasis on web services. Next, the concept of a monolithic architecture is discussed alongside the Nevada Research Data Center (NRDC), the monolithic data system MESA was designed to interface with, in Section 2.3. Following up after is Sections 2.4 and 2.5, where it provides a technical look at a Microservice Architecture and the role it plays in MESA. This includes the decision of using the architecture, the benefits, and potential detriments. Moving on, Section 2.6 briefly goes over key information that is vital for the project domain that MESA is designed for. Finally, Section 2.7 covers a detailed listing of the components and libraries that enabled MESA to provide the necessary support that it does.

2.1 Web Services

Before getting started with the thesis, it would best serve to start off with a bit of some light introduction into the world of web development. With the dawn of the current digital age, technology shifted towards the exchanging of data and the integration of the internet into just about everything. One major innovation was the idea in which communication between software was no longer determined by physical hardware connection, but through transfers of data across a network, or the internet in this case. These transfers were conducted through intermediaries, known as web services, originating from the Remote Procedure Call (RPC) component of

the Distributed Computing Environment (DCE), which was a framework for software development [29]. RPC was developed for DCE in order to conduct distributed computing on a single system around the early 1990s, however Microsoft actually retooled this component into the Microsoft RPC (MSRPC) in order to handle interprocess communication across separate systems. It is through this ironic twist that the first foundations of web services were laid. Eventually this technology would find its way into virtually all forms of frameworks at the time such as Java SE/EE and Microsoft's own DotNet. In the end, this process evolved further into the beginnings of the XML-RPC protocol, which is used even today [3, 29]. Web services have slowly established themselves to be the most common and reliable form of data transfer and middleware connection between remote software systems. Even now, web services are involved in many fields regarding the handling of data, such as Machine Learning, High Performance Computing, and Virtual Reality [5, 53, 55].

Although an interesting history lesson, that does not really explain what a web service is. To understand what a web service is, the most important task is to first understand what a service is. A service by technical definition refers to a software

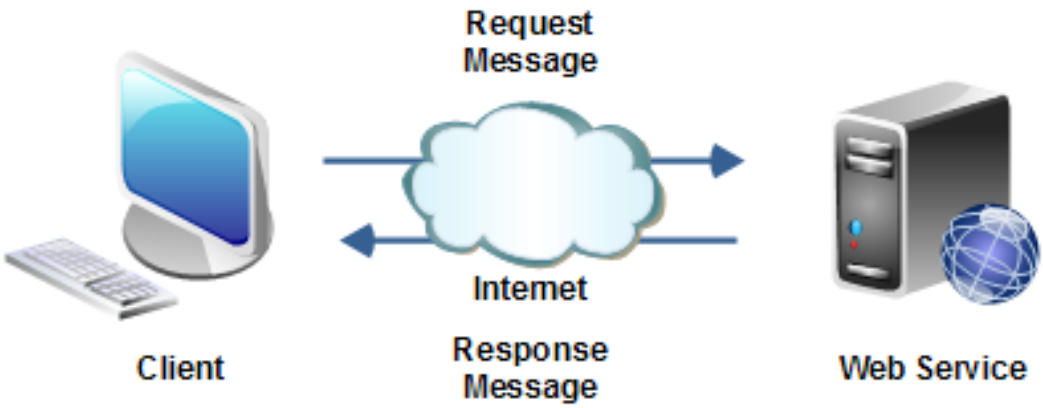

Figure 2.1: This is a representation of the interaction between web service and client [25].

function that executes and accomplishes a business task, also called a business requirement in certain disciplines [16]. A business task is the smallest identifiable and essential job within a component of the overarching project. A web service differs from a normal service in that it seeks to connect whole separate systems and exposes reusable business functions over HyperText Transfer Protocol, or HTTP [6]. Essentially, web services bind complex software functionalities on actual URL endpoints and by calling these URLs, the web service would invoke a software function and relay the result back to a client. An example of this is seen in Figure 2.1. Now that a web service is properly defined, it is worth noting that web services do come in two different flavors of communication protocol. These communication protocols, Simple Object Access Protocol (SOAP) and Representational State Transfer (REST) determine the ways in which a web service are invoked.

2.1.1 SOAP

SOAP was originally developed by Microsoft 1998 and is a legacy protocol that utilizes very strict and structured XMLs in order to invoke functionality within a web service [18]. SOAP consists of three major nodes: SOAP Sender, Receiver, and Intermediary. The SOAP sender functionality generates and transmit SOAP messages. The SOAP Receiver retrieves messages and processes them, and it may optionally generate a SOAP response. The SOAP Intermediary can serve as both SOAP Receiver and Sender, where it receives the headers from incoming messages and forwards it to the receiver. It is through these three nodes that SOAP operates. However, when data is passed through the entire protocol of SOAP, a side effect is that the data is over-inflated due to the standards set by SOAP [32]. Because of this and many other performance related issues, SOAP has long since fallen out of favor of current trending technology practices in favor of REST and now is used almost exclusively in legacy systems [46]. A comparison is shown in Figure 2.2

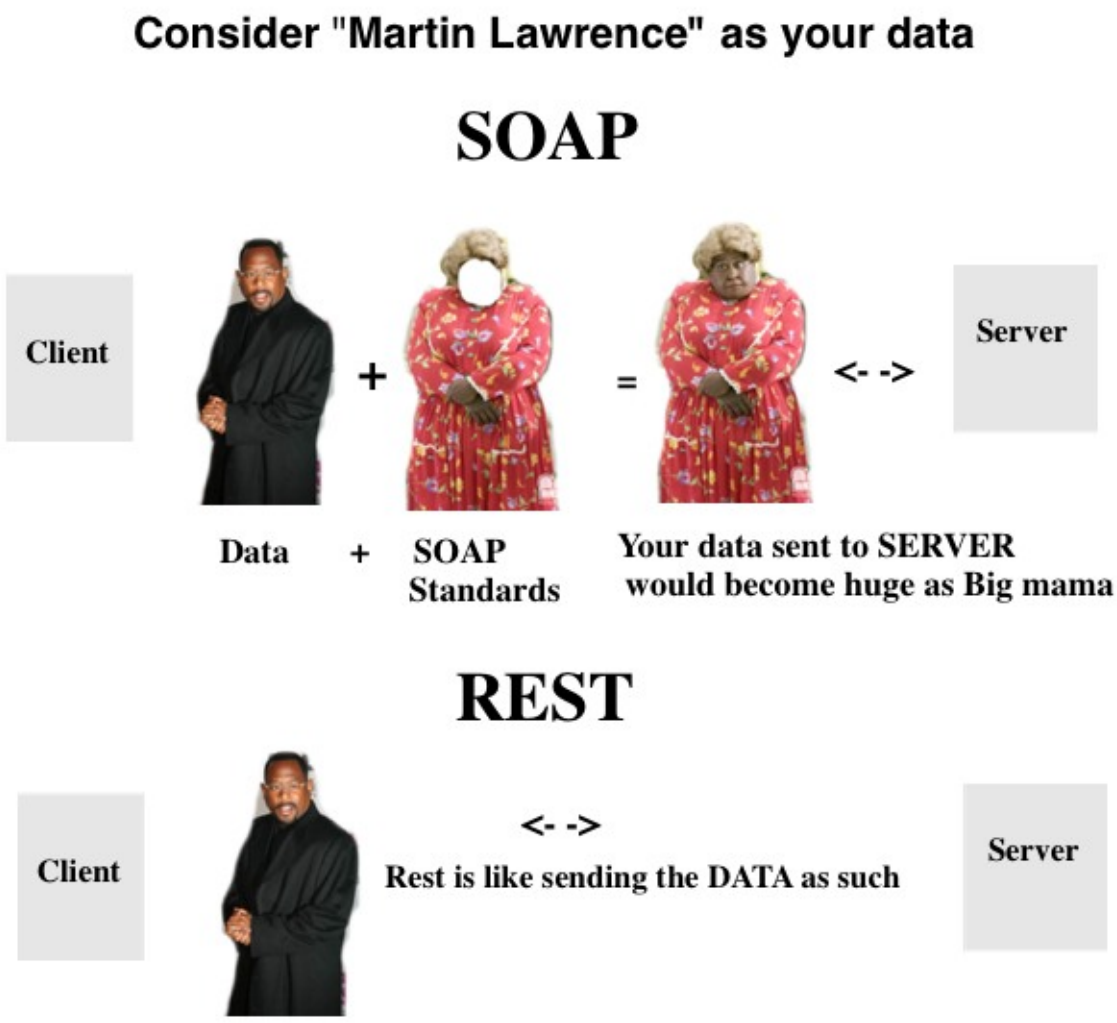

Figure 2.2: This is a comparison between rest and soap when it comes to data transfer [32].

2.1.2 REST and RESTFUL

REST was originally conceived by Roy Fielding at the University of California, Irvine in 2000 for his Ph.D dissertation. This architectural style was designed to be applied with the recently developed HTTP and Uniform Resource Identifier (URI) [11, 52]. The idea of REST is to provide interoperability between any form of software system connected via the internet and this is done through web service instances that utilize the REST architecture, dubbed "RESTFUL Services". Systems will then be able

to query these RESTFUL services through appropriate requests, the main form of interfacing with HTTP, and invoke the intended functionality. With REST, services can now operate with resources other than XML, such as HTML and Javascript Object Notation (JSON), which is the most common form of file format to transfer data today. REST requests are also unique in that each request utilizes a method, which itself is derived from verbs that describe actions to be done on the resource. These methods are: GET, which signifies data retrieval; POST, which signifies data upload; PUT, which signifies the altering of existing data; and DELETE, which signifies the deletion or removal of existing data. Finally, REST provides response back after each request that provides both a status code that shows the current state of the request and a more detailed feedback in plain text. This combination of features allow REST to create a more intuitive style of web service and makes it so that RESTFUL services are considered industry standard even today.

2.2 Modern Web Services

In today's development climate, web services, especially RESTFUL ones, are commonly used for all sorts of backend, middleware, and applications development. Some of the largest tech companies out there today, such as Google, Twitter, Netflix, and Amazon all support RESTFUL APIs for most of their publicized features. Interestingly enough, none of these APIs, despite being advertised as RESTFUL, are not actually RESTFUL. Although accurate, a better phrasing would be that the idea of REST has evolved over the years since it's coining in 2000 [11, 33].

Originally, REST was developed with a handful exceedingly strict rules such as services requiring actual representations of a data instance and that an incoming resource must be absolutely self-describing without any outside context. These rigid pillars of REST quickly fell out of favor as today's technological climate shifted towards an applications and standalone software packages. In fact, very few web services developed by large tech companies even follow the canonically accurate ideals set down by Roy Fielding, one such example being Netflix's API. REST quickly

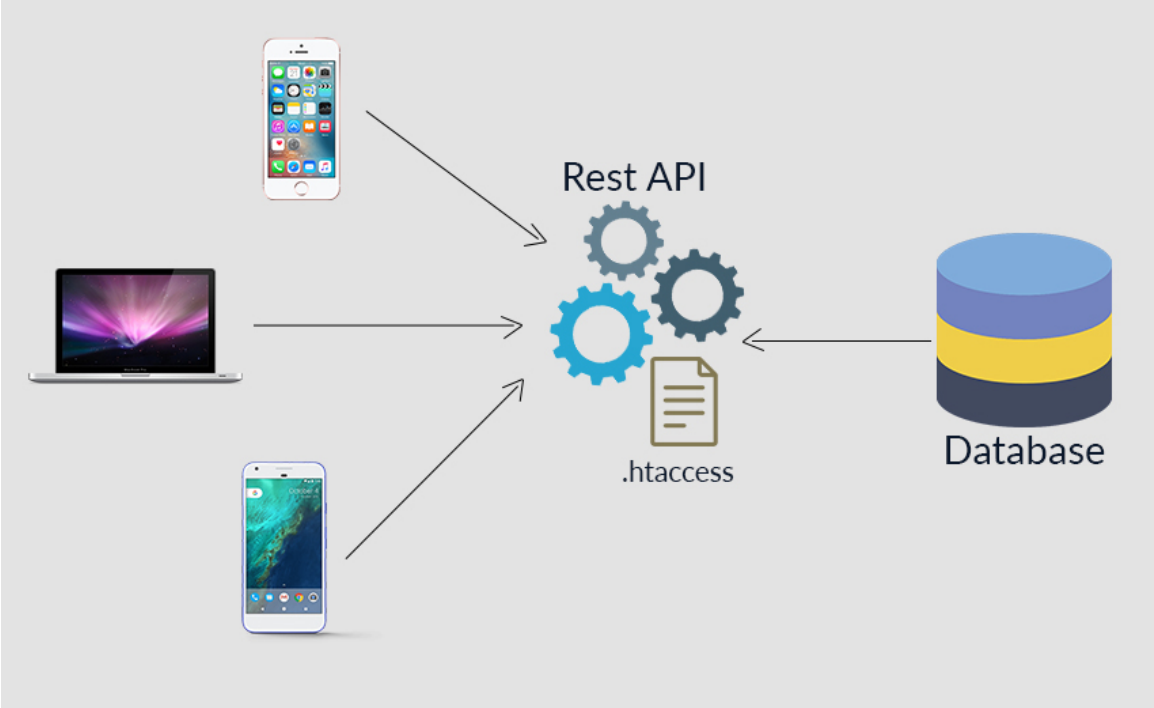

Figure 2.3: The modern approach of rest, aiming towards applications development [26].

became associated with the idea of resources and functionality made accessible by URI endpoints on the internet. Ironically enough, this revolutionary concept made in 2000 would actually be a detriment to our modern web development and through it's bastardization, an architecture style more conducive was born. Figure 2.3 shows the common and modern approach at RESTful APIs.

2.3 Monolithic Architecture

A monolithic architecture revolves around the idea of "End-to-end", where the system's functional aspects are thoroughly interwoven [51]. By being interwoven, this meant the vital components that makes up the system's functionalities are tightly-coupled to each other and requires each others' presence to operate. To give a good example, imagine the idea of the inputs and outputs of a system being closely tied to both the error handling and the user interface. Should one of these functionalities fail, the entire interwoven structure would collapse alongside it. A sample of a mono-

lithic system can be seen in Figure 2.4. Now, this approach is not necessarily bad as it works very well when designing systems based around monetary management. Should something fail during a money transaction, then the entire process would as well. This would be preferable then accidentally being charged for the transaction despite failing. Despite being acceptable for monetary transactions, a monolithic architecture is not the best choice for a growing environmental data collection system.

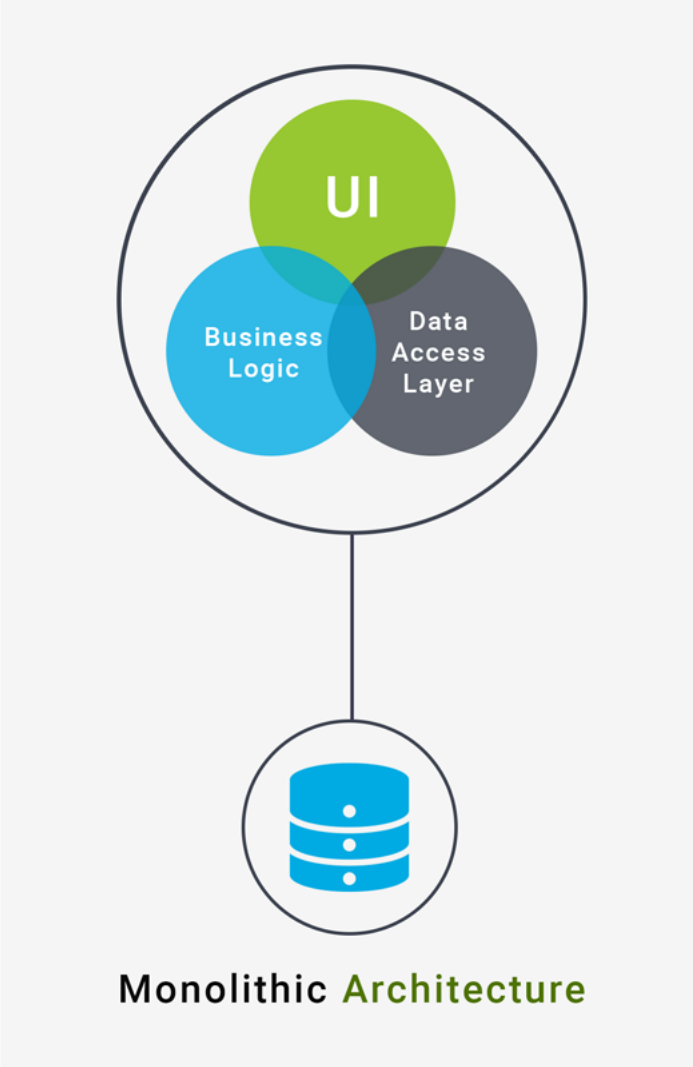

Figure 2.4: This is a representation of a typical monolithic system. The interwoven nature of this monolithic system is shown through the venn diagram encompassing the main functionalities [25].

An example of a monolithic architecture is the Nevada Research Data Center.

The NRDC is the central data hub of the Solar Energy-Water-Environmental Nexus project, where environmental sensor data from various research teams is collected and stored [10, 43]. Unfortunately, the NRDC is the descendant of an older monolithic system and inherited its predecessor's interwoven approach [8]. Due to its interwoven nature, the NRDC system has great trouble even maintaining itself. In order to conduct maintenance on the system, the monolithic nature forces the system to suspend the data collection services. During this time, the NRDC could miss several dozen data entries, which is especially disconcerting for the scientists who are expected to conduct research on the data. The monolithic NRDC had virtually no form of actively monitoring its services' health which made it hard for to tell if a functionality was even online. Finally, the NRDC had no form of supporting the development of tools or application without a base-level modification to the entire system.

Fortunately, this thesis was not the first time this problem was recognized, and there was prior work on proposing a more distributed reformation of the NRDC [38]. However, the development team could not just suddenly scrap the NRDC system and suddenly create a new one. The NRDC was already involved in the recording of research data every minute and many teams depended on this data to continue their research. Therefore, a support system was needed to handle the brunt of the new developments while the NRDC could make careful and deliberate approaches at a sustainable pace to rectify the oversights originating from the legacy system.

2.4 Microservices

The term "Microservice" was traced back to a Microsoft Service Edge Conference presentation in 2005 by Dr. Peter Rodgers. Dr. Rogers referred to the concept of granular web services that remained independent of each other as "Micro-Web-Services" [49]. These granular web services could then be mapped to a specific functionality and the inter-switching of them created a new option for modularity within application. An example of this can be seen in Figure 2.5. This term eventually would evolve and emerge around 2011-2013 in multiple conferences and workshops

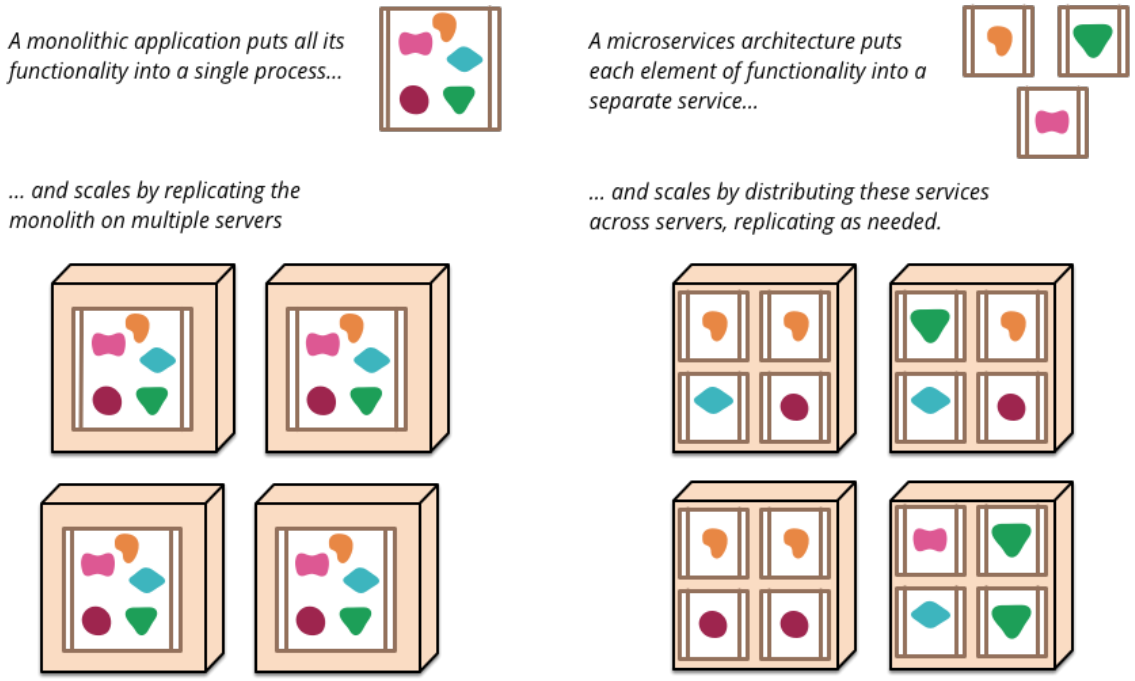

Figure 2.5: High-level look at microservices [13].

around the world. The idea of Microservices generally referred to web services that had singular simple roles that remained independent of each other. By having the web services all independent and granular, each microservice could have varying code bases and differing databases. This would allow each development team in charge of a service to choose the best language or database for the job. Even though the code base and database could be different, each service still communicates through a REST API so there is very little change among the microservices. In addition, often associated with Microservices was the concept of having an overseer service called the Service Discovery which had the role of registering, monitoring, and sometimes testing the health of each microservice.

However, the microservice architecture has its own failing in the form of it being incredibly difficult to implement at first. The microservice architecture style currently does not have any associated frameworks or libraries that enables the fast and easy deployments of its implementations. This is due it's nature as a technology-agnostic architecture so pretty much any programming language with web capabilities can

construct microservices [44]. In order to deploy a microservice-based implementation, it would require the actual construction of such a system by using the concepts as a guideline. This would make it incredibly inefficient to implement for smaller systems but generally very effective in larger and growing ones such as a system handling social media content or an entire state's worth of environmental data.

2.5 A Distributed Approach for the NRDC

Combating the severe limitations placed on the NRDC by its monolithic legacy and its own rapid growth required a distributed approach instead of an interwoven one. There needed to be a way in which the NRDC could have persisting cyberinfrastructure (shortened to CI) options for the Nexus Project even during maintenance. At the same time, many research teams look to the NRDC for their CI support and each team had requirements unique to their own field. There had to be a way so that the NRDC could scale to the needs of it's own members. These concerns were well addressed in the past and plans were made to move forward with a service-oriented approach [38]. Finally, the NRDC simply could not make the time to unravel an old legacy monolith while research groups were still using the stored data. It is because of these reasons that the decision was made to create an external support system based on the Microservice Architecture. The Microservice Architecture was chosen because it's unique ability to separate a monolithic structure into independent components based on business requirements. A comparison diagram of a monolithic system versus a microservice-based on can be seen in Figure 2.6. In this section, a technical overview of a microservices-based system is discussed, as well as the benefits and detriments that this approach brings.

2.5.1 Technical Overview

With the choice made for a microservice-based system to handle NRDC's monolithic problem, what exactly goes into a system like this? Interestingly enough, microservices are so recent and steeped in theory that there is not even a concrete framework

or set of rules that a system utilizing them has to follow. In common practice, development teams assigned to a microservice are often given full reign to the complete development of their service. This means that each service could be entirely different codebases or libraries, as long as they have an exposed HTTP API. The nature of their HTTP API approach allows intense flexibility, but this freedom often comes at the price of standardization. Simply put, no one has ever agreed on a recipe for a good microservice-based system. Fortunately, among the many industry practitioners and researchers, there are common guidelines that are composed of popular practices used by industry. Bear in mind that the following practices are of a smaller set that are deemed significant and relevant to understanding the architecture.

Generally, a good practice for developing a microservice-based system is to actually remain mindful of the decoupling concept. Microservices are designed to combat the rigidness of the monolithic interwoven structure. This implies that when designing microservices, each microservice should remain independent of its other services. By remaining independent, the microservice would not drag down the system when

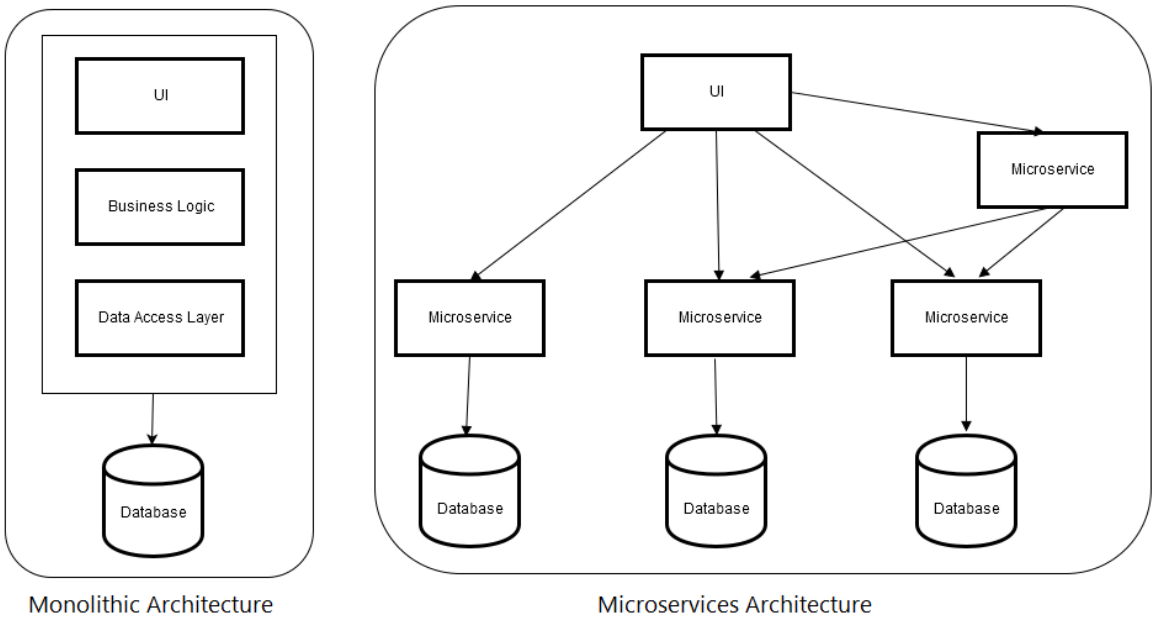

Figure 2.6: This is a visual comparison of a microservice system versus a monolithic one. Should a microservice fail, it would not compromise the overall system [56].

it eventually fails. However, the independence of the microservices make it difficult to manage without a way to locate and monitor them. Often coupled with this practice is the concept of a service discovery. A service discovery is another microservice whose task is to monitor certain ports that are registered on it. Without interfering with the other microservices, a service discovery pings and record the results of all services registered to it. Additionally, service discoveries are often used by industry to set specialized autonomous tests to further check the health of their microservices. A common example of this would be the Consul Service Discovery, a third-party service developed by HashiCorp [19]. A high level diagram of a microservice-based system with a service discovery is shown in Figure 2.7.

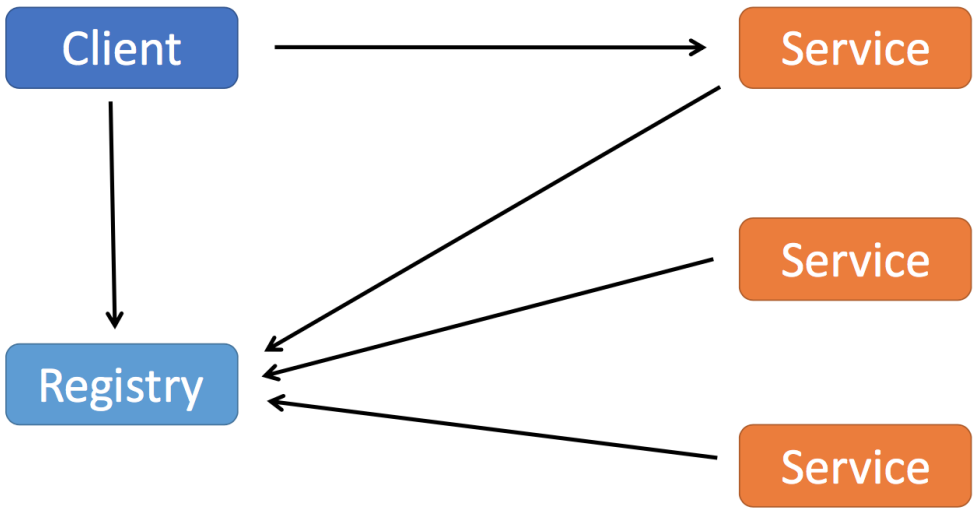

Figure 2.7: This is a common and simplified design of a microservice-based system. The diagram does not cover the database integration but instead focuses on the relationship of the microservices with the service discovery and a client [19].

Continuing on, a good practice for microservices is the separation of critical and secondary functionalities. To elaborate, when designing a microservice-based system, developers should identify the core functions that are essential to its operation. Once identified, these core functions should each be encompassed by a set of microser-

vices and left alone. The separation of these functions give development teams more breathing room to operate without fear of shutting down any portion of the grander system. Before moving on to the benefits, a final good practice would be clarifying the responsibility of your microservice. A microservice by its very nature is the representation of a singular business requirement. That means the microservice being developed should not overstretch its scope and fit in anything else.

2.5.2 Benefits

Often times, the NRDC staff is contacted by the various other groups within it's parent project, the Solar Energy-Water-Environment Nexus, for software support in the form of quality control, data analytics, or some manner of visualization. Usually, these requests are met with strained commitment due to the NRDC system not having the capacity to adapt without severe base-level modifications. However, the addition of a distributed support system can alleviate the burden of these requests by providing the following features.

Microservices provide the means to separate business requirements imposed on the NRDC by distributing them to an individual microservice. The very nature of distributing business requirements to independent entities within the system brings the benefit of scalability, should the business requirements change or increase in the future. Each microservice is independent of each other and can be specialized to handle the software needs that a Nexus component may require, such as refined searching of data entries, the uploading of files, or even the security access of certain data. Additionally, these microservices are exposed over the internet through HTTP endpoints and lessens the need for technical coding expertise among scientists. More importantly though, these microservices can be combined together to create more complex software solutions and are compatible with all forms of programming languages. With this system, Nexus CI developers can now create tools to curate incoming or stored data, design applications to support Nexus fieldwork, or even generate customizable data visualizations without needing to deeply modify the greater NRDC system. Fi-

nally, this system was designed with failure in mind, so a Nexus CI developer or technician can simply restart a microservice without fear of compromising any other microservice or the entire system.

2.5.3 Detriments

As one may expect, the implementation of a microservice-based system like MESA comes with it's own set of drawbacks that could deter some environmental projects, namely smaller ones, from adopting this approach. The first hurdle of the microservice approach comes along with its lack of standardization. This means that the development of a microservice-based system is incredibly hard because there is virtually no support and only a sparse amount of documentation to work off of. The next hurdle, when creating the system, it also requires a significant amount of technical knowledge that exists well beyond the realm of most environmental scientists. For smaller environmental projects without a major CI component, this approach would be extremely detrimental because the burden of establishing the system would then fall to the researchers. Finally, another hurdle is the sheer amount of time that is required for designing this system. Before implementing a system like MESA, it requires a great deal of planning, knowing that each requirement can be broken into a microservice. Each service must be designed so that there is no dependencies on other microservices. This could be especially disadvantageous to smaller environmental groups because this process would literally take manpower away from actual research.

2.6 Additional Information

Although not having much to do with the development of a microservice-based system, there is a bit of sub-domain specific information that holds critical importance to understanding MESA and the data it handles. There will be many mentions of a research site within this thesis, however the idea of a research site can vary from discipline to discipline. To be more specific, a research site in regards to environ-

mental sciences refers to the autonomous sites deployed throughout a region, Nevada being the case for this project. These research sites are configured with numerous third-party sensor equipment that are situated around a centralized relay tower to monitor the area immediately around it. An example of a Nexus research site can be seen in Figure 2.8. These sensors are then configured to broadcast measurement reading every set interval by using the tower as a medium. The tower then sends these readings to specific repositories to be read in and interpreted by environmental scientists. This process is called In-situ Environmental Monitoring, where the in-situ portion implies the research site measures only the immediate area around it instead of the region itself. This leads to more refined time-series analysis under ambient conditions, but would require more technical expertise as a trade-off. This is where Cyberinfrastructure and the NRDC comes in!

2.7 Technology Utilized

Developing MESA as a microservice-based system was quite the challenge due to the architecture being technology agnostic and not having explicit support from any language or libraries. The architecture cares only for HTTP calls through a RESTful API so the development of a microservice is language-agnostic in nature. With that in mind, the construction of the MESA system used common and modern web technologies compatible with the database manager utilized by the NRDC. It should be noted that even though this section lists several programming languages and frameworks, a system like MESA can be executed by virtually any RESTful web service development platform.

2.7.1 C# Programming Language

The C# programming language is a powerful and modern choice for web development and has great compatibility with Microsoft products [54]. C# provides access to robust, industry-level software, such as the entity framework for object-relational mapping and the .NET framework for applications development. Much like Java,

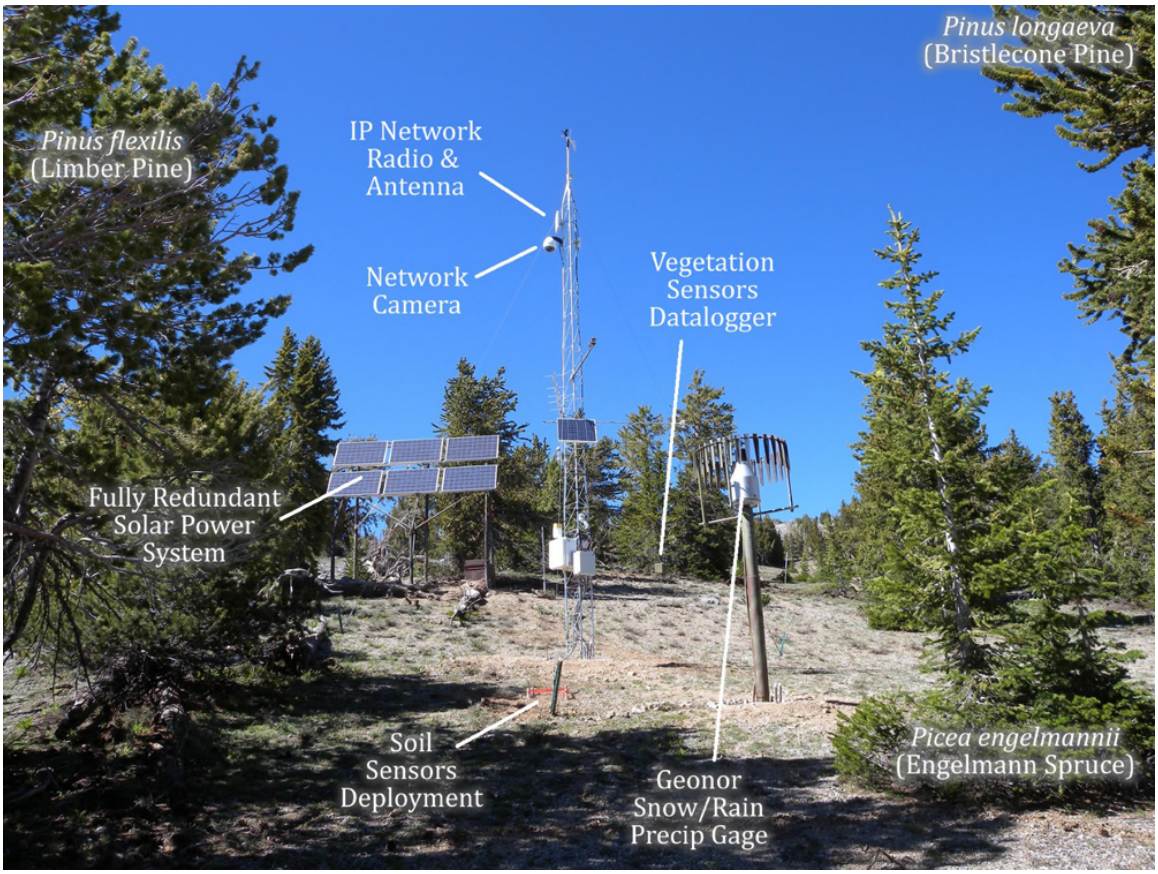

Figure 2.8: This is an example of a research site installed by Nexus Project technicians and environmental researchers. This photo showcases one of the multiple research sites located in Snake Range over at White Pine County, Nevada [42].

C# is both an object-oriented language and generates an intermediate language code after compilation. MESA utilized C# as the codebase for the majority of its data and infrastructure-based microservices due to it's reputation as a dependable web development language, it's massive amount of support libraries, and it's convenient interfacing with the NRDC's Database Management System: Microsoft SQL Server.

2.7.2 Python Programming Language

Although not as popular as C#, the Python programming language is still a strong contender in industry as a web development and backend server codebase [12]. Python, unlike C#, is actually an interpreted language, where code is compiled line by line instead of being done after the entire code is collected. Additionally, Python enjoys

the benefits of having a very popular user-generated content community, leading to an abundance of modules and frameworks available for download. Due to the highly readable and straightforward nature of Python's syntax, it is often chosen by scientists and technicians as their choice of scripting language. MESA utilized Python to develop several microservices designed around supporting Nexus environmental fieldwork. The choice to use Python for MESA's microservices was due to it's many modern web technology modules and that it is a development language that scientists or technicians would feel familiar with.

2.7.3 Windows Communication Foundation

Windows Communication Foundation (WCF) is a toolset developed in the .NET Framework that specializes in implementing and deploying service-oriented architectures (SOA) [37]. This is especially useful since the Microservice Architecture is a variant of SOA and benefits greatly from it's features. This approach differs from the common .NET Web API, a newer and lighter web api framework, in that WCF uses safer protocols and is geared more towards backend development. WCF provides the necessary components for a developer to deploy a web service while making it discoverable via HTTP. Originally, WCF was designed with SOAP in mind, but in 2007, WCF added the JSON support needed for REST implementations. MESA uses WCF alongside C# to deploy microservices that provide advanced data infrastructure solutions to other Nexus CI developments.

2.7.4 Flask

Flask is a python-based microframework that supports the development of web services [1]. By microframework, this means Flask does not require any tools to install it, and does not provide any form validation, database abstraction, or any components for that matter. Flask by itself only provides the bare-bones structure of service creation, however Flask is very modular in that it can be combined with various other Python modules to generate a RESTful service. Several microservices tailored for ap-

plications were developed by using various Python modules alongside Flask, such as the JSON module to handle JSON serialization, the PYODBC module for database connection and abstraction, and the Requests module to handle HTTP calls. However, a major drawback from this approach is that the state of the service depends entirely on whether these modules are kept up to date. Should a module lose support from its developers, it can potentially cause severe issues for the microservice in the future.

2.7.5 Microsoft SQL Server

Microsoft SQL Server (MSSQL) is the main Database Management System for the NRDC and as a product, it's primary function is to store data inside a relational database and retrieve it when necessary [36]. MSSQL is considered to be one of the most popular database systems in the market today and boasts an excellent reputation when it comes to communication across other software applications. As noted from the name, MSSQL is a relational database, but uses an altered SQL querying language, called Transact SQL (TSQL). TSQL not only provides MSSQL with the same features as regular SQL but expands SQL further by allowing transaction control, error handling, row processing, and the creation of declared variables. MSSQL as a Microsoft product shows incredible compatibility with other Microsoft products, making it an excellent choice when deploying WCF microservices from the Windows operating system. The choice of using MSSQL was determined based off of the hardware that was purchased or allocated at the start of the project.

2.7.6 Consul

The Consul Service Discovery was developed by HashiCorp to provide the DevOps and applications development communities an out of the box solution when it comes to managing web services [19]. Consul comes fully loaded with a host of features, such detecting and registering services on a network, applying customized health tests on service autonomously, and displaying relevant maintenance information on a

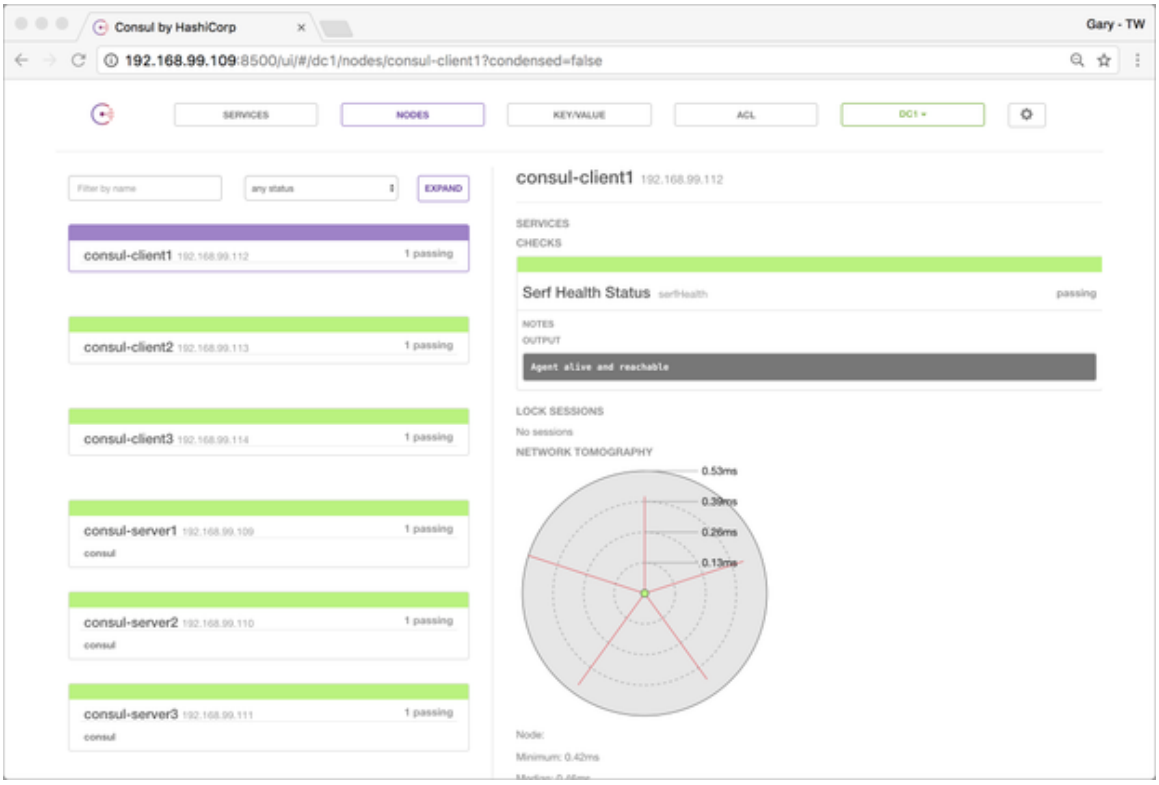

Figure 2.9: This is an example of an interface generated by Consul [19]. On this web page, users can view the current state and health of a service, as well as set up various settings.

generated web interface, as shown in Figure 2.9. This third-party software is free and available for use as soon as it is downloaded without any major installations. MESA decided on Consul as a service discovery option due to it's free nature and it's ability to administer autonomous tests. The decision to use Consul against creating a service discovery was made based on the amount of additional technical knowledge needed to complete a comparable solution. Simply put, it would have expended too much time and effort to create anything exceptional, while Consul is free and robust.

Chapter 3

Related Work

Present in this chapter is an overview of similar distributed systems existing in and out of the Earth sciences. To be specific, the systems presented are distributed systems that enable support for applications, visualizations, or any form of development work. Additionally, due to the strict criteria of covering microservice-based architectures for environmental research, this chapter will instead focus on any form of microservice-based system offering support to applications. Section 3.1 covers several systems that fit the criteria previously mentioned, but a technical comparison between MESA and this selection will be covered in Chapter 6. Section 3.2 provides three similar systems implemented outside the domain of environmental research. Microservices being used for application development has been a hot topic for close to a decade now, so a limit on related work will be placed in this section.

3.1 Similar Work in the Earth Sciences

This section covers similar systems to MESA that were also developed for environmental research. Specifically, each of these systems were created with the idea of providing a platform for additional software development in mind. For each of these systems, a brief overview of the similarities and differences to the proposed MESA solution is presented. As mentioned, this section contains several selections that are covered in further technical detail and compared in Chapter 6.

3.1.1 WC-WAVE Hydrological Model Framework

A framework to visualize hydrological models was developed for the WC-WAVE Project, an environmental collaborative project across three states: Idaho, New Mexico, and Nevada [20]. WC-WAVE is a research project that falls under the same EPSCoR umbrella as the Nexus Project and are often collaborators on certain research ventures. This system used a microservice architecture to generate models based off of standards set by the research domain. The system then returns the models back through HTTP and disseminates them on their client web application. The modeling framework utilizes a suite of microservices to provide security, retrieve information, generate models, and send models to a web application designed to visualize them. Currently, the web application is the only client that utilizes this architecture.

This system bears a striking similarity to MESA because it is also a platform for the development of additional software solutions for an environmental research project. MESA supports a wide number of microservices that share similar functionalities, such as security, data management, and data visualization. However, as much as this system holds similarities with MESA, it also holds distinct differences. The WC-WAVE Modeling Framework focuses primarily on allowing users to execute and store experimental models in a contained environment, while MESA focuses primarily on providing a platform for software solutions. The Modeling Framework appears to be designed with the idea of supporting only one application client in mind, unlike MESA. In addition, MESA provides data infrastructure solutions that this system does not. Finally, this system show any indication of possessing a means to discover and register microservices in order to monitor health.

3.1.2 Generic Service Infrastructure for PEIS

A team of researchers at the Karlsruhe Institute of Technology developed a microservice-based architecture based around a massive nation-wide research project whose purpose is to study environmental information [4]. This system, dubbed Public Environmental Information System (PEIS), uses microservices to provide infrastructure

and application support to various client applications, such as sensor networks, web applications, and mobile devices. PEIS is a redesign of an original monolithic architecture and inherits an established network of sensors that span the entire southwest region of Germany. PEIS focuses on four main microservices that covers time-series operations, asset management, data analytics, and a centralized service called master. Additionally, this system uses both containerization and service discoveries to better manage the independent web services. Finally, PEIS's data analytic features are centered around computing cluster that was made part of their project.

The PEIS system is remarkably similar to MESA in that it uses an approach that was also considered by the Nexus Project and it falls under the umbrella of a larger environmental project. The approach adopted by PEIS involved the complete refactoring and rebuilding of a sensor collection system. Ultimately though, the Nexus project sought to avoid the rebuilding a system that actively collected data every few minutes in fear of losing a sizable amount of research. Additionally, the PEIS system provides much of the functionality that is provided by MESA, including a service discovery and excellent support for current and future applications. A major difference between the two system is the amount of microservices being employed by PEIS is significantly less than MESA. It is not clear as to whether PEIS does not require that much business logic or that the web services are somewhat interwoven. Regardless, PEIS represents an excellently constructed and well designed microservice-based development platform for applications.

3.1.3 OceanTEA: Exploring Ocean-Derived Climate Data

OceanTEA is a support system designed to aid researchers with processing data from a ocean monitoring system designed by the University of Kiel in Germany [28]. The ocean monitoring system deployed by the University of Kiel consists of more than 3000 floating sensors that concurrently stream time-series data at regular intervals each day. The collection of time-series data has created a surge that negatively impacted the management of data within their systems. This support system uses microservices to

structurize the data presented to researchers based on specified criteria. OceanTEA then presents this information through an intuitive and responsive web interface. The system utilizes a handful of microservices that handle four major jobs: authentication, time series conversion, spatial analysis, and time series pattern discovery. Each of these microservices are encompassed in a docker container and feature a different codebase. Finally, these microservices are made available through unspecified cloud technologies or as a desktop application.

OceanTEA greatly resembles MESA in both the motive of why the system was developed and portions of the execution. Both MESA and OceanTEA were designed as a relief to the monolithic system that gathers the actively streaming sensor data from associated areas of research. These two systems also share similarities in that they leverages application support, through microservices, to tools designed to ease the burdens of their project's researchers.

Despite the similarities shared by these two systems, there is an abundance of differences between MESA and OceanTEA. First, the design of OceanTEA gives off the impression that the system is tailored only towards to the structuring and presenting of data to researchers, rather than providing the groundwork for other future tool development. Next, several of the microservices designed for OceanTEA seem dependent on the functionalities of other microservices. This breaks the independent nature of the microservices and opts towards a interwoven relationship. Finally, the OceanTEA system does not utilize a service discovery in order to register services, apply tests, or monitor health.

3.2 External Work

This section provides a look into systems that are similar to MESA in their application support, but differ in the domain of research. It is important to consider that there has been significant development in the last decade regarding Microservice-based Architectures and the applications supported by them. In light of this consideration, this system selection presented has been limited to relatively recent works. Much like

the previous section, choice systems from this selection are given a technical overview and comparison in a later chapter.

3.2.1 DIMMER Smart City Platform

The DIMMER Smart City Platform is the initial case study in which members of the European DIMMER Smart City Project applied a microservice architecture to provide support to various sensor networks placed around affiliated cities [31]. Additionally, the DIMMER platform was also designed to interface with smart city applications that in theory would be used by city officials for administrative work. In order to achieve this, around a dozen microservices were developed between two major subgroups: Middleware for data management and Smart City for applications. Middleware microservices include functionalities such as the management of sensor data, active clients, and the service discovery. It should be noted here that it is unclear as to whether the service discovery encompasses just the Middleware microservices or the entire system. The Smart City microservices provide application support that varies based on the many applications that each microservice is tethered to. These microservices include functionalities that provide energy simulations, create energy optimization strategies, and expose data models of geography, buildings, or systems.

The DIMMER platform shares similar features to the proposed MESA system despite not being part of the environmental research domain. Both systems offer data infrastructure solutions and application support through microservice REST APIs. Additionally, MESA and DIMMER provide a service discovery element to better track the state of their microservices, however it is still unknown as to whether this feature is extended to DIMMER's IoT application microservices. In regards to the differences between the two systems, DIMMER collects the data of the sensor network unlike MESA who relies on the NRDC's system to autonomously collect sensor data from its various research sites. Another difference that follows DIMMER's one system approach is that the autonomous collection of data from DIMMER's sensor network and the support of the IoT applications creates intense network overhead that affects

the quality of life for researchers. MESA does not share this disadvantage because the NRDC system curates data in time-series batches instead of the real-time approach that DIMMER uses. This creates fewer times in which applications or sensors queue for the attention of the central repository.

3.2.2 Open IoT Framework based on Microservices

An IoT platform created by using microservices and the Jolie programming language was presented by a software team from the Innopolis University in Russia. This platform was designed to provide concurrent support to IoT devices and an advance sensor network placed inside a smart building as part of a Human-Building Interaction study [30]. The Jolie programming language is presented as the main component of establishing this microservice-based platform and is described as a language that is designed entirely to develop service-oriented architectures. In this platform, several microservices are written for each connected device as a data abstraction layer, but this amount is not covered in their work. This is possibly due to the sheer amount of devices present within a smart building. Jolie extracts data from connected devices by executing contained Java code to interface with the IoT devices. Alongside each sensor and application, a raspberry pi is connected in order to serve as a reliable way to connect to the main microservice system through their network.

In terms of similarities, this IoT platform provides dedicated application support, much like MESA. Each application connected to the IoT platform and MESA is provided a dedicated data abstraction layer among other feature support. Additionally, both MESA and this IoT platform were developed with powerful programming languages that specialized in web technologies. This IoT approach however, shares far more differences with MESA than similarities despite being a microservice-based platform. A major difference between MESA and this IoT platform is that MESA provides more features in its execution than just data abstraction layers between a tethered device and the main system. Additionally, this IoT platform does not make mention of a service discovery portion, which makes the tracking of system health

and applying tests difficult. Another difference is that the IoT platform collects the data from its sensor network unlike MESA who relies on the NRDC system to collect data. Finally, the IoT is highly reliant on using raspberry pis as relays between the microservices and the IoT devices.

3.2.3 Danske Bank Migration

A microservice-based architecture was developed for the migration of a monolithic banking system out of Danske Bank in Denmark [9]. Danske Bank is the largest bank in Denmark and one of the largest financial institutions based out of Northern Europe. Over the years, the main system of the bank has had troubles in scalability, the interwovenness of their features, and riddled with indecipherable business logic. The system, dubbed FXCore, reinvents the old monolith of Danske Bank into a distributed microservice architecture with the latest technologies. FXCore reinvents the old interwoven services into independent microservices and registers each in a service discovery that monitors the health of the system. The banking system is now fully equiped with automated continous integration and delivery pipeline that is hosted on their internal server. Each of the services are fully containerized with Docker and loaded into Docker Swarm's Orchestration to handle both the service discovery and the load balancing. With these new improvements in technology, FXCore provides Danske Bank with a scalable future that it desperately wanted.

MESA and FXCore shares many similarities in that both systems were designed around the idea that changes were needed in order to handle the limitations of scalability within their respective domains. Both systems separate their respective business requirements and encompasses each one in an independent microservice to be monitored by the service discovery. However, there is a vast difference between the two systems. FXCore provides Danske Bank's systems with the latest and greatest technological advancements in microservice technology, while MESA serves as a application support platform for the NRDC system. FXCore rewrote and replaced the original monolith created by Danske Bank, while MESA preserves some of the critical

systems that the NRDC original implemented. Interestingly enough, this approach actually may not be the best for FXCore. Although an excellent means to enable scalability, microservices do not provide the system critical support that a banking system necessarily needs. Monoliths in fact are often used by banking systems because when a portion of the transaction fails, the entire process fails. This allows banks to narrow the actions caused by transactions to either a full success or a full failure. Anything in between can be disastrous for banking business requirements.

Chapter 4

Software Design

In this chapter, an overview of the software design specifications that defines the operational details of MESA is covered. Section 4.1 of this chapter provides a deeper look into the requirements placed on this system. This entails the functional requirements that describes the required system features, and the non-functional requirements that describes the constraints on the system. Section 4.2 covers the use cases of the MESA system in the form of a diagram and a brief overview. After which, Section 4.3 covers the sequence of invoking one of the microservices. Finally, the Section 4.4 covers the architectural design of the MESA system. This will entail a high-level diagram of the microservices, component and activity diagrams of affiliated tools developed with the microservices, as well as an entity relationship diagram for the NRDC database that MESA interfaces with. To have a further look at a sample microservice, source code is shown in Appendix A.

4.1 Requirements

MESA was developed with several key requirements in mind, and among these requirements, each were separated into three major types: base level functionalities, secondary functionalities, and constraints. In this section, an in-depth look at the requirements is presented in two parts. The first part covers the functional requirements, which consists of the base level functionalities and the secondary or future functionalities. The second part details the non-functional requirements whom rep-

resent the hardware and software constraints placed on the MESA system.

4.1.1 Functional Requirements

The functional requirements in the context of software systems are the set of technical functionalities provided by the described system. In the case of MESA, the functional requirements are split between three levels. The first level denotes the functionalities that are considered basic and vital to the very operation of MESA. The second level of the functional requirements are functionalities that are not vital to the operation of MESA and provide an additional feature to the system. The third level are functionalities that are considered low-priority or unfinished, and are often associated with future work. In fact, these level three requirements and further plans are discussed later on in Chapter 7. A table showing the functional requirements of all levels is shown in Table 4.1. In the following text, the requirements of each level will be given a detailed overview.

In regards to the level one functional requirements, there are a total of ten that make up the core features of MESA. The first requirement establishes that MESA provides applications with the standard features found within a database without having to tether directly to it. The second requirement ensures that the data retrieved from the data source is not a massive pull, but rather a curated batch that follows the standards set by the application. The third requirements enables MESA to be language-agnostic by setting communications to HTTP protocols only. The fourth requirement combats the interwoven nature of monoliths by having the microservices remain independent of each other. The fifth requirement involves the registering of each microservice inside a service discovery service. The sixth requirement entails the tracking of each microservice through regular heath checks via the service discovery. The seventh requirement is the automated and custom testing of the microservices through scripts written in Bash and Python. The eighth requirement involves the system remaining operational even if a microservice fails. The ninth requirement is the detection of conflicts within requests issued by multiple applications. The tenth

requirement established that MESA provides more than just simple data operations and these capabilities include automated testing, data curation, photo operations, and data visualization. More on this is covered in Chapter 5 .

Unlike the level one functional requirements, there are only two level two requirements that serve as additional features outside the base level ones. Both of these level two requirements involve a specific type of data gathered by research sites: high definition photos captured by station cameras. The first requirement covers the functionality of storing and retrieval photos. This is significant because the photos are not stored as actual images and are instead encoded as byte arrays before storing then decoded once retrieved. The second requirement covers a more unique feature involving cropping or compressing a high definition image as a preview on an application. This was created in order to address the massive amount of data involved in loading a high definition picture. Instead of loading all images at once, a smaller

Name	**Level**	**Description**
FR1	1	The system will handle the data abstraction processes between application and data source
FR2	1	The system will structure retrieved data based on the needs of the application
FR3	1	The system will handle all communication via HTTP protocols
FR4	1	The system microservices will remain independent of each other
FR5	1	The system will register each of the microservices
FR6	1	The system will track the state of each of the microservices
FR7	1	The system will test each of the microservices autonomously
FR8	1	The system will remain operational even when a microservice is offline
FR9	1	The system will detect conflicts in the submission of data
FR10	1	The system will provide complex software solutions to applications
FR11	2	The system will handle the storage and retrieval of high-definition images
FR12	2	The system will provide compressed or cropped versions of present images
FR13	3	The system will utilize Continuous Integration
FR14	3	The system will utilize Cloud Technologies

Table 4.1: Functional Requirements

preview is set as the main display, but if the user wants to expand further, it would load a high definition image.

Much the like the level two requirements, the level three requirements consists of only two features that represent work considered to be low-priority or unfinished. The two requirements presented cover the work usually practiced in microservice-based systems found in industry. The first requirement covers the usage of continuous Integration. This feature is not present within MESA and involves using devops tools such as Jenkins [21] to automate the process of merging working developer copies into a shared mainline. This feature, although critical in most engineer teams in modern companies, was not added to this thesis implementation due to it possessing only one developer and was created as a proof of concept. The second requirement proposes the idea of utilizing cloud technologies such as Amazon AWS and Microsoft Azure to host microservices instead of running them on the NRDC physical servers. This was not implemented because the servers at the NRDC are not currently in any risk of running out of memory. Additionally, the implementation of cloud technologies from third party owners cost a set amount of money each month that would not outweigh the costs of maintaining it on the physical servers. This will be discussed further in Chapter 7.

4.1.2 Non-functional Requirements

While the functional requirements are quite numerous, the non-functional requirements number under half of the former. Each of the six non-functional requirements serve as constraints that are placed on the MESA system, as shown in Table 4.2. It should be noted that even though these constraints specify certain frameworks and languages, a microservice-based system can be written in any language and it's appropriate web framework. The first requirement states that MESA was developed with the C# and Python programming language. Alongside the first, the second requirement describes the frameworks, Flask and WCF, that were mainly used in development of the microservices. The third requirement explains that MESA was

developed with a combination of both Linux and Windows. The fourth requirement covers the type of database management system, SQL Server 2012 specifically, in which the MESA system interfaces with. The fifth requirement describes the third-party software used to implement the service discovery. The sixth and final requirement explains on how the MESA system was deployed, one portion with IIS and the other through NGINX reverse proxy.

Name	Description
NFR1	The system will use the C# and Python programming languages
NFR2	The system will use the Flask Microframework and the WCF Toolset
NFR3	The system will operate on both Linux and Windows
NFR4	The system will communicate with a SQL Server 2012 database
NFR5	The system will use Consul 1.2.1 as a service registry
NFR6	The system will deploy via IIS or NGINX reverse proxy

Table 4.2: Non-Functional Requirements

4.2 Use Case Modeling

In this section, the use case modeling is covered for the MESA system and the following descriptions are presented as a use case diagram in Figure 4.1. In the following text, the use cases for the MESA system are broken up between it's two actors and are briefly discussed.

The first actor is the Application Developer, or development team depending on the project and various logistical elements. As an application developer, they are able to perform a GET request on a MESA microservice to invoke it's data retrieval operations. The developer is also able to perform a POST or PUT request on the MESA microservice to make new changes or update older entries, respectively, on the database. Additionally, POST requests are also the central way for developers to invoke functionality outside that of database accessing. The next use case is the ability for Developers to invoke a DELETE request on the MESA microservice to remove a data entry or setting, depending on the feature that the microservice

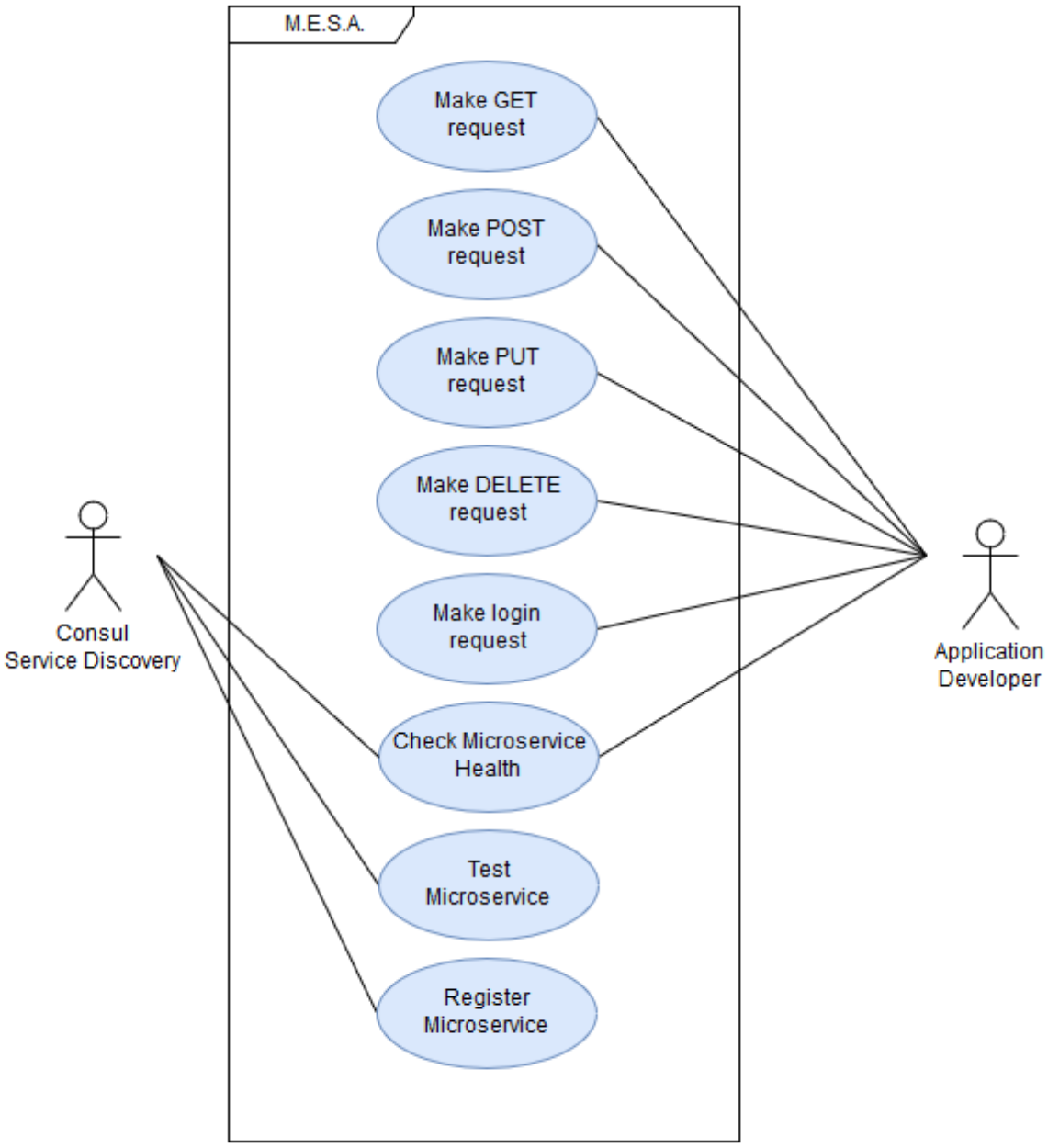

Figure 4.1: Use Case Diagram of the MESA system.

governs. The next use case for the developer is to log-in into the MESA system with an authorized username and password by interfacing with the Login Microservice. Finally, the last use case for developer is also one shared by it's other actor, the Consul Service Discovery, and that use case is the ability to check the health of a microservice by querying a specific URI set by the MESA system.

As mentioned, the other actor for the MESA system is the Consul Service Discovery and aside from the one shared with the Application Developer, it only has two use cases. The first use case involves the service discovery using the system's ability to set autonomous tests that are configurable by user-made scripts. The second and last use case is the ability to register a microservice under the service discovery and apply the appropriate settings by sending a detailed JSON to the MESA system.

4.3 Sequence of Microservice Invocation

In this section, the invocation of a microservice is covered in detail, with a POST request to the Site Network microservice serving as the primary example. A sequence diagram is also provided, shown in Figure 4.2, to illustrate the flow of events within the microservice. The following text briefly describes the sequence from the start of a client request to response made by the microservice.

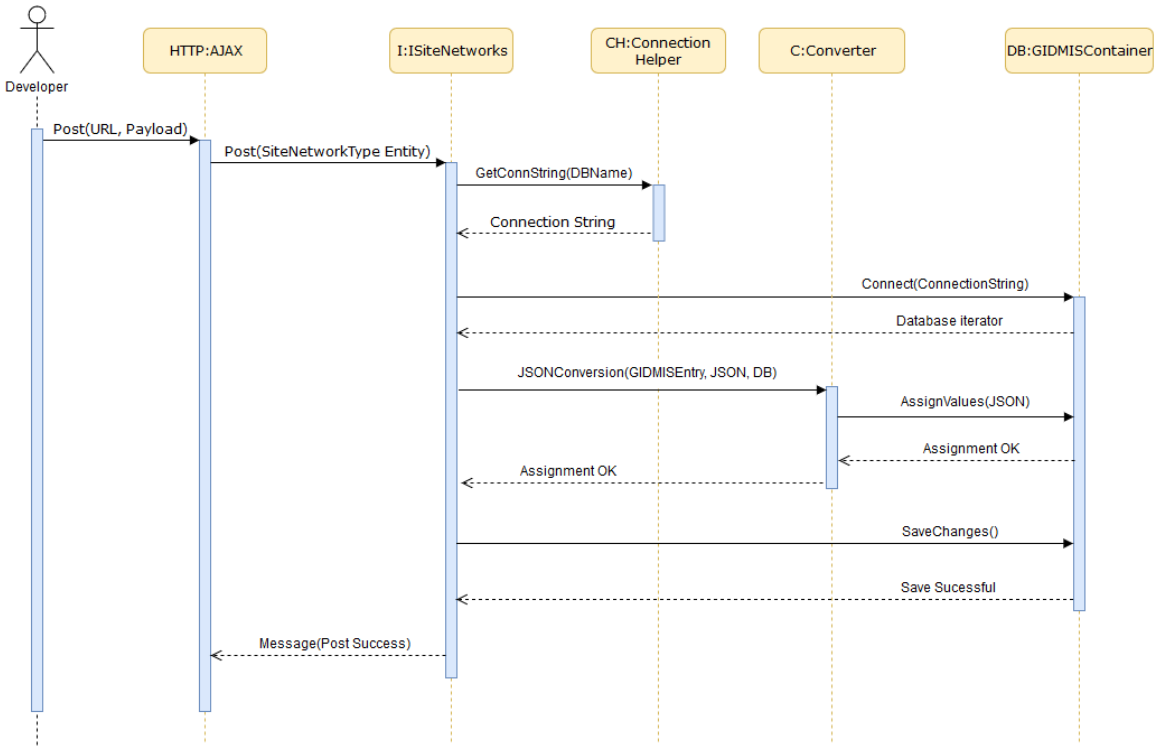

Figure 4.2: Sequence Diagram of a generic POST request to a microservice.

Starting first with the client, a developer's client constructs a JSON payload and

sends a HTTP POST request with the JSON attached to appropriate POST URI. Then the payload is stored as a SiteNetworkType entity and the Post functionality of the SiteNetwork microservice is called. Once called, the GetConnString function of the Connection Helper class is activated to retrieve the proper connection string. The connection string is then used by the Connect functionality of the GIDMISContainer class in order to retrieve an iterator to navigate the database. With this iterator, the system queries whether or not an entry matching the SiteNetworkType entity exists. If it does not exist inside the database, the iterator is used to create a new entry instance and calls the JSONConverter function of the Converter class. This function then translates the string based values of the SiteNetworkType entity into the proper data types and sets them to the newly made entry instance. Once this is complete, it returns back to scope of the new instance and then attempts to save the changes made to the database with the SaveChanges function of the database iterator. Once saved, the SiteNetwork microservice sends back an appropriate success or fail response to the client.

4.4 Architecture

In this section, the architecture of MESA is presented through three main segments. The first segment will showcase the high-level design of MESA with help from a design diagram. This provides a generic overview of MESA and relations it shares with various supported applications. The second segment will give a deeper look into MESA's microservices and some ways in which they are utilized for associated applications. This will entail the component-based diagrams and activity charts of two applications to serve as a proof of concept. The last segment will cover the NRDC Database which stores both the Metadata and the data retrieved from the research sites. A set of entity-relationship diagrams will follow this segment and show the overall layout of the NRDC Database.

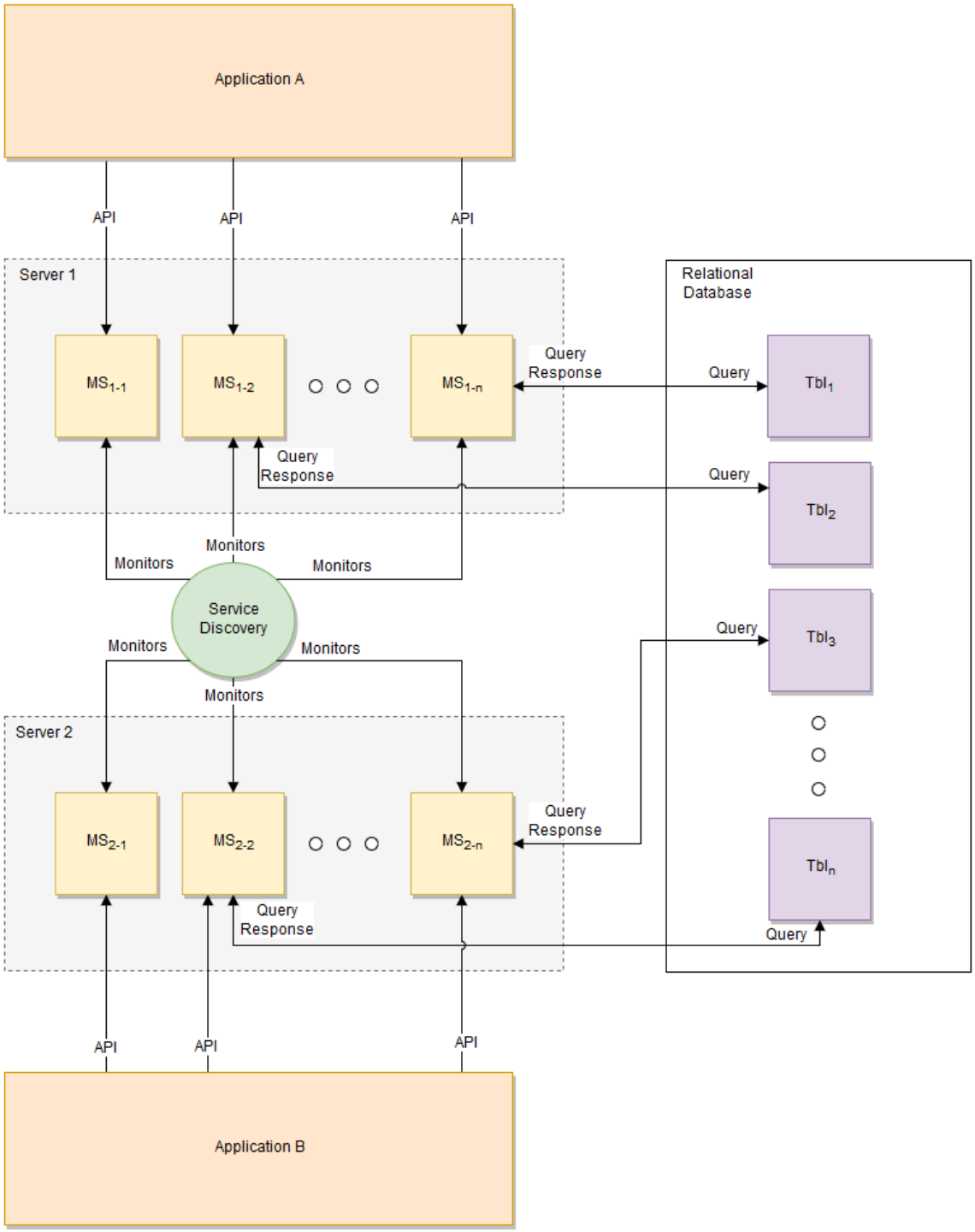

Figure 4.3: High Level Design of the MESA system. Each of the microservices are denoted by a label in the format of MS_{x-y} where x indicates the generic application it services and y being a count that totals up to n.

4.4.1 High-Level Design

MESA as a system has four major components, and a high-level representation of this is provided in Figure 4.3. At the center of MESA is the Service Discovery, which serves as a both registry and monitor for all of the microservices. The Service Discovery does not actively entangle itself with any services, aside from scheduled tests, and provides necessary metadata regarding location and health. The next and most crucial portion of MESA is the microservices. The microservices are all independent and represent the main features of the MESA system. The microservices are often shuffled into servers associated with development on specific applications. It is here where the microservices expose their functionalities to the application through it's HTTP API. This is shown on the diagram where microservices are labeled in the format, MS_{x-y}, to indicate the generic setups between microservice and application. Additionally, microservices are occasionally tethered to a database table to serve as an abstraction layer, however some provide other complex software functionalities and this is denoted in the figure. Moving on to the next component, the tables of the database are not only utilized as a general data abstraction between client applications. Tables, such as those containing the sensor measurements of NRDC sites, are called and utilized in certain microservices to perform complex calculations or data management operations. Finally, the last component are the multiple applications that utilize MESA as their support. The applications in mind are not limited to web applications, but also include phone applications and actual separate systems. These applications all share one aspect and that is the sending of RESTful requests to the exposed APIs of MESA in order to activate it's many features.

4.4.2 Application Design

The cluster of microservices that are allocated towards an application end up serving as the entire server backend of the client application. Simply put, each microservice represents a major operation that performs on the more powerful server machine in order to fulfill a feature provided by the client. To show off an example of this setup,

the architecture of the NRDC QA Application, an application designed to support Nexus technicians, is shown. More on this application is described in the next chapter. In the NRDC QA Application, the client was created using the Model-View-Controller (MVC) architectural pattern, as seen in Figure 4.4. While the view portion handles the formal frontend website and the controller deals with event handling, the model ties in the the server backend. Through the model, request JSONs are crafted and then sent through HTTP to invoke the functionality of the microservices. This can be anywhere from storing or retrieving data to handling conflicts. The microservices performs the appropriate action based on the URI invoked by the HTTP request and the result is sent back to the client as a response. This data can then be used by the client's controller and view to affect the overall interface.

So far, the role of the microservices in a client application is discussed, but acting as a server backend contains more than just free-floating microservices. The microservices leveraged for a client application are designed much like other systems with relationships across the other components. The only difference in this situation, is that the microservices are loosely coupled and do not critically depend on one another to stay online. An example of this loosely coupled setup of microservices is the backend that provides support to the NRAQC system, as seen in Figure 4.5. NRAQC is a system that handles the automated quality control operations on the sensor data, and is also described in the next chapter. NRAQC in this instance shows that there are not only free-floating microservices such as the Flag Handler and the DataVis Handler, but microservices whom are designed to support other microservices. An example of this would be the Data Manager microservice, where it's job is to be a communication/transfer hub between the datasource, the Configuration Manager, and the Testing/Flagging Service.

4.4.3 NRDC Database

The data utilized by MESA's affiliated applications are drawn from a database that is replicated regularly from the main NRDC data stores. The database utilizes Microsoft

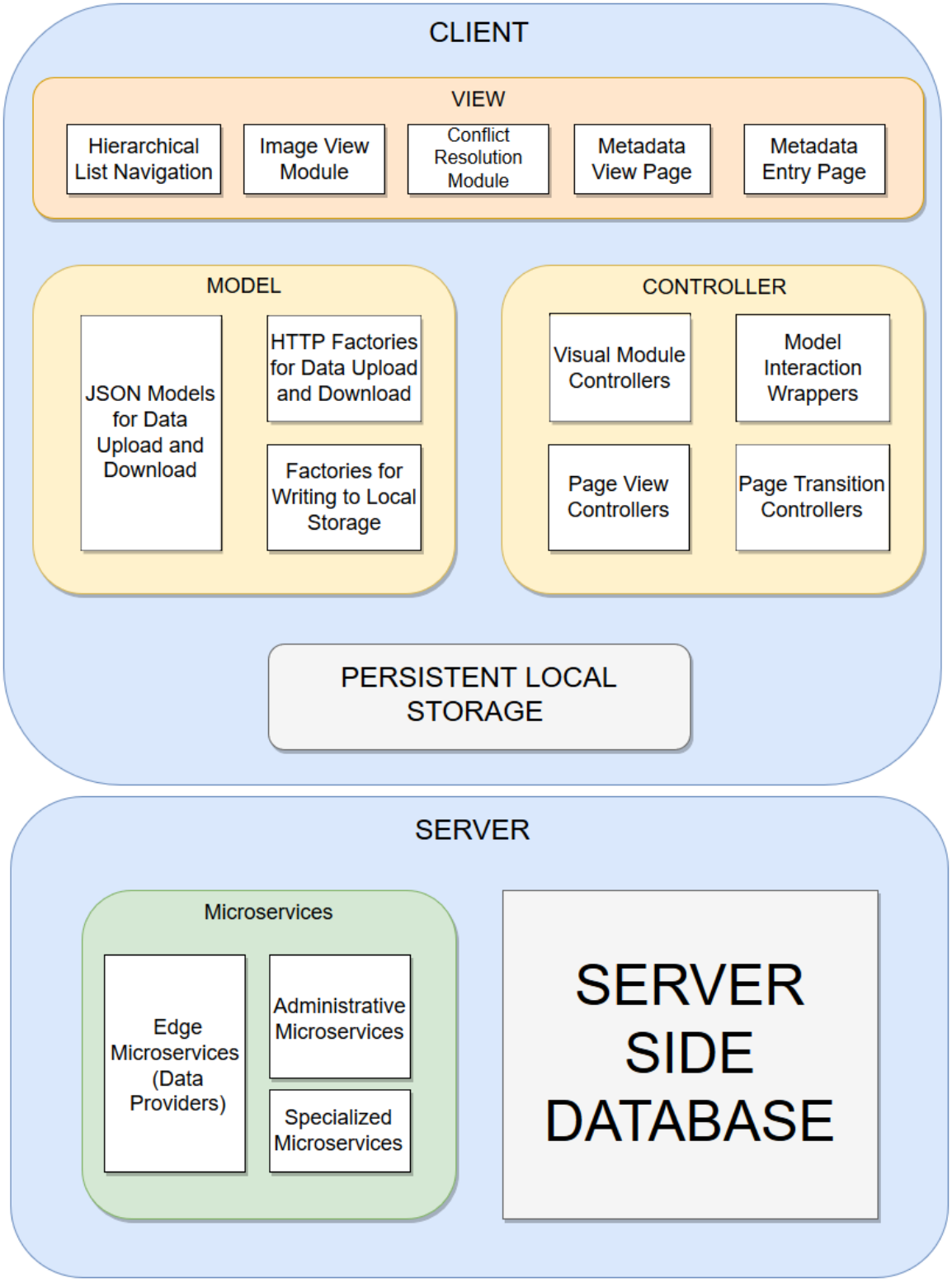

Figure 4.4: Component Diagram of the NRDC QA Application.

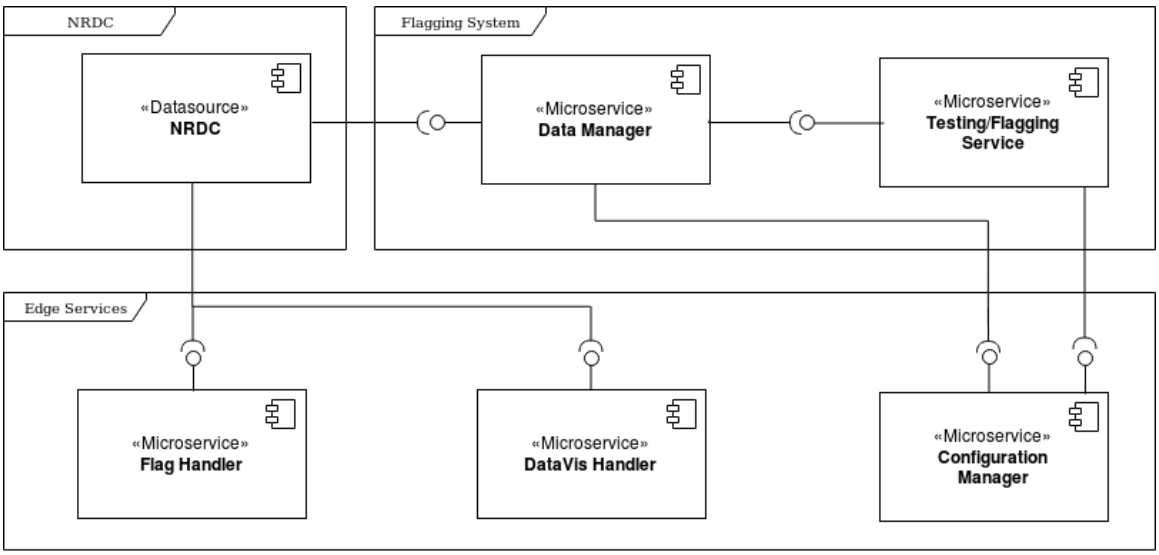

Figure 4.5: Component Diagram of the NRAQC System.

SQL Server 2012 and houses both the metadata gathered by Nexus technicians and the streamed sensor data from the research sites around Nevada. The NRDC's collection of metadata, shown in Figure 4.6, is separated into eight tables: People, Site Network, Site, System, Deployment, Component, Service Entry, and Document. The first six tables are considered infrastructure and are connected to one another through a one-to-many relationship. This creates a hierarchy of data access and is used to help narrow navigation through the large quantities of data. The last two tables, Document and Service Entry, exists as foreign key tables to most of the other infrastructure tables. They provide necessary information regarding each time a tower is serviced, as well as store documentation on any part of the research equipment.

Differing from the metadata tables, the sensor data is situated around a central Data Streams table, shown in Figure 4.7. The Data Streams table has a foreign key relationship with the Deployment infrastructure table. From the Data Streams table, four tables are referenced through foreign keys to determine units of measurement, categories of data, data types, and the properties of the data. Additionally, the Data Streams table shares a one-to-many relationship with the Measurements table. It is this table where all of the data from the research sites are stored. Finally, the

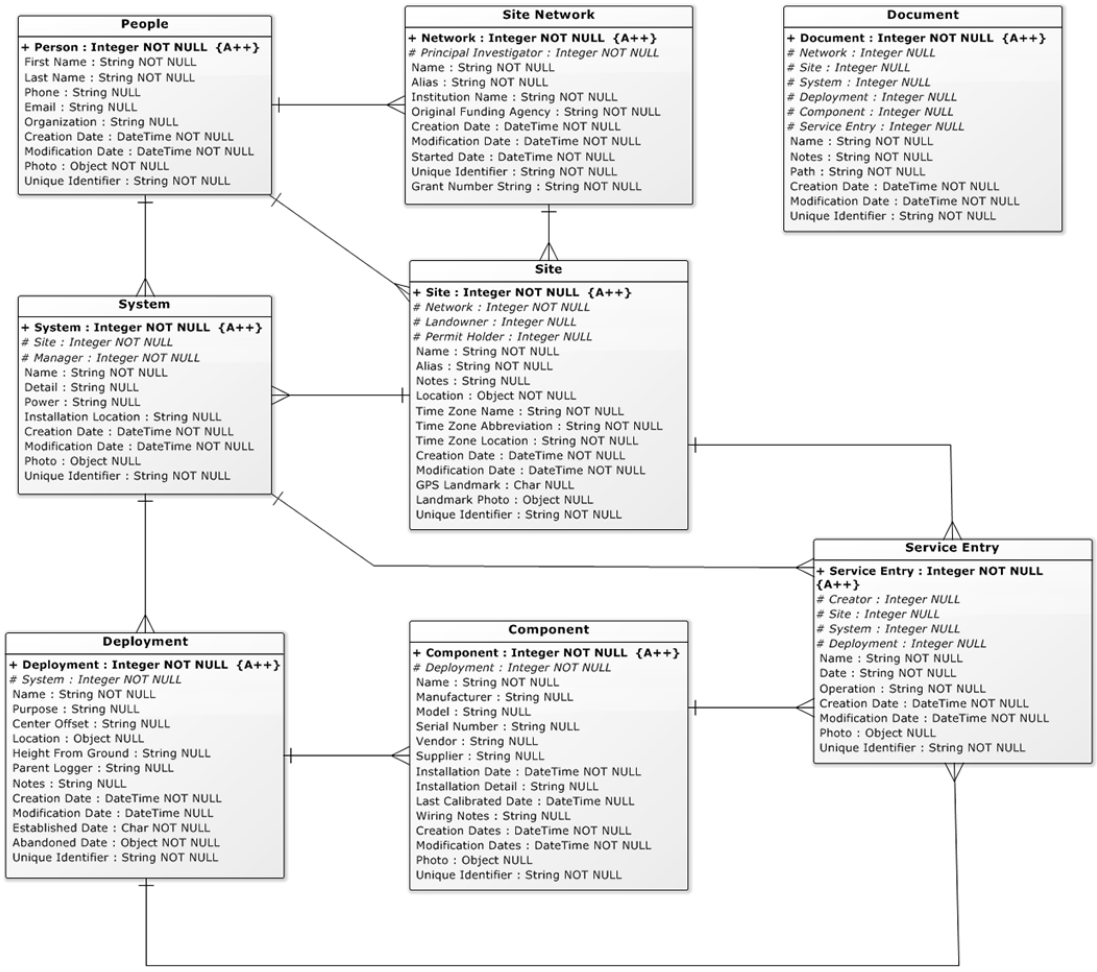

Figure 4.6: Entity-Relationship Diagram of the NRDC's Metadata Storage.

Measurements table also references two tables that hold information about quality control flags, but this section is optional and made entirely to support the NRAQC system.

Originally, the database schema did not account for the refined data querying and massive batches of data were the medium in which data was transferred. Through these tables, applications can now structure the querying of data to their client interface. Additionally, each of the tables are related to one another by using only one-to-many relationships. This choice allows each table to easier support the microservice that utilizes it. By using one-to-many relationships, the tables themselves

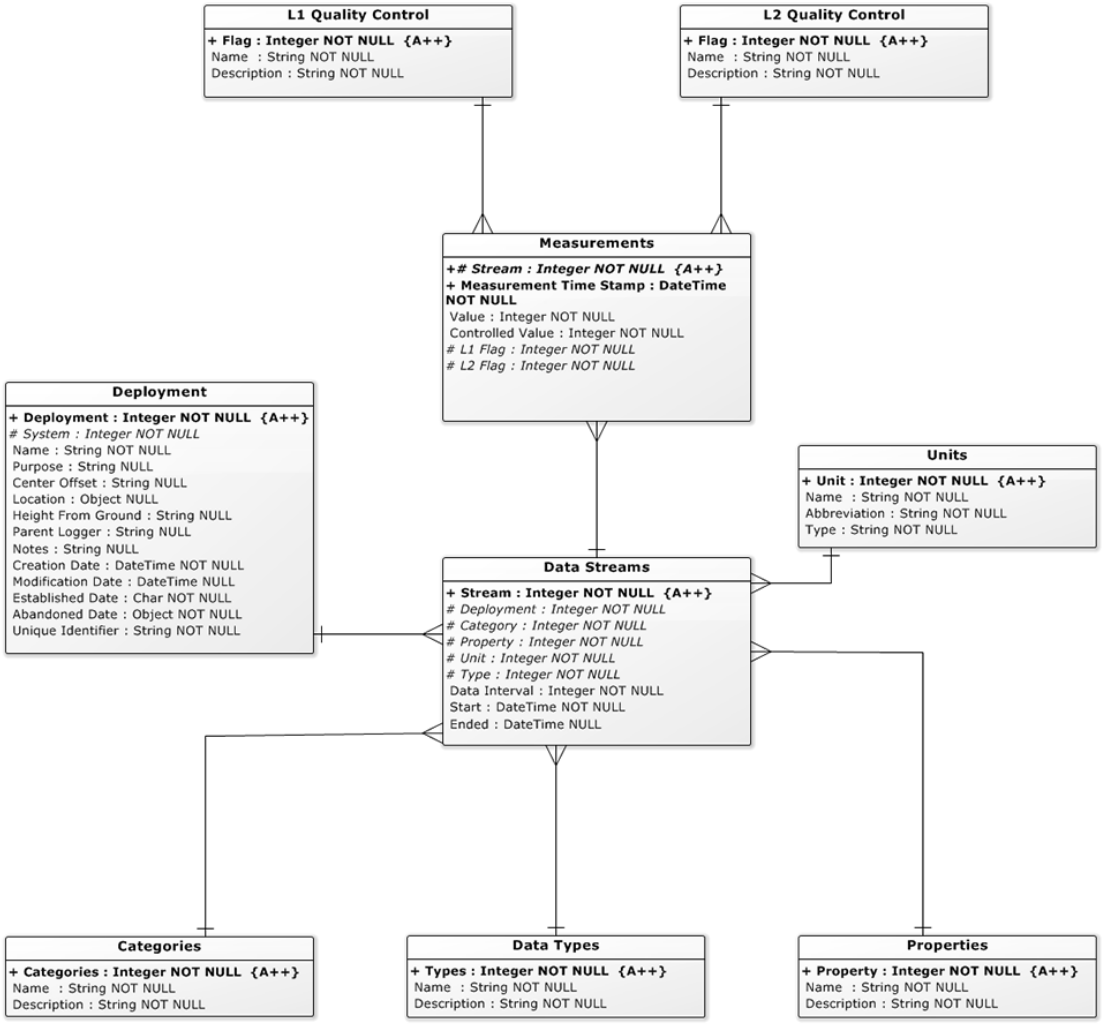

Figure 4.7: Entity-Relationship Diagram of the NRDC's Sensor Data Storage.

still contain foreign keys to reference other tables, and not require a join table to be referenced. This allows microservices to bring the referencing of tables down to a minimum.

Chapter 5

Implementation

In this chapter, a detailed walk-through of the MESA system is covered. As previously mentioned at the beginning of the thesis, the solution that MESA provides can be split into three major parts. The first part, covered in Section 5.1, is the current roster of Microservices developed as part of MESA. Sixteen microservices make up MESA, and each of the microservices are mapped to a specific application or development with some room for reuse. The second part, covered in Section 5.2, of MESA is the Service Discovery and Registry portion of the system that serves as the main management and testing hub. This section will cover the features provided by Consul and a brief overview of the customized tests. The third part, covered in Section 5.3, of MESA is the application support that is provided to other projects affiliated with the NEXUS Project. The applications mentioned are the NRDC Quality Assurance Application, the Near Real-time Autonomous Quality Control Application, and the SEPHAS Lysimeter Visualization. This section specifically will cover the microservices' place within the application's workflow. Wrapping up the chapter will be Section 5.4 describing software practices that were utilized but are not part of the core functionality of the system.

5.1 Microservices

The first major part of MESA is the collection of microservices that comprise the modular parts of the system. Each of the microservices presented in this section are only accessible via HTTP REST protocols, such as GET or POST. Additionally, the microservices present within MESA were developed with an agnostic view towards coding languages in mind. This means that the certain batches of microservices differ in terms of frameworks and actual programming languages. This was done in order to match effective tools and libraries with powerful languages without being tied down by a specific codebase.

There are a total of sixteen microservices that each represent a feature of an application or some form of development supported by MESA. The first eight microservices were developed to serve as curators of metadata gathered by Nexus technicians. These microservices not only provide the typical create, read, update, and delete (CRUD) functionalities, but further refines the data searching options of the NRDC by enforcing a hierarchical structure of accessing data. The ninth microservice manages images captured by Nexus technicians during routine maintenance. The tenth microservice formats and feeds data into web-based environmental data visualizations. The eleventh microservice handles conflicts from concurrent data entries. The last five microservices provide differing functionalities but center around an autonomous quality control application developed for sanitizing environmental sensor data. These microservices provide support to the various features offered by the quality control application, such as handling of erroneous data, managing incoming data, formatting data sources, configuring the output, and displaying the corrected values.

5.1.1 Metadata Infrastructure

The eight Metadata Infrastructure Microservices revolve around a specific Nexus application designed to alleviate the troubles faced by Nexus technicians during their routine maintenance of research equipment at the remote research sites. Each of the

microservices are mapped to the eight categories of metadata in which the technicians monitor for: People, Site Networks, Sites, Systems, Deployments, Components, Service Entries, and Documents. These metadata categories govern an aspect of the research site and it's maintenance, such as when it was created, how many sensors are on it, or when was the last time it was serviced. More on the metadata specifically is referenced further in the chapter as part of the detailing of the application. The microservices fit into this metadata management scenario by providing CRUD functionalities to each of the metadata categories.

By calling specific REST protocols (GET, POST, PUT, DELETE), applications can store and retrieve metadata from the NRDC infrastructure database. It should be noted here that the data stored inside the NRDC was converted upon entry to generic formats in order to ensure consistency. This means that each of the microservices autonomously converts all of the data, including date time and geographical coordinates, to standardized formats when used. Through a GET request, the microservices will retrieve an unaltered list of the entries or a singular entry within that metadata category. By issuing a POST request, the microservices will attempt to send a specified JSON object to the database and add the data inside the JSON as a new entry to the database. A similar action is conducted when issuing a PUT requests, but instead of adding a new entry, an existing one will be altered using the data from the JSON. Finally, the DELETE request identifies an existing entry and attempts to remove the entry from the database. Each response from the microservices are retrieved and displayed as an object containing both the status code and the appropriate text.

However, merely retrieving entire batches of data will easily flood the application with unnecessary clutter since metadata gathered regularly by multiple technicians tends to accumulate quickly. To combat this, each of the microservices has functionality that pertains to a hierarchy. This hierarchy is described as a tree-like structure based around one-to-many relationships of the metadata categories. An example of this type of relationship can be how Site Networks may contain many sites, while each

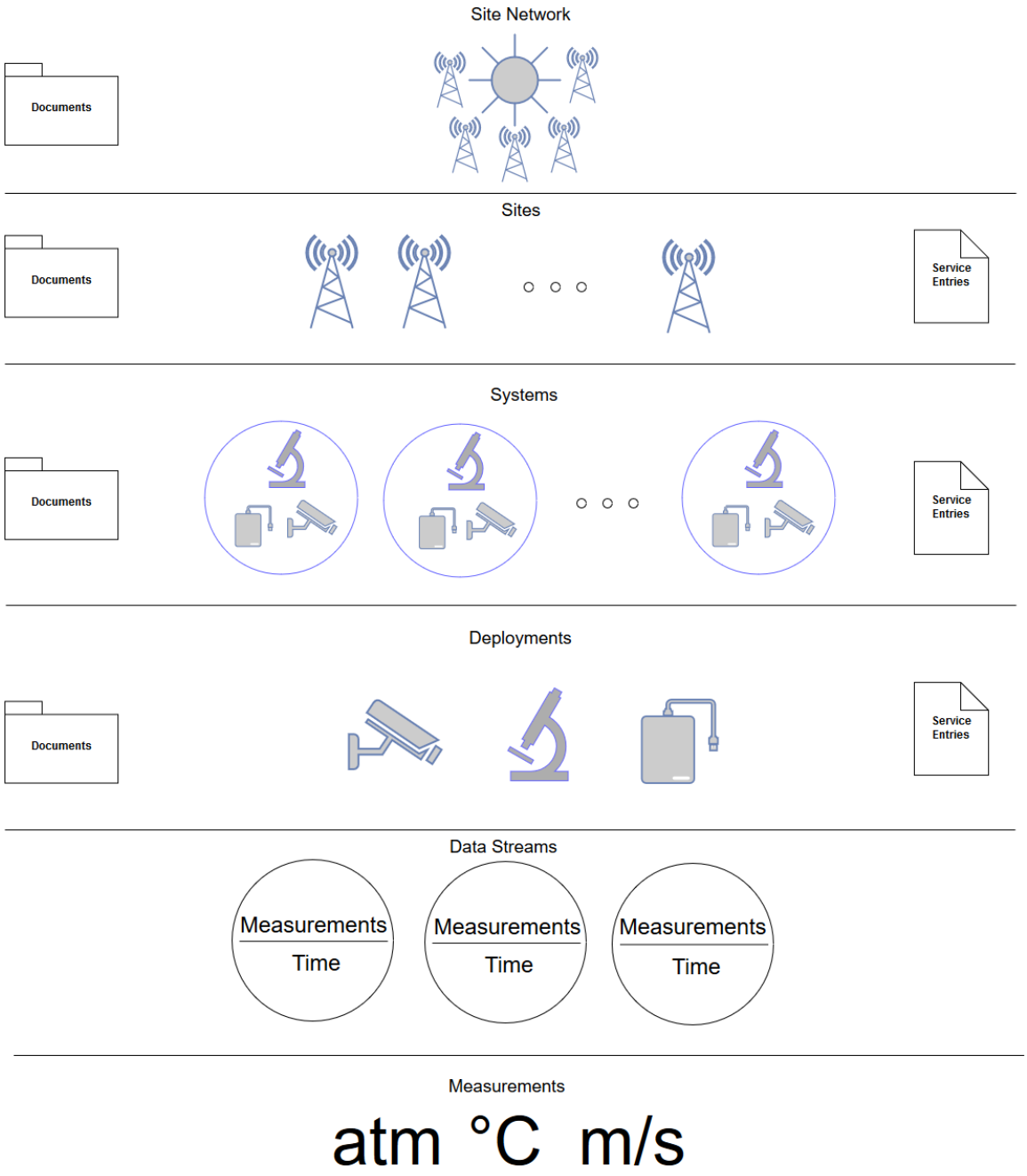

Figure 5.1: This figure showcases the hierarchy of both the metadata and the sensor data that exists in the NRDC.

site may contain many systems. This relationship can then be followed down further until reaching a specific component or even the sensor data, as seen in Figure 5.1. By narrowing searches down to a hierarchy, applications can have more fine-grained

control as to what data is displayed at a given time. Instead of having a list of a hundred components, the application can receive just the components that are associated with a specific deployment. To implement this on the microservices, each were given a functionality to peer into the foreign keys of their entries and display the associated IDs of the entries within the category that immediate follows them in the hierarchy. By alternating between this hierarchy function and the other CRUD functions, a hierarchical search pattern can be implemented.

5.1.2 Imagery

The Imagery microservice was constructed to handle the uploading, downloading, and searching of images associated with the metadata stored on the NRDC. The images in questions are not to be confused with the high definition image repository that the NRDC active populates every couple minutes from research sites. These images are instead those taken by Nexus technicians of physical components or installations during their maintenance of the research sites. Originally, the NRDC did not have the capability to support the spontaneous uploading of user-generated imagery. The Imagery microservice was developed to address that oversight and create an overall quality-of-life improvement for Nexus technicians.

To enable the Imagery microservice, an application needs to only send a JSON object containing a unique identifier to the two exposed URIs bound to the service. The first URI, Photosearch, invokes two different functionalities depending on the suffix appended to the end of the URI. The second URI activates without a suffix by merely receiving the JSON directly to the Preview URI. For Photosearch, should the "/Post" suffix be activated, the microservice will convert an active image into a UTF-8 byte string and store it on the database. However, when activating the "/Get" suffix, this would prompt the microservice to retrieve an image with an ID matching the one on the JSON after searching through each of the metadata tables. The located image would then be decoded and converted back to an arbitrary image format, such as PNG or JPG. The other URI, Preview, replicates the exact functionalities of Photosearch's

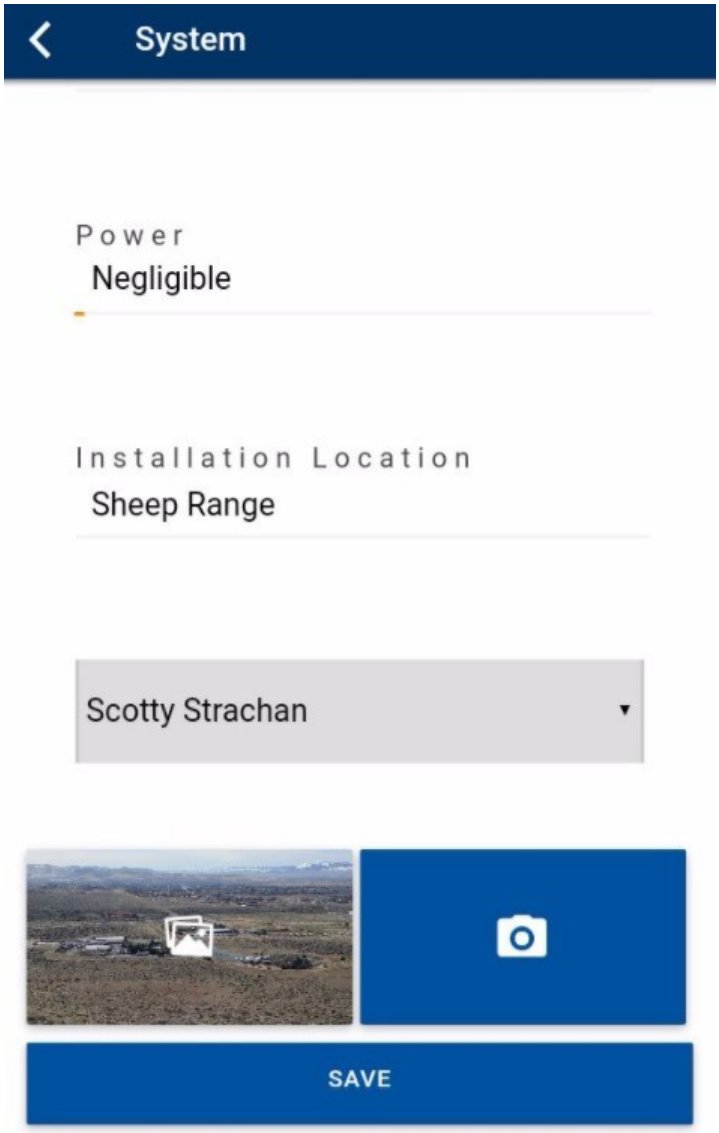

Figure 5.2: This figure showcases the Photosearch's preview functionality on the NRDC QA Application, shown in the bottom left.

"/Get" suffix, but instead, formats the converted image's height while preserving the ratio it shares with the width. The final cropped photo is then returned in the response JSON as a preview of the original image, shown in Figure 5.2. These two features are especially useful to mitigate the usage of memory inside an application, by first giving users a thumbnail then the option to download the full resolution photo.

5.1.3 Data Visualization

The Data Visualization microservice was constructed to aid in the visualization of environmental data gathered over several years at the SEPHAS Facility located in Las Vegas. The data provided by SEPHAS consists of three major measurements: solar radiation, air temperature, and liquid precipitation. The objective of this microservice was to separate the three measurements and format the data so that it can fed into three responsive graphs. When a user zooms into a region, the data must also be updated appropriately to match their current view.

To activate the Data Visualization microservice, the application must make a post request that consists of a JSON containing sample rates and a datetime range. This would prompt the microservice to draw from the data source and separate it into three data structures. This step is ignored after the first request call, as the data is temporarily loaded in memory. Then, the data is cloned to a final three data structures based off of the sample rate and the datetime range. By having an appropriate sample rate, the amount of data returned will be less and retain the appropriate shape for the visualization. If a datetime range is also specified, then the amount of data returned will potentially be less as well. Once fully configured, the data is packaged into a JSON and sent back as a response to the application. The frontend will then process this data and load it into the visualization, as shown in Figure 5.3. This entire process is repeated constantly and made faster as the user navigates through the data.

5.1.4 Conflict Management

The Conflict Management microservice was created to resolve conflicts that result from multiple users enacting change on a database with an application. Conflict Management was developed largely for the metadata application uploading multiple entries at one time. The main process operates in a similar manner as most version control software. When the submission of a data entry whose modification date is earlier than what is listed inside the database, a conflict is flagged. Much like

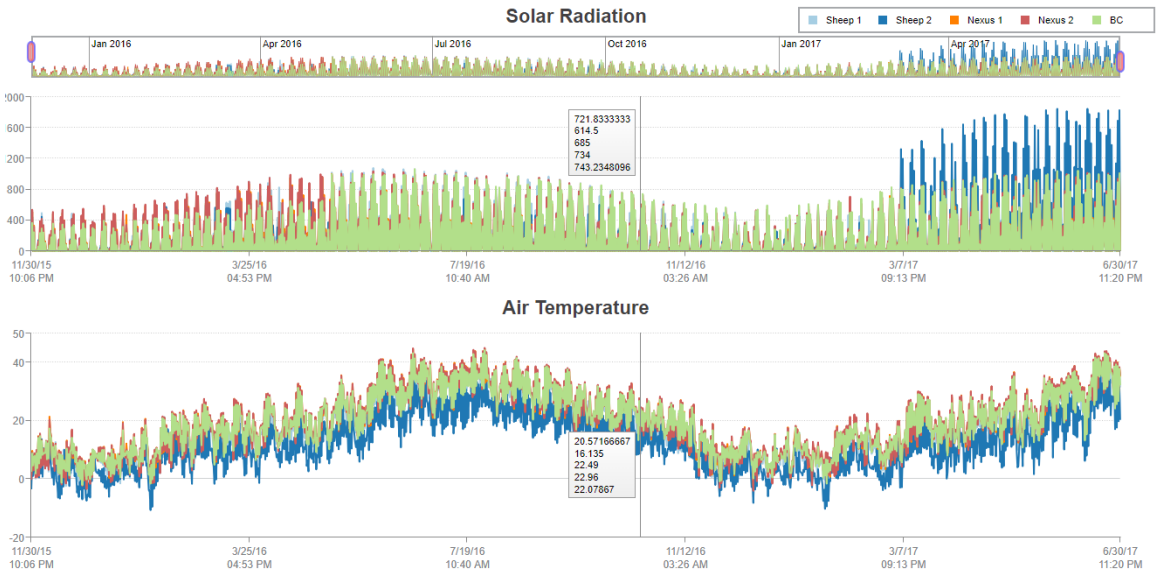

Figure 5.3: This figure showcases the data visualization functionality on the Lysimeter Data Display.

version control software, the user is given the option to continue with their flagged copy or merge their version with the current canon. Once a selection is chosen, the microservices locates the database table in which the data resides and overwrites the entry with the selection made. This is then repeated for each of the multiple entries being uploaded by the application during that one transaction.

Calling the Conflict Management microservice involves sending a POST request consisting of a list of entries to submit to the database. Also inside the JSON, descriptive information is given to locate the entry's associated database table. The microservice then goes through each of the submissions and compare the modification dates. Should a conflicting modification date be found, the microservice appends that entry to a flagged list. Meanwhile, the passing submissions are added to their respective database tables via the appropriate microservices. At this point, the microservice will return a response detailing specific information about the conflict, and a copy of both what was sent and what currently exists within the database. The response returned provides the necessary information for a front end to create a conflict resolution interface 5.4. Once a finalized choice has been made, a POST request to

another URI within the microservice allows for the overwriting or updating of what currently exists within that database entry.

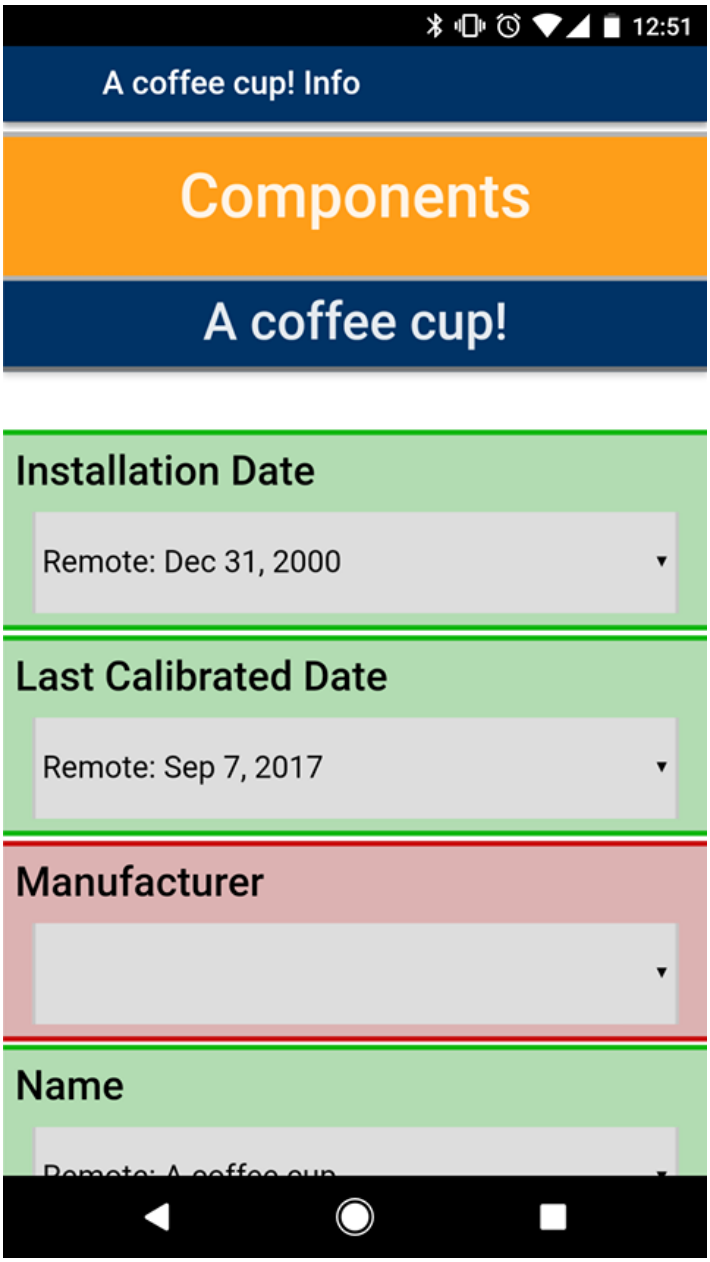

Figure 5.4: This figure showcases the conflict management functionality on the NRDC QA Application.

5.1.5 Quality Control

The Quality Control (QC) portion of MESA are comprised of five microservices that provides functionalities to the Near Real-time Autonomous Quality Control Application (NRAQC). NRAQC is a QC system that provides the NRDC with the means to transform raw sensor data into a quality data product [50]. The purpose of these microservices is to provide all of the necessary feature support that NRAQC requires, such as flagging incoming data, formatting data sources, visualizing data, and exporting data.

The first QC microservice is the Data Manager service. The Data Manager provides all functionality of retrieving, formatting, disseminating and writing measurement data, flag data and related metadata. This service provides a flexible interface to the data source that can be reconfigured to accommodate an API data source or a flat file system. It requires the implementation of two other interfaces: one which allows it to transfer data between itself and the Testing/Flagging Service component and another which allows it to receive data from the Configuration Manager component.

The second QC microservice is the Testing/Flagging service. The Testing/Flagging Service provides the core processing functionality of NRAQC by testing and flagging data. It runs autonomously and requires no direct interaction from the user at any time. It provides one interface allowing the Data Manager to send measurements to it for testing and retrieve flagged measurements from it for writing to the data source. It requires the implementation of one interface from the Configuration Manager to allow the upload and modification of tests from the client.

The third QC microservice is the Flag Handler service. The Flag Handler provides an interface between the client and server. Its primary role is to handle the client's JavaScript code which queries for flagged data. It will filter, format and bundle flags so that they may be quickly rendered upon being downloaded by the client. Formally, it implements an interface to connect to the data source.

The fourth QC microservice is the DataVis Handler service. The DataVis Handler implements an interface to connect with the data source component and retrieve desired information. This microservice handles all requests from the data visualization component of the UI and performs necessary formatting and organization to ensure that the Web page can focus on rendering data points instead of wasting power processing them. This service also provides functionality to classify flagged data points as distinct from normal data points so they will be rendered differently by the data visualization component.

The fifth QC microservice is the Configuration Manager service. The Configuration Manager interfaces with the web client – and user – by receiving user configuration requests and sending down stored user configuration info. Additionally, the Configuration Manager implements interfaces to distribute configuration objects to the Data Manager component and the Testing/Flagging Service. It can also notify the two services of a change to the configuration and request a temporary cease of the automated flagging cycle while re-configuration occurs.

5.2 Consul Service Discovery

The Consul Service Discovery is the second major part of the MESA system and is actually another independent service much like the microservices. In fact, it is often called the service discovery service by industry developers. The purpose of Consul in regards to MESA is be the service registry and discovery for each of the developed microservices. Consul's primary job is to register the independent microservices based on the IP and port in which they were deployed on. It's secondary job is to then regularly monitor the state of the microservice, determining if the service is functioning or not. This is done with autonomous checks that are configured during a microservice's first register. Another role that Consul plays in MESA is that of an autonomous testing platform. Through the configuration options of Consul, a developer can have a customized testing script, written in any language, run at set intervals and record the results appropriately.

Currently, the MESA system has an instance of the Consul Service Discovery active on the SENSOR domain where it monitors the microservices deployed. The instance used by MESA accepts two forms of communication with it, one through its HTTP API and one through it's generated web interface. The web interface is useful to show statistics of activity and status, but is weaker compared to the HTTP API. The HTTP API however, gives developers access to all Consul commands, such as retrieving connection information about a specific microservice, viewing all active microservices, or even the health history of a microservice. To view statuses and health information, this would only require a GET request to the correct URI. To register a service with Consul, a developer would only need to send a PUT request alongside a detailed JSON object to the appropriate URI endpoint. The detailed JSON would require information such as location, port, name, description. It is here where the autonomous options can be configured.

Consul allows for the loading of specialized scripts written in virtually any scripting language to be loaded into the configuration settings. Once a file is designated, the developer can then specify the execution details of the script and the re-occurring times it is executed, by altering the configuration JSON. As of now, each of the microservices are configured with customized testing scripts written in Python that routinely test all of the features. The scripts developed for the microservices are all unique to the individual service, but essentially involves sending a sequence of requests while monitoring the response codes. These results are then compiled on Consul and made viewable by the HTTP API or the web interface.

5.3 Application Support

The third and last major part of the MESA system is the Application Support that it provides to affiliated Nexus applications. This section will show off three projects developed with the microservices provided by MESA. The first project is the NRDC Quality Assurance Application, which was commonly referred to in this chapter as the metadata application. The second project is the SEPHAS Lysimeter Visual-

ization, which was a data visualization of compiled data provided by the SEPHAS Facility in Las Vegas. The third project is the Near Real-time Autonomous Quality Control Application, or NRAQC. This section will not cover the intricacies of each project. Instead, it will provide a brief overview while describing the role which the microservices play.

5.3.1 NRDC Quality Assurance Application

The NRDC Quality Assurance (QA) Application, or QA App for short, is an application developed by the Cyberinfrastructure component of the Nexus project to handle metadata [39]. Previously in this chapter, there were mentions of Nexus technicians going out to research sites for maintenance, installations, and other forms of configurations. Albeit accurate, a significant issue was not brought up, which is the fact that there is no form of internet access in the areas that house the research sites. Technicians would have to manually write down entries on a notebook and then transcribe those notes into a database sometime after. The QA App was developed for the express reason of alleviating the troubles faced by Nexus technicians. The application narrows down metadata on research sites specific to the user and allows them to alter entries or add new ones right at the tower. A visual representation is shown in Figure 5.5. Since there is limited internet access however, this application stores the changes locally and syncs them to the database when appropriate internet connection is made available.

The microservices play a vital role as server backend for the QA application. The QA application upon initial activation calls upon each of the eight microservices to store a local copy of the relevant data entries within the metadata database. It is here where the microservices converts data from the NRDC and presents it to the QA application. When changes are made in the application and the sync button is pressed, the application then uses the eight microservices alongside the Conflict Management microservice to verify and submit the changes to the database. Should the Conflict Management microservice return a merge issue, the response from the

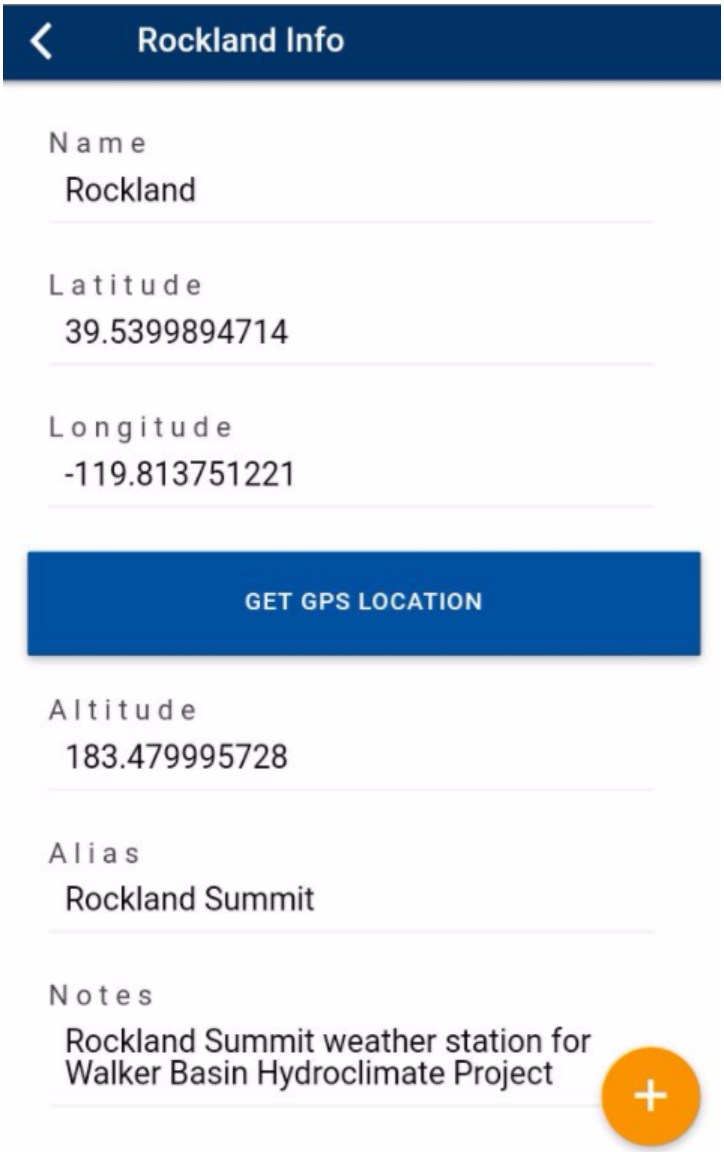

Figure 5.5: This figure showcases the NRDC QA Application navigating through a site entry.

microservice is parsed and then used to generate a merging interface. Additionally, the Imagery microservice is called when an entry features an image and handles the storage and retrieval of that image into the database. The Imagery microservice is also called when the a entry is viewed by the user, where it retrieves a preview image instead of the original.

5.3.2 SEPHAS Lysimeter Visualization

The SEPHAS Lysimeter Visualization is a data visualization developed to better visualize the environmental data gathered over the span of several years by the SEPHAS facility in Las Vegas [27]. The main page is shown in Figure 5.6. The Lysimeter Visualization was given a strict set of requirements that were determined by the SEPHAS team. The first requirement was that the Lysimeter Visualization had to parse a flat file and this process had to be generic so that these files can be interchanged by a non-technical staff member. The second requirement was that the data must be separated into three categories, one for each instrument, and fed into three web-based visualizations. The third requirement was that the visualizations were interactive and when affecting one, the other visualizations will share that same change. This is done so that researchers may view the environmental effects concurrently.

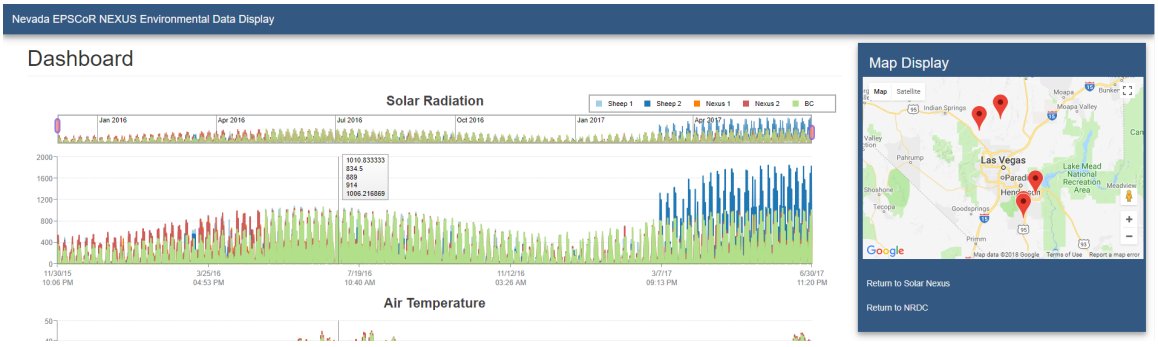

Figure 5.6: The SEPHAS Lysimeter Data Visualization webpage.

The microservice played a non-vital but very significant role in the visualization of the lysimeter data. By using powerful frontend visualization libraries such as D3.js, the data file could merely be loaded and visualized [2]. However, this would cause an almost unbearable lag times between actions issued by the user on the visualization. Even zooming further into the graph would cause several minutes of delay. Although the visualization can operate without the need for a microservice, the usage of a microservice in this case was able to cut down virtually all of the lag time between the user actions and the visualization library. The Data Visualization microservice was utilized to handle all of the data loading and transformation operation, so the

web client only needed query the microservice for all of its needs. Once the client contacted the microservice, the service would then return limited amounts of data to only preserve the shape of the visualization. However, once the user explored further into visualization, the microservice would then alter the range and the amount of data presented to match what the user viewed. This allowed the client to levy all of its intensive actions onto the server and provide an accurate, responsive, and swift visualization.

5.3.3 Near Real-time Autonomous Quality Control

Occasionally, sensor readings received by the NRDC from the remote research sites show signs of erroneous data. This can be missing values, values outside possible bounds, or even repeats of past values. To address these problems, the Near Real-time Autonomous Quality Control (NRAQC) System was developed for the NRDC [50]. NRAQC tests incoming data points logged autonomously at a research site to see if they meet the criteria of an invalid measurement. The system, with aid of user specified configurations, flags all invalid measurements with metadata that specifies the nature of the invalidity. The service provided by NRAQC is necessary for the production and distribution of a quality data product. A visual of NRAQC's main menu and data visualization page are shown in Figures 5.8 and 5.7.

Much like the QA Application, the microservices play the vital role of server backend for the NRAQC system. NRAQC utilizes an intuitive web interface as the main client, but splits its main features into microservices that support it. These features includes the handling of differing data sources, enable autonomous flagging of measurements, interfacing with the client, enabling a data visualization, and formatting the results based on the user specifications. While the microservices deal with the computationally and memory intensive portions of NRAQC, it does not govern the entire system itself. The NRAQC client presents a number of features to the user and when the user selects a task, the NRAQC client then communicates to the microservices via HTTP.

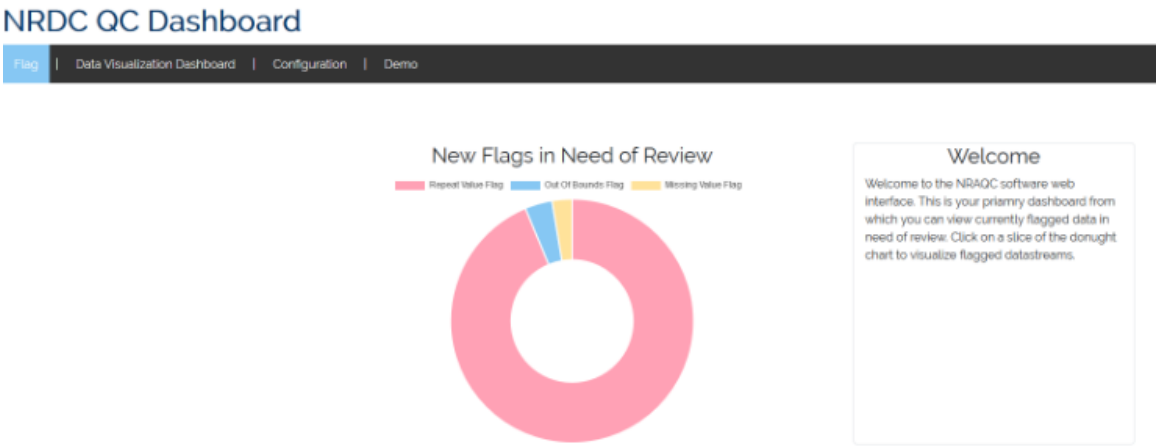

Figure 5.7: The main menu of the NRAQC system.

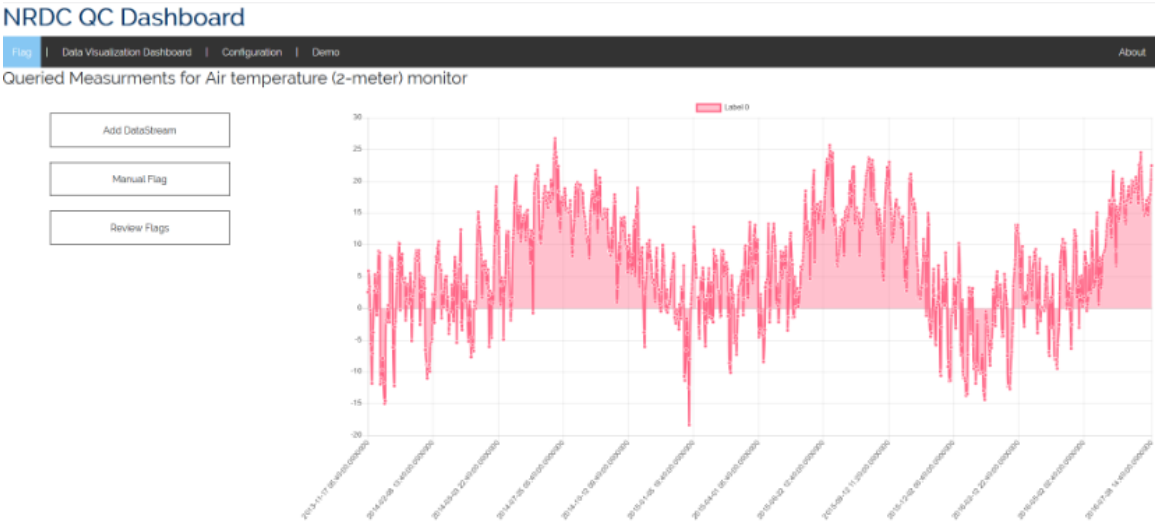

Figure 5.8: The main visualization component of the NRAQC system.

5.4 Containerization & Continuous Integration

Although originally designed as prototype, MESA does utilize two major industry practices: containerization and continuous integration. Containerization is creating

a virtual environment that ensures the software running on it operates as intended. Continuous Integration is tied to the idea of software development teams and constructing code among multiple developers. Essentially, continuous integration is the practice of merging all developer code copies into one mainline multiple times a day.

To execute the containerization of microservices, the Docker Containerization software was used. A diagram showing Docker's generic use is shown in Figure 5.9. Docker is the premier brand of containerization software that is used in industry software today, with famous consumers including General Electric, Lyft, and BBC News. Docker primarily operates on Linux environments, but their Windows equivalents do exist although utilized less. Currently, the microservices deployed in the Linux environment are containerized with Docker, but aside from the basic setup, no additional

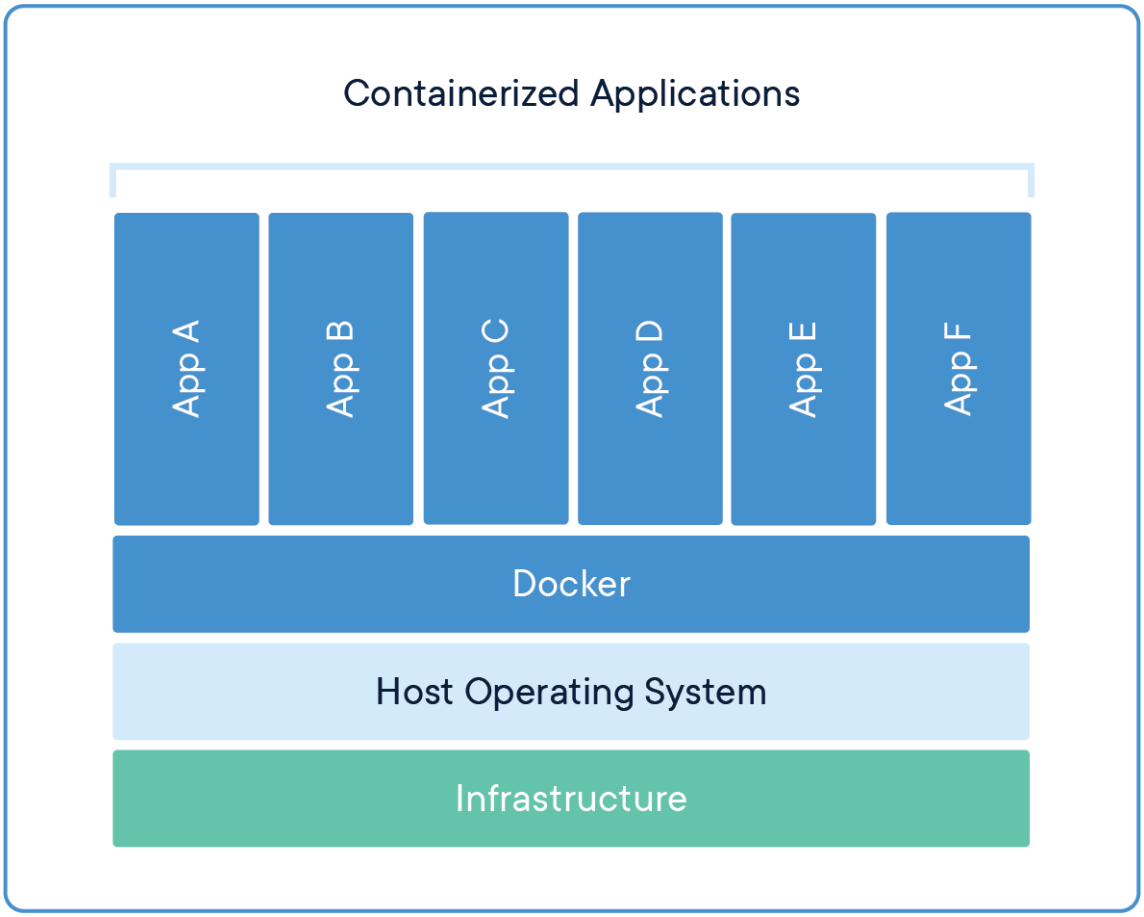

Figure 5.9: This figure shows off the generic use of Docker [22].

development was made. The containerization of the microservice, although very significant, were not made a core functional requirement for this iteration of the MESA system.

In regards to the continuous integration, the Travis C.I. software was the third-party software used to enable the merging of a mainline [15]. Travis C.I. links to a Github repository and must be configured to match the compilation of the linked software [24]. An example of this integration is show in Figure 5.10. Once properly configured, the software will activate on each successful commit to the code repository. In this case, Travis C.I, has been configured to test python code for the microservices supporting the NRAQC system. The addition of continuous integration to MESA was made to show that continuous integration is possible, but was not fully implemented due to this portion not being a core functional requirement.

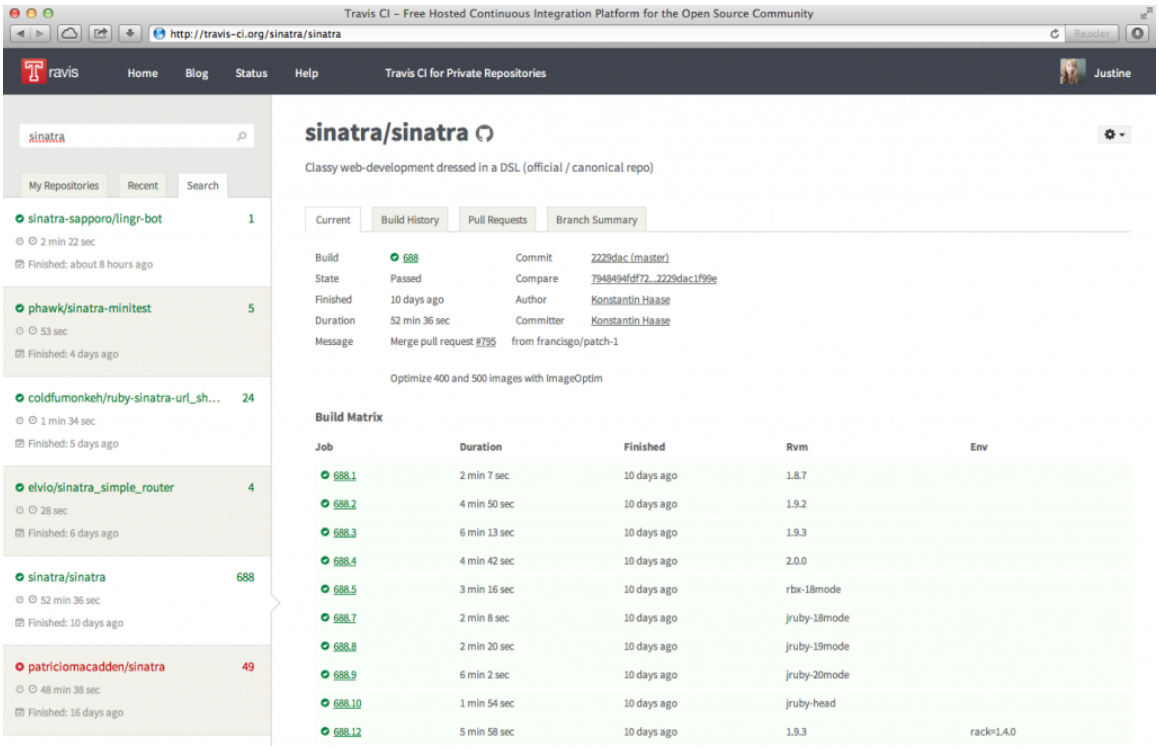

Figure 5.10: This figure shows off the integration of Github with testing frameworks through TravisCI [15].

Chapter 6

Discussion and Validation

So far in this thesis, a background of the organizations behind MESA has been discussed, as well as similar works inside and outside the field of environmental research. On top of this, the details behind the software design has been covered, and the implementation has also been showcased. The next step is to understand the novelty in which this solution brings to the world of Environmental Research and Microservice Architectures. In this chapter, three systems are discussed in Sections 6.1, 6.2, and 6.3 from the six covered in Chapter 3. The selected three are given a more in-depth look at the technical set up of each system. After which, a discussion is presented in Section 6.4 to review the contributions brought by MESA. To wrap this chapter up, a feature-based comparison table is made using the technical traits of the three systems covered and MESA itself.

6.1 Generic Service Infrastructure for PEIS

The new microservice architecture for PEIS, described in Chapter 3, is a very modern approach at developing a distributed system and supports the latest technologies. The project in which PEIS originates from involves funding on a national level and would be considered a very large environmental project. PEIS itself covers an entire portion of Germany so this system is vastly larger and more well maintained comparably to MESA whom draws funding from NSF resources. Starting from the bottom, the PEIS system be broken down into five major parts, a diagram of this structure can be seen

in Figure 6.1. These parts are: Hardware, OS & Cluster Management, Big Data & Base-level Software Infrastructure, Services, and Applications.

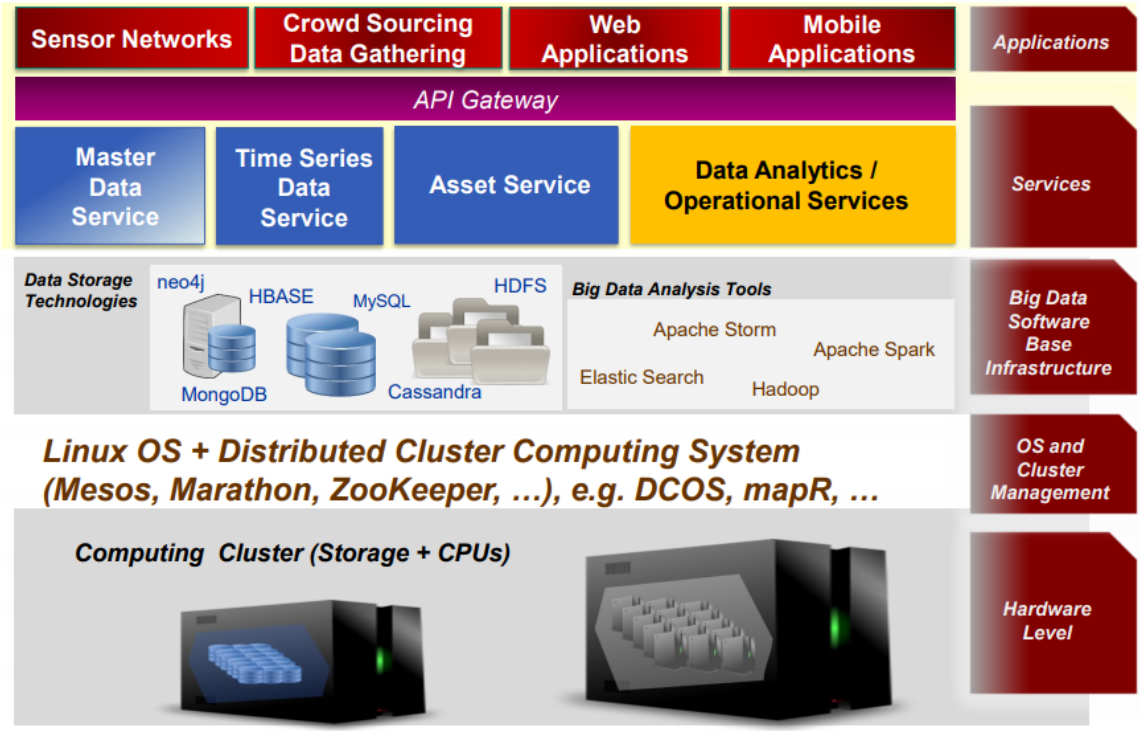

Figure 6.1: The new microservice architecture for PEIS [4].

For the first part, Hardware, PEIS utilizes it's own computing cluster, which in this case is a network of tightly connected CPUs and Storages. This is particularly interesting, because a computing cluster is a very expensive investment that ranges in the thousands of dollars to own and maintain. For the second part, OS & Cluster Management, PEIS utilizes Linux-based operating systems and uses a distributed cluster computing system. These cluster computing systems can be popular third-party solutions, such as Mesos, Marathon, or Zookeeper. For third part, Big Data & Base-level Software Infrastructure, PEIS allows for the usage of virtually any database, be it relation or non-relational, and some examples of supported databases can include MongoDB, HBASE, MySQL, or Cassandra. Additionally, the third part also mentions the usage of third-party big data analysis tools, such as Apache Spark, Apache Storm, Elastic Search, or Hadoop. For the fourth part, Services, PEIS uses

three major services: a Master Data Service, a Time-series Data Service, and an Asset Service. Any additional services developed are encompassed under a category called Data Analytics & Operational Services. Finally, for the last part, Applications, PEIS lends support to Mobile Applications, Web Applications, Crowd Sourcing Data Gathering, and Sensor Networks. Two examples of the applications that PEIS supports are the "Umweltnavigator Bayern" web application that dissemninates environmental data and the "Building Energy Dashboard" data visualization that displays measurement readings from components within a building.

In addition to the five parts, there are also features of the PEIS system that exist outside of the provided diagram. PEIS supports service discovery and registry, continuous integration, continuous deployment, containerization, and diagnostic tools. For a system managing data from such a massive project, it would only be natural to borrow the same practices that large industries use for managing microservices.

6.2 OceanTEA: Exploring Ocean-Derived Climate Data

OceanTEA utilizes a microservice architecture to support a web application that disseminates the oceanographic measurement data gathered by the University of Kiel. This data is collected from multiple remote sensors placed on over three thousand specially designed floats. The purpose of the OceanTEA systems is to create an accessible interface where users can navigate high-dimensional data in real-time. The microservice architecture the makes up OceanTEA involves only five microservices and one support service handling the user authentication. There is no form of Service Discovery or additional industry software practices, such as continuous integration and delivery. A diagram depicting OceanTEA's architecture is shown in Figure 6.2.

The five microservices that make up the OceanTEA system are as follows: a Time-series Conversion microservice, a Univariate Time-series Management microservice, a Multivariate Time-series Management microservice, a Spatial Analysis microservice, and a Time-series Pattern Discovery microservice. The Time-series Con-

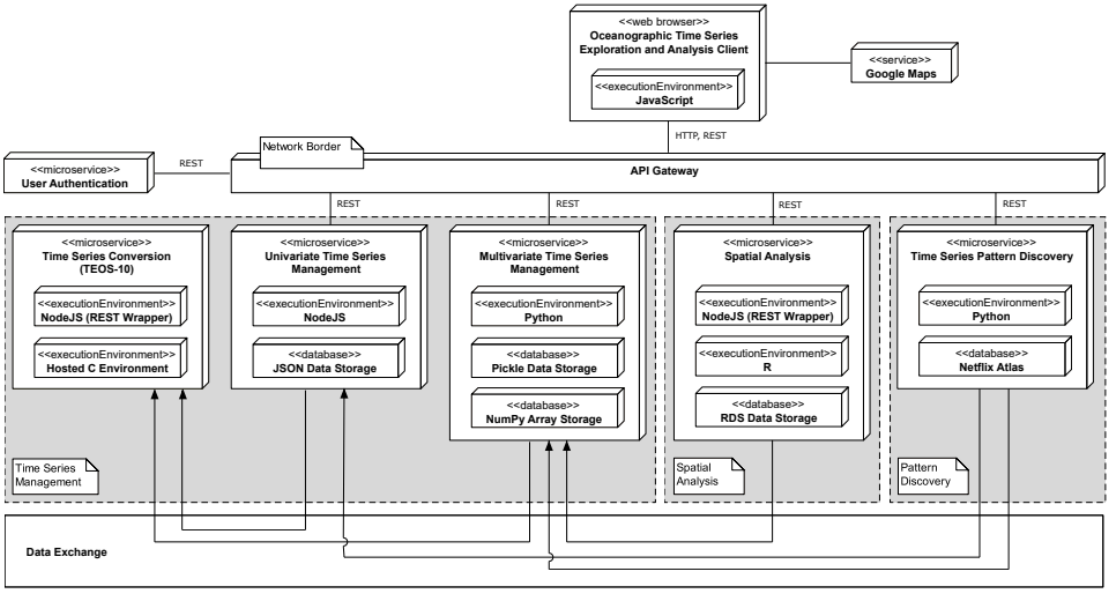

Figure 6.2: The microservice architecture for OceanTEA [28].

version microservice is the main form of handling time-series manipulation by the two management microservices. This service was written in Javascript with Node.js as its primary framework. The Univariate Time-series Management microservice is the service which handles scalar observations recorded sequentially over equal time increments. This service is also written in Javascript with Node.js as the primary framework. The Multivariate Time-series Management microservice is very similar to the Univariate one in that it also handles observations made over time, but the difference is that Multivariate can handle multiple observations at a given time. This microservice was written in Python with the Pickle and NumPY Array module for data storage. The Spatial Analysis microservice is OceanTEA's primary mode of conducting numerical operations on the spatial data gathered. This microservice was written in Javascript with Node.js, but executes code in the R programming language to do numerical analysis. The last microservice, Time-series Pattern Discovery, uses the Python programming language and the Netflix Atlas module to perform automatic pattern recognition on time-series data.

6.3 DIMMER Smart City Platform

The DIMMER Smart City Platform uses a microservice architecture to provide support to various IOT devices within a city. DIMMER is not a research project based around environmental science like the other two, but was chosen regardless because of DIMMER's multiple application support. Furthermore, it is an example of microservice usage outside the Earth sciences. DIMMER is designed to be an approach at large-scale IOT city management, where it increases the quality of the services offered to citizens, while reducing the costs of public administration. The DIMMER system's microservices are separated into two main categories: Middleware Services and Smart City Services. A diagram of the DIMMER architecture is shown in Figure 6.3 alongside associated components.

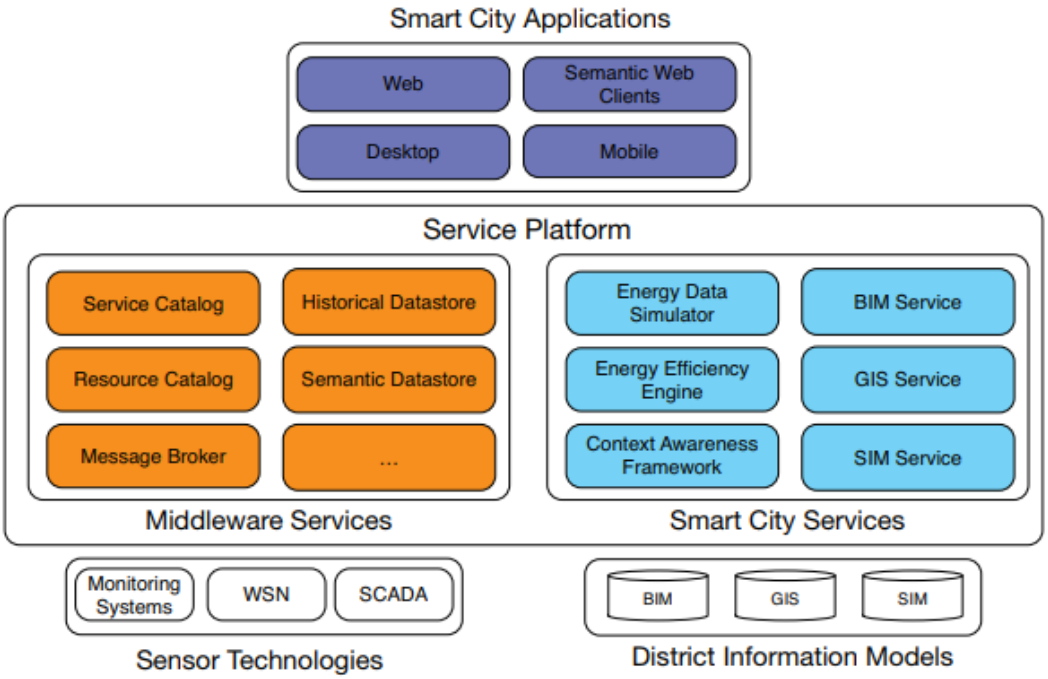

Figure 6.3: The microservice architecture for DIMMER [31].

Starting with the Middleware Services, this grouping of microservices are composed of five services that govern data abstraction and service/device discovery. The

first service in this group is the Service Catalog which discovers and registers all microservices present within the system. This is essentially the service discovery under a different name. The next service is the Resource Catalog, which is a form of device discovery that discovers and registers all physical IOT devices connected to the system. The third service is Message Broker service, and this microservice handles the publishing and subscribing of the sensor data. The fourth service is the Historical Datastore service that is used to query for or store time-series sensor data. The last service is the Semantic Datastore service that uses semantic web technologies to provide higher level abstractions of the connected devices in relation to other entities.

The next and last grouping of microservices are the Smart City Services, which are built off of the Middleware services and provide the core functionality of the platform. There are six microservices and each are mapped to an IOT device present within the city. Three of the services are centered around Geographical Information Systems (GIS), Building Information Models (BIM), and System Information Models (SIM). The GIS, BIM, and SIM microservices provide the appropriate models when invoked by the application. The fourth microservice is the Energy Data Simulator, which generates a simulation for the gathered sensor data when called upon by other microservices. The fifth microservice is the Energy Efficiency Engine, and this service provides optimization solutions for energy consumption. The sixth and final microservice is the context awareness framework which handles the modeling for real-world situations by defining a set of context properties.

6.4 Novelty Discussion

Throughout this chapter, it is abundantly clear that the research of microservices inside and outside of the earth sciences has advanced significantly. However, these approaches bring along with them certain noticeable trends. In the environmental field, microservice architecture is often used to bring about solutions in the form of singular applications. Multiple microservices are usually developed and used to create web-based support for a singular application, such as the ones described in

OceanTEA. For research outside the earth sciences, the microservice architecture is often used as platform for providing an abstract data layer between an application and the data source of the project, as shown with the DIMMER Smart City Platform. Interestingly enough, both research inside and outside the Earth sciences share a common trend of refactoring systems into microservice architectures when the problem involves an existing monolith. This approach usually involves the decomposition of a monolith into requirements that are mapped to microservices in the hopes of enabling scalability. This conversion can especially be seen in the PEIS system described above. To sum up the trends shown by both research inside and outside the Earth sciences, microservices are usually used to either provide application support or shift an aging monolith to a distributed system.

While both of the described trends showcase two valid and intended use cases of the Microservice Architecture, the complete refactoring of a system from monolithic to microservice-based brings about serious concerns. In environmental research projects, especially projects ranging from a single university to an entire state, monolithic system designs are common due to unexpected growth in a project or from a sheer lack in pooled technical knowledge. Many of these projects simply do not have the resources or people to simply halt progress and perform an entire system overhaul. Additionally, many environmental projects often autonomously collect data from sensor networks and shutting down these systems, even briefly, can cause detrimental effects on the research produced by the overall project.

It is through these concerns that brings to light the novelty of MESA as a microservice-based system. MESA is designed as a scalable application development platform to support critical systems, especially monoliths, without having the need to tear down the existing system. MESA connects to a regularly updated replication of the main NRDC database that houses copies of the incoming data, so it does not interfere with the monolith whom autonomously gathers data at set intervals. To clarify, this approach does not eliminate the option of full system migration, and provides the means to lessen the burden of the demands placed on developers until

a solution is decided upon and implemented. Should a full migration be decided, MESA can be utilized to shoulder the burdens of inactive systems and can eventually become decommissioned once the migration is complete. To sum up, the MESA system provides an alternative approach to providing scalable options to monolithic systems without having to tear down the monolith itself. This solution can serve a full system solution or simply as a temporary until a development team rebuilds the monolith from the ground up.

In addition to the novelty as a microservice-based system, MESA also brings a unique contribution to the field of environmental research. Usually, the option of supporting multiple applications with a service-based platform is common for industry, but uncommon for environmental research projects with modest funding. This is mainly due to the lack of manpower and technical skill within a modest sized environmental project to fully rebuild a system to support a suite of microservices. However, MESA does not require any retooling other than a connection to a database. In this regard, MESA provides a less technically-intensive option for environmental scientists to customize and develop multiple tools, instead of providing support to a singular application. As a platform, MESA seeks to address the infrastructure needs of an entire project and the potentially multiple scientific teams within it. In short, the novelty of MESA to the environmental research field specifically, is that the MESA support system bridges the gap between modern development practices and the cyberinfrastructure needs of environmental researchers without enforcing any of overhead.

To properly know where the MESA system stands with fellow microservice-based systems, a feature comparison table is provided in Figure 6.1. Through this comparison, it is shown that MESA brings forward a unique contribution to Environmental Research in the form of it not requiring a complete system refactoring in order to be used. Additionally, the MESA system carries with it many of the modern features and industry practices that are present in microservice development. It is through these features that MESA is able to offer more functionality than two similar sys-

Feature Description	**MESA**	**DIMMER**	**OceanTEA**	**PEIS**
Requires refactoring entire system		x	x	x
Support multiple applications	x	x		x
Oriented toward environmental research	x		x	x
Technology-agnostic	x	x	x	x
Service Discovery features	x	x		x
Supports multiple databases	x	x		x
Uses Containerization	x		x	x
Uses Continuous Integration	x			x
Capable of High Performance Computing solutions		x		x
Machine Learning Capabilities				x

Table 6.1: Feature-based Comparison Table

tems: OceanTEA and DIMMER. OceanTEA, although a very effective system in environmental data gathering, does not offer platform support, a service discovery, or the continuous integration features that MESA does. Similarly so with DIMMER, it does not provide as much features as MESA, lacking in areas like supporting multiple databases, containerization, and continuous integration features. Despite performing comparably to other environmental systems that also use microservices, MESA still is outperformed by microservice systems used by the larger environmental projects, like PEIS who is able to support advance features such as HPC or Machine Learning. Overall, MESA's abundance of features makes it a solid and modern alternative to a development platform for small to medium scale environmental projects.

Chapter 7

Conclusions and Future Work

In this final chapter, a concluding piece wrapping up the thesis is presented in Section 7.1. This section briefly goes over what was showcased during the extent of this thesis. Finally, Section 7.2 of this chapter is the future work. This section covers samples of additional work and approaches as potential extensions for the MESA system further down the line.

7.1 Conclusions

The system described in this thesis, the Microservice-based Envirosensing Support Architecture, focuses on creating an alternative approach to cyberinfrastructure for environmental research projects without enforcing a costly system refactoring. The central idea of this system is to create an applications platform centralized around a regularly replicated data source without having to tear down a monolith. The option to completely refactor a system and break apart an active monolith is expensive and requires a massive amount of technical knowledge and time to execute. This application platform was created by using a series of microservices that break up the business requirements into individual and independent web services. The microservices answer to a central service discovery and are generally mapped to a feature within a client application. The whole system was developed in Python and C# and revolves around a MSSQL database. The features provided by MESA encompass a wide range of functionalities, from merely fetching data from a database to automating tests to sanitize

data points within a data stream.

MESA is a very relevant and beneficial system to environmental research projects due to it's ability to provide platform support to a field that is often limited by the technical aspects of software development. While the idea of switching to a distributed architecture, like microservices, can be attractive to growing environmental projects, the reality of the matter comes down to whether the project has the time to halt progress while development is made and if there are adequate resources available to achieve this result. So oftentimes, environmental scientists are forced to choose between two extremities: a limited older system or an expensive new system. MESA brings to the Earth sciences, a third choice that can bridge the gap between the two while providing industrial practices, such as containerization and continuous integration.

7.2 Future Work

The MESA system is still undergoing development and requires much more enhancements in order to provide the best possible development platform for the environmental scientists at it's affiliated research projects. MESA was created in order to address the need for application development within it's projects and to highlight a new approach in distributed systems. In the rush to achieve this main functionality of this system, many planned secondary non-vital features were pushed back into future developments. Each of the following sections covers one of the features that has been evaluated in-depth and is considered to be a significant enhancement to the MESA system.

7.2.1 Security

As MESA was developed to show a proof of concept, the steps to ensure the security of the microservices and to prevent the interception of data was overlooked. Currently, the microservices have minimal security with all of it's services placed in a domain with SSL and a singular login service handling it's encryption with SHA-256.

Unfortunately, with the advancements in GPUs, the encryption algorithms like SHA-256 are often cracked through brute force annually. In this case, SHA-256 has quickly fallen out of favor with most web developers and alternatives such as SCRYPT and BCRYPT are preferred [48, 34, 35, 47]. This is due to their nature as slow hashing algorithms and are limited when attempting to parallelize the cracking of a password.

To prevent the interception of data and verify that the client has clearance to interact with data, most modern RESTful-based software practices token-based authentication. This is the passing of a server generated token to the client and used as a verification component to ensure that the client does in fact have clearance to access the server's data. Once the server recognizes the token after comparing it to the stored list, the server then performs the action requested by the client. Unfortunately, MESA does not utilize this industry-standard practice as of yet. Currently, the services perform this actions without verification and are highly susceptible to being intercepted. Again, this is due to MESA being a prototype to show a proof of concept and security additions are considered secondary features.

In the future, a major enhancement to the MESA system would be the application of modern security practices. Each of the described solutions above would be a vital standard feature within the next iteration of the MESA system. However, it must be noted that these practices are based off of recent advancements in cyber-security, but procedures change almost annually and must be updated as such.

7.2.2 More Containerization

As mentioned, MESA uses containerization technology in the form of Docker to ensure that software will run the same, regardless of environment [22]. Currently, MESA has Docker containers operating on approximately a third of the active microservices. This is due to the compatibility of deploying containers on a Linux environment. Docker has commonly been used for Linux environments and while it does have Windows versions, it requires the deft hand of a system administrator and much configuration. During the development of MESA's Windows-based microservices,

this task was deemed secondary due to it's non-vital nature. However, there has been recent talks within the development team at the Nexus project regarding steering towards the containerization of the entire Windows virtual servers. This decision will be made with careful consideration and extensive research at a much later date. One thing is for sure though, any future iterations of the MESA system will have full containerization of each and every microservice. It is most likely that Docker will continue to be used by MESA, as well as it's derivative software, Docker Swarm.

7.2.3 More Applications

Currently, MESA support three applications officially for the environmental research projects that it is affiliated with. However, there are much more tools and applications have been requested by environmental scientists and are scheduled for completion. The following text showcases the various application that are scheduled to join the covered applications from this thesis at a later date.

Dr. Bob Boehm, a distinguished professor at the University of Nevada, Las Vegas, also doubles as the Director at the Center for Energy Research (also in Las Vegas) [41]. Dr. Boehm and his team are fellow members of the Nexus project and are solar energy researchers that have stored up a significant amount of solar data. Much like the Lysimeter Visualization, Dr. Boehm's data is scheduled to be a part of an advance data visualization with a server backend. This application will contain the same visualization approach as the Lysimeter Visualization by emphasizing on the handling of large data sets and simultaneous comparisons with other provided data. The server backend will be a singular microservice that will handle any conversions of time-series data, as well as handling the navigation aspect of the client. This will involve the constant passing of data ranges, sample rate, and the altered data to be displayed. Additional features will be added according the specification elicited by Dr. Bob Boehm and his team, and this can range from tying in a google earth display to adding in various data models.

A project was proposed by Dr. Scotty Strachan of UNR to leverage the use of data

visualization software developed by a fellow NSF environmental research group. The software mentioned is the Cloud-Hosted Real-time Data Services for the Geosciences (CHORDS) system developed by the EarthCube project [7, 14]. This software provides a free alternative to data visualizations for environmental scientists by creating a powerful, fast, automatic, and cost-less visualization solution. This software is quite significant in that there is very few data visualization solution for such a specific subdomain like environmental research. However, this software as a standalone does not contain any of the necessary components pre-built within it to interface the NRDC. It is here where a connection package was created for the NRDC to interface with a CHORDS instance. The original package was developed as a part of a year-long senior project consisting of undergraduates. As such, the software is not fully deployed and many features were ignored to construct a proof of concept. This software would be considered preliminary work and a future goal would be to break this software into it's base requirements and appropriately map them to a microservice for both preservation and usability.

Last on this list of immediate applications to finish and integrate into MESA, is the NRAQC system that was previously covered. Although significant progress has been made into the development of NRAQC, the system is not fully operational and still requires additional development work. The base system which discovers, gathers, and manages flagged data is complete, but many of the additional subsystems that allow it's various primary requirements are yet to be addressed. The system still needs to add the component of allowing concurrent processes to be made in the testing of the flagged data. Additionally, the interface in which allows environmental scientists to affect the outputs of the system non-programmatically is not yet complete. Finally, the ability to generate a powerful, dynamic, and responsive data visualization is not yet implemented for the NRAQC system. As a part of future developments, addressing these features and finalizing NRAQC for use by environmental scientists would be placed very high in the list of priorities.

7.2.4 High Performance Computing

Taking much inspiration from the PEIS system proposed by Braun et. al., a potential enhancement for the MESA system is to provide High Performance Computing (HPC) solutions to environmental scientists. This can perhaps take the form of generating data models through the use of multiple GPUs as part of a computing cluster. Another use could be in the form of data imputation where missing values are then filled by a combination of machine learning and HPC. The usage of GPUs is often a standard when it comes to complex calculations on server machines and has been a fairly common approach even in environmental science.

As great as it is to use GPU-based solutions for environmental research, there is a significant start-up cost to even set up the capability to do HPC. Starting up an appropriately sized computing cluster alone can take upwards of several tens of thousands of dollars. The upkeep of a computing cluster will require a significant amount of money, as well as dedicated technicians to maintain the hardware. Once the hardware is set up, the cluster will also require the attention of specialized system administrative staff to run. Overall, this approach would require significant investment to make this solution viable.

MESA currently does not have any access to any computing clusters, so this would require collaboration with existing clusters. A potential candidate would be the computing cluster provided by the University of Nevada, Reno. Another would be the computing cluster created and maintained in the University of Nevada, Las Vegas. Regardless, this would be a very fascinating addition to the MESA system and would be interesting approach further along into the future.

7.2.5 Automatic Code Generation

Automatic code generation, or commonly known as Automatic Programming, is the concept of feeding a high-level abstraction into some sort of software to produce a working program or code portion [40]. The study of Automatic Programming has been traced to the 1940s with the automation of paper tape [45] and is still a modern topic

of research in the field of Computer Science. There have been notable advancement in automatic code generation, such as Eclipse where it can generate textual languages with it's own proprietary modeling software [23].

To understand the process behind automatic code generation, a key concept to know is the idea of an ontology. An ontology is the explicit formal specifications of the terms in the domain and relations among them [17]. Basically, this means that an ontology, in Computer Science at least, is essentially a schematic of everything that composes a computer program, such as variables, relationships, and entities. The general idea behind the generation of code is that a programming tool such as a template processor uses an ontological model to load in all of the information about the program. Then, the programming tools use the mapping provided by the ontology to map the given information to equivalent source code. This process is very similar to a compiler, whom uses specified rules to map one coding language to it's machine code equivalent.

The way that Automatic Programming ties into MESA as a future development, would be as a potential feature where environmental scientists may generate their own microservices by providing an ontological model. To implement this, a standard must be set for the ontological models that MESA will accept. Additionally, there must be developed a unique programming tool that converts the standardized ontological model into a Python or C microservice. This task will be especially challenging since this would essentially entail the development of a pseudo-compiler. This development would be a significant addition to the MESA system in that would allow environmental scientist to create new programmatic features by just submitting a high-level abstraction.

Bibliography

[1] Ronacher Armin. Flask microframework. URL: `http://flask.pocoo.org/`. [Online; accessed August 17, 2018].

[2] Mike Bostock. Data-driven documents. URL: `https://d3js.org/`. [Online; accessed August 17, 2018].

[3] Don Box. A brief history of soap, 2001. URL: `http://www.xml.com/pub/a/ws/2001/04/04/soap.html`. [Online; accessed August 17, 2018].

[4] Eric Braun, Thorsten Schlachter, Clemens Düpmeier, Karl-Uwe Stucky, and Wolfgang Suess. A generic microservice architecture for environmental data management. In *Environmental Software Systems. Computer Science for Environmental Protection: 12th IFIP WG 5.11 International Symposium, ISESS 2017, Zadar, Croatia, May 10-12, 2017, Proceedings 12*, pages 383–394. Springer, 2017.

[5] Chase Carthen, Vinh Le, Richard Kelley, Tomasz Kozubowski, and Frederick C Harris Jr. Rewind: a transcription method and website. *International Journal of Computers and Their Applications*, 24:20–30, 2017.

[6] Robert Daigneau. *Service Design Patterns: fundamental design solutions for SOAP/WSDL and restful Web Services*. Addison-Wesley, 2011.

[7] Michael Daniels. Cloud-hosted real-time data services for the geosciences. URL: `https://www.earthcube.org/group/chords`. [Online; accessed August 17, 2018].

[8] Sergiu Dascalu, Frederick C Harris Jr, Michael McMahon Jr, Eric Fritzinger, Scotty Strachan, and Richard Kelley. An overview of the nevada climate change portal. *7th International Congress on Environmental Modelling and Software*, 2014.

[9] Nicola Dragoni, Schahram Dustdar, Stephan T Larsen, and Manuel Mazzara. Microservices: migration of a mission critical system. *arXiv preprint arXiv:1704*, 2017. URL: `https://arxiv.org/abs/1704.04173`.

[10] Nevada EPSCoR. Solar Energy Water Environment Nexus in Nevada. `https://solarnexus.epscorspo.nevada.edu/`. [Online; accessed August 17, 2018].

[11] Roy Thomas Fielding. *REST: architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[12] Python Software Foundation. Python. URL: https://www.python.org/. [Online; accessed August 17, 2018].

[13] Martin Fowler. Microservices: a definition of this new term. URL: https://martinfowler.com/articles/microservices.html. [Online; accessed August 17, 2018].

[14] NSF Geo and NSF ACI. Earthcube. URL: https://www.earthcube.org/. [Online; accessed August 17, 2018].

[15] Travis CI GmbH. Travis ci - test and deploy with confidence. URL: https://travis-ci.com/. [Online; accessed August 17, 2018].

[16] The Open Group. Service-oriented architecture standards. URL: http://www.opengroup.org/standards/soa. [Online; accessed August 17, 2018].

[17] Thomas R Gruber. A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2):199–220, 1993.

[18] Festim Halili and Erenis Ramadani. Web services: a comparison of soap and rest services. *Modern Applied Science*, 12(3):175, 2018.

[19] Hashicorp. Consul. https://www.consul.io/. [Online; accessed August 17, 2018].

[20] Moinul Hossain, Rui Wu, Jose T Painumkal, Mohamed Kettouch, Cristina Luca, Sergiu M Dascalu, and Frederick C Harris Jr. Web-service framework for environmental models. In *Internet Technologies and Applications (ITA), 2017*, pages 104–109. IEEE, 2017.

[21] CloudBees Inc. Jenkins. URL: https://jenkins.io/. [Online; accessed August 17, 2018].

[22] Docker Inc. Docker. URL: https://www.docker.com/. [Online; accessed August 17, 2018].

[23] Eclipse Foundation Inc. The platform for open innovation and collaboration — the eclipse foundation. URL: https://www.eclipse.org/. [Online; accessed August 17, 2018].

[24] Github Inc. Github: build software better, together. URL: https://github.com/. [Online; accessed August 17, 2018].

[25] Quora Inc. Quora. URL: https://www.quora.com/. [Online; accessed August 17, 2018].

[26] Cygnet Infotech. Rest- an ideal web service to integrate applications/product. URL: https://www.cygnet-infotech.com/blog/rest-an-ideal-web-service-to-integrate-applications. [Online; accessed August 17, 2018].

[27] Desert Research Institute. Scaling environmental processes in heterogeneous arid soils (sephas). URL: https://www.dri.edu/sephas. [Online; accessed August 17, 2018].

[28] Arne Johanson, Sascha Flögel, Christian Dullo, and Wilhelm Hasselbring. Oceantea: exploring ocean-derived climate data using microservices. *International Workshop on Climate Informatics (CI 2016)*:24–29, 2016.

[29] Martin Kalin. *Java Web Services: Up and Running: A Quick, Practical, and Thorough Introduction*. O'Reilly Media, Inc., 2013.

[30] Kevin Khanda, Dilshat Salikhov, Kamill Gusmanov, Manuel Mazzara, and Nikolaos Mavridis. Microservice-based iot for smart buildings. In *Advanced Information Networking and Applications Workshops (WAINA), 2017 31st International Conference on*, pages 302–308. IEEE, 2017.

[31] Alexandr Krylovskiy, Marco Jahn, and Edoardo Patti. Designing a smart city internet of things platform with microservice architecture. In *Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on*, pages 25–30. IEEE, 2015.

[32] Arun Kumar. Soap vs rest. URL: `https://aruniphoneapplication.blogspot.com/2017/03/soap-vs-rest.html`. [Online; accessed August 17, 2018].

[33] Mobomo LLC. Rest isn't what you think it is, and that's ok, 2017. URL: `https://www.mobomo.com/2010/04/rest-isnt-what-you-think-it-is/`. [Online; accessed August 17, 2018].

[34] Node Package Manager. Bcrypt. URL: `https://www.npmjs.com/package/bcrypt`. [Online; accessed August 17, 2018].

[35] Node Package Manager. Scrypt. URL: `https://www.npmjs.com/package/scrypt`. [Online; accessed August 17, 2018].

[36] Microsoft. Microsoft sql server. URL: `https://www.microsoft.com/en-us/sql-server/sql-server-2017`. [Online; accessed August 17, 2018].

[37] Microsoft. Windows communication foundation. URL: `https://docs.microsoft.com/en-us/dotnet/framework/wcf/`. [Online; accessed August 17, 2018].

[38] Rakhi Motwani, Mukesh Motwani, Frederick C Harris Jr, and Sergiu Dascalu. Towards a scalable and interoperable global environmental sensor network using service oriented architecture. In *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2010 Sixth International Conference on*, pages 151–156. IEEE, 2010.

[39] Hannah Munoz, Connor Scully-Allison, Vinh Le, Frederick C Harris Jr, and Sergiu Dascalu. A mobile quality assurance application for the nrdc. *Proceedings of the ISCA 26th International Conference on Software Engineering and Data Engineering (SEDE 2017)*:61–66, 2017.

[40] Ricardo Aler Mur. Automatic inductive programming. In *Proceedings of the 23rd international conference on machine learning, tutorial (ICML 2006)*, Carnegie Mellon University, Pittsburgh, Pennsylvania, 2006.

[41] University of Nevada Las Vegas. Center for energy research. URL: `https://www.unlv.edu/cer`. [Online; accessed August 17, 2018].

[42] University of Nevada Reno. Nevada Climate Change Portal. www.sensor.nevada.edu/NCCP. [Online; accessed August 17, 2018].

[43] University of Nevada Reno. Nevada Research Data Center. www.sensor.nevada.edu. [Online; accessed August 17, 2018].

[44] Biplab Pal. Tools and techniques for building microservices - dzone microservices, 2018. URL: https://dzone.com/articles/tools-and-techniques-to-build-microservices. [Online; accessed August 17, 2018].

[45] David Lorge Parnas. Software aspects of strategic defense systems. *Communications of the ACM*, 28(12):1326–1335, 1985.

[46] Bruno Pedro. Is rest better than soap? yes, in some use-cases, 2015. URL: https://nordicapis.com/rest-better-than-soap-yes-use-cases/.

[47] Niels Provos and David Mazieres. A future-adaptable password scheme. In *USENIX Annual Technical Conference, FREENIX Track*, pages 81–91, 1999.

[48] Frank Rietta. Use bcrypt or scrypt instead of sha* for your passwords, please! URL: https://rietta.com/blog/2016/02/05/bcrypt-not-sha-for-passwords/. [Online; accessed August 17, 2018].

[49] Peter Rogers. Service-oriented development on netkernel- patterns, processes products to reduce system complexity. URL: http://www.cloudcomputingexpo.com/node/80883. [Online; accessed August 17, 2018].

[50] Connor Scully-Allison, Vinh Le, Frederick C Harris Jr, and Sergiu Dascalu. Near real-time autonomous quality control for streaming environmental sensor data. *22nd International Conference on Knowledge-Based and Intelligent Information & Engineering Systems*, 2018.

[51] Rod Stephens. *Beginning software engineering*. John Wiley & Sons, 2015.

[52] Erik Sundvall, Mikael Nyström, Daniel Karlsson, Martin Eneling, Rong Chen, and Håkan Örman. Applying representational state transfer (rest) architecture to archetype-based electronic health record systems. *BMC medical informatics and decision making*, 13(1):57, 2013.

[53] Haishan Tian, Yuanjun He, and Hongming Cai. A vr web service for active scene using x-vrml. In *Communications and Information Technology, 2005. ISCIT 2005. IEEE International Symposium on*, volume 1, pages 405–408. IEEE, 2005.

[54] Bill Wagner. C 6.0 draft language specification. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/. [Online; accessed August 17, 2018].

[55] Rui Wu, Jose T Painumkal, Nimrat Randhawa, Lisa Palathingal, Sage R Hiibel, Sergiu M Dascalu, and Frederick C Harris Jr. A new workflow to interact with and visualize big data for web applications. In *Collaboration Technologies and Systems (CTS), 2016 International Conference on*, pages 302–309. IEEE, 2016.

[56] Rajesh Kumar Yadav. What is the difference between web services and micro services ? URL: http://rajitblog.com/2018/03/17/puspdeep-raj/difference-between-web-services-and-micro-services/. [Online; accessed August 17, 2018].

Appendix A

Microservice Code Sample

```
using Newtonsoft.Json;
using NRDC.Models.GIDMIS;
using NRDC_Infrastructure.Core_Assets;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.ServiceModel.Web;

namespace NRDC_Infrastructure.Service_Contracts
{
    public class Systems : ISystems
    {
        // Convert Strings back
        public void JsonConversion(NRDC.Models.GIDMIS.Systems Instance,
            ↪  SystemType Entry, NRDC.Models.GIDMIS.GIDMISContainer db
            ↪ )
        {
            // Retrieve Foreign Key
            var SiteRow = (from x in db.Sites where x.Unique_Identifier
                ↪ .ToString() == Entry.Site_ID select x).FirstOrDefault
                ↪ ();
            Instance.Site = SiteRow.Site;

            // Retrieve Foreign Key
            var PeopleRow = (from x in db.People where x.
                ↪ Unique_Identifier.ToString() == Entry.Manager_ID
                ↪ select x).FirstOrDefault();
            Instance.Manager = PeopleRow.Person;
```

```
        Instance.Unique_Identifier = new Guid(Entry.
            ↪ Unique_Identifier);
        Instance.Name = Entry.Name;
        Instance.Details = Entry.Details;
        Instance.Power = Entry.Power;
        Instance.Installation_Location = Entry.
            ↪ Installation_Location;
        Instance.Modification_Date = Convert.ToDateTime(Entry.
            ↪ Modification_Date);
        Instance.Creation_Date = Convert.ToDateTime(Entry.
            ↪ Creation_Date);
        Instance.Photo = Entry.Photo;
    }

    // Get All
    public Stream GetData()
    {
        try
        {
            // Initialize Return List
            List<SystemType> Results = new List<SystemType>();

            // Connection Helper
            ConnectionHelper Conn = new ConnectionHelper();
            String ConnStr;
            ConnStr = Conn.getConnectionString("ProtoNRDC");

            // LINQ Retrieve
            using (var db = new GIDMISContainer(ConnStr))
            {
                var table = (db.Systems.Any()) ? (from x in db.
                    ↪ Systems select x) : null;
                if(table != null)
                {
                    foreach (var row in table)
                    {
                        var Entry = new SystemType(row, db);
                        Results.Add(Entry);
                    }
                }
            }
```

```
            // Serialize JSON
            string JSON = JsonConvert.SerializeObject(Results);
            WebOperationContext.Current.OutgoingResponse.
                ↪ ContentType = "application/json; charset=utf-8";
            return new MemoryStream(System.Text.Encoding.UTF8.
                ↪ GetBytes(JSON));
        }
        catch (Exception e)
        {
            Console.WriteLine("An error occurred: '{0}'", e);
            throw e;
        }
    }

    // Get Singular
    public Stream GetSingular(string ID)
    {
        try
        {
            // Initialize Result
            SystemType Result;

            // Connection Helper
            ConnectionHelper Conn = new ConnectionHelper();
            String ConnStr;
            ConnStr = Conn.getConnectionString("ProtoNRDC");

            // LINQ Retrieve
            using (var db = new GIDMISContainer(ConnStr))
            {
                var Row = (from x in db.Systems where x.
                    ↪ Unique_Identifier.ToString() == ID select x).
                    ↪ FirstOrDefault();
                Result = new SystemType(Row,db);
            }

            // Serialize JSON

            string JSON = JsonConvert.SerializeObject(Result);
            WebOperationContext.Current.OutgoingResponse.
                ↪ ContentType = "application/json; charset=utf-8";
            return new MemoryStream(System.Text.Encoding.UTF8.
                ↪ GetBytes(JSON));
```

```
        }
        catch (Exception e)
        {
            Console.WriteLine("An␣error␣occurred:␣'{0}'", e);
            throw e;
        }
    }

    // Post
    public string Post(SystemType Entry)
    {
        try
        {

            // Connection Helper
            ConnectionHelper Conn = new ConnectionHelper();
            String ConnStr;
            ConnStr = Conn.getConnectionString("ProtoNRDC");

            // LINQ Delete, Update, or Create
            using (var db = new GIDMISContainer(ConnStr))
            {
                // Check if Entry exists
                var Exist = db.Systems.Any(x => x.Unique_Identifier.
                    ↪ ToString() == Entry.Unique_Identifier);

                // Delete if Delete is present
                if (Exist && Entry.Delete == true)
                {
                    var Row = (from x in db.Systems where x.
                        ↪ Unique_Identifier.ToString() == Entry.
                        ↪ Unique_Identifier select x).FirstOrDefault
                        ↪ ();
                    db.Systems.Remove(Row);
                    db.SaveChanges();
                }

                // Otherwise Update if exists
                else if (Exist)
                {
                    var Row = (from x in db.Systems where x.
                        ↪ Unique_Identifier.ToString() == Entry.
                        ↪ Unique_Identifier select x).FirstOrDefault
```

```
                    ↪ ();
                JsonConversion(Row, Entry, db);
                db.SaveChanges();
            }

            // Else Add if it doesn't
            else
            {
                NRDC.Models.GIDMIS.Systems Instance = new NRDC.
                    ↪ Models.GIDMIS.Systems();
                JsonConversion(Instance, Entry, db);
                db.Systems.Add(Instance);
                db.SaveChanges();
            }
        }

        return "Success";
    }
    catch (Exception e)
    {
        Console.WriteLine("An␣error␣occurred:␣'{0}'", e);
        return e.ToString();
    }
}

// Find Associated Deployments
public Stream Deployments(SearchType Entry)
{
    try
    {
        // Initialize Return List
        List<DeploymentType> Results = new List<DeploymentType
            ↪ >();

        // Connection Helper
        ConnectionHelper Conn = new ConnectionHelper();
        String ConnStr;
        ConnStr = Conn.getConnectionString("ProtoNRDC");

        // LINQ Delete, Update, or Create
        using (var db = new GIDMISContainer(ConnStr))
        {
```

```
                // Check if Entry exists
                var Exist = db.Systems.Any(x => x.Unique_Identifier.
                    ↪ ToString() == Entry.Unique_Identifier);
                if (Exist)
                {
                    var Rows = (from x in db.Deployments where x.
                        ↪ Systems.Unique_Identifier.ToString() ==
                        ↪ Entry.Unique_Identifier select x);
                    foreach (var Row in Rows)
                    {
                        DeploymentType Instance = new DeploymentType(
                            ↪ Row, db);
                        Results.Add(Instance);
                    }
                }
            }

            // Serialize JSON
            string JSON = JsonConvert.SerializeObject(Results);
            WebOperationContext.Current.OutgoingResponse.
                ↪ ContentType = "application/json;␣charset=utf-8";
            return new MemoryStream(System.Text.Encoding.UTF8.
                ↪ GetBytes(JSON));
        }
        catch (Exception e)
        {
            Console.WriteLine("An␣error␣occurred:␣'{0}'", e);
            throw e;
        }
    }

    // Find Associated Service Entries
    public Stream ServiceEntries(SearchType Entry)
    {
        try
        {
            // Initialize Return List
            List<ServiceEntryType> Results = new List<
                ↪ ServiceEntryType>();

            // Connection Helper
            ConnectionHelper Conn = new ConnectionHelper();
            String ConnStr;
```

```
            ConnStr = Conn.getConnectionString("ProtoNRDC");

            // LINQ Delete, Update, or Create
            using (var db = new GIDMISContainer(ConnStr))
            {
                // Check if Entry exists
                var Exist = db.Systems.Any(x => x.Unique_Identifier.
                    ↪ ToString() == Entry.Unique_Identifier);
                if (Exist)
                {
                    var Rows = (from x in db.Service_Entries where x
                        ↪ .Systems.Unique_Identifier.ToString() ==
                        ↪ Entry.Unique_Identifier select x);
                    foreach (var Row in Rows)
                    {
                        ServiceEntryType Instance = new
                            ↪ ServiceEntryType(Row, db);
                        Results.Add(Instance);
                    }
                }
            }

            // Serialize JSON
            string JSON = JsonConvert.SerializeObject(Results);
            WebOperationContext.Current.OutgoingResponse.
                ↪ ContentType = "application/json;␣charset=utf-8";
            return new MemoryStream(System.Text.Encoding.UTF8.
                ↪ GetBytes(JSON));
        }
        catch (Exception e)
        {
            Console.WriteLine("An␣error␣occurred:␣'{0}'", e);
            throw e;
        }
    }

    // Find Associated Documents
    public Stream Documents(SearchType Entry)
    {
        try
        {
            // Initialize Return List
            List<DocumentType> Results = new List<DocumentType>();
```

```
            // Connection Helper
            ConnectionHelper Conn = new ConnectionHelper();
            String ConnStr;
            ConnStr = Conn.getConnectionString("ProtoNRDC");

            // LINQ Delete, Update, or Create
            using (var db = new GIDMISContainer(ConnStr))
            {
                // Check if Entry exists
                var Exist = db.Systems.Any(x => x.Unique_Identifier.
                    ↪ ToString() == Entry.Unique_Identifier);
                if (Exist)
                {
                    var Rows = (from x in db.Documents where x.
                        ↪ Systems.Unique_Identifier.ToString() ==
                        ↪ Entry.Unique_Identifier select x);
                    foreach (var Row in Rows)
                    {
                        DocumentType Instance = new DocumentType(Row,
                            ↪  db);
                        Results.Add(Instance);
                    }
                }
            }

            // Serialize JSON
            string JSON = JsonConvert.SerializeObject(Results);
            WebOperationContext.Current.OutgoingResponse.
                ↪ ContentType = "application/json;␣charset=utf-8";
            return new MemoryStream(System.Text.Encoding.UTF8.
                ↪ GetBytes(JSON));
        }
        catch (Exception e)
        {
            Console.WriteLine("An␣error␣occurred:␣'{0}'", e);
            throw e;
        }
    }

}
```

```
// Connection Helper Class

    public class ConnectionHelper
    {
        //Members
        public string storedConnString { get; set; }
        public string SQLConnString { get; set; }
        public string EntityClientConnString { get; set; }

        public string getConnectionString(string DBName)
        {
            //get a sql connection string matching DBName
            return String.Format(ConfigurationManager.ConnectionStrings
                ↪ ["GIDMISContainer"].ConnectionString, DBName);
        }

        public string getManagementConnectionString(string DBName)
        {
            //get a sql connection string matching DBName
            return String.Format(ConfigurationManager.ConnectionStrings
                ↪ ["ProjectManagementContainer"].ConnectionString,
                ↪ DBName);
        }

        public HttpResponseMessage BuildJsonResponse(string JSON)
        {
            HttpResponseMessage Response = new HttpResponseMessage(
                ↪ System.Net.HttpStatusCode.OK);
            Response.Content = new StringContent(JSON, System.Text.
                ↪ Encoding.UTF8, "application/json");
            return Response;
        }
    }
```